Texts in Computer Science



David P. Voorhees

Guide to Efficient Software Design

An MVC Approach to Concepts, Structures, and Models

Springer

Texts in Computer Science

Series Editors

David Gries, Department of Computer Science, Cornell University, Ithaca, NY, USA
Orit Hazzan, Faculty of Education in Technology and Science, Technion—Israel Institute of Technology, Haifa, Israel

More information about this series at http://www.springer.com/series/3191

David P. Voorhees

Guide to Efficient Software Design

An MVC Approach to Concepts, Structures, and Models

Springer

David P. Voorhees
Le Moyne College
Syracuse, NY, USA

ISSN 1868-0941 ISSN 1868-095X (electronic)
Texts in Computer Science
ISBN 978-3-030-28500-5 ISBN 978-3-030-28501-2 (eBook)
https://doi.org/10.1007/978-3-030-28501-2

© Springer Nature Switzerland AG 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my parents for teaching me the value of effort and persistence.
To my wife for your friendship and love, and for putting up with me while I think about yet another pedagogical approach to teaching software design.
To my kids, I don't expect you to read this!

Preface

Why This Book?

This book is intended to be used by someone that has done some programming but has not thought seriously about software design. This book:

- Uses a case study to illustrate the proper and improper use of design concepts. The case study is intentionally small, allowing the reader to quickly understand the scope of the case study. Even though the case study is small, various alternative designs are presented and discussed to help illustrate the use of the design concepts.
- Describes software design from two perspectives: object-oriented and structured.
  The organization of this book allows you to focus on one of these software design approaches while ignoring the other approach. This is accomplished by having chapters that introduce a design concept without regard to how it may apply in an object-oriented or structured paradigm. Immediately following most of the *concept* chapters are two chapters that apply the design concepts by exploring the case study. These two chapter titles have one of the following prefixes—OOP, OOD, SP, or SD—to indicate whether the chapter describes the case study from an object-oriented programming, object-oriented design, structured programming, or structured design perspective, respectively. The expectation is that a course that uses this book will focus on only one of these two software design paradigms.
- Focuses on a bottom-up approach to describing software design concepts.
  The first chapter introduces software design by describing many perspectives on what software design is and what it is not. Chapters 2 through 10 discuss program design, which represents the bottom-up learning approach. We start with topics you are familiar with and move onto topics that may be new, while relating the new topics back to your existing knowledge. Chapter 11 begins the discussion on software design concepts that are further developed through the remaining chapters. The expectation is that the program design concepts described in Chaps. 2 through 10 are a review; when the author teaches a software design course, these chapters are covered in the first two weeks of a 15-week semester.

- Emphasizes Model–View–Controller (MVC) as an underlying architectural principle.
  This book focuses on describing fundamental software design concepts and then applies these to a stand-alone software application (i.e., an *old-fashioned* desktop application that is not using any networking technologies). This book does not discuss the development of distributed software applications. Since MVC is often used as a software architecture for network-based applications, why is MVC being emphasized in this book? Empirically, MVC continues to be a widely used software architecture. The fact that MVC promotes good software design (more about this in later chapters) leads the author to conclude that learning to apply MVC in a stand-alone software application is a nice way to reinforce software design concepts. There will be more to say about MVC in the chapter that introduces this architectural style.
- Describes design concepts that may be applied to many types of software development projects.
  A *software development process (SDP)*, also known as a software development life cycle (SDLC), describes the steps to follow when developing software. There are many different types of SDPs that have been created, documented, and used in projects. The emphasis of this book is on learning to design software that lead to object-oriented or structured solutions, irrespective of the type of SDP being used. Chapter 1 will describe how software design fits into the broader topic of SDPs.

This book is not intended to be an encyclopedia of software design concepts. It does not attempt to provide descriptions of all the various design characteristics or modeling techniques one could use to create a software design. It also does not attempt to provide descriptions of all the various processes one could use to create a software design.

Learning Research

The approach used in this book is based on findings from learning research [1], which strongly suggests that people learn best when able to use their preexisting knowledge to make connections to new information/knowledge being learned. These connections between preexisting knowledge and new knowledge are strengthened, and hopefully get stored in long-term memory, as a result of repeated use of the new concepts and their links to preexisting knowledge.

For this book, the preexisting knowledge is that the reader already knows how to develop software code. The book starts by reviewing small programming solutions that reinforce the reader's existing knowledge. A few criteria are introduced to evaluate these program designs. In addition, a small number of modeling techniques are introduced to represent the code in a more abstract form.

After the readers' preexisting knowledge is reinforced, software design concepts are introduced. Early on in the transition to learning to design software, the book continues to use program code as a vehicle for understanding the more abstract software design concepts. As the book iterates through more complex software designs, code snippets are used less frequently. Instead, the book begins to emphasize the use of design characteristics and modeling techniques as a way to represent higher levels of design abstraction.

Your Preexisting Knowledge

If you do not know how to program, STOP! Go learn how to develop software code and then come back to this book.

Active Learning

The best way to learn to program is to write and test code. If you are a first-time programmer, simply reading a book generally does not make you a good programmer. Similarly, the best way to learn to design software is to create software designs and to evaluate these designs based on some agreed upon criteria.

The case study is used throughout this book to illustrate various design concepts being introduced. With this case study, you get an in-depth look at the pros and cons of various software designs. Discussion questions and exercises in each chapter ask you to extend the case study in various ways. Additional discussion questions and exercises will ask you to apply what you've learned to other problem domains. You are encouraged to think about, discuss, and apply the concepts covered in this book. As a famous English proverb says "use it or lose it".[1]

How to Use This Book

After learning the material in Chap. 1, you may decide to focus on one of two types of software design covered in this book.

- The structured design (SD) topics presented in this book are based on structured analysis and design [3, 4] and information engineering [5] approaches to developing software.
- The object-oriented design (OOD) topics presented in this book are based on the object-oriented approach to developing software [6, 3].

[1]Apparently, the actual proverb said "skills and knowledge that are seldom applied are likely to be lost with time". *Use it or lose it* sounds so much cooler [2].

Incremental and Iterative Development

Regardless of the software design approach you chose to learn, applying the concepts introduced in this book is the best way for you to reinforce your learning. In using draft versions of this book, the author had students develop a software design incrementally and iteratively. Students would be given a few requirements to design, code and test. As more design concepts are introduced, students would refine their existing software design and be given more requirements to incrementally add to their project.

There are generally two challenges in learning to produce a good software design.

- Learning how to use abstraction when creating your design models. In particular, deciding which details should be generalized into a common design element.
- Learning the Model–View–Controller architectural style.

Using an incremental and iterative software development approach is a great way to address these two challenges. As you learn and apply the concepts presented in this book, you should expect to take the longest time in developing your understanding of how to use abstraction and in applying the Model–View–Controller architectural style.

Book Parts

Table 1 describes the four parts of this book. The duration column indicates the number of weeks the author uses to cover the book part in a 3 credit-hour 15-week semester.[2] While Parts I and II are covered quickly when the author teaches a software design course, it is important that each learner take the time to assess their own strengths and weaknesses in both program design and software design. The learner can then practice both kinds of design while applying design perspectives (i.e., see Part III) that are related to their interests.

To summarize, this book starts with a review of program design concepts to reinforce your existing programming knowledge. This knowledge is then used to introduce software design concepts. The book introduces many foundational software design concepts and provides design examples, both good and bad, to help you assess your own software designs. Before starting this book, you should have some experience writing program code. After reading this book you should have learned some foundational software design concepts, seen how these concepts are applied to a case study, and practiced applying these concepts to the case study (and hopefully) to other application domains.

[2]A 3 credit-hour class meets for 3 h per week. The 3 h of class sessions include time for lectures, reviews, and discussions.

Table 1 Four book parts

Part Title	Chapters	Duration	Description
Program Design Fundamentals	2–10	Weeks 1–2	Reinforces your understanding of some basic program design concepts. It is expected that these concepts will be a review for many of you
Introduction to Software Design	11–16	Weeks 3–5	Introduces the characteristics of a good software design and the Model–View–Controller (MVC) architectural pattern. These chapters represent the core software design topics that are further explored in Part III
Software Design Perspectives	17–33	Weeks 6–14	Introduces five additional software design topics: human–computer interaction design; quality assurance; secure design; design patterns; and persistent data storage design. *Four of these five topics are covered independent of each other, allowing the learner to cover these topics in any order they choose. The design patterns topic is the exception; the design patterns chapters refer to chapters on HCI and secure design.* See the Part III description for recommendations on which chapters to read for each of the five topics included in this portion of the book Learning one or more of these design perspectives will improve your understanding of the software design topics introduced in Chaps. 11–16. In the software design course the author teaches, all five perspectives are covered. Given the time it takes to develop a GUI, students are given the option of developing a graphical user interface for bonus points
Wrap-Up	34–35	Weeks 14–15	Introduces the notion of a software design document and presents some ideas for what you should learn next

Acknowledgements

I would like to thank the anonymous reviewers for your feedback on earlier drafts of this book. Your comments and constructive criticisms have made this a better book. Thanks to two authors of software engineering books—Shari L. Pfleeger and Roger S. Pressman—for your tireless efforts in explaining the world of software development. I know I have benefited greatly from your books, and hope in 10–20 years someone could say similar things about this book. Thanks to Louise Corrigan at Apress for forwarding my book proposal to Springer Nature. Finally, thanks to Wayne Wheeler and Simon Rees at Springer for all of your efforts in helping this book get published.

Syracuse, NY, USA
August 2019

David P. Voorhees

References

1. National Research Council (2000) How people learn: brain, mind, experience, and school, expanded edition. In: Bransford JD, Brown AL, Cocking RR (eds) Committee on developments in the science of learning
2. Wiktionary.org: use it or lose it. In: Wiktionary the free dictionary. Wikimedia foundation (2015) Available via https://en.wiktionary.org/wiki/use_it_or_lose_it. Accessed 1 June 2015
3. Pfleeger SL (1991) Software engineering: the production of quality software, 2nd edn. Macmillan, New York
4. Vick CR (1984) Introduction: a software engineering environment. In: Vick CR, Ramamoorthy CV (eds) Handbook of software engineering, Van Nostrand Reinhold, New York, pp xi–xxxii
5. Martin J (1989) Information engineering, book I: introduction. Prentice Hall, US
6. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. McGraw-Hill, New York

Contents

1 Introduction to Software Design 1
1.1 Preconditions 1
1.2 Concepts and Context 1
1.2.1 What is Software Design? 2
1.2.2 What is Not Software Design? 5
1.2.3 What are Some Approaches for Creating a Software Design? 6
1.2.4 What is Abstraction? 10
1.2.5 Software Design Within a Software Development Process 11
1.3 Post-conditions 14
References 15

Part I Program Design Fundamentals

2 Program Design Criteria and Simple Design Models 19
2.1 Preconditions 19
2.2 Concepts and Context 20
2.2.1 Case Study and Bottom-Up Approach 20
2.2.2 Evaluating Program Designs 20
2.2.3 Design Models that Describe Structure 21
2.2.4 Design Models that Describe Behavior 22
2.3 Post-conditions 24
2.4 Next Chapter? 25
Reference 25

3 OOP Case Study: Use Program Design Criteria and Simple Models 27
3.1 OOP Preconditions 27
3.2 OOP Case Study and Bottom-Up Approach 27
3.2.1 OOP Very Simple Address Book Application (ABA) 28

3.3 OOP Simple Object-Oriented Programming Designs 28
3.3.1 OOP Version A 28
3.3.2 OOP Version B: Same Simple ABA, Better Program Design 32
3.3.3 OOP Version C: No Duplicate Names 35
3.4 OOP Post-conditions 38

4 SP Case Study: Use Program Design Criteria and Simple Models 41
4.1 SP Preconditions 41
4.2 SP Case Study and Bottom-Up Approach 41
4.2.1 SP Very Simple Address Book Application (ABA) . . . 42
4.3 SP Simple Structured Programming Designs 42
4.3.1 SP Version A 42
4.3.2 SP Version B: Same Simple ABA, Better Program Design 45
4.3.3 SP Version C: No Duplicate Names 48
4.4 SP Post-conditions 51

5 Program Design and Performance 55
5.1 Preconditions 55
5.2 Concepts and Context 55
5.2.1 Performance: Time 56
5.2.2 Performance: Memory 58
5.3 Post-conditions 59
5.4 Next Chapter? 59
References 60

6 OOP Case Study: Considering Performance 61
6.1 OOP Pre-conditions 61
6.2 OOP Simple Designs 61
6.2.1 OOP Version A 62
6.2.2 OOP Version B 66
6.2.3 OOP Version C 70
6.3 OOP Post-conditions 75
Reference 75

7 SP Case Study: Considering Performance 77
7.1 SP Preconditions 77
7.2 SP Simple Designs 77
7.2.1 SP Version A 78
7.2.2 SP Version B 82
7.2.3 SP Version C 86
7.3 SP Post-conditions 90
Reference 90

8 Program Design and Security 91
8.1 Preconditions 91
8.2 Concepts and Context 91
8.2.1 Security 91
8.2.2 Information Security 93
8.3 Post-conditions 100
8.4 Next Chapter? 100
References 101

9 OOP Case Study: Considering Security 103
9.1 OOP Preconditions 103
9.2 OOP Simple Object-Oriented Programming Designs 103
9.2.1 OOP Version A 104
9.2.2 OOP Version B: A More Secure ABA 108
9.3 OOP Post-conditions 114

10 SP Case Study: Considering Security 115
10.1 SP Preconditions 115
10.2 SP Simple Structured Programming Designs 115
10.2.1 SP Version A 115
10.2.2 SP Version B: A More Secure ABA 119
10.3 SP Post-conditions 123

Part II Introduction to Software Design

11 Characteristics of Good Software Design 127
11.1 Preconditions 127
11.2 Concepts and Context 127
11.2.1 What are Characteristics of a Good Software Design? 127
11.2.2 Abstraction: The Art of Software Design 132
11.3 Post-conditions 134
References 135

12 OOD Case Study: Transition to Software Design 137
12.1 OOD Preconditions 137
12.2 OOD Transition to Software Design 137
12.2.1 OOD ABA Structure Design Models 138
12.2.2 OOD ABA Structure and Behavior Design Models . . . 138
12.2.3 OOD ABA Behavior Design Models 140
12.2.4 OOD Evaluate ABA Software Design 140
12.2.5 OOD ABA Software Design: Summary 146
12.3 OOD Top-Down Design Perspective 147
12.3.1 OOD Personal Finances: A Second Case Study 147
12.3.2 OOD Personal Finances: Structure Design Models 147

12.3.3 OOD Personal Finances: Structure and Behavior Design Models . . . 149
12.3.4 OOD Personal Finances: Behavior Design Models . . . 150
12.3.5 OOD Personal Finances: Summary . . . 151
12.4 OOD Post-conditions . . . 153
References . . . 155

13 SD Case Study: Transition to Software Design . . . 157
13.1 SD Preconditions . . . 157
13.2 SD Transition to Software Design . . . 157
13.2.1 SD ABA Structure Design Models . . . 158
13.2.2 SD ABA Structure and Behavior Design Models . . . 158
13.2.3 SD ABA Behavior Design Models . . . 160
13.2.4 SD Evaluate ABA Software Design . . . 161
13.2.5 SD ABA Software Design: Summary . . . 166
13.3 SD Top-Down Design Perspective . . . 167
13.3.1 SD Personal Finances: A Second Case Study . . . 167
13.3.2 SD Personal Finances: Structure Design Models . . . 168
13.3.3 SD Personal Finances: Structure and Behavior Design Models . . . 168
13.3.4 SD Personal Finances: Behavior Design Models . . . 169
13.3.5 SD Personal Finances: Summary . . . 170
13.4 SD Post-conditions . . . 172
References . . . 174

14 Introduction to Model-View-Controller . . . 175
14.1 Preconditions . . . 175
14.2 Concepts and Context . . . 175
14.2.1 Introduction to Model–View–Controller (MVC) . . . 176
14.3 Post-conditions . . . 179

15 OOD Case Study: Model–View–Controller . . . 181
15.1 OOD: Preconditions . . . 181
15.2 OOD: Transition to MVC . . . 181
15.3 OOD: ABA MVC Design Version A . . . 183
15.3.1 OOD: In What Order Do MVC Objects Get Created? . . . 183
15.3.2 OOD: How Do MVC Objects Communicate with Each Other? . . . 186
15.3.3 OOD: ABA MVC Version A Software Design . . . 189
15.3.4 OOD: Evaluate ABA MVC Version A Software Design . . . 190

15.4 OOD: ABA MVC Design Version B 196
15.4.1 OOD: In What Order Do MVC Objects Get Created? 198
15.4.2 OOD: How Do MVC Objects Communicate with Each Other? 199
15.4.3 OOD: Evaluate ABA MVC Version B Software Design 202
15.5 OOD Top-Down Design Perspective 204
15.5.1 OOD Personal Finances: A Second Case Study 205
15.5.2 OOD Personal Finances: Structure 205
15.5.3 OOD Personal Finances: Structure and Behavior 209
15.5.4 OOD Personal Finances: Behavior 210
15.5.5 OOD Personal Finances: Summary 211
15.6 OO Language Features 212
15.6.1 A Simple Example 213
15.7 Post-conditions 216

16 SD Case Study: Model–View–Controller 219
16.1 SD: Preconditions 219
16.2 SD: Transition to MVC 219
16.3 SD: ABA MVC Design Version A 221
16.3.1 SD: How Do MVC Components Communicate with Each Other? 221
16.3.2 SD: ABA MVC Version A Software Design 225
16.3.3 SD: Evaluate ABA MVC Version A Software Design 226
16.4 SD: ABA MVC Design Version B 231
16.4.1 SD: How Do MVC Objects Communicate with Each Other? 233
16.4.2 SD: Evaluate ABA MVC Version B Software Design 235
16.5 SD Top-Down Design Perspective 242
16.5.1 SD Personal Finances: A Second Case Study 242
16.5.2 SD Personal Finances: Structure 243
16.5.3 SD Personal Finances: Structure and Behavior 243
16.5.4 SD Personal Finances: Behavior 244
16.5.5 SD Personal Finances: Summary 245
16.6 Post-conditions 246

Part III Software Design Perspectives

17 Introduction to Human–Computer Interaction (HCI) Design 251
17.1 Preconditions 251
17.2 Concepts and Context 252
17.2.1 HCI Evaluation Criteria 252
17.2.2 Software and HCI Design Goals 255
17.2.3 HCI Design Steps 257
17.3 Post-conditions 257
References 259

18 OOD Case Study: Text-Based User Interface 261
18.1 OOD: Preconditions 261
18.2 OOD: Concepts and Context 262
18.2.1 OOD: TUI Design Alternatives 262
18.3 OOD: ABA TUI Designs 262
18.3.1 OOD: Menu and Guided Prompts 263
18.3.2 OOD: Commands and Guided Prompts 264
18.3.3 OOD: Evaluate ABA TUI Designs 265
18.3.4 OOD: Commands Design Revisited 266
18.4 OOD Case Study: TUI Design Details 268
18.5 OOD Top-Down Design Perspective 270
18.5.1 OOD Personal Finances: A Second Case Study 270
18.5.2 OOD Personal Finances: TUI Design Choices 271
18.5.3 OOD Personal Finances: Evaluate TUI Design Choices 273
18.5.4 OOD Personal Finances: TUI Design Details 274
18.6 Post-conditions 275

19 SD Case Study: Text-Based User Interface 279
19.1 SD: Preconditions 279
19.2 SD: Concepts and Context 280
19.2.1 SD: TUI Design Alternatives 280
19.3 SD: ABA TUI Designs 280
19.3.1 SD: Menu and Guided Prompts 281
19.3.2 SD: Commands and Guided Prompts 282
19.3.3 SD: Evaluate ABA TUI Designs 283
19.3.4 SD: Commands Design Revisited 285
19.4 SD Case Study: TUI Design Details 286
19.5 OOD Top-Down Design Perspective 287
19.5.1 OOD Personal Finances: A Second Case Study 288
19.5.2 OOD Personal Finances: TUI Design Choices 289

19.5.3 OOD Personal Finances: Evaluate TUI Design Choices . . . 291
19.5.4 SD Personal Finances: TUI Design Details . . . 291
19.6 Post-conditions . . . 292

20 Model–View–Controller: TUI Versus GUI . . . 297
20.1 Preconditions . . . 297
20.2 MVC User Interface: Concepts and Context . . . 298
20.2.1 Text-Based User Interface . . . 298
20.2.2 Graphical-Based User Interface . . . 298
20.2.3 Summary: TUI versus GUI . . . 299
20.2.4 Impact on MVC Design . . . 299
20.2.5 MVC Summary: TUI Versus GUI . . . 300
20.3 GUI: More Concepts and Context . . . 300
20.3.1 GUI Design Alternatives . . . 301
20.4 Post-conditions . . . 303

21 OOD Case Study: Graphical-Based User Interface . . . 305
21.1 OOD: Preconditions . . . 305
21.2 OOD: ABA GUI Design . . . 306
21.3 OOD: ABA GUI Version A . . . 306
21.3.1 OOD: Evaluate Version A GUI Design . . . 312
21.4 OOD Top-Down Design Perspective . . . 312
21.4.1 OOD Personal Finances: A Second Case Study . . . 312
21.4.2 OOD Personal Finances: GUI Design . . . 313
21.4.3 OOD Personal Finances: Evaluate GUI Design . . . 314
21.5 OOD: Post-conditions . . . 316

22 SD Case Study: Graphical-Based User Interface . . . 319
22.1 SD: Preconditions . . . 319
22.2 SD: ABA GUI Design . . . 320
22.3 SD: ABA GUI Version A . . . 320
22.3.1 SD: Evaluate Version A GUI Design . . . 326
22.4 SD Top-Down Design Perspective . . . 326
22.4.1 SD Personal Finances: A Second Case Study . . . 326
22.4.2 SD Personal Finances: GUI Design . . . 327
22.4.3 SD Personal Finances: Evaluate GUI Design . . . 328
22.5 SD: Post-conditions . . . 330

23 Is Your Design Clear, Concise, and Complete? . . . 333
23.1 Preconditions . . . 333
23.2 Concepts and Context . . . 334
23.2.1 Software Quality Assurance . . . 335
23.2.2 Formal Review . . . 335
23.2.3 Informal Review . . . 341

23.2.4 Design/Code Walkthrough . . . 341
23.2.5 Customer Survey . . . 342
23.2.6 Software Testing . . . 342
23.2.7 Summary of QA Methods . . . 344
23.3 Software QA in Software Design . . . 345
23.3.1 Formal Review of ABA Software Design . . . 346
23.3.2 Design Walkthrough of ABA Software Design . . . 347
23.4 Software QA in Software Development . . . 348
23.5 Post-conditions . . . 349
References . . . 349

24 Software Design and Security . . . 351
24.1 Preconditions . . . 351
24.2 Concepts and Context . . . 352
24.2.1 Information Security Foundations . . . 352
24.2.2 Designing Information Security in Software . . . 352
24.2.3 Cryptographic Concepts . . . 354
24.3 Use in Software Designs . . . 356
24.3.1 Data-Flow Diagram . . . 356
24.3.2 IDEF0 Function Model . . . 357
24.3.3 UML State Machine Diagram . . . 357
24.3.4 Using Other Design Models . . . 358
24.4 Post-conditions . . . 359
References . . . 360

25 OOD Case Study: More Security Requirements . . . 361
25.1 OOD: Preconditions . . . 361
25.2 OOD: ABA Security Design . . . 362
25.2.1 OOD: ABA Design Models . . . 362
25.3 OOD Top-Down Design Perspective . . . 370
25.3.1 OOD Personal Finances: A Second Case Study . . . 370
25.3.2 OOD Personal Finances: Security Design . . . 371
25.4 OOD: Post-conditions . . . 372
Reference . . . 374

26 SD Case Study: More Security Requirements . . . 375
26.1 SD: Preconditions . . . 375
26.2 SD: ABA Security Design . . . 376
26.2.1 SD: ABA Design Models . . . 376
26.3 SD Top-Down Design Perspective . . . 384
26.3.1 SD Personal Finances: A Second Case Study . . . 384
26.3.2 SD Personal Finances: Security Design . . . 384
26.4 SD: Post-conditions . . . 386
References . . . 388

27 Introduction to Design Patterns . . . 389
27.1 Preconditions . . . 389
27.2 Concepts and Context . . . 390
27.2.1 GoF Design Patterns . . . 390
27.2.2 Larman Design Patterns . . . 394
27.2.3 Fernandez Design Patterns . . . 396
27.2.4 Summary of Design Pattern Templates . . . 399
27.3 Post-conditions . . . 400
References . . . 402

28 OOD Case Study: Design Patterns . . . 403
28.1 OOD: Preconditions . . . 403
28.2 OOD: ABA Design Patterns . . . 404
28.2.1 OOD: ABA GoF Patterns . . . 404
28.2.2 OOD: ABA Larman Patterns . . . 406
28.2.3 OOD: ABA Fernandez Patterns . . . 407
28.3 OOD Top-Down Design Perspective . . . 408
28.3.1 OOD Personal Finances: A Second Case Study . . . 408
28.3.2 OOD Personal Finances: Design Patterns . . . 409
28.4 OOD: Post-conditions . . . 413
References . . . 414

29 SD Case Study: Design Patterns . . . 415
29.1 SD: Preconditions . . . 415
29.2 SD: Design Patterns . . . 416
29.2.1 SD Versus OOD . . . 416
29.3 SD: ABA Design Patterns . . . 416
29.3.1 SD: ABA GoF Patterns . . . 417
29.3.2 SD: ABA Larman Patterns . . . 420
29.3.3 SD: ABA Fernandez Patterns . . . 421
29.4 SD Top-Down Design Perspective . . . 422
29.4.1 SD Personal Finances: A Second Case Study . . . 422
29.4.2 SD Personal Finances: Design Patterns . . . 423
29.5 SD: Post-conditions . . . 425
References . . . 426

30 Modeling Persistent Data . . . 427
30.1 Preconditions . . . 427
30.2 Concepts and Context . . . 428
30.2.1 Externally Versus Internally Stored Data . . . 428
30.2.2 Logical Data Modeling . . . 428
30.2.3 Physical Data Modeling . . . 434
30.2.4 Normalization . . . 435
30.2.5 Third Normal Form (3NF) . . . 436

30.2.6 Boyce-Codd Normal Form (BCNF) . . . 437
30.2.7 Fourth Normal Form (4NF) . . . 438
30.2.8 Apply Normalization . . . 438
30.3 Post-conditions . . . 440
Reference . . . 441

31 Persistent Data Storage . . . 443
31.1 Preconditions . . . 443
31.2 XML Concepts . . . 444
31.2.1 What is XML? . . . 444
31.2.2 XML Files . . . 444
31.2.3 Document Object Model (DOM) . . . 445
31.2.4 Java Examples . . . 447
31.2.5 Python Examples . . . 451
31.3 Relational Database Concepts . . . 455
31.3.1 What is a Relational Database? . . . 455
31.3.2 Relational Database Model (aka: Physical Data Model) . . . 456
31.3.3 Structured Query Language (SQL) . . . 460
31.3.4 Embedded SQL . . . 466
31.3.5 Java Examples . . . 466
31.3.6 Python Examples . . . 470
31.4 Post-conditions . . . 473
References . . . 474

32 OOD Case Study: Persistent Storage . . . 475
32.1 OOD: Preconditions . . . 475
32.2 OOD: ABA Persistent Storage Designs . . . 476
32.2.1 OOD: ABA Data Models . . . 477
32.2.2 OOD: ABA Design—XML . . . 480
32.2.3 OOD: ABA Design—Rdb . . . 482
32.2.4 OOD: ABA Design—Summary . . . 482
32.3 OOD Top-Down Design Perspective . . . 483
32.3.1 OOD Personal Finances: A Second Case Study . . . 483
32.3.2 OOD Personal Finances: Data Models . . . 484
32.4 Post-conditions . . . 486

33 SD Case Study: Persistent Storage . . . 489
33.1 SD: Preconditions . . . 489
33.2 SD: ABA Persistent Storage Designs . . . 490
33.2.1 SD: ABA Data Models . . . 491
33.2.2 SD: ABA Design . . . 494

33.3 SD Top-Down Design Perspective 497
33.3.1 SD Personal Finances: A Second Case Study 497
33.3.2 SD Personal Finances: Data Models 498
33.4 Post-conditions 499

Part IV Wrap-Up

34 Software Design Document 505
34.1 Preconditions 505
34.2 Concepts and Context 506
34.2.1 What Would a Software Design Artifact Describe? . . . 506
34.2.2 What Would a Software Design Artifact Look Like? 506
34.3 Post-conditions 510
Reference 510

35 What's Next? 511
35.1 Preconditions 511
35.2 What Should You Do Now That You've Been Introduced to Software Design? 511
35.2.1 Dive Deeper into Book Topics 511
35.2.2 Explore Other Design Topics 513
35.3 Post-conditions 514
Reference 514

Index 515



Introduction to Software Design

1

The objective of this chapter is to provide context for what is and what is not software design, to introduce the notion of abstraction, and to position software design within a software development process.

1.1 Preconditions

The following should be true prior to starting this chapter.

- You have experience developing and testing program code.

1.2 Concepts and Context

This section presents an introduction to software design by answering the following questions.

1. What is software design?
2. What is not software design?
3. What are some approaches for creating a software design?
4. What are some examples to help me better understand the concept of abstraction?
5. How does software design fit into a software development process?
6. How does the selection of a software development process affect your software design approach?

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_1

1.2.1 What is Software Design?

There are many good answers to this question. The answers below are from four perspectives.

1.2.1.1 Software Design is a Process that, When Performed, Translates Requirements into a Design. A Software Design Process is a Bridge Between What the Software Needs to do (e.g., Requirements) and the Software Implementation (e.g., Code)

This answer explains software design from the perspective of a software development process (SDP). An SDP is a description of the steps to be performed in order to develop software.

At this point in your learning, you've likely been given programming assignments that describe what the code should do. Your task was to write the code that meets the assignment description. In essence, the instructor completed many of the SDP steps for you, leaving you with the task of writing and testing your code.

There are many SDPs, each having a unique list of steps to be performed, and entire books have been written to explain a single SDP. SDPs also vary in the types of artifacts that are created, where an artifact is an output produced by performing one or more SDP steps. The SDP topic is large and complex, and there is a significant debate about which type of SDP is better to use. The following paragraph provides a simplified description of some SDPs.

Many SDP descriptions will group steps together that are logically related to each other and give this group of steps a name. These SDPs will likely have a group that contains planning steps, analysis steps, design steps, coding steps, and testing steps. These groups are often called phases. Thus, many SDP descriptions include a plan phase, an analysis phase, a design phase, a code phase, and a test phase.

This first answer to the question *what is software design?* can be explained using these SDP phases. Before software code can be written, the developers need to know what the software needs to do. These needs are often called requirements, and requirements are identified by performing the steps in the analysis phase. Once we know the requirements, we can begin to formulate how the software code should be constructed. These formulations, called software design models, would be created by performing the steps in the design phase. Once we have design models, we can perform the steps in the code phase to translate the design models into code.[1]

[1] As mentioned, there is much debate about which SDP is best to use. Some researchers and practitioners promote the agile SDPs as being better than the more traditional approaches. I'll address this debate later in this chapter (see Sect. 1.2.5), focusing specifically on how the different types of SDPs address the need for software design. This book is focused on learning how to design software; this focus lends itself to some of the more traditional SDPs that have fairly clear descriptions for the SDP phases just described.

1.2.1.2 Software Design is a Description of the Structure and Behavior of Software at a Higher Level of Abstraction Than the Code.

This answer explains software design from the perspective of a designer. A software designer wants to describe software elements found in the solution and how these elements are associated with each other. Descriptions of software elements within a design document (i.e., artifact) are used to depict the solution at various levels of abstraction. A high-level design artifact may describe the major components and how these are associated with each other. A lower level design artifact may describe the modules/classes and how these more detailed design elements are associated with each other.

When you write software code, you are focusing on building functions (in procedural code) or methods (in object-oriented code) that are responsible for certain processing. In procedural code, functions are grouped together to create a module, and many modules may be combined to solve the problem. In object-oriented code, methods are grouped together to create a class, and many classes may be combined to solve the problem. Thus, describing the structure of a software design depends on the type of programming paradigm[2] used in the code.

- Procedural code is represented by having functions within a module and by having these modules interact with each other to solve the problem. A software design artifact that describes the *structure* of a procedural solution would need to accurately represent the notion that functions are found within modules, that modules may be combined to represent a component, and that components may be combined to represent the entire software application.
- Object-oriented code is represented by having methods within a class and by having these classes interact with each other to solve the problem. A software design artifact that describes the *structure* of an object-oriented solution would need to accurately represent the notion that methods are found within classes, that classes may be combined to represent a package, and that packages may be combined to represent the entire software application.

Software *behavior* is represented by the detailed interactions between modules/functions (in procedural code) or between classes/methods (in object-oriented code). These behaviors manifest themselves in code in three different ways.

- Software code uses three types of language statements to represent behavior. These statement types are sequence, selection, and iteration. A software design artifact describing behavior may choose to document the algorithm being implemented in the code. An algorithm may be documented using a natural language, pseudo-code, or a diagram to represent logic expressed using sequence, selection, and iteration.

[2] The two programming paradigms discussed in this book are procedural (aka imperative) and object-oriented. Procedural code may be written in languages like C, COBOL, C++, and Python. Object-oriented code may be written in languages like C++, Java, and Python. Other language paradigms include functional (e.g., lisp, scheme), logical (e.g., prolog), and declarative (e.g., HTML, SQL).

- Software code uses function or method calls, including passing parameters and optionally returning a value/object, to represent behavior. A software design artifact that describes behavior may choose to document these function/method calls.
- Software code uses variables to represent the state of the software. This notion of program state is another form of software behavior that may be described in a software design artifact.

At this point in your learning, you may have developed some program code that had a handful of functions in a module or a handful of methods in a class. You may have developed some code that required a couple of modules or classes with each containing a bunch of functions or methods. In these cases, the amount of code you needed to solve the problem was not excessive—you were able to understand the structure and behavior of your solution by reading your code.

However, when your code is orders of magnitude larger, it becomes much harder to understand the structure and behavior of a solution by reading the code. An inexperienced programmer may find it very difficult to comprehend the structure and behavior of a solution with thousands of source lines of code (SLOC), whereas a solution of a few hundred SLOC is relatively easy for them to read and understand. An experienced programmer may be able to understand a solution of a few thousand SLOC by reading the code, but a solution of tens of thousands (or larger) of SLOC would be difficult for any programmer to read and understand in a short amount of time.

The ability to quickly understand the structure and behavior of a solution is one reason why software design is important to learn. A good software designer will accurately represent the implementation code while allowing someone to understand the solution structure and behavior in far less time than having to read the code to get their understanding. A software designer needs to develop the ability to create abstractions that accurately represent the code.

1.2.1.3 Software Design is a Collection of Artifacts that Describe the Architecture, Data, Interfaces, and Components of the Software

The brief description of SDPs found above included using the term artifacts to refer to items that are produced when performing steps in a SDP. Software design artifacts should include a description of the four design categories [1] described in Table 1.1.

A software design artifact may combine text and graphical notations (i.e., models) to describe the structure and/or behavior associated with one of the four design categories. Examples of this type of design artifact are class diagrams, entity–relationship diagrams, data flow diagrams, state machine diagrams, flowcharts, data dictionaries, and pseudo-code.

A software design artifact may be a single document that contains a section for each design category—architecture, data, interfaces, and algorithms. This type of software design artifact consolidates various descriptions of the structure and/or behavior into one coherent document. Having a single design document is simply a

Table 1.1 Four Design Categories

Architecture	A high-level description of the design elements found within a system
Data	A description of the logical and physical representations of the data structures used by the system. These data structures may be used for persistent or volatile storage of data. An example of persistent data storage is a database stored on a hard drive. An example of volatile data storage is a data structure created and used during program execution. When the program execution ends, the memory-based data structure is destroyed, causing any data stored in the data structure to be lost
Interfaces	A description of the human–computer interface, the interfaces between the high-level design elements (described in the architecture), and any interfaces to external systems
Components	A description of the significant or unique processing steps contained within a high-level design element

way to organize the many individual design artifacts into a coherent whole. Chapter 34 presents a template for a single document to contain an entire software design.

1.2.1.4 Software Design is a High-Level Description of the Knowledge Represented by the Code

Many researchers consider the process of developing software to be a knowledge acquisition activity [2–4]. Writing code requires knowledge of the application domain and programming language. The code represents the storage of knowledge about the application domain. Since a software design is an abstraction of the code, software design artifacts also represent knowledge about the application domain. In addition, these artifacts represent knowledge about the software development process being used.

1.2.2 What is Not Software Design?

While it is critical to know what software design is, it is just as important to know what software design is not.

1.2.2.1 Software Design is Not Software Analysis

In software analysis, we focus on understanding *what* the software must do. Software analysis gathers and organizes information that describes the needs (i.e., requirements) that the software must fulfill. In contrast, software design focuses on the structures and behaviors that explain *how* the software satisfies the requirements. Software design applies technical and application domain knowledge to translate the needs into a design.

1.2.2.2 Software Design is Not Program Design

In program design, we focus on the coding details of our solution. Things like good variable names, good method/function names, clean method/function signatures, and the reuse of methods/functions are things we think about when doing program design. In software design, we focus on a higher level of abstraction.

1.2.2.3 Software Design is Not Programming

When programming, we focus on the syntax and semantics of a particular programming language. We use this language and its features to construct code. In software design, we focus on the syntax and semantics of modeling techniques and our natural language to communicate in graphical and written/verbal forms.

1.2.3 What are Some Approaches for Creating a Software Design?

Seven approaches for developing a software design are described in this section, including the advantages and challenges of each approach. The approaches described below are not intended to be a complete list of design approaches. Rather, these seven approaches are used to highlight the range of possible ways a design may be constructed.

1.2.3.1 Top-Down Design

A top-down approach develops a design through a process of subdivision. You start with a high-level context of the system and subdivide this context into logically connected design elements. You may then take each of these design elements and subdivide again, and so on. At some point, each subdivided design element is small enough to be well understood.

The top-down approach is also called *stepwise refinement* or *decompositional*. You produce a high-level design (e.g., identify components of the system). You then refine/decompose this design into a more detailed design (e.g., identify subcomponents for each component). You then refine/decompose this design into a more detailed design (e.g., identify design elements within each subcomponent). At some point, you've refined/decomposed into design elements that are small enough to be correctly implemented via code.

Doing stepwise refinement (i.e., taking a general concept and breaking it down into smaller concepts) is a natural process for us to do; it is a natural way for us to think. In addition, there are many examples in the physical world that allow us to see a device in terms of an entire system, its major components, and its detailed parts (e.g., automotive designs, house designs).

Doing a top-down design does have its challenges.

- Decisions made at a higher level of design will directly influence the lower levels of design. For example, perhaps we've decided that our high-level design should

consist of three components. As we begin to further divide each component, we identify a design element that does not fit nicely within one of the three components. How do we deal with this? Do we need to explore design options as fully as possible at each stage of decomposition?
- Knowing when to stop decomposing is also difficult to judge. How small should each decomposed design element be before we stop? How much detail do we need in our design? How much design detail is too much?
- Finally, decomposing may result in duplicate design elements in two distinct parts of a system. How do we avoid the possibility of duplicate design elements?

1.2.3.2 Bottom-Up Design

A bottom-up approach develops a design through a process of building up from basic elements to the entire system. You start with basic elements and combine these into logically connected subsystems (or subcomponents). You then take each subsystem and combine again, and so on. At some point, the entire system is described as a collection of basic elements.

The bottom-up approach is also called *compositional*. You produce specific design elements (e.g., data structures, methods/functions, classes). You then combine/compose these into a more general design (e.g., physical data model, class diagrams). You then combine/compose these into even more general design elements (e.g., logical data model, package diagram).

Following a bottom-up approach allows us to immediately start to code, thinking about design only when we need to combine individual code elements into a larger solution. Since most software developers start their career by learning to write code, this is a more comfortable starting point. Typically, someone has written code longer than they've done any other software development skill.

Doing a bottom-up design does have its challenges.

- Combining small design elements into a larger design may result in more rework of existing code/design. In fact, the term *refactoring* describes this exact situation. Doing lots of refactoring each time you add code may suggest that a fundamental design flaw exists in your higher level design.
- How do we identify basic design elements from the problem description? Some code/design elements will be apparent by reading the problem statement or by your prior knowledge about the problem domain. However, there are likely code/design elements that are necessary just to make the design work. How and when is this *glue* code identified?
- Identifying and combining basic design elements may be a less intuitive approach when compared to thinking top-down (i.e., hierarchically). Is learning to design using a bottom-up approach harder to learn and apply?

1.2.3.3 Process-Oriented Design

A process-oriented approach develops a design by focusing on the processes/functions being automated. Here, the terms process and function refer to the activities being performed within an organization. These activities are candidates for automation. Design models created in a process-oriented approach emphasize how processing steps are organized (i.e., performed sequentially or concurrently) to meet the needs of the organization. While this approach focuses on process/function, many of the process-oriented design models will show how information/data is being created or manipulated by the activities being automated.

Performing a process-oriented design is beneficial for organizations that have consistent (and documented) policies and procedures for the work being performed. This allows someone to translate the policies and procedures into automated steps. The terms *work flow* and *business process reengineering* are often used to describe this approach.

Following a process-oriented approach to designing software may place constraints on your design, limiting your ability to innovate. This may happen if you focus on automating existing processes without keeping an eye on making the process more efficient. Another possible weakness is that data tends to be included in a design only when viewed from the perspective of the processes that create or use the data. This could result in data structures that are designed based on work flow without regard to how the data is related to each other.

1.2.3.4 Data-Driven Design

A data-driven approach develops a design by focusing on the information/data being automated. Design models created in a data-driven approach emphasize how information/data are organized (i.e., relationships between data elements) to meet the needs of the organization. While information/data is the focus, many of the data-oriented design models will show how processing steps use the information/data.

Doing a data-driven design approach allows a designer to develop the data structures first, and then apply these structures within the appropriate processing elements identified in the design. Since software exists to act upon data (i.e., can you think of a software program that does not use any data?), thinking about data first and processing second may lead to a well-designed system.

Creating a data-driven design may limit your ability to streamline the processing involved in using the data. This is because the design is focusing on data relationships instead of processing steps. The weaknesses of the data-driven approach is analogous to the process-oriented approach. Both of these design approaches can result in weaker designs since their focus is on only one aspect of software.

1.2.3.5 Object-Oriented Design

An object-oriented approach develops a design by focusing on both process and data. In this approach, processing steps and information/data are combined into design elements called classes. A class encapsulates the processing and information into a single software element.

Creating an object-oriented design has the benefits associated with the process-oriented and data-driven approaches, while addressing many of the weaknesses of these two approaches. However, the notion of a class as a software abstraction is not an easy thing to learn. It often takes years to develop the experience necessary to consistently develop good object-oriented software designs.

1.2.3.6 Structured Design

A structured approach develops a design by identifying modules that are often arranged in a hierarchy. Thus, this has much in common with the top-down design approach previously discussed. A structured design may focus on process, data, or both.

1.2.3.7 Hybrid Design Approaches

Often, a design phase in an SDP does not strictly adhere to just one of the above approaches. For example, a design phase may describe a series of steps to be performed where some of the steps ask a designer to think about the entire system, with the goal of establishing some high-level design structures. These steps have the designer following a top-down approach. Other steps in the design phase may ask a designer to think about specific details of the system, with the goal of establishing some detailed design elements. These steps have the designer following a bottom-up approach. The use of both top-down and bottom-up approaches is a natural way to develop a software design. It allows you to think about the entire system while also focusing on those details that may be critical to the success of the software system.

Similarly, a design phase in an SDP may include steps describing the use of process-oriented, data-driven, and object-oriented approaches. For example, a design phase may include steps describing the: development of design models using the Unified Modeling Language (UML) (i.e., object-oriented approach); development of logical data models using entity–relationship notation (i.e., data-driven approach); and development of work flow diagrams representing activities to be automated (i.e., process-oriented approach).

As stated above, the design approaches just described represent some of the common ways a software design is created. Other design approaches included in an SDP may involve doing a function-oriented design or doing an event-driven design. A function-oriented design focuses on functions as the primary software abstraction. Functions can be described at a high-level of abstraction and then be subdivided into smaller functions that may be directly translated into code. An event-driven design focuses on how software should react to external stimuli. Certain types of software, including embedded software and graphical user interfaces, use this design approach. The design identifies the types of events to occur along with how the software should react to each type of event.

1.2.4 What is Abstraction?

The fourth definition of abstraction found at [5] fits the software design notion perfectly: "The act of comparing commonality between distinct objects and organizing using those similarities; the act of generalizing characteristics; the product of said generalization."

1.2.4.1 What Are Some Examples to Help me Better Understand the Concept of Abstraction?

An example of abstraction can be found in the brief SDP explanations given above. I can describe an SDP to you by talking about all of the detailed steps that must be performed. For example, when talking to another software developer I can say "I'm currently interviewing users to understand their needs" or "I'm currently modeling the manual steps we plan to automate." The software developer hearing these statements may not remember the specific details, but they will likely know that I am doing analysis. Alternatively, I can describe an SDP to you by talking about the phases that must be performed. When talking to another software developer, I can say "I'm currently doing analysis." The software developer hearing this would understand that I am currently learning about what the software must do. This person could then ask questions about the types of analysis steps we are performing (if they care to know about the details).

In software design, abstraction is about identifying broader themes from the specific requirements. For example, if the requirements indicate the need for a baseball, softball, and kickball, then we should see an opportunity to generalize these into the need for a ball that is a sphere.

As mentioned above, the term program state is used to describe the execution state of a software program. A program state is represented by the set of variables and their respective values at a given point in time. For example, an assignment statement (e.g., varName = expression) is used to alter a program state. Software designs should show the *significant* changes in state within the program or system being designed. For example, a human resources and benefits system would need to store a change in state when an employee retires. This would likely be a significant state change as the benefits available for an employee will likely be different from the benefits available to a retiree. This form of abstraction will ignore certain details, either because the detail is considered insignificant or because it is not associated with the design element being described.

Another example of abstraction is in describing programming languages. All programming languages allow a person to develop a computational model that may be executed on a computing device. The level of abstraction for a programming language is based on how close the programming language is to the hardware capabilities of the device.

In Fig. 1.1, machine language is at a low level of abstraction; this language is closest to the computing hardware. That is, the instructions found in machine language have a direct relationship to the capabilities of the hardware. In contrast, Python is at

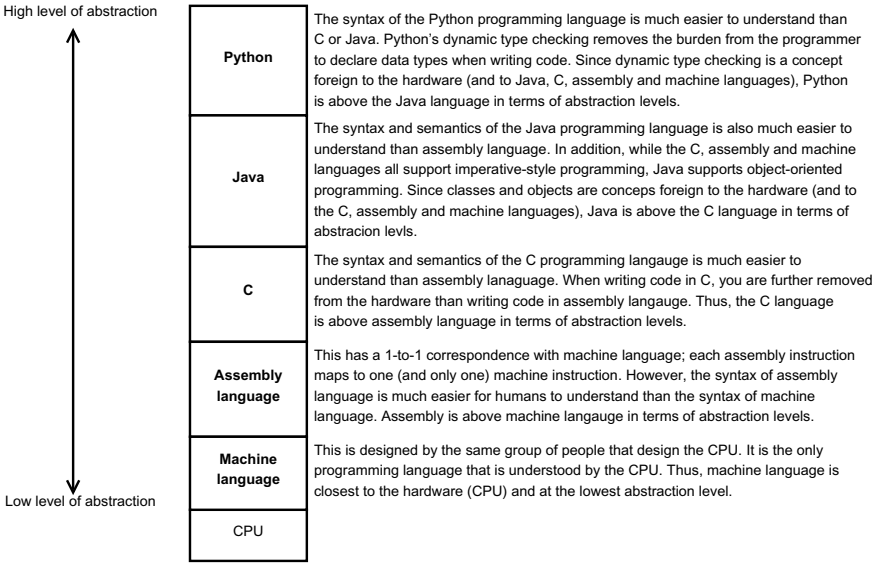


Fig. 1.1 Levels of abstraction—programming languages

a high level of abstraction; this language is further removed from the hardware than the other languages listed in the figure. While Python statements are ultimately executed on hardware, this execution occurs by using an interpreter and other software components that provide translations of Python statements into an executable form that is understood by the hardware.

1.2.5 Software Design Within a Software Development Process

This section provides context by describing software design within the broader topic of software development processes.

1.2.5.1 How Does Software Design Fit into a Software Development Process?

A software development process (SDP) is a series of steps describing the tasks to perform to create a software product. There are many different types of SDPs, each describing a unique approach to developing software. While each type of SDP describes a different approach, all SDPs have a few key things in common. Figure 1.2 shows three common steps performed in most types of SDPs. Each type of SDP describes an approach to gather information about what the software product needs to do. This information is often called requirements. The requirements are then used to produce a design, which is then implemented to produce the software product.

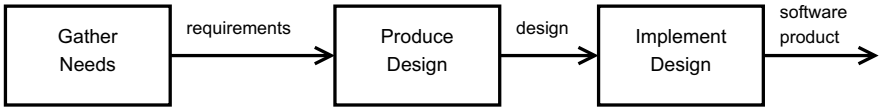


Fig. 1.2 Common steps in SDPs

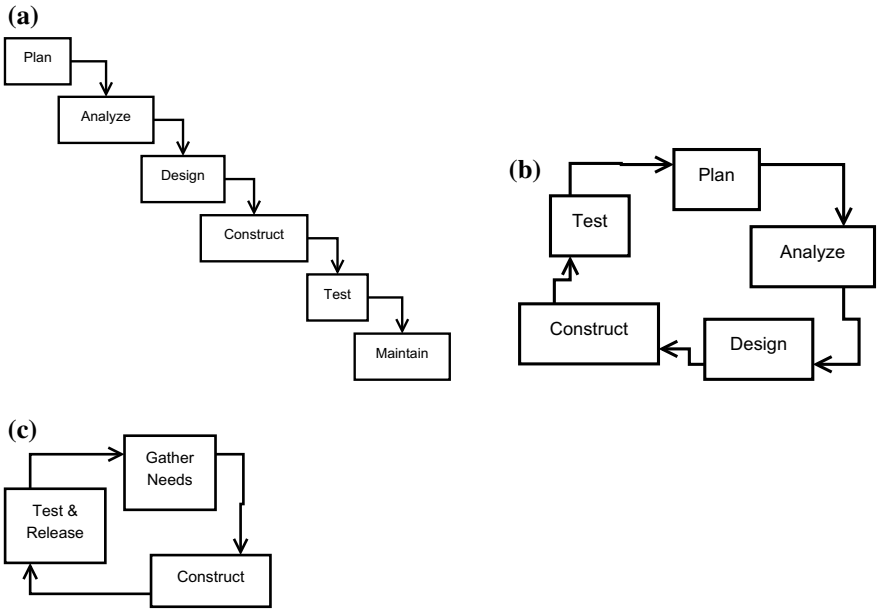


Fig. 1.3 (**a**)Waterfall, (**b**) Incremental and (**c**) Agile

Figure 1.2 shows how the process of software design acts as a bridge between gathering requirements and writing code. A software design process translates the requirements, often written in a native language (e.g., English), into a technical description that is better understood by people responsible for writing code.

1.2.5.2 How Does the Selection of a Software Development Process Affect Your Software Design Approach?

Figure 1.3a, b, and c shows three generic SDPs for a waterfall, incremental, and agile approach, respectively. These three figures are used to illustrate the similarities and differences between these approaches and, in particular, how software design fits within these approaches.

A waterfall SDP is the oldest approach for developing a software product. Its two primary characteristics are: (1) each step/phase produces one or more software documents input into the next step; and, (2) each step/phase is performed only once. Thus, a software product is produced through the sequential completion of the steps described in a waterfall SDP.

Software design in a waterfall SDP adheres to the common steps in SDPs as shown in Fig. 1.2. The *Design* step would receive as input a requirements document produced by the *Analyze* step. The Design step would then produce one or more software design documents, which are input into the *Construct* step. Some waterfall approaches will have two distinct design steps, a high-level design and a detailed design. The high-level design phase would focus on the entire system and its components. The detailed design would decompose the components into subcomponents and smaller design elements, with the goal of having each design element be detailed enough to allow for construction.

An incremental SDP is a commonly used approach for developing a software product. Its two primary characteristics are: (1) each step/phase produces one or more software documents that are then input into the next step; and, (2) the steps/phases are performed many times in a cycle, until the entire software product has been created. Thus, a software product is produced through the iterative completion of the steps described in an incremental SDP.

Software design in an incremental SDP adheres to the common steps in SDPs as shown in Fig. 1.2. The *Design* step would receive as input a requirements document produced by the *Analyze* step. The Design step would then produce one or more software design documents, which are input into the *Construct* step. The significant difference between a waterfall and incremental SDP is the contents of these software documents. In a waterfall SDP, each document produced by a step contains all of the pertinent information related to the software product being produced. In contrast, the documents produced by a step in an incremental SDP contains only the information needed for this particular cycle (or iteration) of the steps. As the iterations continue, each document is updated to reflect additional information needed to incrementally update the software product. Thus, an incremental SDP produces a software design in an evolutionary way, gradually adding design knowledge as more becomes known about the application domain.

An agile SDP is a relatively new approach for developing a software product. Its four primary characteristics are: (1) gather needs through human interactions; (2) code is the only form of technical documentation; (3) incremental development; and, (4) small increments with frequent releases of the software product.

Software design in an agile SDP does *not* adhere to the common steps in SDPs as shown in Fig. 1.2. In an agile SDP, no software design documents are typically created. Instead, the agile manifesto promotes the code as the only form of technical documentation that is needed. Agile software development is an example of following a bottom-up design approach. An agile project will typically experience lots of refactoring[3] as more becomes known about the capabilities being added to the

[3] Refactoring is the process of altering the structure of code to address changes in domain knowledge and/or technical details that were not known at the time the code was first developed. Examples of refactoring in an object-oriented language include splitting a class into many classes, combining many classes into one, splitting a method into many methods, combing many methods into one, and moving methods to a different class. Refactoring in a structured language includes splitting and combining modules and functions in ways analogous to OOP languages.

software product. From my perspective, a controversial aspect of the agile manifesto is its reliance on code as the only form of technical documentation. For projects with a relatively small scope and no more than 2–3 project team members, relying only on the code to document your design may result in a successful project. However, as project scope and/or project team size increases, the need to create design documents to effectively communicate important technical information will be important for the successful completion of the project.

1.3 Post-conditions

The following should have been learned after completing this chapter.

- Software design is a process. When the steps in a software design process are performed, software artifacts are produced to describe how the software will fulfill the requirements.
- Software designs are more beneficial as the size of the software increases.
- A software design describes the structure and behavior of the software. These structures and behaviors represent a higher level of abstraction when compared to the code.
- Software design artifacts should collectively describe the architecture, data, interfaces, and components of the software.
- Top-down is a software design approach that subdivides the problem domain into smaller design elements. For example, the domain is split into components, then the components are split into subcomponents, and so on, until the design elements are small enough to be well understood.
- Bottom-up is a software design approach that creates basic design elements and then combines these into larger design elements. Creating and combining basic design elements continues until all of the requirements have been fulfilled.
- Structured design artifacts represent abstractions of code written using modules and functions.
- Object-oriented design artifacts represent abstractions of code written using classes and methods.
- The following terms and acronyms were introduced in this chapter.

Design category	Identifies the types of design knowledge that are collected and stored in an artifact. There are four categories of design knowledge: 1. Architecture 2. Data 3. Interfaces 4. Components
SDP	Software development process
SLOC	Source lines of code

Exercises

Discussion Questions

1. From a code perspective, what are some examples of a structure? Of a behavior?
2. A *flowchart* or *Nassi–Shneiderman diagram* may be used to model the code using graphic notations that represent sequence, selection, and iteration statements. Typically, these two modeling techniques are used to graphically represent the individual statements/steps found in your code/algorithm. Using the Wikipedia pages as your source, describe how these two modeling techniques could be used to represent more abstract design models instead of the detailed processing steps they were originally intended to convey.
3. According to Pressman (2005), a design should contain artifacts that describe four design categories: architecture, data, interfaces, and components. Based on Table 1.1 and your programming experience, give an example of something you've done in your code to represent:

    a. The architecture of your solution.
    b. The data of your solution.
    c. An interface of your solution.
    d. A component of your solution.

4. An SDP provides guidance to a software development team by describing the types of steps to perform and the types of artifacts to be produced. Can you think of a situation where it may not be necessary to adhere to an SDP when developing software? For example, can you think of a situation where you may want to develop software in an ad hoc manner?

References

1. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. McGraw-Hill, New York
2. Abbas N, Andersson J, Weyns D (2011) Knowledge evolution in autonomic software product lines. In: Proceedings of the 15th international software product line conference (SPLC '11), Munich, Germany, vol 2, no 36
3. Armour PG (2000) The business of software: the case for a new business model. Commun ACM 43(8)
4. Armour PG (2004) Beware of counting LOC. Commun ACM 47(3)
5. Wiktionary.org: abstraction (2019) In: Wiktionary The free dictionary. Wikimedia Foundation. https://en.wiktionary.org/wiki/abstraction. Accessed 6 Jan 2019

Part I
Program Design Fundamentals

This first part reinforces your understanding of some basic program design concepts. The author expects these concepts to be familiar to many of you. Chapters 2 through 10 are covered in the first two weeks of a 15-week semester for a 3 credit-hour course on software design. While reviewing program design fundamentals, you may choose to focus only on object-oriented program (OOP) design or structured program (SP) design by using only those chapters whose title includes the prefix OOP or SP.



Program Design Criteria and Simple Design Models

2

The objective of this chapter is to review the criteria used to evaluate a program design and to introduce a few simple design models.

2.1 Preconditions

The following should be true prior to starting this chapter.

- You have experience developing software code.
- You understand that a structured program contains functions that are grouped into modules. An entire solution may consist of many modules where each module contains many functions.
- You understand that an object-oriented program contains methods that are grouped into classes. An entire solution may consist of many classes, where each class contains many methods.
- You understand that software design is a:
  - Process that, when followed, will produce artifacts representing an abstraction of the implementation code. These abstractions describe the structure and/or behavior of the solution.
  - Description of the structure and behavior of software at a higher level of abstraction than the code.
  - Collection of artifacts that describe the architecture, data, interfaces, and components of the software.
  - High-level description of the knowledge represented by the code.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_2

2.2 Concepts and Context

The bottom-up learning approach used in this book is briefly described. A description of program design criteria to evaluate program code is then described. This is followed by a description of a few simple design models to describe the structure or behavior of a solution.

2.2.1 Case Study and Bottom-Up Approach

The same case study is used throughout the remainder of this book to help illustrate software design thinking and alternatives. This case study is a simple Address Book Application (ABA) that will store contact information of people that you know. In part I of this book, the focus is on an ABA that does not store contact information persistently. The contact data is stored only when the application is running; each time the application is started the address book contains no data.

Storing data non-persistently was chosen as the first bottom-up design to avoid having to introduce persistent data storage technologies (e.g., XML, relational database). This should allow you to use the material in part I as a review of prior programming knowledge, which hopefully includes program design concepts.

2.2.2 Evaluating Program Designs

We'll assess the quality of the program designs using the following three criteria. Note that both structured and object-oriented program designs are discussed for each criterion. The acronyms SP (structured programming) and OOP (object-oriented programming) are used throughout this chapter to denote whether the concept is related to structured and/or object-oriented programming.

2.2.2.1 Separation of Concerns

Each function (in SP) or method (in OOP) should do one, and only one, thing. When you see a function or method that is designed to do X, but also does Y, split the function/method into smaller functions/methods so that the Y processing is in a separate function/method (which may be called from the function/method doing X). As you put distinct processing in separate functions/methods, look for opportunities to create functions/methods that are reusable.

As you group functions into modules (when doing SP) or methods into classes (when doing OOP), each module or class should have one, and only one, responsibility. For example, a Student module/class would contain functions/methods that perform computations on student data. Another module/class might be needed to represent processing performed on data for a Teacher. The Student module/class should not have any functions/methods that process Teacher data, and the Teacher module/class should not have any functions/methods that process Student data.

2.2.2.2 Design for Reuse

Redundant code should be eliminated, when possible. When you see code duplicated in two or more places, create a function or method definition and replace the duplicated code with a function or method call. A more challenging application of design for reuse is when two segments of code are similar but not identical. In this case, these similar segments of code should be put into a function/method. This new function/method will likely include parameter variables that are used to distinguish the processing described by the two (or more) original code segments.

2.2.2.3 Design Only What is Needed

Only write code for the processing that is required. Do not add extra functionality. Why? This extra functionality needs to be tested and debugged, even though it is not required. Since testing and debugging code is hard (and time consuming), this places an extra burden on the developer and tester. As Benjamin Franklin wrote many years ago—"time is money" [1]. Since developing software is a labor-intensive effort, no need to do any more than what is necessary!

2.2.3 Design Models that Describe Structure

A hierarchy chart shows the structure of an SP solution while a class diagram shows the structure of an OOP solution.

2.2.3.1 Hierarchy Chart (SP)

A hierarchy chart depicts the structure of a program by showing the call hierarchy of the functions. It shows a top-down view of the functions where the function at the top of the chart is the function that begins to execute when the program is run.

For example, Fig. 2.1 shows that Func_A is the first function to execute when the program executes since it appears at the top of the hierarchy chart. The left-to-right ordering of Func_B and Func_C in the chart indicates that Func_B is called before Func_C is called. Since both functions are immediately below Func_A, Func_A calls each of these functions. Finally, the chart shows that Func_C calls Func_D.

A hierarchy chart does not show how many times a function is called. In the example, Func_A could have a call to Func_C inside a loop statement. During program execution this may result in Func_C being called many times. A hierarchy chart does not convey this iterative program design element.

2.2.3.2 Class Diagram (OOP)

A class diagram shows the structure of a class by listing its attributes and methods (aka operations). In Fig. 2.2, Class_A is a class that contains three attributes and four methods.

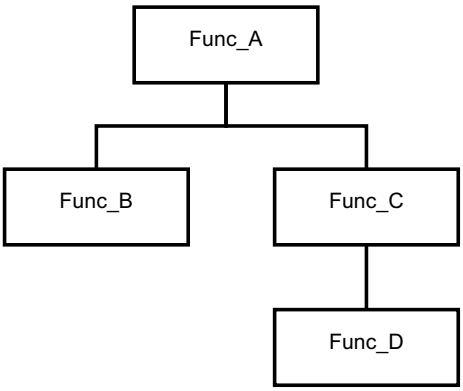


Fig. 2.1 Hierarchy chart example

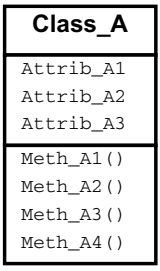


Fig. 2.2 Class diagram example

When a class diagram includes two or more classes, the diagram may use a class relationship type to describe how the two classes are related to each other. These relationship types include association, aggregation, composition, inheritance, and realizes. These relationship types will be discussed in later chapters. In addition, a class diagram will show an access modifier for each attribute and method. An access modifier indicates the code that can reference/use the attribute or method. Access modifiers will also be discussed in later chapters.

2.2.4 Design Models that Describe Behavior

The two design models described below express the behavior of a solution and may be used to describe a structured (SP) or object-oriented (OOP) solution.

2.2.4.1 Nassi–Shneiderman Diagram (SP or OOP)

This modeling technique is used to describe an algorithm performed within a solution. The entire diagram is a rectangle that contains only three types of shapes.

1. A rectangle that represents statements being executed in sequence.
2. A carpenter's square (or a sideways capital "L") that represents a loop condition and a block of statements executed each time the loop condition is true.
3. A shape that looks like the back of an envelope that represents conditional logic.

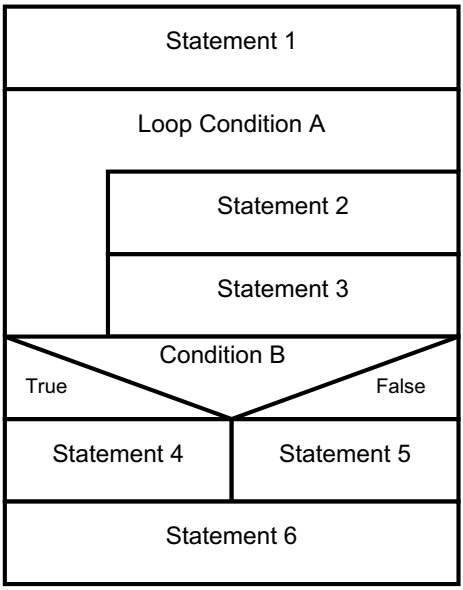


Fig. 2.3 Nassi–Shneiderman diagram example

Execution of an algorithm starts at the top of the diagram and ends when execution falls out of the bottom of the diagram.

In Fig. 2.3, execution would begin with Statement 1 and then continue to Loop Condition A. As long as this condition remains true, iteration occurs on Statement 2 and Statement 3. When Loop Condition A becomes false, execution continues with Condition B. When Condition B is true Statement 4 then Statement 6 is executed. When Condition B is false Statement 5 then Statement 6 is executed. In either case, the algorithm ends after Statement 6.

2.2.4.2 Statechart Diagram (SP or OOP)

This modeling technique is used to describe states and transitions between these states. A state is a discrete condition or situation during the life of a system or object. A transition is an action or event that is applied to a state. A transition usually results in a change to a different state. However, a transition may result in no change in state (i.e., the transition points back to the same state that initiated the action or event). Each state is given a name and most transitions are labeled to indicate the action or event that causes the transition. A statechart diagram is a powerful modeling technique that can describe the behavior of many different types of systems. Examples in Chaps. 3 and 4 will show how this type of model can be used to describe the behavior of the case study software.

In Fig. 2.4, the start state immediately transitions to state-1. When one of three events occurs (either event 1, event 2, or event 3) a transition would occur to a new state (either state-2, state-3, or state-4, respectively). Once in state-2, an occurrence of event 4 results in no change in state. The end state is reached by being in state-2 and event 6 occurring or by being in state-4 and event 7 occurring.

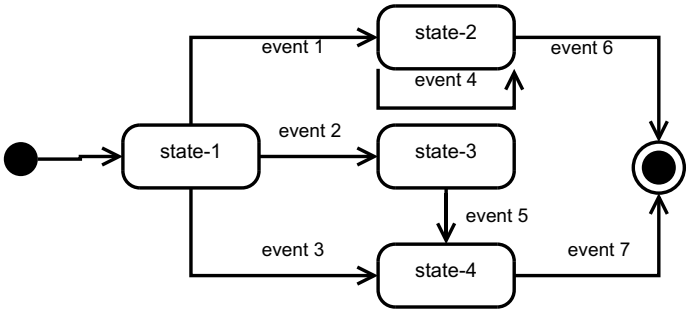


Fig. 2.4 Statechart diagram example

2.3 Post-conditions

The following should have been learned after completing this chapter.

- Program design should be evaluated using at least three criteria:
  - Separation of concerns
    Each named element of code (e.g., function or method) should do one, and only one, thing.
    When functions are grouped into a module, or when methods are grouped into a class, each module or class should have one, and only one, responsibility.
  - Design for reuse
    Redundant code should be eliminated. Parameters should be used to generalize a function or method so that it may be called in situations that are similar (but not exactly the same).
  - Design only what is needed
    Adding code for processing that is not required will increase the time needed to test the software. This extra cost should be avoided whenever possible.

- Program design models include the following.
  - A hierarchy chart is used to show the structure of a solution by illustrating the call hierarchy of functions in a structured programming solution.
  - A class diagram is used to show the structure of a solution by identifying the attributes and methods for each class in an object-oriented programming solution.
  - A Nassi–Shneiderman diagram is used to show the behavior of a solution by explaining the algorithmic processing.
  - A statechart diagram is used to show the behavior of a solution by describing the states the software is in and how the solution transitions to different states during execution.

2.4 Next Chapter?

If you are interested in seeing how the program design criteria and program design models may be applied to a small object-oriented programming solution, continue with Chap. 3. This chapter will introduce you to the Address Book Application case study using Python and Java.

If you are interested in seeing how the program design criteria and program design models may be applied to a small structured programming solution, continue with Chap. 4. This chapter will introduce you to the Address Book Application case study using Python and C++.

Exercises

Discussion Questions

1. What are the benefits to applying *separation of concerns* to a program design?
2. Can you think of a situation where you would want to ignore separation of concerns when developing a program design?
3. Besides the "time is money" argument for the *design only what is needed* program design criteria, are there other reasons why a developer should only implement what is needed?
4. What are the benefits to applying *design for reuse* to a program design?
5. Can you think of a situation where you would want to ignore design for reuse when developing a program design?

Reference

1. Franklin B (1748) Advice to a young tradesman. Retrieved 30 May 2017. https://founders.archives.gov/documents/Franklin/01-03-02-0130



OOP Case Study: Use Program Design Criteria and Simple Models

3

The objectives of this chapter are to introduce the Address Book Application case study and to apply the program design criteria and object-oriented design models to the case study.

3.1 OOP Preconditions

The following should be true prior to starting this chapter.

- You understand that an object-oriented program contains methods grouped into classes. An entire solution may consist of many classes, where each class contains many methods.
- Program design should be evaluated using at least three criteria: separation of concerns; design for reuse; and design only what is needed.
- Object-oriented design models include: class diagrams, Nassi–Shneiderman diagrams, and statechart diagrams.

3.2 OOP Case Study and Bottom-Up Approach

The same case study is used throughout the remainder of this book to help illustrate software design thinking and alternatives. This case study is a simple address book application (ABA) that will store contact information of people that you know. In this first bottom-up design chapter, the focus is on an ABA that does not store contact information persistently. The contact data is stored only when the application is running; each time the application is started, the address book contains no data.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_3

Storing data non-persistently was chosen as the first bottom-up design to avoid having to introduce new technologies (e.g., XML, relational database). This should allow you to use the material in this chapter as a review of prior programming knowledge, which hopefully includes program design concepts.

3.2.1 OOP Very Simple Address Book Application (ABA)

This very simple ABA shall

- Allow for entry and nonpersistent storage of people's names.
- Use a simple text-based user interface to obtain the names.
- Not check to determine if a duplicate name was entered.
- Not retrieve any information from the address book.

Both Python and Java code will be used to introduce OOP design concepts.

3.3 OOP Simple Object-Oriented Programming Designs

3.3.1 OOP Version A

A Python solution for the case study requirements is shown in Listing 3.1. We can execute this Python solution in a Python interpreter environment by loading the source code file and then entering addressBook() at the interpreter prompt. This Python solution has one class named ABA_OOP_A and one function named addressBook.

Listing 3.1 ABA OOP solution—Python Version A

```
#ABA_OOP_A.py: very simple object–oriented example.
# Entry and non–persistent storage of name.

def addressBook():
  aba = ABA_OOP_A()
  aba.go()

class ABA_OOP_A:
  def go(self):
    book = []
    name = input("Enter contact name ('exit' to quit): ")
    while name != "exit":
      book.append(name)
      name = input("Enter contact name ('exit' to quit): ")

    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    for name in book:
      print(name)
```

A Java solution for the case study requirements is shown in Listing 3.2. We can execute this Java solution using any Java development environment by loading the source code file, compiling the code, and then executing the program from within the development environment. This Java solution has two classes.

Listing 3.2 ABA OOP solution—Java Version A

```
//ABA_OOP_A_solution.java: very simple object–oriented example.
//  Entry and non–persistent storage of name.

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class ABA_OOP_A_solution
{
   public static void main(String[] args)
   {
      ABA_OOP_A aba = new ABA_OOP_A();
      aba.go();
   }
}

class ABA_OOP_A
{
  public void go()
  {
     ArrayList<String> book = new ArrayList<String>();
     Scanner console = new Scanner(System.in);

     System.out.print("Enter contact name ('exit' to quit): ");
     String name = console.nextLine();

     while (! name.equals("exit"))
     {
       book.add(name);
       System.out.print("Enter contact name " +
         "('exit' to quit): ");
       name = console.nextLine();
      }

      System.out.println();
      System.out.println("TEST: Display contents of address book");
      System.out.println("TEST: " +
        "Address book contains the following contacts");
      Iterator<String> iter = book.iterator();
      while (iter.hasNext())
        System.out.println(iter.next());
    }
}
```

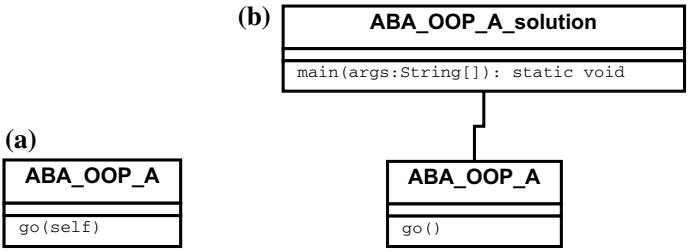


Fig. 3.1 **a** Python version A class diagram **b** Java version A class diagram

The Python solution has one function and one class. When evaluating this program design, we will ignore the addressBook function since its purpose is simply to construct an ABA_OOP_A object and then execute the go() method. The Java solution has two classes. We will evaluate this program design based on these two classes. These two OOP solutions are about as simple you can get with object-oriented programming.

3.3.1.1 Design Models

The Python and Java solutions can be described using the design modeling techniques introduced in Sects. 2.2.3 and 2.2.4. The class diagrams for these two solutions are in Fig. 3.1a, b. The Python class diagram shows the single class while the Java class diagram shows an association relationship between the class that contains the main method (which is where execution starts) and the class that contains the ABA logic.

Both the Python and Java solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 3.2 describes the algorithm used in either solution. The statechart diagram in Fig. 3.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and the software application.

The Python and Java solutions are now evaluated using the program design criteria described in Sect. 2.2.2.

3.3.1.2 Separation of Concerns

All four ABA requirements (see Sect. 3.2.1) are implemented in a single Python class. Note that we ignore the AddressBook function as this is used to simply construct an object then invoke the go method in the ABA_OOP_A class. Given this, one could argue that the Python program meets the separation of concerns criteria. However, the first requirement—allow for the entry and nonpersistent storage of people's names—states two distinct concerns. One, the ABA needs to obtain people's names and two, the ABA needs to non-persistently store these names. Given this, the Python program could be better designed.

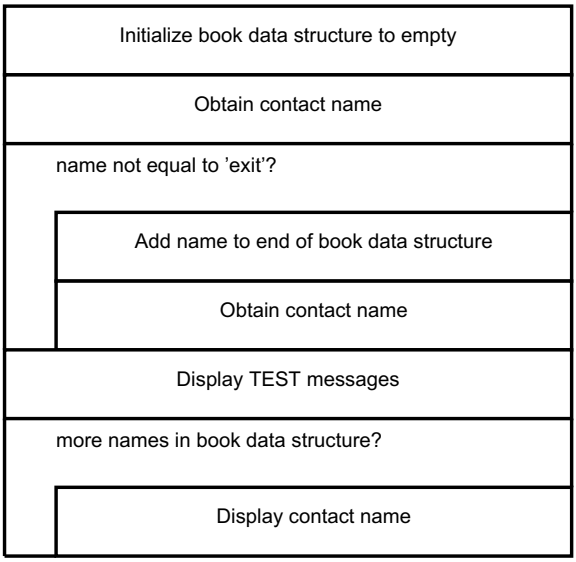


Fig. 3.2 Version A Nassi–Shneiderman diagram

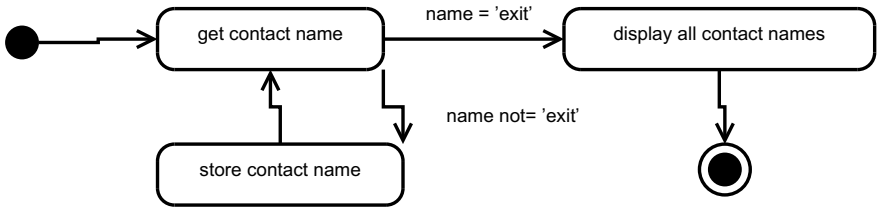


Fig. 3.3 Version A statechart

Similarly, we ignore the Java class that contains the main method, resulting in all four ABA requirements implemented in a single Java class. Like the Python solution, the Java solution could be better designed since the first ABA requirement expresses two distinct concerns.

3.3.1.3 Design for Reuse

Both the Python and Java versions contain duplicate code. In Python the *name = input(...)* statement is repeated, while in Java the two statements *System.out.print ("Enter ...");* and *String name = console.nextLine();* are repeated. Eliminating this redundant code will improve the quality of the program design.

3.3.1.4 Design Only What is Needed

Both programs include code to display the contents of the address book. This code is not necessary as it is not stated as a requirement. However, this code is included in the programs to ensure each name entered was in fact stored (not persistently) in the address book. In essence, the code displaying the address book is test code. Given this, a better design would be to put this test code in a separate function. This provides more flexibility—we could then call this test function whenever and wherever we want.

3.3.2 OOP Version B: Same Simple ABA, Better Program Design

Version B is an improved program design based on the evaluation just described for Version A. The Python code in Listing 3.3 still shows a single class, but now this class has three methods.

Listing 3.3 ABA OOP solution—Python Version B

```
#ABA_OOP_B.py: very simple object–oriented example.
# Entry and non–persistent storage of name (better).

def addressBook():
  aba = ABA_OOP_B()
  aba.go()

class ABA_OOP_B:
  def go(self):
    self.book = []
    name = self.getName()
    while name != "exit":
        self.book.append(name)
        name = self.getName()

     self.displayBook()

    def getName(self):
      return input("Enter contact name ('exit' to quit): ")

    def displayBook(self):
      print()
      print("TEST: Display contents of address book")
      print("TEST: Address book contains the following contacts")
      for name in self.book:
          print(name)
```

The Java code in Listing 3.4 still has two classes, with one class containing a very simple main method and the other class now having three methods.

Listing 3.4 ABA OOP solution—Java Version B

```
//ABA_OOP_B_solution.java: very simple object–oriented example.
//  Entry and non–persistent storage of name (better).

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class ABA_OOP_B_solution
{
    public static void main(String[] args)
    {
      ABA_OOP_B aba = new ABA_OOP_B();
      aba.go();
    }
}

class ABA_OOP_B
{
  ArrayList<String> book;
  Scanner console;

  public void go()
  {
     book = new ArrayList<String>();
     console = new Scanner(System.in);
     String name;
     name = getName();

     while (! name.equals("exit"))
     {
       book.add(name);
       name = getName();
     }

       displayBook();
  }

  public String getName()
  {
     System.out.print("Enter contact name ('exit' to quit): ");
     String name = console.nextLine();
     return name;
  }

  public void displayBook()
  {
    System.out.println();
    System.out.println("TEST: Display contents of address book");
    System.out.println("TEST: " +
      "Address book contains the following contacts");
```

```
        Iterator<String> iter = book.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

The go() methods in each version B solution controls the overall flow of execution. However, we now have two additional methods that are called by the go() method.

3.3.2.1 Design Models

The class diagrams in Fig. 3.4a, b show the structure of the Version B solutions. The Python class diagram shows the ABA_OOP_B class having a single instance variable—self.book—representing the address book data structure. This instance variable is needed since the go method stores names in self.book and the displayBook method uses this same data structure to display all of the names stored in the data structure.

The Java class diagram shows the ABA_OOP_B class having two instance variables—book and console—representing the address book data structure and a Scanner object, respectively. The book instance variable is needed since the go method stores names in book and the displayBook method uses this same data structure to display all of the names stored in the data structure. Similarly, the console instance variable is needed since the Scanner object is constructed in the go method and then used to obtain user-supplied input data via the getName method.

The Nassi–Shneiderman and statechart diagrams for the version B solutions are the same as the version A solutions.

3.3.2.2 Separation of Concerns

The first requirement—allow for the entry and non-persistent storage of people's names—is now split between two methods. The go method is responsible for storing

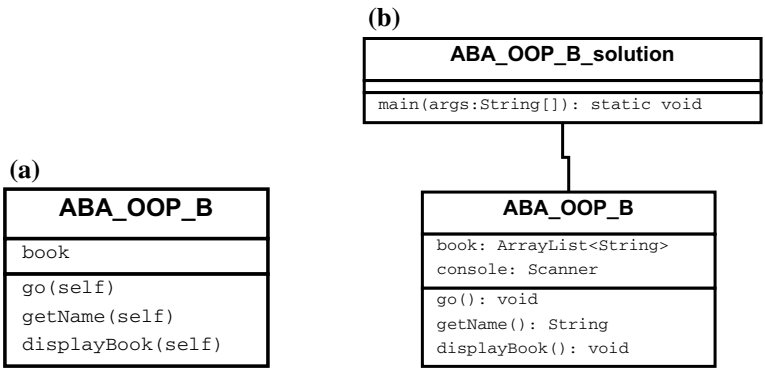


Fig. 3.4 **a** Python version B class diagram and **b** Java version B class diagram

each person's name while the getName method is responsible for obtaining a name from the user.

3.3.2.3 Design for Reuse

The duplicate code (found in version A) is now put in the getName method. Prompting for and obtaining a contact name is now done in one method; any changes to how this should be done is now localized to a single method.

3.3.2.4 Design Only What is Needed

The test code to display the contents of the address book is now all in one method. If we want to remove this test code, we can simply comment out the displayBook() call statement contained in the go method.

3.3.3 OOP Version C: No Duplicate Names

We'll now change one of the requirements (identified in italics). The ABA shall

- Allow for entry and nonpersistent storage of people's names.
- Use a simple text-based user interface to obtain the names.
- *Prevent a duplicate name from being stored in the address book.*
- Not retrieve any information from the address book.

The Python code shown in Listing 3.5 takes advantage of the *in* operator and the fact that book is a list data structure. The changed requirement is implemented in the go method via the if statement found inside the while statement.

Listing 3.5 ABA OOP solution—Python Version C

```
#ABA_OOP_C.py: very simple object–oriented example. Entry and
 non–persistent storage of name, no duplicate names (i.e., doing a
 sequential search).

def addressBook():
  aba = ABA_03c()
  aba.go()

class ABA_OOP_C:
  def go(self):
    self.book = []
    name = self.getName()
    while name != "exit":
      if name not in self.book:
        self.book.append(name)
      name = self.getName()
```

```
        self.displayBook()

    def getName(self):
      return input("Enter contact name ('exit' to quit): ")

    def displayBook(self):
      print()
      print("TEST: Display contents of address book")
      print("TEST: Address book contains the following contacts")
      for name in self.book:
         print(name)
```

The Java code shown in Listing 3.6 takes advantage of the ArrayList class and its *contains* method. The changed requirement is implemented in the go method via the if statement found inside the while statement.

Listing 3.6 ABA OOP solution—Java Version C

```
//ABA_OOP_C_solution.java: very simple object-oriented example.
//  Entry and non-persistent storage of name, no duplicate names
//  (i.e., doing a sequential search).

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Scanner;

public class ABA_OOP_C_solution
{
  public static void main(String[] args)
  {
     ABA_OOP_C aba = new ABA_OOP_C();
     aba.go();
  }
}

class ABA_OOP_C
{
   ArrayList<String> book;
   Scanner console;

   public void go()
   {
     book = new ArrayList<String>();
     console = new Scanner(System.in);
     String name;
     name = getName();

     while (! name.equals("exit"))
     {
        if (! book.contains(name))
          book.add(name);
        name = getName();
```

```
      }

      displayBook();
    }

    public String getName()
    {
       System.out.print("Enter contact name ('exit' to quit): ");
       String name = console.nextLine();
       return name;
     }

     public void displayBook()
     {
       System.out.println();
       System.out.println("TEST: Display contents of address book");
       System.out.println("TEST: " +
         "Address book contains the following contacts");
       Iterator<String> iter = book.iterator();
       while (iter.hasNext())
          System.out.println(iter.next());
     }
}
```

3.3.3.1 Design Models

Neither Version C solution results in any changes to the class diagrams shown for Version B.

The two behavior diagrams—the Nassi–Shneiderman diagram in Fig. 3.5 and the statechart in Fig. 3.6—need to show the additional logic associated with not allowing duplicate names to be stored in the address book data structure.

3.3.3.2 Separation of Concerns

The go method in both solutions are now implementing two requirements. This method implements the first requirement—allow for entry and nonpersistent storage of people's names—via the use of the append (Python) or add (Java) method. The third requirement—prevent a duplicate name from being stored in the address book—is implemented in the if statement. However, since the third requirement is implemented in Python using a built-in operator and in Java using the Java ArrayList application programming interface, this program design satisfies this criteria.

3.3.3.3 Design for Reuse

There is no duplicate code in this solution. Thus, design for reuse is satisfied in this program design.

3.3.3.4 Design Only What is Needed

Besides the displayBook method used for testing purposes, this solution contains no extra processing.

3.4 OOP Post-conditions

The following should have been learned after completing this chapter.

- Program design should be evaluated using at least three criteria:
  - Separation of concerns

    Each named element of code (e.g., method) should do one, and only one, thing.
    When methods are grouped into a class, each class should have one, and only one, responsibility.

  - Design for reuse

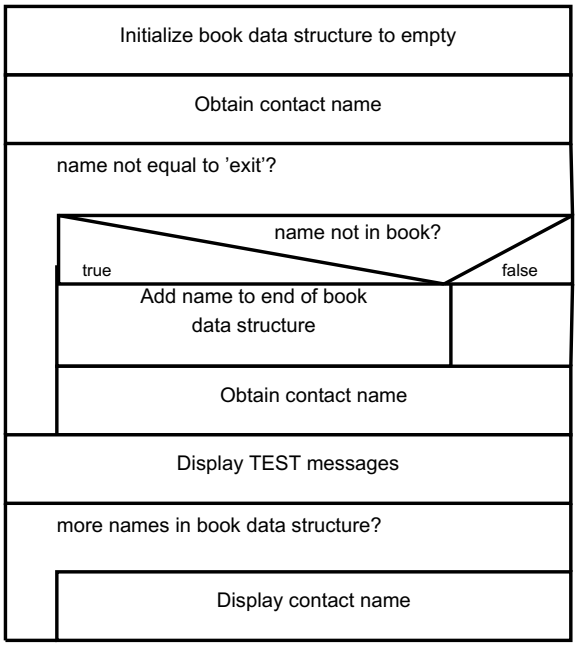


Fig. 3.5 Version C Nassi–Shneiderman diagram

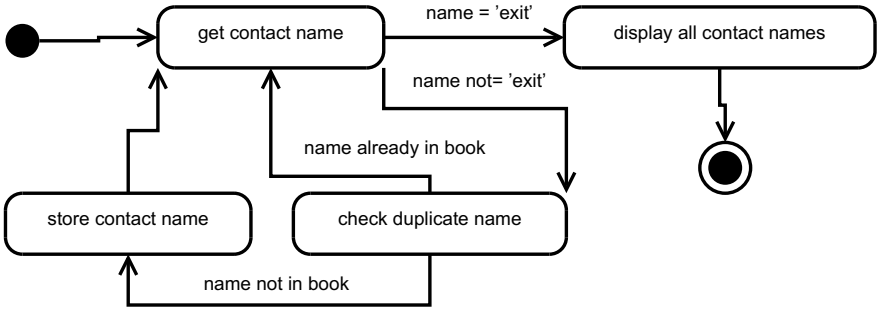


Fig. 3.6 Version C statechart

Redundant code should be eliminated. Parameters should be used to generalize a method so it may be called in situations that are similar (but not exactly the same).

– Design only what is needed

Adding code for processing that is not required will increase the time needed to test the software. This extra cost should be avoided whenever possible.

- Object-oriented program design models include the following.
  - A class diagram is used to show the structure of a solution by identifying the attributes and methods for each class in an object-oriented programming solution.
  - A Nassi–Shneiderman diagram is used to show the behavior of a solution by explaining the algorithmic processing.
  - A statechart diagram is used to show the behavior of a solution by describing the states the software is in during execution and how the solution transitions to different states during execution.

Exercises

Discussion Questions

1. What are the benefits to applying separation of concerns to a program design?
2. Can you think of a situation where you would want to ignore separation of concerns when developing a program design?

3. Identifying similar code segments that can be generalized into a single method is challenging.

   a. What types of code features should you look for when trying to identify multiple code segments that are similar enough to generalize into a method?
   b. Can you explain the steps one might take to generalize code segments into a single method?

Hands-On

1. Using an existing code solution that you've developed:

   a. Draw an appropriate model that describes the structure of your solution.
   b. Draw a Nassi–Shneiderman diagram that depicts the algorithmic processing of your solution.
   c. Draw a statechart diagram that depicts the user interactions in your solution.
   d. Improve its program design based on the criteria introduced in Sect. 2.2.2 and used in the case study in this chapter.

2. Select an application domain from the list below or select an application domain based on your interests. Write a shortlist of requirements for this application domain and then create program design models and code that meet your list of requirements. For this exercise, it is important that you keep the list of requirements short and simple. You want to focus on thinking about program design models instead of trying to figure how to implement a requirement.

   - A simple dice game like Bunco, Pig, or Poker Dice.
   - A simple card game like Concentration/Memory, Snap, or War.



SP Case Study: Use Program Design Criteria and Simple Models

4

The objectives of this chapter are to introduce the Address Book Application case study and to apply the program design criteria and structured design models to the case study.

4.1 SP Preconditions

The following should be true prior to starting this chapter.

- You understand that a structured program contains functions that are grouped into modules. An entire solution may consist of many modules where each module contains many functions.
- Program design should be evaluated using at least three criteria: separation of concerns; design for reuse; and design only what is needed.
- Structured design models include: hierarchy charts, Nassi–Shneiderman diagrams, and statechart diagrams.

4.2 SP Case Study and Bottom-Up Approach

The same case study is used throughout the remainder of this book to help illustrate software design thinking and alternatives. This case study is a simple Address Book Application (ABA) that will store contact information of people that you know. In this first bottom-up design chapter, the focus is on an ABA that does not store contact information persistently. The contact data is stored only when the application is running; each time the application is started, the address book contains no data.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_4

Storing data non-persistently was chosen as the first bottom-up design to avoid having to introduce new technologies (e.g., XML, relational database). This should allow you to use the material in this chapter as a review of prior programming knowledge, which hopefully includes program design concepts.

4.2.1 SP Very Simple Address Book Application (ABA)

This very simple ABA shall

- Allow for entry and nonpersistent storage of people's names.
- Use a simple text-based user interface to obtain the names.
- Not check to determine if a duplicate name was entered.
- Not retrieve any information from the address book.

Both Python and C++ code will be used to introduce SP design concepts.

4.3 SP Simple Structured Programming Designs

4.3.1 SP Version A

A Python solution for the case study requirements is shown in Listing 4.1. We can execute this Python solution in a Python interpreter environment by loading the source code file and then entering addressBook() at the interpreter prompt. This Python solution has one function named addressBook.

Listing 4.1 ABA SP solution—Python Version A

```
#ABA_SP_A.py: very simple structured example.
# Entry and nonpersistent storage of name.

def addressBook():
    book = []
    name = input("Enter contact name ('exit' to quit): ")
    while name != "exit":
        book.append(name)
        name = input("Enter contact name ('exit' to quit): ")

    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    for name in book:
        print(name)
```

A C++ solution for the case study requirements is shown in Listing 4.2. We can execute this C++ solution using any C++ development environment by loading the

source code file, compiling and linking the code, and then executing the program from within the development environment. This C++ solution has one function named main.

Listing 4.2 ABA SP solution—C++ Version A

```
//ABA_SP_a.cpp: very simple structured example.
// Entry and nonpersistent storage of name.

#include <iostream>
#include <string>
#include <list>

using namespace std;

int main()
{
    list<string> book;
    string name;
    cout << "Enter contact name ('exit' to quit): ";
    getline(cin, name);
    while (name != "exit")
    {
        book.push_back(name);
        cout << "Enter contact name ('exit' to quit): ";
        getline(cin, name);
    }

    cout << endl;
    cout << "TEST: Display contents of address book\n";
    cout << "TEST: Address book contains the following contacts\n";
    for (list<string>::iterator iter = book.begin();
         iter != book.end(); iter++)
        cout << *iter << endl;

    return 0;
}
```

Both Python and C++ solutions contain just one function.

4.3.1.1 Design Models

The Python and C++ solutions can be described using the design modeling techniques introduced in Sects. 2.2.3 and 2.2.4. Since each solution contains one function, each hierarchy chart (see Fig. 4.1a, b) shows just one function at the top of the hierarchy.

Both the Python and C++ solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 4.2 describes the algorithm used in either solution. The statechart diagram in Fig. 4.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and the software application.

The Python and C++ solutions are now evaluated using the program design criteria described in Sect. 2.2.2.



Fig. 4.1 **a** Python Version A hierarchy chart **b** C++ Version A hierarchy chart

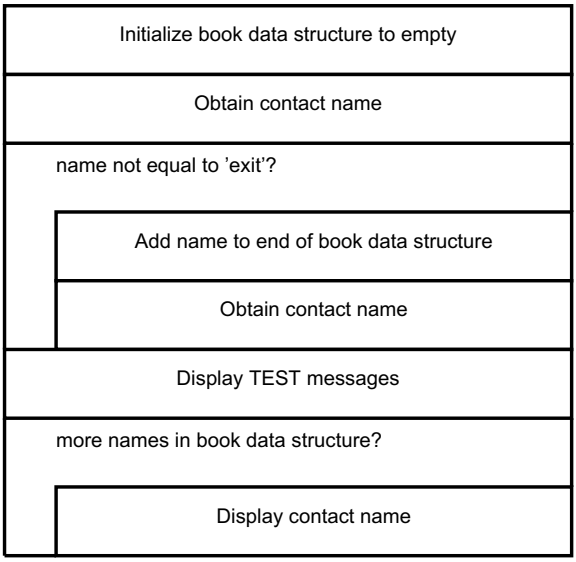


Fig. 4.2 Version A Nassi–Shneiderman diagram

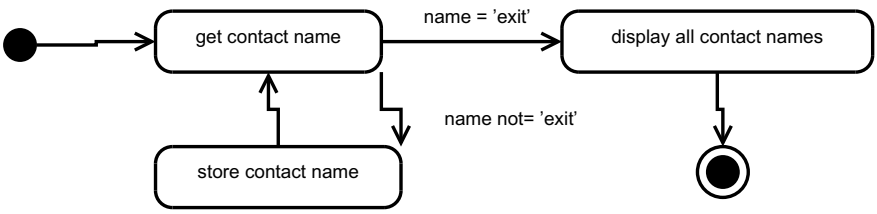


Fig. 4.3 Version A statechart

4.3.1.2 Separation of Concerns

All four ABA requirements (see Sect. 4.2.1) are implemented in a single Python function. Given this, one could argue that the Python program meets the separation of concerns criteria. However, the first requirement—allow for the entry and nonpersistent storage of people's names—states two distinct concerns. One, the ABA needs to obtain people's names and two, the ABA needs to non-persistently store these names. Given this, the Python program could be better designed.

Similarly, the C++ solution implements all four ABA requirements in a single function. Like the Python solution, the C++ solution could be better designed since the first ABA requirement expresses two distinct concerns.

4.3.1.3 Design for Reuse

Both the Python and C++ versions contain duplicate code. In Python the *name = input(...)* statement is repeated, while in C++ the two statements *cout « "Enter..."*; and *getline(...);* are repeated. Eliminating this redundant code will improve the quality of the program design.

4.3.1.4 Design Only What is Needed

Both programs include code to display the contents of the address book. This code is not necessary as it is not stated as a requirement. However, this code is included in the programs to ensure each name entered was in fact stored (not persistently) in the address book. In essence, the code displaying the address book is test code. Given this, a better design would be to put this test code in a separate function. This provides more flexibility—we could then call this test function whenever and wherever we want.

4.3.2 SP Version B: Same Simple ABA, Better Program Design

Version B is an improved program design based on the evaluation just described for Version A. The Python code in Listing 4.3 now has three functions.

Listing 4.3 ABA SP solution—Python Version B

```
#ABA_SP_B.py: very simple structured design example.
# Entry and nonpersistent storage of name (better).

def addressBook():
    book = []
    name = getName()
    while name != "exit":
        book.append(name)
        name = getName()

    displayBook(book)

def getName():
    return input("Enter contact name ('exit' to quit): ")

def displayBook(book):
    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    for name in book:
        print(name)
```

The C++ code in Listing 4.4 now has three functions.

Listing 4.4 ABA SP solution—C++ Version B

```
//ABA_SP_B.cpp: very simple structured design example.
// Entry and nonpersistent storage of name (better).

#include <iostream>
#include <string>
#include <list>

using namespace std;

string getName();
void displayBook(list<string> book);

int main()
{
    list<string> book;
    string name;
    name = getName();
    while (name != "exit")
    {
        book.push_back(name);
        name = getName();
    }

    displayBook(book);

    return 0;
}

string getName()
{
    string name;
    cout << "Enter contact name ('exit' to quit): ";
    getline(cin, name);
    return name;
}

void displayBook(list<string> book)
{
    cout << endl;
    cout << "TEST: Display contents of address book\n";
    cout << "TEST: Address book contains the following contacts\n";
    for (list<string>::iterator iter = book.begin();
         iter != book.end(); iter++)
        cout << *iter << endl;
}
```

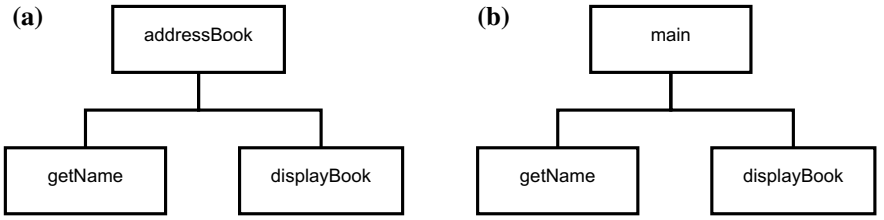


Fig. 4.4 **a** Python Version B hierarchy chart **b** Java Version B hierarchy chart

The addressBook and main functions in each Version B solution control the overall flow of execution. However, we now have two additional functions that are called by the addressBook (or main) function.

4.3.2.1 Design Models

The hierarchy charts in Fig. 4.4a, b shows the structure of the Version B solutions. The Python hierarchy chart shows the addressBook function calling the getName function and then the displayBook function. Similarly, the C++ hierarchy chart shows the main function calling the getName function and then the displayBook function.

The Nassi–Shneiderman and statechart diagrams for the Version B solutions are the same as the Version A solutions.

4.3.2.2 Separation of Concerns

The first requirement—allow for the entry and nonpersistent storage of people's names—is now split between two functions. The addressBook (or main) function is responsible for storing each person's name while the getName function is responsible for obtaining a name from the user.

4.3.2.3 Design for Reuse

The duplicate code (found in Version A) is now put in the getName function. Prompting for and obtaining a contact name is now done in one function; any changes to how this should be done is now localized to a single function.

4.3.2.4 Design Only What is Needed

The test code to display the contents of the address book is now all in one function. If we want to remove this test code, we can simply comment out the displayBook(book) call statement contained in the addressBook (or main) function.

4.3.3 SP Version C: No Duplicate Names

We'll now change one of the requirements (identified in italics). The ABA shall

- Allow for entry and nonpersistent storage of people's names.
- Use a simple text-based user interface to obtain the names.
- *Prevent a duplicate name from being stored in the address book.*
- Not retrieve any information from the address book.

The Python code in Listing 4.5 takes advantage of the *in* operator and the fact that book is a list data structure. The changed requirement is implemented in the addressBook function via the if statement found inside the while statement.

Listing 4.5 ABA SP solution—Python Version C

```
#ABA_SP_C.py: very simple structured design example.
# Entry and nonpersistent storage of name, no duplicate names
# (i.e., doing a sequential search).

def addressBook():
    book = []
    name = getName()
    while name != "exit":
        if name not in book:
            book.append(name)
        name = getName()

    displayBook(book)

def getName():
    return input("Enter contact name ('exit' to quit): ")

def displayBook(book):
    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    for name in book:
        print(name)
```

The C++ code in Listing 4.6 includes a fourth function named notFound that does a sequential search of the list data structure looking for a duplicate name. The main function calls this new function via the if statement found inside the while statement.

Listing 4.6 ABA SP solution—C++ Version C

```
//ABA_SP_C.cpp: very simple structured design example.
// Entry and nonpersistent storage of name, no duplicate names
// (i.e., doing a sequential search).

#include <iostream>
#include <string>
#include <list>
```

```
using namespace std;

string getName();
void displayBook(list<string> book);
bool notFound(list<string> book, string name);

int main()
{
    list<string> book;
    string name;
    name = getName();
    while (name != "exit")
    {
        if (notFound(book, name))
            book.push_back(name);
        name = getName();
    }

    displayBook(book);

    return 0;
}

string getName()
{
    string name;
    cout << "Enter contact name ('exit' to quit): ";
    getline(cin, name);
    return name;
}

void displayBook(list<string> book)
{
    cout << endl;
    cout << "TEST: Display contents of address book\n";
    cout << "TEST: Address book contains the following contacts\n";
    for (list<string>::iterator iter = book.begin();
         iter != book.end(); iter++)
        cout << *iter << endl;
}

//pre: book contains zero or more string values.
//post: returns false when name is in book.
//    otherwise, returns true (i.e., didNotFind is true).
bool notFound(list<string> book, string name)
{
    bool didNotFind = true;
    list<string>::iterator iter = book.begin();
    while (iter != book.end() && *iter != name)
        iter++;
    if (iter != book.end())
```

```
        didNotFind = false;
    return didNotFind;
}
```

4.3.3.1 Design Models

In Python, changing this requirement does not affect the hierarchy chart. However, the C++ solution contains an additional function to deal with the sequential search of the address book data structure. These two hierarchy charts are in Fig. 4.5a, b.

The two behavior diagrams—the Nassi–Shneiderman diagram in Fig. 4.6 and the statechart in Fig. 4.7—need to show the additional logic associated with not allowing duplicate names to be stored in the address book data structure.

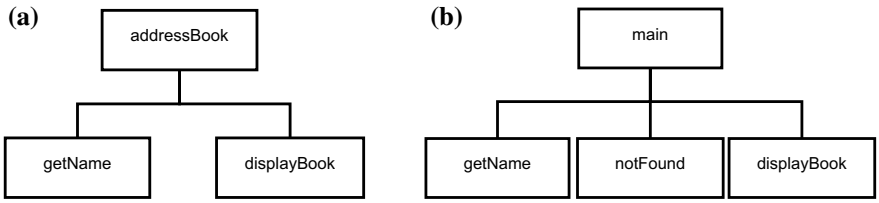


Fig. 4.5 **a** Python Version B hierarchy chart **b** Java Version B hierarchy chart

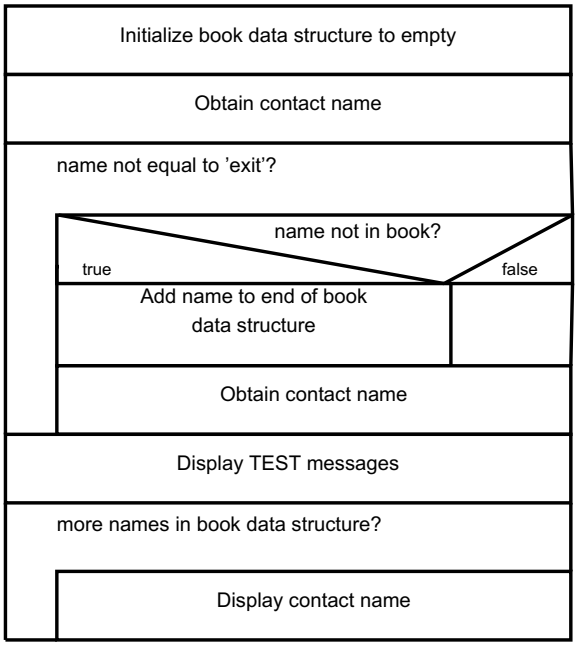


Fig. 4.6 Version C Nassi–Shneiderman diagram

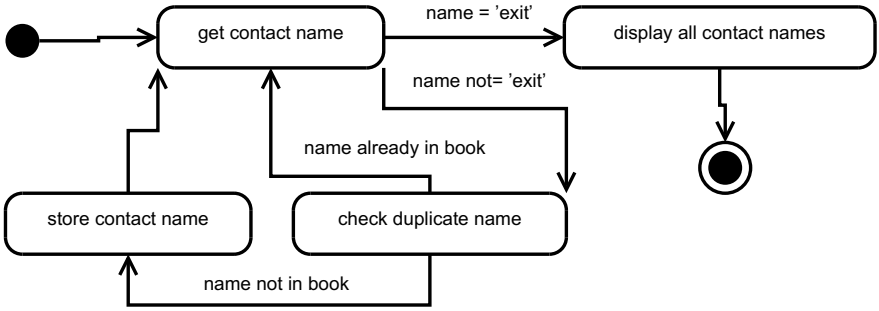


Fig. 4.7 Version C statechart

4.3.3.2 Separation of Concerns

The addressBook and main functions in these solutions are now implementing two requirements. These functions implement the first requirement—allow for entry and nonpersistent storage of people's names—via the use of the append (Python) or push_back (C++) method. The third requirement—prevent a duplicate name from being stored in the address book—is implemented in the if statement. However, since the third requirement is implemented in Python using a built-in operator and in C++ by calling the programmer-defined function notFound, this program design satisfies this criteria.

4.3.3.3 Design for Reuse

There is no duplicate code in this solution. Thus, design for reuse is satisfied in this program design. In addition, the C++ notFound function could be used on any list <string>data structure when we want to know if a particular string value is already in the list. While it's unlikely we would need to reuse this function within the ABA (i.e., using something other than the book list structure), it may be useful in other applications.

4.3.3.4 Design Only What is Needed

Besides the displayBook function used for testing purposes, this solution contains no extra processing.

4.4 SP Post-conditions

The following should have been learned after completing this chapter.

- Program design should be evaluated using at least three criteria:

- Separation of concerns
  Each named element of code (e.g., function) should do one, and only one, thing.
  When functions are grouped into a module, each module should have one, and only one, responsibility.
- Design for reuse
  Redundant code should be eliminated. Parameters should be used to generalize a function so it may be called in situations that are similar (but not exactly the same).
- Design only what is needed
  Adding code for processing that is not required will increase the time needed to test the software. This extra cost should be avoided whenever possible.

- Structured program design models include the following.

  - A hierarchy chart is used to show the structure of a solution by illustrating the call hierarchy of functions in a structured programming solution.
  - A Nassi–Shneiderman diagram is used to show the behavior of a solution by explaining the algorithmic processing.
  - A statechart diagram is used to show the behavior of a solution by describing the states the software is in during execution and how the solution transitions to different states during execution.

Exercises

Discussion Questions

1. What are the benefits of applying separation of concerns to a program design?
2. Can you think of a situation where you would want to ignore separation of concerns when developing a program design?
3. Identifying similar code segments that can be generalized into a single function is challenging.

   a. What types of code features should you look for when trying to identify multiple code segments that are similar enough to generalize into a function?
   b. Can you explain the steps one might take to generalize code segments into a single function?

Hands-on

1. Using an existing code solution that you've developed:

   a. Draw an appropriate model that describes the structure of your solution.
   b. Draw a Nassi–Shneiderman diagram that depicts the algorithmic processing of your solution.
   c. Draw a statechart diagram that depicts the user interactions in your solution.
   d. Improve its program design based on the criteria introduced in Sect. 2.2.2 and used in the case study in this chapter.

2. Select an application domain from the list below or select an application domain based on your interests. Write a short list of requirements for this application domain and then create program design models and code that meet your list of requirements. For this exercise, it is important that you keep the list of requirements short and simple. You want to focus on thinking about program design models instead of trying to figure out how to implement a requirement.

   - A simple dice game like Bunco, Pig, or Poker Dice.
   - A simple card game like Concentration/Memory, Snap, or War.



Program Design and Performance

5

The objective of this chapter is to discuss performance as another program design criteria.

5.1 Preconditions

The following should be true prior to starting this chapter.

- You understand three program design criteria: separation of concerns, design for reuse, and design only what is needed. You've evaluated program code using these criteria.
- If you are learning about structured design, then you know that a hierarchy chart may be used to model the structure of your program design.
- If you are learning about object-oriented design, then you know a class diagram may be used to model the structure of your program design.
- You know that a Nassi-Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

5.2 Concepts and Context

The performance of software should consider runtime and memory usage.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_5

5.2.1 Performance: Time

All of the ABA Chaps. 3 and 4 program solutions use a sequence data structure to store the address book data. The Python and C++ solutions use a list data structure while Java uses an ArrayList data structure. When the code needs to determine whether a contact name is already in the data structure, a sequential search is performed of the data structure. When the number of entries in the address book is large (i.e., think hundreds of thousands and larger), the linear time (e.g., $O(n)$) behavior of a sequential search could have an observable affect on the responsiveness of the ABA.

It may be impractical to think of an address book as containing hundreds of thousands, millions, or even billions of contacts. It may also be unreasonable to assume that a search taking a few seconds means that the application is no longer useful to its users; having one poor performing software application on a computing device may be acceptable to a user. But if software developers ignore performance considerations, then users would end up using many poor performing applications. Given the desire to have computing platforms capable of operating many software applications simultaneously, having many poorly performing applications would compound the responsiveness issues for a user. Even with the increase in hardware computing power, the *time complexity* (i.e., time performance) of software applications is still something that software programmers and designers should pay attention to.

When looking to correct poor performing software, there are a handful of choices a software designer may opt to implement:

- Change the algorithm to one that has better performance.
- Change the data structure to one that has better performance.
- Implement multithreading and move the poor performing code to a separate thread.
- Target a computing device that has significantly more processing capabilities.

We will focus on the first two items in the above list.

5.2.1.1 Algorithm Performance

What if we change the sequential search to a binary search? A binary search exhibits $O(\log_2 n)$ time, which is significantly faster than $O(n)$ for large n. For example, for n=10 a linear search would take 10 time units while a binary search would take about 4.3 time units. However, when n=1000 a binary search would only take about 11 time units! A potential issue with doing a binary search is that the values being searched must be in sort order. To answer the question about switching to a binary search, we need to look at each programming language.

The Python list data structure supports indexing (using the [ ] operator) and has a sort method that sorts the list contents in place (i.e., it does not use any extra memory to perform the sort). Python also has a built-in function, named sorted, that will sort the contents into a new list object. The indexing operator exhibits constant time $O(1)$ performance while both the sort method and the sorted() function have $O(n \log n)$ time performance [1].

With Python lists, we would first do a binary search to see if the contact name is already in the ABA. This would have $O(\log_2 n)$ time performance. When the contact name is not in the ABA, we would add the new contact name to the end of the list data structure (constant time $O(1)$ performance) and then sort the list (with $O(n \log n)$ performance) to ensure the next binary search works correctly. Thus, switching from a sequential search to a binary search in the Python solution takes us from $O(n)$ to a worse time performance of at least $O(n \log n)$. (When n=1000, $n \log n$ is 3000.)

The Java ArrayList data structure supports indexing (using the get method) and the Java Collections class has a static sort method that will sort an ArrayList. The ArrayList get method exhibits constant time performance while the Collections sort method has $O(n \log n)$ time performance [2]. Thus, switching from a sequential search to a binary search in the Java solution also takes us from $O(n)$ to $O(n \log n)$ performance.

The C++ list data structure is a doubly linked list data structure that does not support indexing. Adding an item to the middle of the list exhibits linear performance while adding an item to the front or back of the list exhibits constant time performance. With no support for indexing, sorting a C++ list data structure is an expensive operation that would result in overall time performance that is worse than linear time.

In the case of these three programming languages and the data structures being used, the faster binary search algorithm would not improve the overall time performance of the ABA.

5.2.1.2 Data Structure Performance

What if we change the type of data structure being used to store the contact names? Would this improve the performance? To answer the question about switching the type of data structure, we need to look at each programming language.

In the Python solution, we can switch from a list data structure to a dictionary data structure. Since a Python dictionary is implemented as a hash table, the hash function used to locate the key value (e.g., the contact name) within the dictionary represents the major impact on time performance. When there are no collisions generated by the hash function, the time performance for storing and retrieving a contact name would be $O(1)$ constant time. However, when the hash function generates many collisions for the different contact names then the time performance would get closer to linear time [1]. Thus, the worst-case performance when using a dictionary matches the best-case performance when using a list.

In the Java solution, we can switch from an ArrayList to a TreeSet. A Java TreeSet implements a red-black tree that guarantees $O(\log_2 n)$ performance when storing and retrieving an item from the data structure [2]. Note that Java also has a HashSet data structure that would exhibit similar time performance as a Python dictionary.

In the C++ solution, we can switch from a list to a set. A C++ set is typically implemented using a red-black tree that guarantees $O(\log_2 n)$ performance when storing and retrieving an item from the data structure [3].

In the case of these three programming languages, other data structures exist that would allow the ABA to exhibit time performance that is better than the linear time associated with the Chaps. 3 (OOP) or 4 (SP) solutions.

5.2.2 Performance: Memory

Another way to evaluate the performance of a program design is to determine its memory usage. Two different solutions to the same set of requirements may use dramatically different amounts of memory. Memory usage can even vary for the same program design implemented in different programming languages.

For example, a Python dictionary is implemented internally as a hash table [4]. A hash table allocates memory for each table entry based on the initial size of the hash table. In Python, the initial size of a hash table is 8 and the collision resolution algorithm uses quadratic open addressing (to avoid looking at contiguous hash table entries for an empty slot). When the hash table is two-thirds full, Python will either quadruple the hash table size (when the hash table has fewer than 50K slots) or double the size (when the hash table has over 50K slots). A Java TreeSet is implemented as a binary search tree (BST). A BST allocates memory for each node in the tree where each node must know if it has child nodes (to allow the BST to be searched quickly).

Both a Python dictionary and a Java TreeSet improve search time, as noted above, but how do these two data structures affect memory usage? Since a hash table has a fixed size, a hash table could have the capacity to store thousands or millions of entries but perhaps there are times when a large hash table is sparsely populated. In this case, one could argue that the hash table is wasting a significant amount of memory. In contrast, a Java TreeSet only has nodes for values that are currently stored in the BST. Thus, a TreeSet makes more efficient use of memory when compared to a sparsely populated hash table.

Parameter passing is another type of memory usage that should be evaluated. This type of memory usage affects the runtime stack, which has a limit to its size. If you've ever written a recursive function that calls itself infinitely many times, you've caused a stack overflow exception that indicates that the size limit to the runtime stack has been exceeded.

In Python and Java, any object (e.g., a Python dictionary or a Java TreeSet) passed as a parameter to a function/method is passed as an object reference. An object reference is a small data value (perhaps 4 bytes in size) that refers to where the object can be found in memory. For example, if we had written the Python displayBook method/function as *def displayBook(book):*, then each call to displayBook would have resulted in a pointer (memory address) value being stored on the runtime stack. This is much more efficient than having to copy the entire data structure onto the runtime stack as part of each method call. Similarly, the Java displayBook method could be written as *public void displayBook(TreeSet book)*. Each call to displayBook would (again) result in a pointer (memory address) value being stored on the runtime stack.

In C++, the default passing mechanism used is *pass by value*. While a C++ set is an object, the default passing mechanism means that a copy of the set will be placed on the runtime stack. This copy is what is used by the method being called. With pass by value, the method does not have access to the original set object. This default behavior can potentially use lots of memory when the data structure contains lots of data. More about this will be discussed in Chap. 7 on structured programming.

5.3 Post-conditions

The following should have been learned after completing this chapter.

- Evaluating the performance of a program design should consider both time and memory usage. More specifically:
  - Adjusting a program design due to poor time performance should look at both the algorithms and data structures being used.
  - Adjusting a program design due to poor memory usage should consider how a programming language does parameter passing and how data structures are being used to store data.

5.4 Next Chapter?

If you are interested in seeing how performance may be applied to a small object-oriented programming solution, continue with Chap. 6. This chapter will continue the Address Book Application case study using Python and Java.

If you are interested in seeing how performance may be applied to a small structured programming solution, continue with Chap. 7. This chapter will continue the Address Book Application case study using Python and C++.

Exercises

Discussion Questions

1. Can you think of a situation where it would be okay to ignore poor time performance in your solution?

2. Can you think of a situation where it would be okay to ignore poor memory performance in your solution?
3. With the fairly rapid increase in processor speed and memory sizes, even in very small computing devices, why is performance still something that should be considered when developing a solution?
4. Cloud computing is a widely used service. Examples of cloud services include email, personal financing, social media, and e-commerce. Since many cloud computing services use large server farms to provide the service, would performance of a cloud-based software app be something that should still be considered? Why?

References

1. Python Software Foundation: TimeComplexity. Python Wiki https://wiki.python.org/moin/TimeComplexity. Accessed 3 Jun 2014
2. Oracle.com: Java Platform Standard Edition 7 API. https://docs.oracle.com/javase/7/docs/api/. Accessed 2 Jun 2014
3. Nyhoff L (2005) ADTs, data structures, and problem solving with C++. Prentice Hall, Upper Saddle River
4. Python.org (2017) Dictionary object implementation using a hash table. http://svn.python.org/projects/python/trunk/Objects/dictobject.c. Accessed 12 Jun 2017



OOP Case Study: Considering Performance

6

The objective of this chapter is to apply all four program design criteria—separation of concerns, design for reuse, design only what is needed, and performance—to the case study.

6.1 OOP Pre-conditions

The following should be true prior to starting this chapter.

- Evaluating the performance of a program design should consider both time and memory usage.
- You understand three program design criteria: separation of concerns, design for reuse, and design only what is needed. You've evaluated program code using these criteria.
- You know that a class diagram may be used to model the structure of your object-oriented program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

6.2 OOP Simple Designs

We'll start with the ABA requirements as stated for Version C in Chap. 3. The ABA shall

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_6

- Allow for entry and (nonpersistent) storage of people's names.
- Use a simple text-based user interface to obtain the names.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

6.2.1 OOP Version A

The Version A solutions shown in Listings 6.1 and 6.2 are a result of changing the data structure being used to store contact names. These changes were made to improve the responsiveness of the ABA based on the time performance improvements described in Chap. 5.

Listing 6.1 ABA_OOP_A.py

```
#ABA_OOP_A.py: very simple object-oriented example.
# Entry and non-persistent storage of name,
# no duplicate names (using dictionary).

def addressBook():
  aba = ABA_OOP_A()
  aba.go()

class ABA_OOP_A:
  def go(self):
    self.book = {}
    name = self.getName()
    while name != "exit":
      if name not in self.book:
        self.book[name] = None
      name = self.getName()

    self.displayBook()

  def getName(self):
    return input("Enter contact name ('exit' to quit): ")

  def displayBook(self):
    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    for name in self.book:
      print(name)
```

The *if name not in self.book* conditional statement in Listing 6.1 results in Python using the built-in hash function __hash__() for the string (str) data type [1]. In a worst-case scenario, the hash function would generate the same hash table location for each contact name. The result would be a sequential search based on the use of the quadratic open addressing collision resolution algorithm. As noted in Chap. 5,

this worst case would result in linear $O(n)$ time performance which matches the performance of the solutions in Chap. 3.

The Java code for Version A is shown in Listing 6.2. This Java program design is significantly different since it is using a TreeSet instead of an ArrayList. As described in Chap. 5, this results in logarithmic time $O(\log_2 n)$ performance when determining if the contact name already exists in the ABA. Specifically, the add method of the TreeSet Java API class will not modify the TreeSet when the name is already in the data structure.

Listing 6.2 ABA_OOP_A.java

```
//ABA_OOP_A_solution.java: very simple object-oriented example.
// Entry and non-persistent storage of name,
// no duplicate names (using TreeSet).

import java.util.Iterator;
import java.util.Scanner;
import java.util.TreeSet;

public class ABA_OOP_A_solution
{
   public static void main(String[] args)
   {
      ABA_OOP_A aba = new ABA_OOP_A();
      aba.go();
    }
}

class ABA_OOP_A
{
  TreeSet<String> book;
  Scanner console;

  public void go()
  {
    book = new TreeSet<String>();
    console = new Scanner(System.in);
    String name;
    name = getName();

    while (! name.equals("exit"))
    {
      book.add(name);
      name = getName();
    }

    displayBook();
  }

    public String getName()
   {
     System.out.print("Enter contact name ('exit' to quit): ");
```

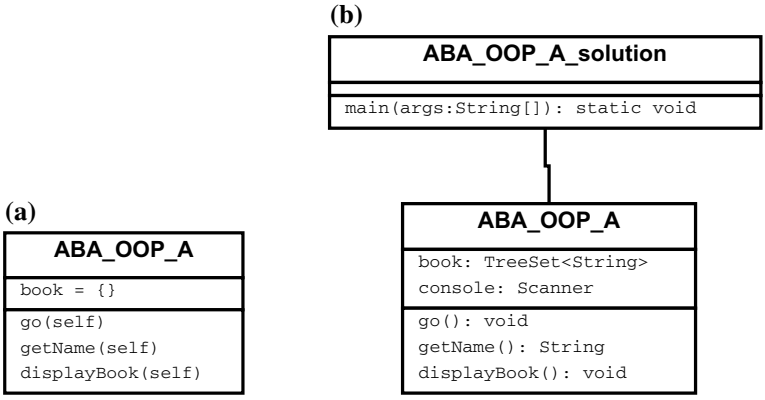


Fig. 6.1 **a** Python version A class diagram. **b** Java version A class diagram

```
        String name = console.nextLine();
        return name;
    }

    public void displayBook()
    {
        System.out.println();
        System.out.println("TEST: Display contents of address book");
        System.out.println("TEST: Address book contains the " +
            "following contacts");
        Iterator<String> iter = book.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

6.2.1.1 Design Models

The class diagrams for these two solutions are in Fig. 6.1a, b. The Python class diagram lists the same attributes and methods as shown in the Version B and C solutions in Sects. 3.3.2 and 3.3.3, respectively. The only difference is that the book attribute is initialized to an empty list in Chap. 3 while in this chapter it is a dictionary. Likewise, the Java class diagram lists the same attributes and methods as shown in the Version B and C solutions in Chap. 3. The only difference is that the book attribute is an ArrayList in Chap. 3 while in this chapter it is a TreeSet.

Both the Python and Java solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 6.2 describes the algorithm used in either solution. The statechart diagram in Fig. 6.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and the software application.

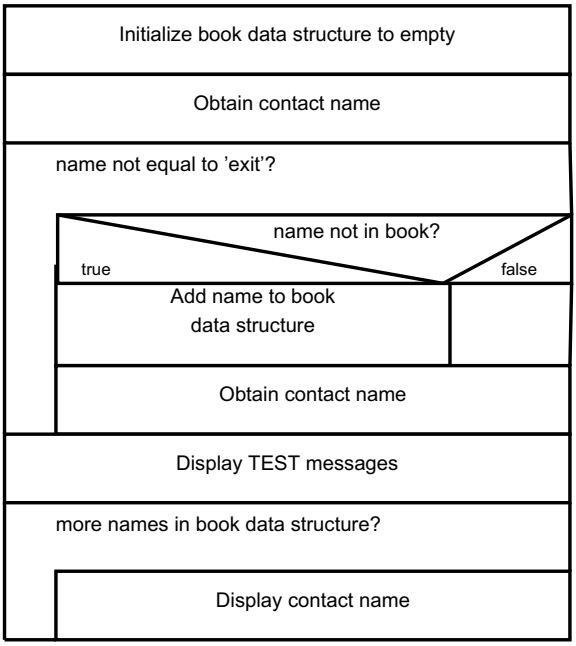


Fig. 6.2 Version A Nassi–Shneiderman diagram

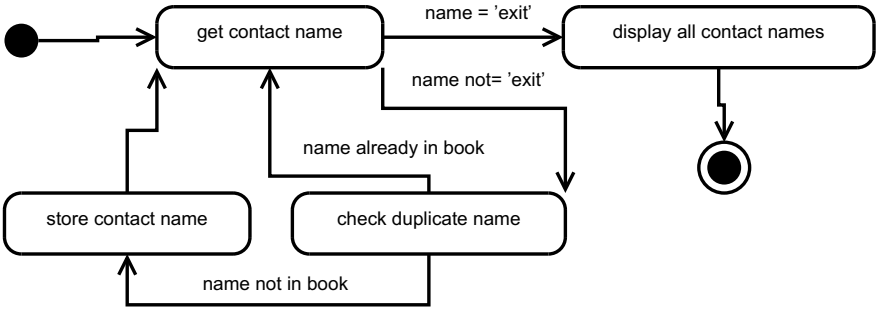


Fig. 6.3 Version A statechart

Note the Nassi–Shneiderman diagram shows a selection statement *name not in book?* inside the while loop. In the Python solution, we see this if statement in the code shown in Fig. 6.1. For the Java solution, the add method of the TreeSet class will not add the element when the element is already in the data structure. In effect, this add method contains a selection statement that prevents a duplicate name from being stored in the data structure.

The Python and Java solutions are now evaluated using the program design criteria described in Chaps. 2 and 5.

6.2.1.2 Program Design Criteria (from Chap. 2)

In terms of separation of concerns, design for reuse, and design only what is needed, changing the type of data structure used to store the address book data has no impact on these program design criteria. This solution exhibits the same program design characteristics as the Version C solution in Chap. 3.

6.2.1.3 Time Performance

Changing the type of data structure used to store the address book data will have a positive impact on the responsiveness of the ABA. To summarize, the time performance of checking to determine if the contact name is a duplicate was studied. Using a Python dictionary instead of a Python list produced a best case of constant time instead of linear time. Using a Java TreeSet instead of a Java ArrayList produced a best case of logarithmic time instead of linear time.

6.2.1.4 Memory Performance

As mentioned in Chap. 5, use of a Python dictionary means that a hash table is being used. Given the nonpersistent storage of contacts at this point in the ABA, it is quite likely that the hash table will contain a small number of entries that will not result in allocating a lot of memory that does not get used.

For the Java solution, the TreeSet data structure will contain a node only for those contacts that are currently in the ABA. Thus, the Java solution will only allocate memory in the TreeSet when it is needed to store contact data.

6.2.2 OOP Version B

Our address book is not very useful since it only stores contact names. Let's add one more requirement, identified in italics below. The ABA shall

- Allow for entry and (nonpersistent) storage of people's names.
- *Store for each person a single phone number.*
- Use a simple text-based user interface to obtain the contact data.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

The Python solution is shown in Listing 6.3. This solution also uses a dictionary, and stores a phone number value instead of None as the mapped data value. Specifically, the statement *book[name] = None* in Version A has been replaced with the statement *book[name] = phone*. The displayBook method has also been modified to display the address book names in alphabetical order. This was done so the Python and Java programs for Version B exhibit the same behavior.

Listing 6.3 ABA_OOP_B.py

```
#ABA_OOP_B.py: very simple object-oriented example.
# Entry and non-persistent storage of name,
# no duplicate names, a phone number.

def addressBook():
  aba = ABA_OOP_B()
  aba.go()

class ABA_OOP_B:
  def go(self):
    self.book = {}
    name = self.getName()
    while name != "exit":
      phone = self.getPhone(name)
      if name not in self.book:
        self.book[name] = phone
      name = self.getName()

      self.displayBook()

    def getName(self):
      return input("Enter contact name ('exit' to quit): ")

    def displayBook(self):
      print()
      print("TEST: Display contents of address book")
      print("TEST: Address book contains the following contacts")
      sortedNames = sorted(self.book.keys())
      for name in sortedNames:
          print(name, self.book[name])

    def getPhone(self, name):
      phone = input("Enter phone number for " + name + ": ")
      return phone
```

The Java solution in Listing 6.4 uses a map instead of a set, since a map stores a key–value pair whereas a set only stores keys. Specifically, the data type *TreeMap<string, string>* indicates the map will store a key that is a string (i.e., contact name) along with a value related to the key (i.e., phone number) which is also a string.

Listing 6.4 ABA_OOP_B.java

```
//ABA_OOP_B_solution.java: very simple object-oriented example.
// Entry and non-persistent storage of name,
// no duplicate names, a phone number.

import java.util.Collection;
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.Scanner;
```

```
import java.util.TreeMap;

public class ABA_OOP_B_solution
{
  public static void main(String[] args)
  {
     ABA_OOP_B aba = new ABA_OOP_B();
     aba.go();
  }
}

class ABA_OOP_B
{
  TreeMap<String,String> book;
  Scanner console;

  public void go()
  {
    book = new TreeMap<String,String>();
    console = new Scanner(System.in);
    String name, phone;
    name = getName();

    while (!name.equals("exit"))
    {
       phone = getPhone(name);
       if (!book.containsKey(name))
         book.put(name, phone);
       name = getName();
    }

    displayBook();
  }

    public String getName()
    {
      System.out.print("Enter contact name ('exit' to quit): ");
      String name = console.nextLine();
      return name;
    }

    public void displayBook()
    {
      System.out.println();
      System.out.println("TEST: Display contents of address book");
      System.out.println("TEST: Address book contains the " +
          "following contacts");

      NavigableSet<String> keySet = book.navigableKeySet();
      Collection<String> valueColl = book.values();

      Iterator<String> iterKey = keySet.iterator();
      Iterator<String> iterValue = valueColl.iterator();
```

```
        while (iterKey.hasNext() && iterValue.hasNext())
          System.out.println(iterKey.next() + " " + iterValue.next());
    }

    public String getPhone(String name)
    {
      System.out.print("Enter phone number for " + name + ": ");
      String phone = console.nextLine();
      return phone;
    }
}
```

The program designs have been modified by adding another method. Below are updated design models and a look at the four program design criteria.

6.2.2.1 Design Models

The getPhone method has been added to each solution. Figure. 6.4a, b shows this change in the class diagrams.

The Nassi–Schneiderman chart and statechart diagrams would need to be changed to show the entry of a phone number by the user. The updates to these models are not shown.

Version B solutions are now evaluated using the four program design criteria.

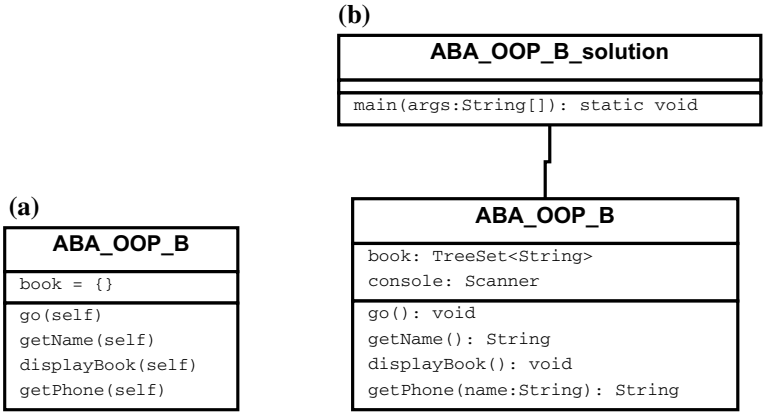


Fig. 6.4 **a** Python Version B class diagram. **b** Java Version B class diagram

6.2.2.2 Separation of Concerns

The additional requirement results in the getPhone method being added to the solution. This new method is called by the go method. This updated program design adheres to the principle of separation of concerns.

6.2.2.3 Design for Reuse

The getPhone method is conceptually similar to the getName method. This is a good example of the more challenging aspect of design for reuse—identifying code segments (or complete methods) that are similar but not the same. To help identify and generalize two methods into a single method, you can write a description of the logic as a series of algorithmic steps. In this case, both methods: display a prompt to the user; obtain data entered by the user; and return this data to the calling function. Given the similar processing performed by the two methods, there is likely a better program design that would eliminate the need to have both of these methods.

6.2.2.4 Design Only What is Needed

The additional method is necessary to fully implement the additional requirement.

6.2.2.5 Time Performance

For the Python solution, there is no change in performance for Version B when compared to Version A since both use a dictionary. The Java program uses a TreeMap instead of a TreeSet. Since maps are typically implemented the same way that sets are (i.e., using a red-black binary search tree algorithm), there is no change in time performance for Version B when compared to Version A.

6.2.2.6 Memory Performance

There is no change in memory performance for these Version B solutions when compared to the Version A solutions.

6.2.3 OOP Version C

We'll add one more requirement to our Address Book Application, identified in italics below, as the last ABA version presented in this chapter. The ABA shall

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number *and a single email address.*
- Use a simple text-based user interface to obtain the contact data.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

The program designs in Listings 6.5 and 6.6 address the problem noted above in the design for reuse criteria for Version B. Instead of having getName and getPhone methods (and a getEmail method for the new requirement), we now have one method getTextLine. The getTextLine method has one parameter variable so the correct prompt value can be passed to it for display to the user.

Listing 6.5 ABA_OOP_C.py

```
#ABA_OOP_C.py: very simple object-oriented example.
# Entry and non-persistent storage of name,
# no duplicate names, a phone number and email(better).

def addressBook():
  aba = ABA_OOP_C()
  aba.go()

class ABA_OOP_C:
  def __init__(self):
    self.book = {}

  def go(self):
   name = self.getTextLine(
       "Enter contact name ('exit' to quit): ")
   while name != "exit":
    phone = self.getTextLine(
        "Enter phone number for " + name + ": ")
    email = self.getTextLine(
        "Enter email address for " + name + ": ")
    if name not in self.book:
      self.book[name] = (phone, email)
    name = self.getTextLine(
        "Enter contact name ('exit' to quit): ")

   self.displayBook()

  def getTextLine(self, prompt):
    return input(prompt)

  def displayBook(self):
    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    sortedNames = sorted(self.book.keys())
    for name in sortedNames:
      print(name, self.book[name])
```

Listing 6.6 shows the Java solution.

Listing 6.6 ABA_OOP_C.java

```
//ABA_OOP_C_solution.java: very simple object-oriented example.
// Entry and non-persistent storage of name,
// no duplicate names, a phone number and email (better).
```

```
import java.util.Collection;
import java.util.Iterator;
import java.util.NavigableSet;
import java.util.Scanner;
import java.util.TreeMap;

public class ABA_OOP_C_solution
{
  public static void main(String[] args)
  {
    ABA_OOP_C aba = new ABA_OOP_C();
    aba.go();
  }
}

class ABA_OOP_contactData
{
    String phone;
    String email;

  public ABA_OOP_contactData(String phone, String email)
  {
     this.phone = phone;
     this.email = email;
  }

  public String toString()
  {
    return "(" + phone + ", " + email + ")";
  }
}

class ABA_OOP_C
{
  TreeMap<String,ABA_OOP_contactData> book;
  Scanner console;

  public ABA_OOP_C()
  {
     book = new TreeMap<String,ABA_OOP_contactData>();
     console = new Scanner(System.in);
  }

  public void go()
  {
     String name, phone, email;
     name = getTextLine("Enter contact name ('exit' to quit): ");

     while (! name.equals("exit"))
     {
       phone = getTextLine("Enter phone number for " +
             name + ": ");
       email = getTextLine("Enter email address for " +
              name + ": ");
       if (! book.containsKey(name))
         book.put(name, new ABA_OOP_contactData(phone, email));
       name = getTextLine("Enter contact name ('exit' to quit): ");
     }
```

```
      displayBook();
    }

    //pre: prompt contains a message (typically instructions)
    //      to be displayed to user.
    //post: returns value entered by user as a String.
    public String getTextLine(String prompt)
    {
      System.out.print(prompt);
      String textLine = console.nextLine();
      return textLine;
    }

    public void displayBook()
    {
       System.out.println();
       System.out.println("TEST: Display contents of address book");
       System.out.println("TEST: Address book contains the " +
            "following contacts");

       NavigableSet<String> keySet = book.navigableKeySet();
       Collection<ABA_OOP_contactData> valueColl = book.values();

       Iterator<String> iterKey = keySet.iterator();
       Iterator<ABA_OOP_contactData> iterValue =
            valueColl.iterator();

       while (iterKey.hasNext() && iterValue.hasNext())
         System.out.println(iterKey.next() + " " + iterValue.next());
    }
}
```

6.2.3.1 Design Models

The class diagrams in Fig. 6.5a, b show the structural changes to the Version C solution. The Python solution uses a tuple to store a phone number and email address for a contact name. In contrast, the Java solution now has a third class used to store a phone number and email address for a contact name. Both solutions now use constructor methods to initialize the instance variables. Python constructor methods are always named __init__ while Java constructor methods have the same name as the class and do not have a method return type.

The Nassi–Schneiderman chart and statechart diagrams would need to be changed to show entry of an email address by the user. The updates to these models are not shown.

Version C solutions are now evaluated using the four program design criteria.

6.2.3.2 Separation of Concerns

The Python solution uses a tuple to store multiple values (e.g., phone number and email address) for a contact name. This results in no impact on the separation of concerns criteria.

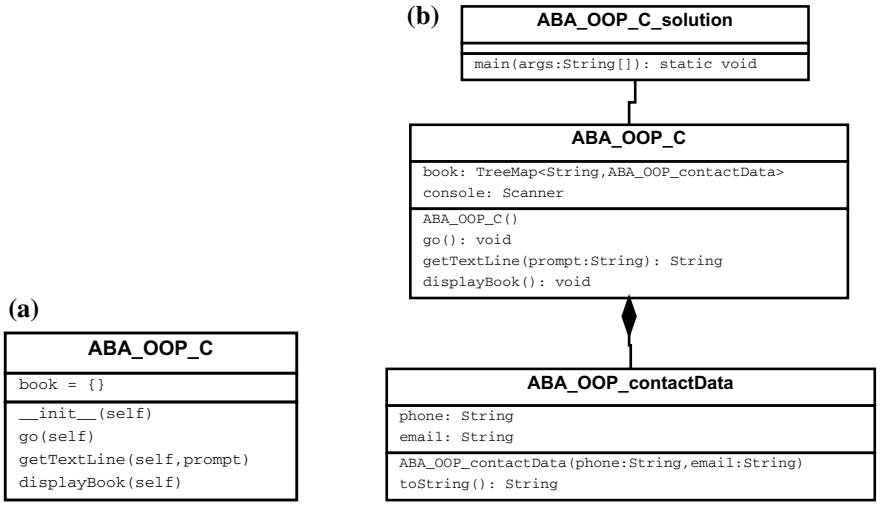


Fig. 6.5 **a** Python Version C class diagram. **b** Java Version C class diagram

The Java solution now has a third class named ABA_OOP_contactData used to store a phone number and email address for each contact name in the TreeMap. This third class is necessary since the TreeMap requires a data type for the key value and the associated data value. This updated program design adheres to the principle of separation of concerns.

6.2.3.3 Design for Reuse

The getTextLine method can now be used whenever there is a need to display a prompt to the user, obtain data entered by the user, and return this data to the calling method as a string value. We have generalized the getName and getPhone methods into a single method, resulting in the getTextLine method being more useful to us. This is evident by the fact that the additional requirement of obtaining an email address was implemented without needing to define a new method.

6.2.3.4 Design Only What is Needed

The Python and Java program designs are performing only those processing steps that are required.

6.2.3.5 Time Performance

There is no change in performance for Version C when compared to Version B since the Python and Java solutions continue to use the same type of data structure.

6.2.3.6 Memory Performance

There is no change in memory performance for these Version C solutions when compared to the Version B solutions.

6.3 OOP Post-conditions

The following should have been learned after completing this chapter.

- Evaluating a program design should consider at least five criteria: separation of concerns, design for reuse, design only what is needed, time performance, and memory performance.
- When adding new methods to a solution, a software engineer should determine whether the new code is similar to an existing method. When similar code segments are found, a more general method should be developed that serves the needs of both code segments.

Exercises

Hands-on Exercises

1. Modify the Version A Nassi–Shneiderman diagram so that it accurately describes the behavior of Version C.
2. Modify the Version A statechart so that it accurately describes the behavior of Version C.
3. Use an existing code solution that you've developed and identify portions of the program design that could be changed to improve its time performance.
4. Use an existing code solution that you've developed and identify portions of the program design that could be changed to improve its memory performance.
5. Select different types of data structures supported by a programming language that you use and identify the types of operations that would result in poor performance. For example, for a given data structure what would the time performance be for accessing a single value via a search? For accessing the first, middle, or last value in the data structure?

Reference

1. Python Software Foundation: The python standard library, version 3.3.5. https://docs.python.org/3.3/library/functions.html. Accessed 4 Jun 2014



SP Case Study: Considering Performance

7

The objective of this chapter is to apply all four program design criteria—separation of concerns, design for reuse, design only what is needed, and performance—to the case study.

7.1 SP Preconditions

The following should be true prior to starting this chapter.

- Evaluating the performance of a program design should consider both time and memory usage.
- You understand three program design criteria: separation of concerns, design for reuse, and design only what is needed. You've evaluated program code using these criteria.
- You know that a hierarchy chart may be used to model the structure of your imperative (i.e., structured) program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

7.2 SP Simple Designs

We'll start with the ABA requirements as stated for Version C in Chap. 4. The ABA shall

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_7

- Allow for entry and (nonpersistent) storage of people's names.
- Use a simple text-based user interface to obtain the names.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

7.2.1 SP Version A

The Version A solutions shown in Listings 7.1 and 7.2 are a result of changing the data structure being used to store contact names. These changes were made to improve the responsiveness of the ABA based on the time performance improvements described in Chap. 5.

Listing 7.1 ABA_SP_A.py

```
#ABA_SP_A.py: very simple structured example.
# Entry and non-persistent storage of name,
# no duplicate names (using dictionary).

def addressBook():
  book = {}
  name = getName()
  while name != "exit":
    if name not in book:
      book[name] = None
    name = getName()

  displayBook(book)

def getName():
  return input("Enter contact name ('exit' to quit): ")

def displayBook(book):
  print()
  print("TEST: Display contents of address book")
  print("TEST: Address book contains the following contacts")
  sortedNames = sorted(book.keys())
  for name in sortedNames:
    print(name)
```

The *if name not in book* conditional statement in Listing 7.1 results in Python using the built-in hash function __hash__() for the string (str) data type [1]. In a worst-case scenario, the hash function would generate the same hash table location for each contact name. The result would be a sequential search based on the use of the quadratic open addressing collision resolution algorithm. As noted in Chap. 5, this worst case would result in linear $O(n)$ time performance, which matches the performance of the solutions in Chap. 4.

The C++ code for Version A is shown in Listing 7.2. This C++ program design is significantly different because it is using a set instead of a list. As described in Chap. 5, this results in logarithmic time $O(\log_2 n)$ performance when determining if

the contact name already exists in the ABA. Specifically, the *notFound* programmer-defined function in Version A uses the find method of the set class to determine if the name is found in the book.

Listing 7.2 ABA_SP_A.cpp

```
//ABA_SP_A.cpp: very simple structured example.
// Entry and non–persistent storage of name,
// no duplicate names (using set).

#include <iostream>
#include <string>
#include <set>

using namespace std;

string getName();
void displayBook(set<string> book);
bool notFound(set<string> book, string name);

int main()
{
  set<string> book;
  string name;
  name = getName();
  while (name != "exit")
  {
     if (notFound(book, name))
       book.insert(name);
    name = getName();
  }

  displayBook(book);

  return 0;
}

string getName()
{
  string name;
  cout << "Enter contact name ('exit' to quit): ";
  getline(cin, name);
  return name;
}

void displayBook(set<string> book)
{
  cout << endl;
  cout << "Test: Display contents of address book\n";
  cout << "Test: Address book contains the following contacts\n";
  for (set<string>::iterator iter = book.begin();
       iter != book.end(); iter++)
```

```
    cout << *iter << endl;
}

//pre: book contains zero or more string values.
//post: returns false when name is in book.
//    otherwise, returns true (i.e., didNotFind is true).
bool notFound(set<string> book, string name)
{
  bool didNotFind = true;
  set<string>::iterator iter = book.find(name);
  if (iter != book.end())
    didNotFind = false;
  return didNotFind;
}
```

7.2.1.1 Design Models

The hierarchy diagrams for these two solutions are in Fig. 7.1a, b. Both of these hierarchy diagrams are the same as the Version C solution found in Sect. 4.3.3.

Both the Python and C++ solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 7.2 describes the algorithm used in either solution. The statechart diagram in Fig. 7.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and the software application.

The Python and C++ solutions are now evaluated using the program design criteria described in Chaps. 2 and 5.

7.2.1.2 Program Design Criteria (from Chap. 2)

In terms of separation of concerns, design for reuse, and design only what is needed, changing the type of data structure used to store the address book data has no impact on these program design criteria. This solution exhibits the same program design characteristics as the Version C solution in Chap. 4.

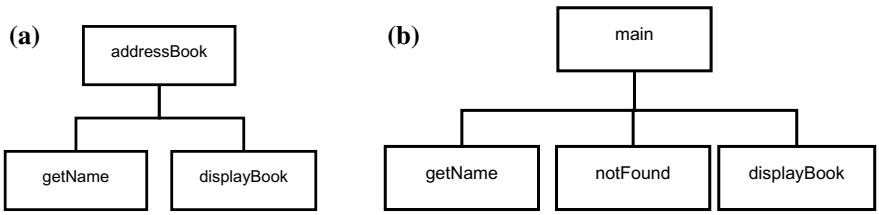


Fig. 7.1 **a** Python version A hierarchy chart. **b** C++ version A hierarchy chart

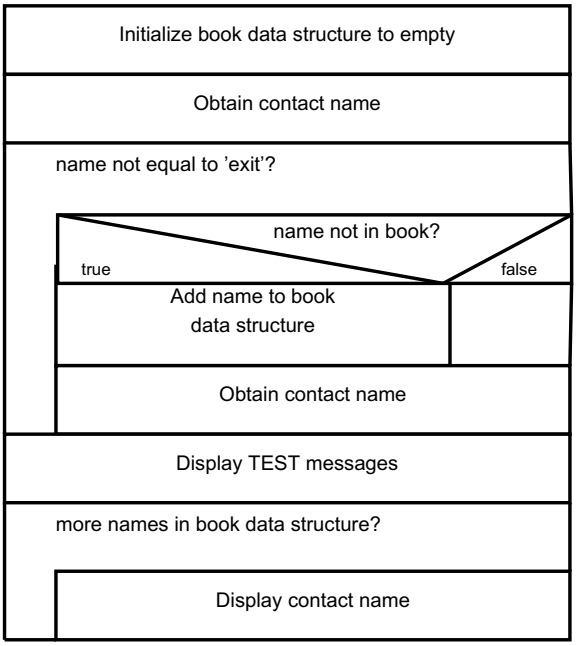


Fig. 7.2 Version A Nassi–Shneiderman diagram

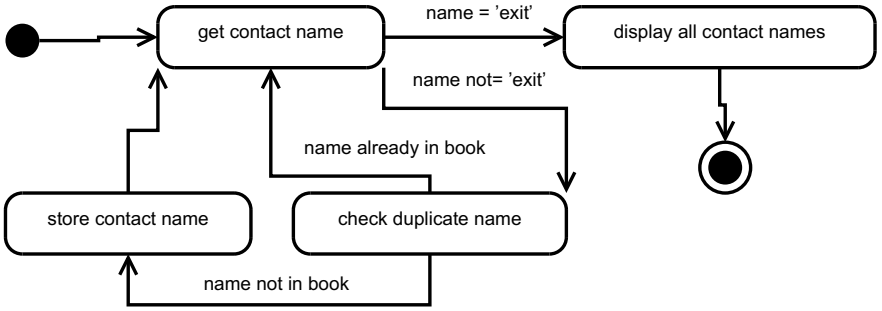


Fig. 7.3 Version A statechart

7.2.1.3 Time Performance

Changing the type of data structure used to store the address book data will have a positive impact on the responsiveness of the ABA. To summarize, the time performance of checking to determine if the contact name is a duplicate was studied. Using a Python dictionary instead of a Python list produced a best case of constant time instead of linear time. Using a C++ set instead of a C++ list produced a best case of logarithmic time instead of linear time.

7.2.1.4 Memory Performance

As mentioned in Chap. 5, use of a Python dictionary means that a hash table is being used. Given the nonpersistent storage of contacts at this point in the ABA, it is quite likely that the hash table will contain a small number of entries that will not result in allocating a lot of memory that does not get used.

For the C++ solution, the set data structure will contain a node only for those contacts that are currently in the ABA. Thus, the C++ solution will only allocate memory in the set when it is needed to store contact data. However, the function calls to both displayBook and notFound pass the book data structure. This will use the C++ default passing mechanism, which means that each call will result in a copy of the data being placed on the runtime stack. This will consume unnecessary memory, and is especially bad for the notFound function, which is called each time the user enters a contact name.

7.2.2 SP Version B

Our address book is not very useful since it only stores contact names. Let's add one more requirement, identified in italics below. The ABA shall

- Allow for entry and (nonpersistent) storage of people's names.
- *Store for each person a single phone number.*
- Use a simple text-based user interface to obtain the contact data.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

The Python solution is shown in Listing 7.3. This solution uses a dictionary, and stores a phone number value instead of *None* as the mapped data value. Specifically, the statement `book[name] = None` in Version A has been replaced with the statement `book[name] = phone`. The displayBook function has also been modified to display the address book names in alphabetical order. This was done so the Python and C++ programs for Version B exhibit the same behavior.

Listing 7.3 ABA_SP_B.py

```
#ABA_SP_B.py: very simple structured example.
# Entry and on-persistent storage of name,
# no duplicate names, a phone number.

def addressBook():
  book = {}
  name = getName()
  while name != "exit":
    phone = getPhone(name)
    if name not in book:
      book[name] = phone
    name = getName()
```

```
  displayBook(book)

def getName():
  return input("Enter contact name ('exit' to quit): ")

def displayBook(book):
  print()
  print("TEST: Display contents of address book")
  print("TEST: Address book contains the following contacts")
  sortedNames = sorted(book.keys())
  for name in sortedNames:
     print(name, book[name])

def getPhone(name):
  phone = input("Enter phone number for " + name + ": ")
  return phone
```

The C++ solution in Listing 7.4 uses a map instead of a set, since a map stores a key–value pair whereas a set only stores keys. Specifically, the data type *map<string, string>* indicates the map will store a key that is a string (i.e., contact name) along with a value related to the key (i.e., phone number) which is also a string.

The other significant change to the C++ Version B solution is the use of the *pass by reference* passing mechanism. The functions displayBook and notFound need to be given the book data structure so they can perform their respective processing. Looking at the C++ code in Listing 7.4, we see an ampersand ("&") has been added to the displayBook and notFound function declarations and definitions. Each ampersand appears immediately before the parameter variable book. The ampersand tells the C++ compiler to pass the book map object by reference. The C++ solution will now mimic the runtime stack usage of the Version A Python solution; it will simply pass the memory location of where the book map object is located instead of copying the entire book map onto the runtime stack.

Listing 7.4 ABA_SP_B.cpp

```
//ABA_SP_B.cpp: very simple structured example.
// Entry and non-persistent storage of name,
// no duplicate names, a phone number.

#include <iostream>
#include <string>
#include <map>

using namespace std;

string getName();
void displayBook(map<string,string> &book);
bool notFound(map<string,string> &book, string name);
string getPhone(string name);

int main()
```

```
{
   map<string,string> book;
   string name, phone;
   name = getName();
   while (name != "exit")
   {
      phone = getPhone(name);
      if (notFound(book, name))
        book[name] = phone;
      name = getName();
    }

    displayBook(book);

    return 0;
}

string getName()
{
  string name;
  cout << "Enter contact name ('exit' to quit): ";
  getline(cin, name);
  return name;
}

void displayBook(map<string,string> &book)
{
  cout << endl;
  cout << "Test: Display contents of address book\n";
  cout << "Test: Address book contains the following contacts\n";
  for (map<string,string>::iterator iter = book.begin();
       iter != book.end(); iter++)
     cout << iter->first << " " << iter->second << endl;
}

//pre: book contains zero or more string values.
//post: returns false when name is in book.
//    otherwise, returns true (i.e., didNotFind is true).
bool notFound(map<string,string> &book, string name)
{
   bool didNotFind = true;
   map<string,string>::iterator iter = book.find(name);
   if (iter != book.end())
     didNotFind = false;
   return didNotFind;
}

string getPhone(string name)
{
  string phone;
  cout << "Enter phone number for " << name << ": ";
```

```
    getline(cin, phone);
    return phone;
}
```

The program designs have been modified by adding another function. Below are updated design models and a look at the four program design criteria.

7.2.2.1 Design Models

The getPhone function has been added to each solution. Figure 7.4a, b shows this change in the hierarchy charts.

The Nassi–Schneiderman chart and statechart diagrams would need to be changed to show the entry of a phone number by the user. The updates to these models are not shown.

Version B solutions are now evaluated using the four program design criteria.

7.2.2.2 Separation of Concerns

The additional requirement results in the getPhone function being added to the solution. This new function is called by the go function. This updated program design adheres to the principle of separation of concerns.

7.2.2.3 Design for Reuse

The goPhone function is conceptually similar to the getName function. This is a good example of the more challenging aspect of design for reuse—identifying code segments (or complete functions) that are similar but not the same. To help identify and generalize two functions into a single function, you can write a description of the logic as a series of algorithmic steps. In this case, both functions: display a prompt to the user; obtain data entered by the user; and return this data to the calling function. Given the similar processing performed by the two functions, there is likely a better program design that would eliminate the need to have both of these functions.

7.2.2.4 Design Only What is Needed

The additional function is necessary to fully implement the additional requirement.

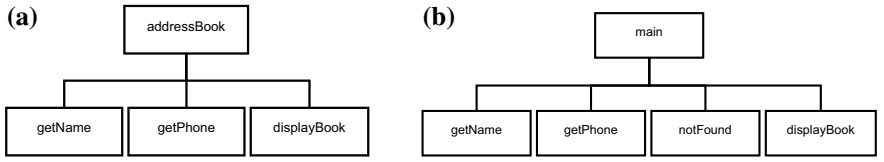


Fig. 7.4 **a** Python Version B hierarchy chart. **b** C++ Version B hierarchy chart

7.2.2.5 Time Performance

For the Python solution, there is no change in performance for Version B when compared to Version A since both use a dictionary. The C++ program uses a map instead of a set. Since maps are typically implemented the same way that sets are (i.e., using a red-black binary search tree algorithm), there is no change in time performance for Version B when compared to Version A.

7.2.2.6 Memory Performance

For the C++ solution, the use of the *pass by reference* notation for the getPhone and notFound functions means the book data structure is not copied onto the runtime stack each time one of these functions is called. This improves memory usage by limiting the growth of the runtime stack.

7.2.3 SP Version C

We'll add one more requirement to our Address Book Application, identified in italics below, as the last ABA version presented in this chapter. The ABA shall

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number *and a single email address.*
- Use a simple text-based user interface to obtain the contact data.
- Prevent a duplicate name from being stored in the address book.
- Not retrieve any information from the address book.

The program designs in Listings 7.5 and 7.6 address the problem noted above in the design for reuse criteria for Version B. Instead of having getName and getPhone functions (and a getEmail function for the new requirement), we now have one function getTextLine. The getTextLine function has one parameter variable so the correct prompt value can be passed to it for display to the user.

Listing 7.5 ABA_SP_C.py

```
#ABA_SP_C.py: very simple structured design example.
# Entry and non-persistent storage of name,
# no duplicate names, a phone number and email (better).

def addressBook():
  book = {}
  name = getTextLine("Enter contact name ('exit' to quit): ")
  while name != "exit":
    phone = getTextLine("Enter phone number for " + name + ": ")
    email = getTextLine("Enter email address for " + name + ": ")
    if name not in book:
      book[name] = (phone, email)
      name = getTextLine("Enter contact name ('exit' to quit): ")

  displayBook(book)
```

```
#pre: prompt contains a message (typically instructions)
#      to be displayed to user.
#post: returns value entered by user as a string.
def getTextLine(prompt):
  return input(prompt)

def displayBook(book):
  print()
  print("TEST: Display contents of address book")
  print("TEST: Address book contains the following contacts")
  sortedNames = sorted(book.keys())
  for name in sortedNames:
     print(name, book[name])
```

Listing 7.6 shows the C++ solution.

Listing 7.6 ABA_SP_C.cpp

```
//ABA_SP_C.cpp: very simple structured design example.
// Entry and non-persistent storage of name,
// no duplicate names, a phone number and email (better).

#include <iostream>
#include <string>
#include <map>

using namespace std;

struct contactData
{
  string phone;
  string email;
};

string getTextLine(string prompt);
void displayBook(map<string,contactData> &book);
bool notFound(map<string,contactData> &book, string name);

int main()
{
  map<string,contactData> book;
  string name;
  contactData data;
  name = getTextLine("Enter contact name ('exit' to quit): ");
  while (name != "exit")
  {
    data.phone = getTextLine("Enter phone number for " +
        name + ": ");
    data.email = getTextLine("Enter email address for " +
        name + ": ");
    if (notFound(book, name))
      book[name] = data;
    name = getTextLine("Enter contact name ('exit' to quit): ");
  }
```

```
  displayBook(book);

  return 0;
}

//pre: prompt contains a message (typically instructions)
//     to be displayed to user.
//post: returns value entered by user as a string.
string getTextLine(string prompt)
{
  string textLine;
  cout << prompt;
  getline(cin, textLine);
  return textLine;
}

void displayBook(map<string,contactData> &book)
{
  cout << endl;
  cout << "Test: Display contents of address book\n";
  cout << "Test: Address book contains the following contacts\n";
  for (map<string,contactData>::iterator iter = book.begin();
       iter != book.end(); iter++)
    cout << iter->first << " (" << iter->second.phone << ", "
      << iter->second.email << ")" << endl;
}

//pre: book contains zero or more string values.
//post: returns false when name is in book.
//    otherwise, returns true (i.e., didNotFind is true).
bool notFound(map<string,contactData> &book, string name)
{
  bool didNotFind = true;
  map<string,contactData>::iterator iter = book.find(name);
  if (iter != book.end())
    didNotFind = false;
  return didNotFind;
}
```

7.2.3.1 Design Models

The hierarchy diagrams in Fig. 7.5a, b shows the structural changes to the Version C solution. The Python solution uses a tuple to store a phone number and email address for a contact name. In contrast, the C++ solution uses a struct to store a phone number and email address for a contact name.

The Nassi–Schneiderman chart and statechart diagrams would need to be changed to show entry of an email address by the user. The updates to these models are not shown.

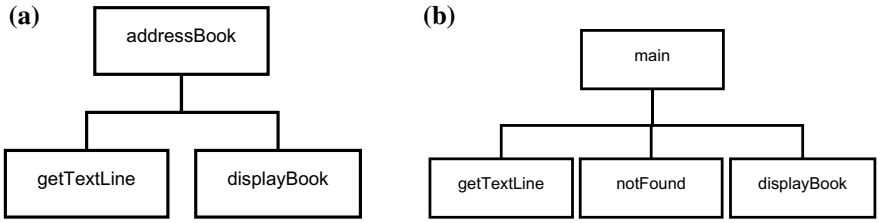


Fig. 7.5 **a** Python Version C hierarchy chart. **b** C++ Version C hierarchy chart

Version C solutions are now evaluated using the four program design criteria.

7.2.3.2 Separation of Concerns

These solutions are using a Python tuple or a C++ struct to store multiple values (e.g., phone number and email address) for a contact name. This results in no impact on the separation of concerns criteria.

7.2.3.3 Design for Reuse

The getTextLine function can now be used whenever there is a need to display a prompt to the user, obtain data entered by the user, and return this data to the calling function as a string value. We have generalized the getName and getPhone functions into a single function, resulting in the getTextLine function being more useful to us. This is evident by the fact that the additional requirement of obtaining an email address was implemented without needing to define a new function.

7.2.3.4 Design Only What is Needed

The Python and C++ program designs are performing only those processing steps that are required.

7.2.3.5 Time Performance

There is no change in performance for Version C when compared to Version B since the Python and C++ solutions continue to use the same type of data structure.

7.2.3.6 Memory Performance

There is no change in memory performance for these Version C solutions when compared to the Version B solutions.

7.3 SP Post-conditions

The following should have been learned after completing this chapter.

- Evaluating a program design should consider at least five criteria: separation of concerns, design for reuse, design only what is needed, time performance, and memory performance.
- When adding new functions to a solution, a software engineer should determine whether the new code is similar to an existing function. When similar code segments are found, a more general function should be developed that serves the needs of both code segments.

Exercises

Hands-on Exercises

1. Modify the Version A Nassi–Shneiderman diagram so that it accurately describes the behavior of Version C.
2. Modify the Version A statechart so that it accurately describes the behavior of Version C.
3. Use an existing code solution that you've developed and identify portions of the program design that could be changed to improve its time performance.
4. Use an existing code solution that you've developed and identify portions of the program design that could be changed to improve its memory performance.
5. Select different types of data structures supported by a programming language that you use and identify the types of operations that would result in poor performance. For example, for a given data structure what would the time performance be for accessing a single value via a search? For accessing the first, middle, or last value in the data structure?

Reference

1. Python Software Foundation: The python standard library, version 3.3.5. https://docs.python.org/3.3/library/functions.html. Accessed 4 Jun 2014



Program Design and Security

8

The objective of this chapter is to introduce security as another program design criteria.

8.1 Preconditions

The following should be true prior to starting this chapter.

- You understand four program design criteria: separation of concerns, design for reuse, design only what is needed, and performance.
- You have evaluated program code using these four criteria.

8.2 Concepts and Context

As is evident in data breaches that are frequently disclosed, developing more secure software is an issue that requires serious attention. This chapter defines security and information security, and explores information security from a programmer's perspective.

8.2.1 Security

Security can be defined as "the condition of not being threatened, especially physically, psychologically, emotionally, or financially" [1]. While security has tradi-

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_8

tionally been thought of in terms of protecting physical assets, the protection of information assets has become an important consideration when developing software.

The need to secure both information and information systems is not a new dilemma. The Morris worm was written by a computer science student in November 1988 [2]. While the author of this worm claims that its purpose was to determine the size of the Internet, the Morris worm became an early example of a *denial-of-service attack*. This worm resulted in a denial of service since it was designed to run multiple times on the same server. Eventually, a server had so many processes running the Morris worm that it was unable to run services related to it its intended purpose. This worm was an early indicator of just how important software security would become as the Internet and the World Wide Web became the communications backbone for personal and business uses.

The latest computer science curriculum guidelines include a knowledge unit called *information assurance and security* (IAS). This knowledge unit is defined as a "set of controls and processes, both technical and policy, intended to protect and defend information and information systems" [3].

The National Institute of Standards and Technology in the U.S.A. provides the following definitions [4].

Information assurance "Measures that protect and defend information and information systems by ensuring their availability, integrity, authentication, confidentiality, and non-repudiation. These measures include providing for restoration of information systems by incorporating protection, detection, and reaction capabilities."

Information security "The protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability."

Information assurance is an umbrella term that includes information security. Information assurance includes understanding the risks associated with information and information systems, privacy, regulatory compliance, standards compliance, and auditing. Professionals that practice information assurance tend to have multidisciplinary knowledge that may include accounting, fraud examination, forensics, management, systems engineering, security engineering, criminology, and computer science. The NIST Cybersecurity Framework [5] illustrates the breadth of information assurance by describing standards, guidelines, and best practices an organization should adhere to in an effort to minimize cybersecurity-related risks.

Information security is the part of information assurance that focuses on the creation and application of security controls. Information security topics tend to be technical in nature and fall largely within the domain of computer science.

8.2.2 Information Security

There are many security concepts/topics that fall within the information security domain. The following descriptions [3,4,6] represent a subset of these concepts/-topics that are directly related to developing a program design and writing code to reduce the number of security vulnerabilities. Note that, at this point in the book, we are focusing on coding practices that will reduce the number of security vulnerabilities. We will discuss software design principles related to developing more secure software in Chap. 24.

Data input validation Data input into a software application, regardless of its source, shall be validated. Examples of input data sources include a user entering data, another software system sending data to our application, and reading data from a persistent data store.

For instance, server software should never assume that data coming from a client has already been validated. This is because a malicious attacker could inject specially formatted data into a client that lacks input validation or could build/use their own client that sends requests to the server [7]. An example of this type of attack was discovered by TripWire that affects three of the top-selling smart home hubs sold on Amazon [8]. As stated in this article, the attacks can be done through malicious web pages or a smartphone application. This implies that the software on the hub is not validating requests it receives from a client.

Data output validation Data output by a software application, regardless of its destination, shall be validated. Examples of output data destinations include displaying data to a user, sending data to another application, and storing data in a persistent data store.

Writing to a log file, data file, or a database can place a "logic bomb" or "data bomb" into a persistent data store that gets activated at some later point based on a specific state being reached [7].

Exception handling Design code to correctly handle software-based exceptions that occur during execution.

Arithmetic errors, buffer overflows, and security exceptions are three types of exceptions that, when not caught and dealt with properly, may result in a situation that a malicious attacker can take advantage of. From a programming perspective, it is easy to detect these types of exceptions. Once the exception is detected (i.e., trapped or caught) it is fairly easy to write code that reacts in a secure way to the error. And yet these types of vulnerabilities continue to exist in software. One of the most popular sets of exploits take advantage of buffer overflows, where the data being stored in memory is larger than the amount of memory allocated for storage of the data. In fact, the book written by Hoglund and McGraw [7] on exploiting software has an entire chapter on buffer overflow.

Fail-safe defaults When a software failure occurs, design the default behavior of the software to fail in a way that denies access to the data.

This coding practice is tied to exception handling and other forms of detecting errors. Any error that is discovered should result in a program state that denies access to data or denies further processing that may lead to accessing data.

Type-safe languages A programming language is type-safe when it raises an exception when an operation is attempted on a value whose data type is not appropriate. The exception may be raised in a static context (i.e., while compiling the code) or in a dynamic context (i.e., while executing the code).

The first four program design security criteria listed above will be further explored in Chaps. 9 and 10 on object-oriented programming (OOP) and structured programming (SP), respectively. Type-safe languages are discussed next.

8.2.2.1 Type-Safe Languages

Of the three programming languages used in this book, Java and Python are type-safe languages. Any attempt at applying an operation on a value not of the appropriate data type will raise an exception. In contrast, C++ is not a type-safe language.

8.2.2.2 Java: A Type-Safe Language

In the case of Java, the compiler will often catch the misuse of a type by reporting a syntax error. (This is called static type checking.) The following examples illustrate the misuse of a type in Java caught by the compiler.

When the Java class in Listing 8.1 is compiled, the compiler generates the syntax error shown in Listing 8.2. Since an *int* in Java is a primitive data type, a variable of type *int* cannot be used to invoke a method.

Listing 8.1 BadMethodCall.java

```
public class BadMethodCall
{
    public BadMethodCall()
    {
        int a = 5;
        a.callMethod();
    }
}
```

Listing 8.2 BadMethodCall.java Syntax Error

```
BadMethodCall.java:6: error: int cannot be dereferenced
        a.callMethod();
```

When the Java class in Listing 8.3 is compiled, the compiler displays the syntax error shown in Listing 8.4. Since the variable *a* has not been defined prior to the *a.callMethod()* statement, the compiler does not know what variable *a* represents.

Listing 8.3 BadVariableName.java

```
public class BadVariableName
{
    public BadVariableName()
    {
        a.callMethod();
    }
}
```

Listing 8.4 BadVariableName.java Syntax Error

```
BadVariableName.java:5: error: cannot find symbol
        a.callMethod();
        ^
    symbol: variable a
```

When the Java class in Listing 8.5 is compiled, the compiler displays the syntax error shown in Listing 8.6. The plus symbol in Java does not have a definition allowing the first operand to be an *int* and the second operand to be a *String*.

Listing 8.5 BadUseOfOperator.java

```
public class BadUseOfOperator
{
    public BadUseOfOperator()
    {
        int a = 5;
        int b = a + "five";
    }
}
```

Listing 8.6 BadUseOfOperator.java Syntax Error

```
BadUseOfOperator.java:6: error: incompatible types: String cannot
be converted to int
     int b = a + "five";
                 ^
  required: int
  found: String
```

There are also cases where Java will catch the misuse of a type while the program is being executed. This is called dynamic type checking. The code in Listing 8.7 illustrates the misuse of a type in Java detected at runtime. When this program is executed, it causes the runtime exception shown in Listing 8.8. This runtime exception occurs on the *sum = sum + (double)numbers.get(idx)* statement during the second iteration of the *for* loop. The Float object added to the *numbers* ArrayList (see *//11* in listing) cannot be cast into a Double object.

Listing 8.7 BadCast.java

```
import java.math.BigDecimal;
import java.util.ArrayList;

class BadCast
{
    public static void main(String[] args)
    {
        ArrayList<Object> numbers;
        numbers = new ArrayList<Object>();
        numbers.add(new Double(3.14159265358979323846264338327795));
        numbers.add(new Float(3.14159265358979323846264338327795)); //11
        numbers.add(new BigDecimal(3.14159265358979323846264338327795));
        double sum = 0;
        for (int idx=0; idx < numbers.size(); idx++)
            sum = sum + (Double)numbers.get(idx); //15
    }
}
```

Listing 8.8 BadCast.java Runtime Error

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.Float cannot be cast to java.lang.Double
     at BadCast.main(BadCast.java:15)
```

8.2.2.3 Python: A Type-Safe Language

In the case of Python, the interpreter will catch the misuse of a type while the program is being executed (i.e., Python performs dynamic type checking). The following examples illustrate misuse of a type in Python.

When the Python interpreter runs the code in Listing 8.9, it displays the exception shown in Listing 8.10. This runtime exception occurs since the Python type *int*, which is implemented as a class, does not have a method whose name is *callMethod*.

Listing 8.9 AttributeError.py

```
def func():
    a = 5
    a.callMethod()

func()
```

Listing 8.10 AttributeError.py Runtime Error

```
Traceback (most recent call last):
  File "AttributeError.py", line 5, in <module>
    func()
  File "AttributeError.py", line 3, in func
    a.callMethod()
AttributeError: 'int' object has no attribute 'callMethod'
```

When the Python interpreter runs the code in Listing 8.11, it displays the exception shown in Listing 8.12. This runtime exception occurs since the variable *a* has not been defined prior to the *a.callMethod()* statement. That is, the interpreter does not know what variable *a* represents.

Listing 8.11 Ch5NameError.py

```
def func():
    a.callMethod()

func()
```

Listing 8.12 NameError.py Runtime Exception

```
Traceback (most recent call last):
  File "NameError.py", line 4, in <module>
    func()
  File "NameError.py", line 2, in func
    a.callMethod()
NameError: global name 'a' is not defined
```

When the Python interpreter runs the code in Listing 8.13, it displays the exception shown in Listing 8.14. This runtime exception occurs since the plus symbol in Python does not include a definition where the first operand is an *int* and the second operand is a *string*.

Listing 8.13 TypeError.py

```
def func():
    a = 5
    b = a + "five"

func()
```

Listing 8.14 TypeError.py Runtime Exception

```
Traceback (most recent call last):
  File "TypeError.py", line 5, in <module>
    func()
  File "TypeError.py", line 3, in func
    b = a + "five"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

8.2.2.4 C++: Not a Type-Safe Language

As mentioned, C++ is not a type-safe language. Listing 8.15 shows one example of why C++ is not a type-safe language. This code declares, initializes, and displays the contents of three variables. First, the *value* variable is declared as an *int* that is initialized to -1. Second, the variable *p_int* is declared as a pointer to an *int* that is initialized to the memory location of the *value* variable. Finally, the variable *p_float*

is declared as a pointer to a *float* that is also initialized to the memory location of the *value* variable. This C++ NotSafe program produces the output shown in Listing 8.16.[1]

Listing 8.15 NotSafe.cpp

```
#include <iostream>

using namespace std;

void displayValues(int* ptr_int, float* ptr_float);

int main()
{
    int value = -1;
    int* p_int = &value;
    float* p_float = (float*)p_int;

    displayValues(p_int, p_float);
    displayValues(p_int+1, p_float+1);
    displayValues(p_int+10, p_float+10);

    return 0;
}

void displayValues(int* ptr_int, float* ptr_float)
{
    cout << ptr_int << "\t" << *ptr_int << "\t"
         << ptr_float << "\t" << *ptr_float << endl;
}
```

Listing 8.16 C++ NotSafe Output

```
0x61ff2c   -1   0x61ff2c   -1.#QNAN
0x61ff30   6422300   0x61ff30   8.99956e-039
0x61ff54   6422368   0x61ff54   8.99965e-039
```

The following explains the output produced from running the code in Listing 8.15.

- The first and third values displayed on each output line show the same values. This confirms the pointer arithmetic being performed on memory address values pointing to an *int* and a *float* are consistently resulting in the same memory location. That is, *int* and *float* values are both stored in 32 bits (i.e., 4 bytes).
- The first call to *displayValues* displays the first line of the output. The *0x61ff2c* value is the memory address where the *value* variable is stored on the

[1]The output shown for the C++ NotSafe program results from compiling and linking this C++ example using GNU C++ (GCC) version 4.8.1 and then executing on Windows 10 Home Edition version 1607 OS Build 14939.1198. The behavior of this C++ program may differ when using a different C++ compiler or operating system.

runtime stack.[2] The *−1.#QNAN* value comes from the **ptr_float* expression in the *cout* statement. This dereferences the pointer to display a floating-point number. Since this memory location contains an integer −1, all 32 bits of the 4-byte integer value are set to a 1. Furthermore, the IEEE floating-point standard defines all 1 bits for a float as representing not-a-number (NaN) value. Specifically, a *quiet NaN* is displayed to indicate the bit-pattern represents an invalid floating-point value.

- The second call to *displayValues* displays the second line of the output. The *0x61ff30* value is the result of adding 1 to the *p_int* and *p_float* pointer values. Since *p_int* is a pointer to an int and *p_float* is a pointer to a float, adding 1 results in adding the size of an int or float (both typically 4 bytes) to the *p_int* and *p_float* memory address value. The *6422300* value (which is *0x61FF1C*) is the integer value of the 4-bytes stored at the *0x61ff30* memory address. This value is stored on the runtime stack and represents the memory location for the *p_int* variable, which contains the address where the *value* variable is found on the stack.
- The last call to *displayValues* displays the third line of the output. The *0x61ff54* value is the result of adding 10 to the *p_int* and *p_float* pointer values. (As mentioned in the previous item, this results in adding 40 to the *p_int* and *p_float* memory address values). The *6422368* value (which is *0x61FF60*) is the integer value of the 4-bytes stored at the *0x61ff54* memory address. The *8.99965e-039* is the floating-point value of the 4-bytes stored at this same memory address.

This C++ program illustrates two language features that make C++ not type-safe. First, the number of bytes reserved for an *int* value is dependent on the compiler implementation. While the GNU C++ compiler treats *int* as a 4-byte value, other C++ compilers may treat this as a 2-byte value. Second, doing pointer arithmetic is a dangerous way to navigate around memory. For example, a programmer can use pointer arithmetic to alter the contents of the runtime stack or the machine code of the executable image currently in the main memory. Depending on the C++ compiler and operating system, it's possible that pointer arithmetic could result in gaining access to memory locations that are outside the memory range being used by your program (i.e., process).

Making a minor change to the code in Listing 8.15 may result in different behavior. For example, changing *displayValues(p_int+10, p_float+10);* to *displayValues(p_int+100, p_float+100);* will produce the error message displayed in Fig. 8.1. In this example, Windows prevented access to a memory location that was outside the process space.

[2] The runtime stack is used to store information related to each function/method call. Each parameter value passed via the call and any local variables defined within the function/method are stored on the runtime stack. In addition, the return address of where the function/method call should return to is also stored on the runtime stack.

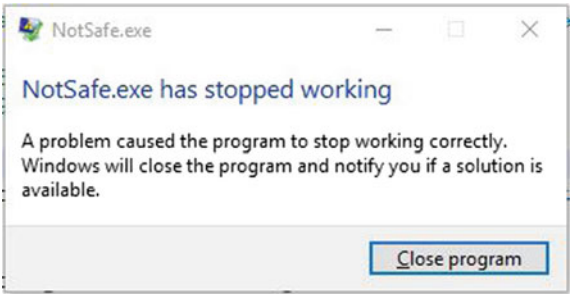


Fig. 8.1 C++ NotSafe windows message

8.3 Post-conditions

The following should have been learned when completing this chapter.

- Developing a safe (more secure) software application involves consideration of a number of factors, including
  - Validate data input into an application.
  - Validate data output by an application.
  - Include an exception handler in code to react in a safe (more secure) way when an exception may occur. In Python, use try-except blocks to react to runtime exceptions. In Java, use try-catch blocks to react to runtime exceptions.
  - Use fail-safe defaults when an error condition or exception is detected, to ensure data is not used in an inappropriate (less secure) manner.
- Making your code more secure often involves adding some complexity to your solution. Testing your code needs to include test cases to determine whether your code responds correctly to both proper and improper use of your application.
- When you have a choice, a type-safe programming language should be used.

8.4 Next Chapter?

If you are interested in seeing how security may be applied to a small object-oriented programming solution, continue with Chap. 9. This chapter will continue the Address Book Application case study using Python and Java.

If you are interested in seeing how security may be applied to a small structured programming solution, continue with Chap. 10. This chapter will continue the Address Book Application case study using Python. Since C++ is not a type-safe pro-

gramming language, C++ is no longer used to demonstrate structured programming and structured design concepts.

Exercises

Discussion Questions

1. Why is it important to use a type-safe language to improve the security of code?
2. Validating input and output data is a critical part of making more secure software. Adding validation of data to a design would appear to be relatively straight forward.

   a. List the steps that should be performed to validate a date in the format YYYY-MM-DD.
   b. Which of these steps must be changed if you needed to validate a date in the format DD MMM YYYY, where *MMM* is the first three letters of the month name?
   c. Search the World Wide Web to identify two types of attacks that would not be possible if data validation is designed and implemented.

References

1. Wiktionary.org: Security (2019) Wiktionary the free dictionary. Wikimedia Foundation. https://en.wiktionary.org/wiki/security. Accessed 10 Feb 2019
2. Wikipedia.org: Morris worm (2015) Wikepedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Morris_worm. Accessed 29 July 2015
3. The joint task force on computing curricula: computer science curricula (2013) Curriculum guidelines for undergraduate degree programs in computer science. ACM and IEEE
4. National institute of standards and technology: computer security resource center glossary (2019) NIST. https://csrc.nist.gov/glossary. Accessed 10 Feb 2019
5. National institute of standards and technology: cybersecurity framework (2019) NIST. https://www.nist.gov/cyberframework. Accessed 09 March 2019
6. Alicherry M, Keromytis AD, Stavrou A (2009) Deny-by-default distributed security policy enforcement in mobile ad hoc networks. http://www.cs.columbia.edu/~angelos/Papers/2009/manet-securecomm.pdf. Accessed 24 June 2014
7. Hoglund G, McGraw G (2004) Exploiting software: how to break code. Addison Wesley, Boston
8. Tripwire: Tripwire uncovers smart home hub zero-day vulnerabilities. Tripwire Press Release (2015). https://www.tripwire.com/company/press-releases/2015/08/tripwire-uncovers-significant-security-flaws-in-popular-smart-home-automation-hub/. Accessed 10 August 2015



OOP Case Study: Considering Security

9

The objective of this chapter is to apply the additional program design criteria—security—to the case study.

9.1 OOP Preconditions

The following should be true prior to starting this chapter.

- You understand four program design criteria: separation of concerns, design for reuse, design only what is needed, and performance. You have evaluated program code using these four criteria.
- You know that a class diagram is used in object-oriented solutions to illustrate the structure of your program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

9.2 OOP Simple Object-Oriented Programming Designs

The Address Book Application will be modified to improve the security of the ABA. The information security programming concepts described in Sect. 8.2.2 will be the focus of these improvements.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_9

9.2.1 OOP Version A

We add two more requirements (identified in italics below) to specify the types of validation required for the input data. The ABA shall

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number and a single email address.
- Use a simple text-based user interface to obtain the contact data.
- *Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no limit on the size of the name entered.*
- *Ensure that each phone number contains only the digits 0 through 9. At least one digit must be entered for a phone number. There is no limit on the size of the phone number entered.*
- Prevent a duplicate name from being stored in the address book.
- Do not retrieve any information from the address book.

9.2.1.1 Python Solution

The Python code in Listing 9.1 shows how the new requirement to validate a person's name is implemented. Included in this listing are two methods: go(self) and getValidName(self). The addressBook() function and other methods of the ABA_OOP_A class are not shown. The *ABA_OOP_A.py* source code file contains the complete solution.

Listing 9.1 ABA_OOP_A.py

```
import re   #regular expression library

class ABA_OOP_A:
  def go(self):
    name = self.getValidName()
    while name != "exit":
      phone = self.getValidPhone(name)
      email = self.getTextLine("Enter email address for " +
            name + ": ")
      if name not in self.book:
        self.book[name] = (phone, email)
      name = self.getValidName()

      self.displayBook()

  #pre: Need to obtain a contact name from the user.
  #post: A valid contact name is returned.
  def getValidName(self):
    #A valid contact name may contain only spaces, uppercase
    #  letters, and lowercase letters.
    pattern = "^[ A-Za-z]+$"
    result = None
```

```
      #Continue asking for a contact name until
      #  valid data is entered.
      while result == None:
        name = self.getTextLine( \
             "Enter contact name ('exit' to quit): ")
        #Remove leading and trailing whitespace.
         name = name.strip()
         if len(name) == 0:
           #contact name must contain at least one letter.
            result = None
            errorMsg = "A contact name must contain at least \
one uppercase or lowercase letter."
         else:
           #Determine if the entered name is valid.
            result = re.match(pattern, name)
            errorMsg = "A contact name must contain only \
uppercase and lowercase letters and spaces."
      if result == None:
        print(errorMsg)

    return name
```

As seen in Listing 9.1, the go method calls getTextLine to obtain the email address, which does not have a validation requirement. To obtain the name and phone number, the go function calls getValidName and getValidPhone, respectively, to ensure that a valid name and phone number is entered by the user.

Implementing the two input validation requirements is done using the regular expression (*import re*) Python module. Looking at the getValidName method, the pattern variable is initialized to "^[A-Za-z]+$". The caret symbol "^" indicates the start of the string value while the dollar sign symbol "$" indicates the end of the string value. The symbols enclosed within square brackets "[" and "]" indicate a *set of characters* that includes space, any uppercase letter, and any lowercase letter. Specifying a set of characters in a pattern will match a single character value. The plus sign "+" after the set notation in the pattern indicates that one or more characters in the set must exist in order for the regular expression to match. The pattern and name entered by the user are passed to the *re.match* method. When this method returns None, the name value does not conform to the pattern specification. Note the use of the strip method to remove leading and trailing whitespace characters from the name entered by the user. This is done prior to checking the pattern specification. This ensures that a name of all spaces is not erroneously deemed a valid value.

The getValidPhone method (not shown in Listing 9.1) performs similar processing with two exceptions. It uses a different pattern value based on the validation requirement and the logic does not need to use the strip method since a space character is not valid in a phone number (i.e., see the validation requirement).

9.2.1.2 Java Solution

The Java code in Listing 9.2 shows how the new requirement to validate a person's name is implemented. Included in this listing are two methods: go() and getValidName(). Other methods of the ABA_OOP_A class are not shown. In addition, the ABA_OOP_A_solution and ABA_OOP_contactData classes are not shown in this code listing. The *ABA_OOP_A.java* source code file contains the complete solution.

Listing 9.2 ABA_OOP_A_solution.java

```
class ABA_OOP_A
{
   public void go()
   {
      String name, phone, email;
      name = getValidName();

      while (! name.equals("exit"))
      {
        phone = getValidPhone(name);
        email = getTextLine("Enter email address for " +
              name + ": ");
        if (! book.containsKey(name))
          book.put(name, new ABA_OOP_contactData(phone, email));
        name = getValidName();
       }

      displayBook();
   }

   //pre: Need to obtain a contact name from the user.
   //post: A valid contact name is returned.
   public String getValidName()
   {
     //A valid contact name may contain only spaces,
     //  uppercase letters, and lowercase letters.
     String pattern = "[ A–Za–z]+";
     String name = "";
     String errorMsg = "";
     boolean okay = false;
     //Continue asking for a contact name until valid data
     //  is entered.
     do
     {
        name = getTextLine(
            "Enter contact name ('exit' to quit): ");
        //Remove leading and trailing whitespace.
        name = name.trim();
        if (name.length() == 0)
        {
          //contact name must contain at least one letter.
          okay = false;
          errorMsg = "A contact name must contain at least" +
```

```
            " one uppercase or lowercase letter.";
        }
        else
        {
            //Determine if the entered name is valid.
            okay = name.matches(pattern);
            errorMsg = "A contact name must contain only" +
               " uppercase and lowercase letters and spaces.";
        }
        if (! okay)
          System.out.println(errorMsg);
      } while (! okay);

      return name;
    }
}
```

As seen in Listing 9.2, the go method calls getTextLine to obtain the email address, which does not have a validation requirement. To obtain the name and phone number, the go method calls getValidName and getValidPhone, respectively, to ensure that a valid name and phone number is entered by the user.

Implementing the two input validation requirements is done using regular expressions. Looking at the getValidName method, the pattern variable is initialized to "[A-Za-z]+" which specifies a *set of characters* that includes space, any uppercase letter, and any lowercase letter. Specifying a set of characters in a pattern will match a single character value. The plus sign ("+") after the set notation in the pattern indicates that one or more characters in the set must exist in order for the regular expression to match. The matches method, which is part of the String class, is called to validate the name given a pattern that it must match. When this method returns false, the name value does not conform to the pattern specification. Note the use of the trim method to remove leading and trailing whitespace characters from the name entered by the user. This is done prior to checking the pattern specification. This ensures that a name of all spaces is not erroneously deemed a valid value.

The getValidPhone method (not shown in Listing 9.2) performs similar processing with two exceptions. It uses a different pattern value based on the validation requirement and the logic does not need to use the trim method since a space character is not valid within a phone number (i.e., see the validation requirement).

9.2.1.3 Object-Oriented Programming Design Models

The class diagrams in Fig. 9.1a, b shows the structure of the Version A solutions. Both solutions continue to use constructor methods to initialize the instance variables. The Python solution is using a tuple to store the phone number and email address for each contact person. The Java solution has a third class named ABA_OOP_contactData used to store the phone number and email address for each contact person. Given this third class, the Java class diagram uses a composition relationship (the filled-in dia-

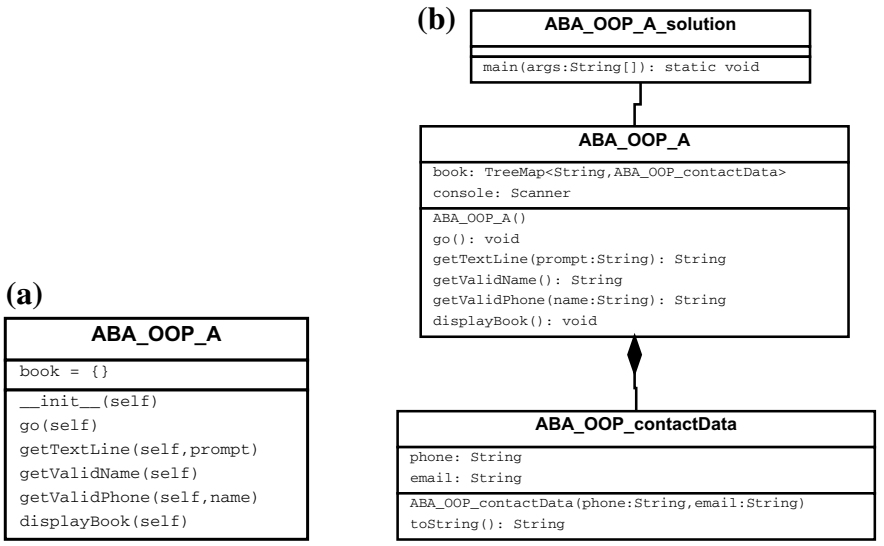


Fig. 9.1 **a** Python Version A class diagram and **b** Java Version A class diagram

mond shape) to indicate the ABA_OOP_A class contains ABA_OOP_contactData instances, where these object instances only exist inside the ABA_OOP_A object. Object-oriented class relationships will be discussed in more detail in Chap. 12.

Both the Python and Java solutions exhibit the same behavior. The Nassi–Shneiderman diagram in Fig. 9.2 describes the algorithm used in either solution while the statechart in Fig. 9.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and software application. Both of these diagrams show that the algorithmic solution now includes the validation of two input values entered by the user.

9.2.2 OOP Version B: A More Secure ABA

The version A solution validates data input by the user, an important security feature to reduce vulnerabilities through an input channel. It may be just as important to validate data output by software. This is especially true when saving data to a persistent data store or sending data to another application. Software applications that retrieve data from a persistent data store often assume that this data is valid. Similarly, receiving data from another application via a network connection is often done with the assumption that the other application will give us only valid data. To create more secure software, it is best to validate any data input into or output by an application.

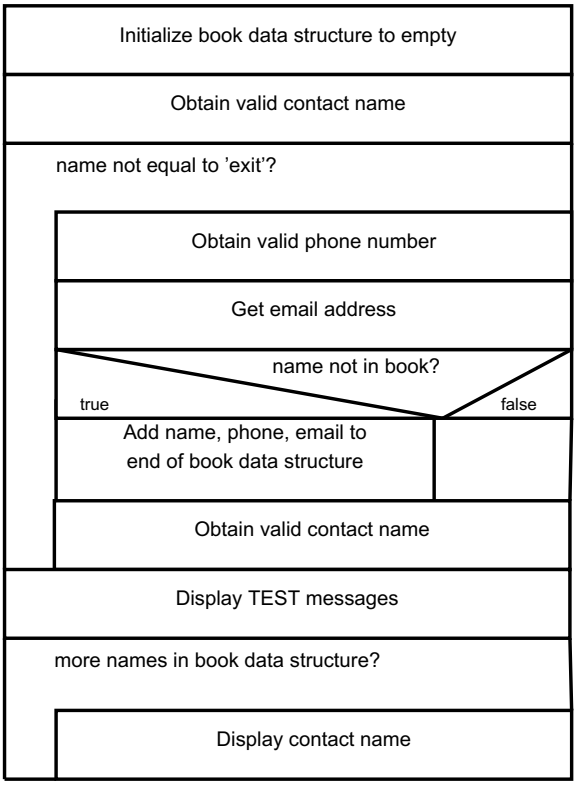


Fig. 9.2 Version A Nassi–Shneiderman diagram

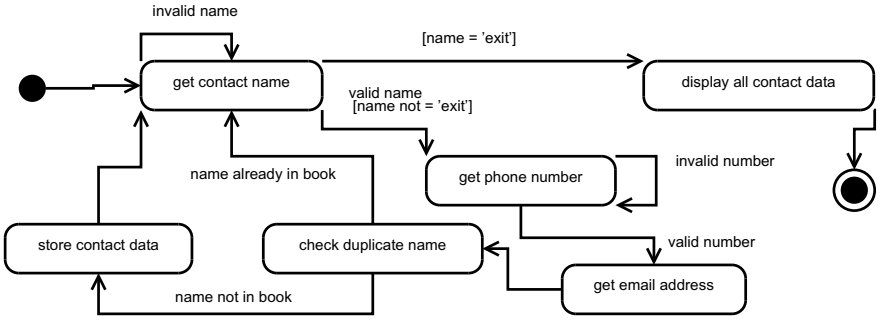


Fig. 9.3 Version A statechart

Many programming languages will raise exceptions during runtime to indicate an error condition has occurred. Handling these runtime exceptions will produce a more secure software application.

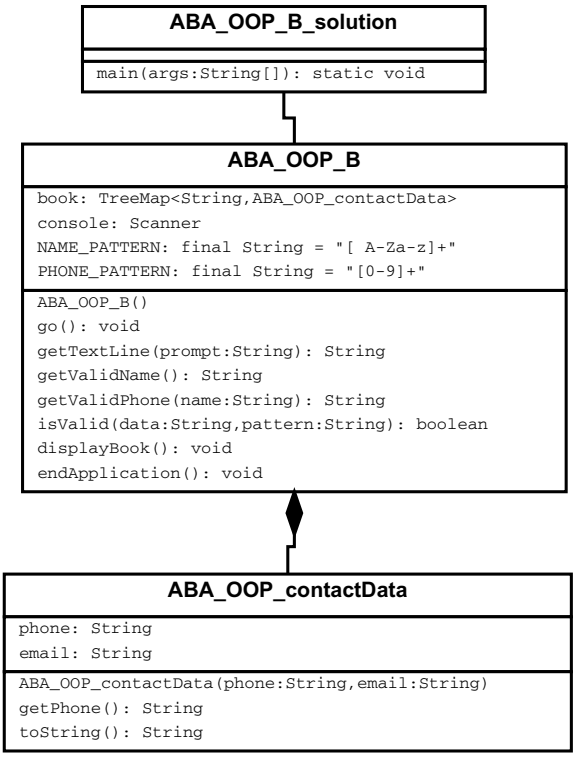


Fig. 9.4 Java Version B class diagram

Finally, recognizing that a software failure (or exception) has occurred is important to do, but this is incomplete unless the design also ensures that the failure does not lead to inappropriate access to the data.

A Java solution (Version B) is described below that includes code that implements input validation, output validation, exception handling, and fail-safe defaults.

9.2.2.1 Object-Oriented Programming Design Models

The class diagram in Fig. 9.4 shows the structure of the Version B solution. The class diagram uses the same two relationship types shown in the Version A Java solution.

The two behavior diagrams—Nassi–Shneiderman and statechart—are not shown for Version B. Instead, sample code listings are included below to explain how each security feature has been added to the solution.

9.2.2.2 Data Input Validation

No changes were made to the validation of data input by the user.

9.2.2.3 Data Output Validation

The displayBook method shown in Listing 9.3 now includes validation of the name and phone values prior to this data being displayed to the user. When either of these values is not valid, neither data value is displayed to the user even though one of the values may be valid! The NAME_PATTERN and PHONE_PATTERN identifiers are two constants defined within the ABA_OOP_B class. See the ABA_OOP_B.java source code file for details on these constants and other portions of the version B solution not described in this section.

Listing 9.3 ABA_OOP_B displayBook method

```
public void displayBook()
{
  System.out.println();
  System.out.println("TEST: Display contents of address book");
  System.out.println("TEST: Address book contains" +
    " the following contacts");

  NavigableSet<String> keySet = book.navigableKeySet();
  Collection<ABA_OOP_contactData> valueColl = book.values();

  Iterator<String> iterKey = keySet.iterator();
  Iterator<ABA_OOP_contactData> iterValue = valueColl.iterator();

  String name, phone, data;
  while (iterKey.hasNext() && iterValue.hasNext())
  {
     try
     {
       name = iterKey.next();
       ABA_OOP_contactData contactData = iterValue.next();
       data = contactData.toString();
       phone = contactData.getPhone();
       if (! isValid(name, NAME_PATTERN) ||
         ! isValid(phone, PHONE_PATTERN))
       {
          name = "[InvalidName]";
          data = "[InvalidContactData]";
       }
     }
     catch (Exception ex)
     {
       name = "[InvalidName]";
       data = "[InvalidContactData]";
     }
     System.out.println(name + " " + data);
  }
}
```

9.2.2.4 Exception Handling

The displayBook method shown in Listing 9.3 includes a try-catch block that will capture exceptions. The expressions *iterKey.next()* and *iterValue.next()* will cause a NoSuchElementExcetion when the iterator object has no more elements. Under normal operation, it would be considered a logic error if this exception was to occur. However, it may be that the memory that contains the book TreeMap has been corrupted in some way, perhaps maliciously. Additionally, any exception that is raised as a result of executing the other statements inside the try block will be caught by the catch block of code.

The getTextLine method shown in Listing 9.4 now has a try-catch block to capture exceptions caused by the Scanner method nextLine. Specifically, holding the Ctrl key down while striking the C key will cause a NoSuchElementException. When this or any other exception is thrown by the nextLine method, an error message is displayed and the program is terminated.

Listing 9.4 ABA_OOP_B getTextLine function

```
//pre: prompt contains a message (typically instructions) to be
//   displayed to user.
//post: returns value entered by user as a String.
public String getTextLine(String prompt)
{
  System.out.print(prompt);
  String textLine;
  try
  {
    textLine = console.nextLine();
  }
  catch (Exception ex)
  {
    textLine = "";
    //Entering Ctrl–Z causes a NoSuchElementException
    System.out.println("\nConsole input has been terminated," +
        " ending application.\n");
    endApplication();
  }
  return textLine;
}
```

The getValidName method shown in Listing 9.5 no longer includes the logic to validate the contact name. Instead, the isValid method (also shown in Listing 9.5) has been added for the purpose of validating a string data value using a regular expression pattern. The try-catch block in the isValid method will detect the use of an invalid regular expression.

Listing 9.5 ABA_OOP_B getValidName and isValid methods

```
//pre: Need to obtain a contact name from the user.
//post: A valid contact name is returned.
public String getValidName()
{
```

```
  String name = "";
  boolean okay = false;
  //Continue asking for contact name until valid data is entered.
  do
  {
     name = getTextLine("Enter contact name ('exit' to quit): ");
     //Remove leading and trailing whitespace.
     name = name.trim();
     okay = isValid(name, NAME_PATTERN);
     if (! okay)
       System.out.println("A contact name must contain only" +
         " uppercase and lowercase letters and spaces.");
  }  while (! okay);

  return name;
}

//pre: User has entered a data value that can be validated
//   using a regular expression.
//post: Return true when name is valid. Otherwise, return false.
public boolean isValid(String data, String pattern)
{
   boolean okay = false;
   if (data.length() > 0)
   {
      try
      {
        //Determine if the entered name is valid.
        okay = data.matches(pattern);
      }
      catch (PatternSyntaxException ex)
      {
         System.out.println("\nLogic error in pattern matching," +
           " ending application.\n");
         endApplication();
      }
   }

   return okay;
}
```

Similarly, the getValidPhone function no longer includes the logic to validate the phone number. Instead, the isValid function is called for the purpose of validating a phone number.

9.2.2.5 Fail-Safe Defaults

The displayBook method shown in Listing 9.3 is an example of fail-safe defaults. When a software failure occurs, in this case when either the name or phone value

stored in the address book data structure is not valid, a generic "invalid data" value is displayed instead of the data currently stored in the book dictionary.

9.3 OOP Post-conditions

The following should have been learned when completing this chapter.

- Developing a safe (more secure) software application involves consideration of a number of factors, including
  - Validate data input into an application.
  - Validate data output by an application.
  - Include an exception handler in code to react in a safe (more secure) way when an exception may occur. In Python, use try-except blocks to react to runtime exceptions. In Java, use try-catch blocks to react to runtime exceptions.
  - Use fail-safe defaults when an error condition or exception is detected, to ensure data is not used in an inappropriate (less secure) manner.
- Making your code more secure often involves adding some complexity to your solution. Testing your code needs to include specific test cases to determine whether your code responds correctly to both proper and improper use of your application.

Exercises

Hands-on Exercises

1. The Java Version B solution has two functions, getValidName and getValidPhone, that appear to contain very similar code. Develop a solution that combines these two functions into one generic function that is able to validate a name or a phone value. Evaluate your solution using the program design criteria.
2. Use an existing code solution you have developed and identify portions of the program design that could be changed to improve its security. Implement these security changes in your solution.



SP Case Study: Considering Security

10

The objective of this chapter is to apply the additional program design criteria—security—to the case study.

10.1 SP Preconditions

The following should be true prior to starting this chapter.

- You understand four program design criteria: separation of concerns, design for reuse, design only what is needed, and performance. You have evaluated program code using these four criteria.
- You know that a hierarchy chart is used in structured solutions to illustrate the structure of your program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

10.2 SP Simple Structured Programming Designs

The Address Book Application will be modified to improve the security of the ABA. The information security programming concepts described in Sect. 8.2.2 will be the focus of these improvements.

10.2.1 SP Version A

We add two more requirements (identified in italics below) to specify the types of validation required for the input data. The ABA shall

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_10

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number and a single email address.
- Use a simple text-based user interface to obtain the contact data.
- *Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no limit on the size of the name entered.*
- *Ensure that each phone number contains only the digits 0 through 9. At least one digit must be entered for a phone number. There is no limit on the size of the phone number entered.*
- Prevent a duplicate name from being stored in the address book.
- Do not retrieve any information from the address book.

Since C++ is not a type-safe language, we'll only use Python to demonstrate structured programming concepts from this point forward.

10.2.1.1 Python Solution

The Python code in Listing 10.1 shows how the new requirement to validate a person's name is implemented. Included in this listing are two functions: addressBook() and getValidName(). The other functions for this solution are not shown. The *ABA_SP_A.py* source code file contains the complete solution.

Listing 10.1 ABA_SP_A.py

```
import re    #regular expression library

def addressBook():
    book = {}
    name = getValidName()
    while name != "exit":
        phone = getValidPhone(name)
        email = self.getTextLine("Enter email address for " + \
                name + ": ")
        if name not in book:
            book[name] = (phone, email)
        name = getValidName()

    displayBook(book)

#pre: Need to obtain a contact name from the user.
#post: A valid contact name is returned.
def getValidName():
    #A valid contact name may contain only spaces, uppercase
    #  letters, and lowercase letters.
    pattern = "^[ A-Za-z]+$"
    result = None
    #Continue asking for contact name until valid data is entered.
    while result == None:
        name = getTextLine("Enter contact name ('exit' to quit): ")
```

```
        #Remove leading and trailing whitespace.
        name = name.strip()
        if len(name) == 0:
            #contact name must contain at least one letter.
            result = None
            errorMsg = "A contact name must contain at least one \
uppercase or lowercase letter."
        else:
            #Determine if the entered name is valid.
            result = re.match(pattern, name)
            errorMsg = "A contact name must contain only uppercase \
and lowercase letters and spaces."
    if result == None:
      print(errorMsg)

  return name
```

As seen in Listing 10.1, the addressBook function calls getTextLine to obtain the email address, which does not have a validation requirement. To obtain the name and phone number, the addressBook function calls getValidName and getValidPhone, respectively, to ensure that a valid name and phone number is entered by the user.

Implementing the two input validation requirements is done using the regular expression (*import re*) Python module. Looking at the getValidName function, the pattern variable is initialized to "^[A-Za-z]+$". The caret symbol "^" indicates the start of the string value while the dollar sign symbol "$" indicates the end of the string value. The symbols enclosed within square brackets "[" and "]" indicate a *set of characters* that includes space, any uppercase letter, and any lowercase letter. Specifying a set of characters in a pattern will match a single character value. The plus sign "+" after the set notation in the pattern indicates that one or more characters in the set must exist in order for the regular expression to match. The pattern and name entered by the user are passed to the *re.match* method. When this method returns None the name value does not conform to the pattern specification. Note the use of the strip method to remove leading and trailing whitespace characters from the name entered by the user. This is done prior to checking the pattern specification. This ensures that a name of all spaces is not erroneously deemed a valid value.

The getValidPhone function (not shown in Listing 10.1) performs similar processing with two exceptions. It uses a different pattern value based on the validation requirement and the logic does not need to use the strip method since a space character is not valid in a phone number (i.e., see the validation requirement).

10.2.1.2 Structured Programming Design Models

The hierarchy diagram in Fig. 10.1 shows the structure of the Version A solution. The Python solution is using a tuple to store the phone number and email address for each contact person.

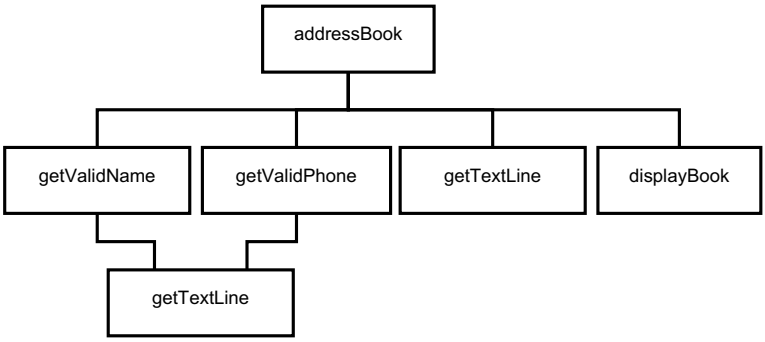
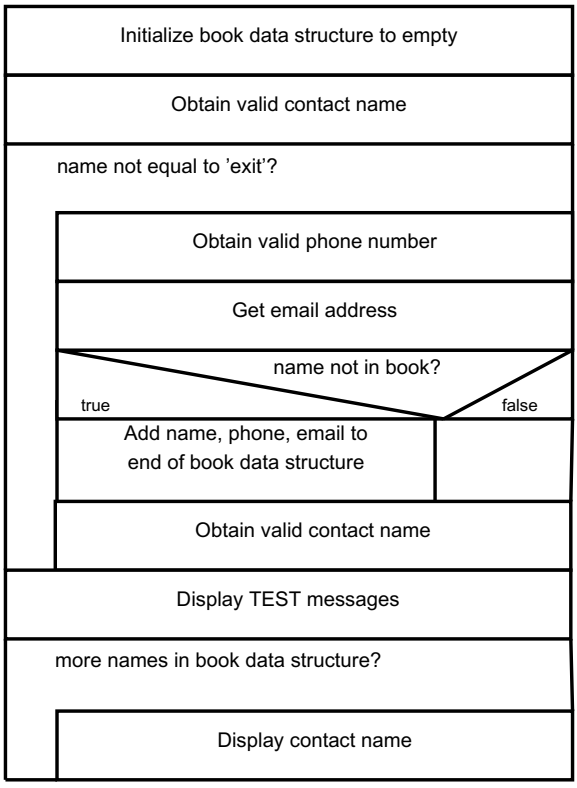


Fig. 10.1 Hierarchy chart—version A python

Fig. 10.2 Version A Nassi–Shneiderman Diagram



The Nassi–Shneiderman diagram in Fig. 10.2 describes the algorithm used in the Version A solution while the statechart in Fig. 10.3 shows the behavior of the human–computer interaction, where each state represents a distinct interaction between the user and software application. Both of these diagrams show that the algorithmic solution now includes validation of two input values entered by the user.

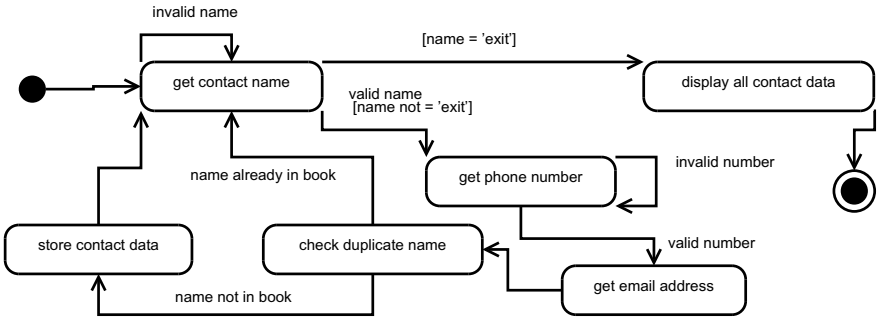


Fig. 10.3 Version A Statechart

10.2.2 SP Version B: A More Secure ABA

The Version A solution validates data input by the user, an important security feature to reduce vulnerabilities through an input channel. It may be just as important to validate data output by software. This is especially true when saving data to a persistent data store or sending data to another application. Software applications that retrieve data from a persistent data store often assume that this data is valid. Similarly, receiving data from another application via a network connection is often done with the assumption that the other application will give us only valid data. To create more secure software, it is best to validate any data input into or output by an application.

Many programming languages will raise exceptions during runtime to indicate an error condition has occurred. Handling these runtime exceptions will produce a more secure software application.

Finally, recognizing that a software failure (or exception) has occurred is important to do, but this is incomplete unless the design also ensures that the failure does not lead to inappropriate access to the data.

A Python solution (Version B) is described below that includes code that implements input validation, output validation, exception handling, and fail-safe defaults.

10.2.2.1 Structured Programming Design Models

The hierarchy diagram in Fig. 10.4 shows the structure of the Version B solution. This solution continues to use a tuple to store the phone number and email address for each contact person.

The two behavior diagrams—Nassi–Shneiderman and statechart—are not shown for Version B. Instead, sample code listings are included below to explain how each security feature has been added to the solution.

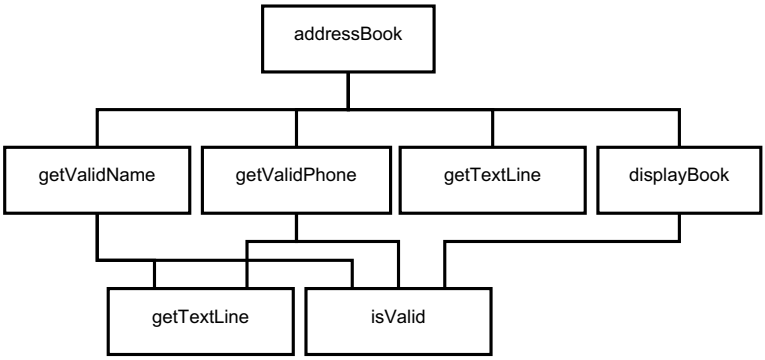


Fig. 10.4 Hierarchy chart—version B python

10.2.2.2 Data Input Validation

No changes were made to the validation of data input by the user.

10.2.2.3 Data Output Validation

The displayBook method shown in Listing 10.2 now includes validation of the name and phone values prior to this data being displayed to the user. When either of these values is not valid, neither data value is displayed to the user even though one of the values may be valid! The RE_PATTERNS.NAME and RE_PATTERNS.PHONE identifiers are two constants defined within the Python code, as shown at the top of Listing 10.2. Refer to the Python Library Reference section on the collections module for more information about the *namedtuple()* factory function.

Listing 10.2 ABA_SP_B displayBook function

```
RE_patterns = collections.namedtuple("RE_patterns", "NAME  PHONE")
RE_PATTERNS = RE_patterns("^[ A−Za−z]+$", "^[0−9]+$")

def displayBook(book):
    print()
    print("TEST: Display contents of address book")
    print("TEST: Address book contains the following contacts")
    sortedNames = sorted(book.keys())
    for name in sortedNames:
        try:
            contactData = book[name]
            validName = isValid(name, RE_PATTERNS.NAME)
            validPhone = isValid(contactData[0], RE_PATTERNS.PHONE)
        except KeyError:
            validName = False
            validPhone = False
        except:
            validName = False
            validPhone = False
```

```
    if not validName or not validPhone:
        name = "[InvalidName]"
        contactData = "[InvalidContactData]"
    print(name, contactData)
```

10.2.2.4 Exception Handling

The displayBook method shown in Listing 10.2 includes a try-except block that will capture exceptions. The statement *contactData = book[name]* will cause a KeyError exception when the name value is no longer found in the book dictionary. Under normal operation, it would be considered a logic error if this exception were to occur. However, it may be that the memory that contains the book dictionary has been corrupted in some way, perhaps maliciously. Additionally, any exception that is raised as a result of executing the other statements inside the try block will be caught by the *except:* block of code.

The getTextLine function shown in Listing 10.3 now has a try-except block to capture exceptions caused by the built-in Python function input. Specifically, holding the Ctrl key down while striking the C key will cause a KeyboardInterrupt exception within a Microsoft Windows environment. When this or any other exception is thrown by the input function, the value returned by the getTextLine function is an empty string.[1]

Listing 10.3 ABA_SP_B getTextLine function

```
#pre: prompt contains a message (typically instructions) to be
#     displayed to user.
#post: returns value entered by user as a string.
#      When an exception occurs (e.g., KeyboardInterrupt via Ctrl–C)
#        an empty string is returned.
def getTextLine(prompt):
    try:
        userData = input(prompt)
    except:
        #Any type of exception is handled by returning an empty
        #  string value.
        userData = ""
    return userData
```

The getValidName function shown in Listing 10.4 no longer includes the logic to validate the contact name. Instead, the isValid function (also shown in Listing 10.4) has been added for the purpose of validating a string data value using a regular expression pattern. While these two functions do not contain any try-except blocks, they are written in a manner that avoids the possibility of exceptions being thrown. For

[1] According to the Python 3.x Library Reference, the built-in input function will throw an EOFError exception when the end-of-file is read. This can be generated in an interactive session by striking Ctrl-Z in Microsoft Windows.

example, the conditional expression found in the if statement of the isValid function ensures that the data variable refers to a str object prior to using the len built-in function. Since the len built-in function will work as long as the object represents a sequence or collection, and since the str type represents a sequence, the code has avoided a TypeError from being thrown. Similarly, the regular expression match function expects it's second argument to be a str object. The same if logic avoids a TypeError for this situation as well.

Listing 10.4 ABA_SP_B getValidName and isValid functions

```
#pre: Need to obtain a contact name from the user.
#post: A valid contact name is returned.
def getValidName():
    #Continue asking for contact name until valid data is entered.
    valid = False
    while not valid:
        name = getTextLine("Enter contact name ('exit' to quit): ")
        name = name.strip()
        valid = isValid(name, RE_PATTERNS.NAME)
        if not valid:
            print("A contact name must contain only uppercase and \
lowercase letters and spaces.")

    return name

#pre: User has entered a data value that needs to be validated.
#post: Return true when data is valid. Otherwise, return false.
def isValid(data, pattern):
    #Assume data is not valid
    valid = False
    if type(data) == str and len(data) != 0 and \
       re.match(pattern, data) != None:
        valid = True

    return valid
```

Similarly, the getValidPhone function no longer includes the logic to validate the phone number. Instead, the isValid function is called for the purpose of validating a phone number.

10.2.2.5 Fail-Safe Defaults

The displayBook function shown in Listing 10.2 is an example of fail-safe defaults. When a software failure occurs, in this case either a KeyError exception or the data somehow has become invalid, a generic "invalid data" value is displayed instead of the data currently stored in the book dictionary.

10.3 SP Post-conditions

The following should have been learned when completing this chapter.

- Developing a safe (more secure) software application involves consideration of a number of factors, including
  - Validate data input into an application.
  - Validate data output by an application.
  - Include an exception handler in code to react in a safe (more secure) way when an exception may occur. In Python, use try-except blocks to react to runtime exceptions. In Java, use try-catch blocks to react to runtime exceptions.
  - Use fail-safe defaults when an error condition or exception is detected, to ensure data is not used in an inappropriate (less secure) manner.
- Making your code more secure often involves adding some complexity to your solution. Testing your code needs to include specific test cases to determine whether your code responds correctly to both proper and improper use of your application.

Exercises

Hands-on Exercises

1. The Python Version B solution has two functions, getValidName and getValidPhone, that appear to contain very similar code. Develop a solution that combines these two functions into one generic function that is able to validate a name or a phone value. Evaluate your solution using the program design criteria.
2. Use an existing code solution you have developed and identify portions of the program design that could be changed to improve its security. Implement these security changes in your solution.

Part II
Introduction to Software Design

This second part introduces the characteristics of a good software design and the Model–View–Controller (MVC) architectural pattern. Chapters 11 through 16 represent the core software design topics that are further explored in part III. The author covers chapters 11 through 16 in weeks 3 through 5 of a 15-week semester for a 3 credit-hour course on software design.



Characteristics of Good Software Design

11

The objective of this chapter is to introduce the characteristics that may be used to determine if a software design is good.

11.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You have evaluated program code using these five criteria.

11.2 Concepts and Context

This chapter will focus on six characteristics of good software design and the concept of abstraction.

11.2.1 What are Characteristics of a Good Software Design?

This section describes six characteristics of good software design: simplicity, coupling, cohesion, information hiding, performance, and security. Table 11.1 describes these characteristics from two perspectives—everyday usage [1] and software design usage.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_11

Table 11.1 Definitions

Term	Everyday usage	Software design usage
Simplicity	The quality or state of being unmixed or uncompounded	The amount and type of software elements needed to solve a problem
Coupling	Act of joining together to form a couple A device that couples two things together	The degree to which each program module relies on other modules [2] Identifies the connectedness of a module to other modules [3] Measure of interdependence between two or more modules [4]
Cohesion	State of cohering or of working together	The degree to which a software module is strongly related and focused in its responsibilities [5] A module that has a small, focused set of responsibilities and single-mindedness [3] An attribute of a software unit that refers to the relatedness of its components [6]
Information hiding	Not applicable	A component encapsulates its behaviors and data, hiding the implementation details from other components [7] Modules should be designed so that information (i.e., algorithms and data) contained within one module is inaccessible to other modules that have no need for such information [3]
Performance	Carrying into execution or action The amount of useful work accomplished estimated in terms of work needed	The analysis of an algorithm to determine its performance in terms of time (i.e., speed) and space (i.e., memory usage). This analysis is useful when comparing two algorithmic approaches that solve the same problem
Security	The condition of not being threatened, especially physically, psychologically, emotionally, or financially	A set of technical controls intended to protect and defend information and information systems [8]

11.2.1.1 Simplicity

The key phrase in the software design usage definition is "amount and type of software elements needed." The term *software elements* refers to processing elements and data structures found in the solution.

When we think about software elements, it's natural to think like a programmer. From this perspective, processing elements include sequence, selection, and iteration statements, as well as the structural elements of functions (for SP) or methods and classes (for OOP). Data structures include hash tables, trees, graphs, and arrays. Our challenge is to also think like a software designer. From this perspective, processing elements include components and subcomponents (for SD) or packages and classes (for OOD), while data structures include persistent data stores.

Our intuition suggests that a simpler design solution is less expensive to implement, test, and maintain. The acronym KIS for *Keep It Simple*, or its more derogatory version KISS for *Keep It Simple Stupid*, is a common phrase that even has a Wikipedia entry. To summarize this software design criteria, as long as your design solves the problem (i.e., satisfies the requirements), a simpler design is likely a better design.

11.2.1.2 Coupling

The key to understanding coupling is the word *module* found in the software design usage description. Unfortunately, the word module has been used to describe a wide range of physical objects (e.g., the Apollo Lunar Module, modular home, power supply module) and software constructs (e.g., component, package, class, function). In this textbook, a software module shall be any of the following.

For an object-oriented software design:

- A module is typically a class (i.e., a group of logically related attributes and methods). For example, a class that contains methods that validate data entered by a user. For very large software systems, a module may be a package of classes.
- A less commonly used interpretation is that a module is a single method found in a class definition.

For a structured software design:

- A module is typically a group of logically related functions. For example, a set of functions that perform validation of data entered by a user could be considered a distinct module within a structured design. This type of module is also known as a component (or in a very large software system a subcomponent). For example, a Python source code file is called a module. Thus, each Python module should contain logically related functions.
- A less commonly used interpretation is that a module is a single function found in a source code file.

The coupling definitions indicate that coupling describes the amount of connectedness between two or more modules. Thus, a design that has only one module cannot exhibit coupling. That is, coupling can only be described between distinct modules.

Our intuition suggests having fewer connections between modules mean there is less to test, maintain, and fix. Having fewer connections between modules is called *low coupling*, *loose coupling*, or *weak coupling*. A good design is one that exhibits low coupling between its modules. In contrast, more connections between modules means there is more to test, maintain, and fix. Having more connections between modules is called *high coupling*, *tight coupling*, or *strong coupling*. A bad design is one that exhibits high coupling between its modules [4].

Craig Larman introduces *General Responsibility Assignment Software Patterns*, abbreviated GRASP [9]. GRASP represents a learning approach to developing object-oriented designs, where software patterns are used to explain the rationale for a design. (Software patterns are discussed more fully in Sect. 11.2.2.2.) One of the nine software patterns included in GRASP is named *Low Coupling*. Table 11.2 summarizes the Low Coupling pattern documented in [9].

Table 11.2 Summary of GRASP low coupling

Solution	Assign a responsibility so that coupling remains low
Problem	How to support low dependency, low change impact, and increased reuse? Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems: • Changes in related classes force local changes • Harder to understand in isolation • Harder to reuse because its use requires the additional presence of the classes on which it is dependent
Discussion	Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider
Contraindications	High coupling to stable elements and to pervasive elements is seldom a problem. For example, a Java J2EE application can safely couple itself to the Java libraries (java.util and so on), because they are stable and widespread
Benefits	• Not affected by changes in other components • Simple to understand • Convenient to reuse
Background	Coupling and cohesion (described next) are truly fundamental principles in design, and should be appreciated and applied as such by all software developers

11.2.1.3 Cohesion

The software design usage description for cohesion also uses the word module. We use the same meaning for module as described above for coupling. Note that the cohesion definitions refer to *a module*. Thus, cohesion is evaluated for each module within a design. Note the contrast between coupling and cohesion—coupling exists between two modules while cohesion exists within one module.

Our intuition suggests a module with one basic purpose and is indivisible (i.e., it's difficult to split the module into separate more basic units [6]), results in a module easier to implement, test, and maintain. Having a module with one basic purpose that is hard to split into separate units is called *high cohesion* or *strong cohesion*. A good design is one that exhibits highly cohesive modules. Having a module that contains many basic purposes (i.e., many responsibilities) is called *low cohesion* or *weak cohesion*. A bad design is one that exhibits low cohesive modules [4,6].

Larman also includes a High Cohesion pattern in GRASP [9]. Table 11.3 summarizes the High Cohesion pattern.

Table 11.3 Summary of GRASP high cohesion

Solution	Assign a responsibility so that cohesion remains high
Problem	How to keep complexity manageable? In terms of object design, cohesion (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems: • Hard to comprehend • Hard to reuse • Hard to maintain • Delicate; constantly affected by change
Discussion	Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider
Contraindications	There are a few cases in which accepting lower cohesion is justified One case is the grouping of responsibilities or code into one class or component to simplify maintenance by one person—although be warned that such grouping may also make maintenance worse Another case for components with lower cohesion is with distributed server objects
Benefits	• Clarity and ease of comprehension of the design is increased • Maintenance and enhancements are simplified • Low coupling is often supported • The fine grain of highly related functionality supports increased reuse because a cohesive class can be used for a very specific purpose

11.2.1.4 Information Hiding

From a software design perspective, information pertains to data and algorithms. A good design should hide data and algorithm details from design elements that do not need to know this information. For example, a software application may need to save data to a persistent data store (e.g., a relational database). Only the design element responsible for saving data needs to know that it's using a database. All other design elements would simply call a method/function (to save data) without knowing details of how this is done. This allows the persistent data store to be changed (e.g., to an XML file) without having to change other parts of the design.

11.2.1.5 Performance

When two software designs provide a solution to the same problem space, and when these two solutions are similar in their simplicity, coupling, cohesion, and information hiding, then differences in performance may decide which design should be implemented. While it is an obvious statement, it is worth saying—a design that has better performance is a better design. Better performance means faster if we're measuring time and less if we're measuring space. Typically, time is measured using Big O notation and space is measured in bytes, kilobytes (KB), megabytes (MB), or gigabytes (GB).

11.2.1.6 Security

Security is not just a concern when we are programming, it is a concern during the entire software development life cycle. As discussed in Chap. 8, this book focuses on information security—the creation and application of security controls within a software solution. Information security within a software design is about the confidentiality, integrity, and availability of data/information.

Chapter 8 discussed security from a program design perspective. The five secure programming practices described in Chap. 8—data input validation, data output validation, exception handling, fail-safe defaults, and type-safe languages—should also be included when developing a software design.

The SEI CERT describes four security practices in their Top 10 Coding Practices that are directly related to software design [10]. These four practices are the following.

- Architect and design for security policies (discussed in Chap. 24).
- Keep it simple (see simplicity in Sect. 11.2.1)
- Practice defense in depth (discussed in Chap. 24)
- Use effective quality assurance techniques (discussed in Chaps. 23 and 24).

11.2.2 Abstraction: The Art of Software Design

Transitioning your thought process from program design to software design involves the use of abstraction. When designing software, you think of ways to abstract away

certain details. This is typically done by grouping certain details together based on shared characteristics or purposes. These shared characteristics or purposes are then a way to generalize your design. For example, a programmer may think about a soccer ball, tennis ball, baseball, and softball as distinct types of items. A software designer would generalize these items by classifying all of them as a type of ball that share certain characteristics (e.g., they are all spheres with a center point and radius) and purposes (e.g., they are all struck by an object).

As a software designer applies abstraction to a problem, different design elements are identified. Each of these design elements represents a certain level of abstraction. For example, a relatively detailed software design element may represent one or more elements found in the code. In an object-oriented design, methods and classes are directly implemented in code. In a structured design, functions are directly implemented in code. Thus, a class diagram (for object-oriented design) and a hierarchy chart (for structured design) represent software design models that are at a detailed level of abstraction. A software designer will produce more general design elements by combining detailed design elements. For example, a bunch of classes in an object-oriented design may be combined into a package. Or, a bunch of functions in a structured design may be combined into a component. As the designer continues to identify elements that represent more abstract designs, the designer produces a very high-level design often called a software architecture.

It is important to remind ourselves that software design abstractions should be created to represent both the structure and behavior of the software. The examples in the previous paragraph focused only on abstractions that describe software structure.

To illustrate the different thinking done by a programmer versus a software designer versus a software architect, the example from the first paragraph in this section is restated. A *programmer* may think about a soccer ball, tennis ball, baseball, and softball as distinct types of items used to play a specific type of game. A *software designer* would generalize these items by classifying all of them as a type of ball that share certain characteristics (e.g., they are all spheres) and purposes (e.g., they are all struck by another object). A *software architect* would abstract away even more details by recognizing that equipment is needed to play certain games. A ball represents one type of equipment needed to play these games.

11.2.2.1 Modeling Abstraction: Software Design Models

As we transition to thinking more about software design and less about program design, we move away from the details of coding. To do this effectively, we will need to introduce and use software design models that allow us to easily represent higher levels of abstraction.

Selected software design models will be introduced in Chaps. 12 and 13. These models will allow us to abstract away some of the details while accurately representing the structure and/or behavior of the software code. Additional software design models will be introduced, as needed, throughout the remaining chapters of this book.

11.2.2.2 Modeling Abstraction: Software Design Patterns

Software design patterns were first documented by Gamma, Helm, Johnson, and Vlissides, often referred to as the *Gang of Four (GoF)*, in their seminal book [11]. Their first paragraph in Chap. 1 Introduction eloquently states the challenges and opportunities for software designers.

> Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get 'right' the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

While all of the patterns this author has seen have been written with object-oriented design and programming in mind, much of the lessons learned and applied to OOD and OOP may be reworded and applied to structured design and programming. Below is the GoF quote, reworded so that it applies to software design regardless of the programming paradigm (e.g., object-oriented, imperative, functional, logical) being used. The *italicized words* identify the changes made to the GoF introductory paragraph.

> Designing software is hard, and designing reusable software is even harder. You must find pertinent *structures and behaviors*, factor these into *design elements* at the right granularity, *define interfaces and hierarchies*, and establish key relationships among *these design elements*. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced software designers will tell you that a reusable and flexible design is difficult if not impossible to get 'right' the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Chapter 27 provides an in-depth introduction to software design patterns. Chapters 28 and 29 apply selected software design patterns to the Address Book Application case study.

11.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as the process of generalizing a concept by removing details that are not needed to properly convey the perspective being emphasized.

Exercises

Discussion Questions

1. A simple measure that may be used to describe simplicity (or complexity) of program code is a count of the source lines of code. Can you think of reasons why counting source lines of code is *not* a good measure for simplicity?
2. Coupling and cohesion are two design concepts that are easy to mistake for each other. Are there examples outside of software design that may be used to better understand the distinction between these two design criteria?
3. Information hiding includes hiding the algorithm being used from other software elements that do not need to know how the processing is being performed. Besides the example used in this chapter—hiding the type of persistence storage being used—what are some other examples of hiding the details of an algorithm from other software elements?
4. Describe a problem statement (i.e., scenario) and have students discuss the use of the six characteristics of a good software design. Are there situations where a balance (i.e., trade-off) needs to be done between two or more of the design characteristics?
5. Use your development of an application that you started in Chap. 3 or 4 for this question. Discuss the six characteristics of a good software design and their use within your current program design and code. Which of these characteristics do you feel your program design adheres to and which do you feel needs to be improved? Explain your answers.
6. Based on the six characteristics of a good software design and your experiences, do you think that a perfect software design can be developed? Explain your answer.

References

1. Wiktionary.org: simplicity, coupling, cohesion, information hiding, performance, and security (2019) In: Wiktionary the free dictionary. Wikimedia Foundation. https://en.wiktionary.org/wiki/Wiktionary:Main_Page. Accessed 09 2019
2. Wikipedia.org: Coupling (computer programming) (2017) In: Wikipedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Coupling_(computer_programming). Accessed 19 2017
3. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. McGraw-Hill, New York
4. Dhama H (1995) Quantitative models of cohesion and coupling in software. J Syst Softw 29
5. Wikipedia.org: Cohesion (computer science) (2017) In: Wikipedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/Cohesion_(computer_science). Accessed 19 2017
6. Bieman JM, Ott LM (1994) Measuring functional cohesion. IEEE Trans Softw Eng 20(8)
7. Pfleeger SL (2001) Software engineering: theory and practice, 2nd edn. Prentice-Hall, Upper Saddle River

8. The Joint Task Force on Computing Curricula, ACM, IEEE Computer Society (2013) Computer science curricula 2013: curriculum guidelines for undergraduate degree programs in computer science
9. Larman C (2002) Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process, 2nd edn. Prentice Hall, Upper Saddle River
10. Software Engineering Institute (2019) Top 10 secure coding practices. Carnegie Mellon University. https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices. Accessed 11 2019
11. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston



OOD Case Study: Transition to Software Design

12

The objective of this chapter is to apply the characteristics of a good software design to the case study and to introduce additional design models that may be used to represent different levels of design abstraction.

12.1 OOD Preconditions

The following should be true prior to starting this chapter.

- You have been introduced to six software design characteristics—simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand five program design criteria—separation of concerns, design for reuse, design only what is needed, performance, and security.
- You have evaluated program code using these five criteria.
- You know that a class diagram is used in object-oriented solutions to illustrate the structure of your program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

12.2 OOD Transition to Software Design

Our strategy in learning more about how to express a software design is to introduce selected design models that will help us evaluate the ABA based on the six software design characteristics. We will evaluate the Chap. 9 ABA Version B solution described in Sect. 9.2.2. Our choice of design models will consider ways to produce abstractions of the ABA that accurately represent its structure and/or behavior.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_12

12.2.1 OOD ABA Structure Design Models

The class diagram for the Chap. 9 ABA Version B solution is shown in Fig. 12.1a. The class diagram design model was introduced in Chap. 3 and used in Chaps. 6 and 9. The only change shown in Fig. 12.1a is the use of a plus sign (+), shown immediately to the left of each attribute and method name, to indicate that the class member has public access.

Another modeling technique that shows the structure of an object-oriented design is a UML (Unified Modeling Language) package diagram. The package diagram for the Chap. 9 ABA Version B solution is shown in Fig. 12.1b. The three classes shown in Fig. 12.1a are now contained within a package notation. This package is unnamed since no *package* statement was used in the Java source code file (i.e., ABA_B_solution.java).

12.2.2 OOD ABA Structure and Behavior Design Models

IDEF0 is a function modeling technique that shows both the structure and behavior of a design. Figure 12.2 shows two activities performed by the Chap. 9 ABA Version B solution. The IDEF0 modeling technique uses four types of arrows:

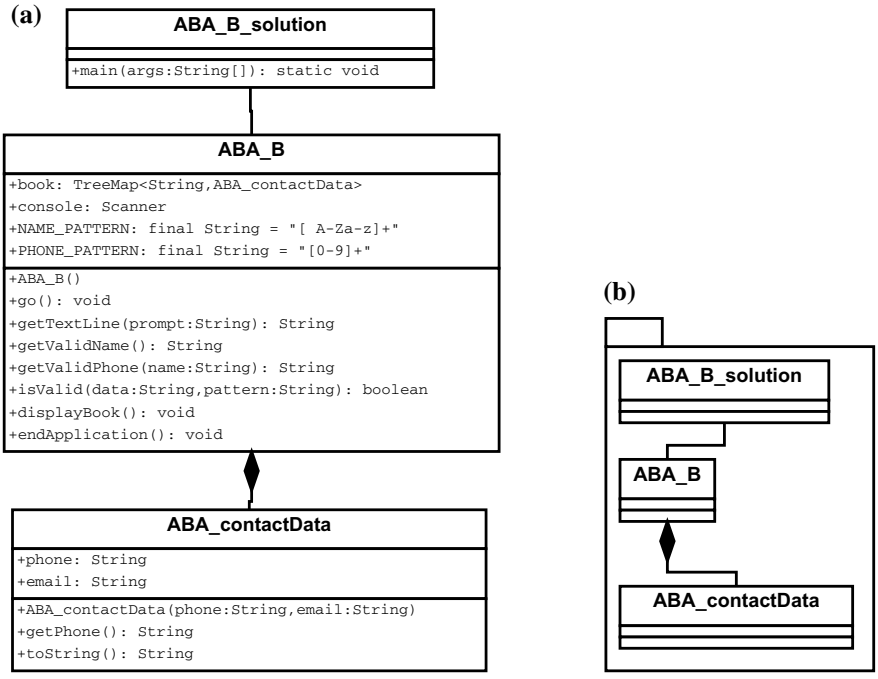


Fig. 12.1 **a** Class diagram. **b** Package diagram

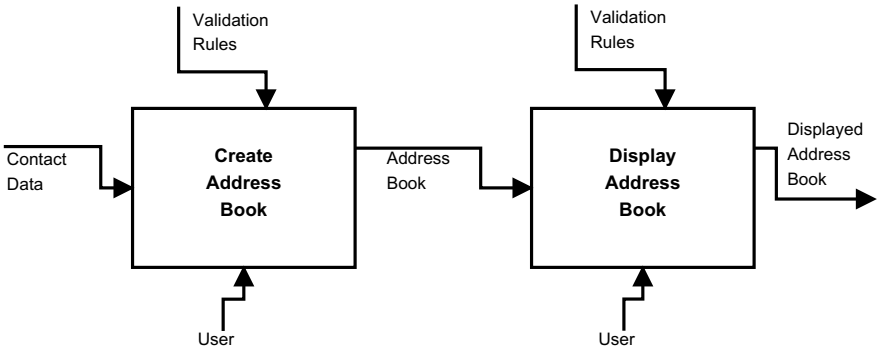


Fig. 12.2 IDEF0 function model for Chap. 9 ABA version B

- Input arrow: The data or objects that are transformed by the function into output. An input arrow points to the left side of an activity notation.
- Control arrow: The conditions required to produce correct output. Data or objects modeled as controls may be transformed by the function, creating output. A control arrow points to the top side of an activity notation.
- Output arrow: The data or objects produced by a function. An output arrow emanates from the right side of an activity notation.
- Mechanism arrow: The means used to perform a function. A mechanism arrow points to the bottom side of an activity notation.

In Fig. 12.2, *Contact Data* input by a *User* is used to create the memory-resident *Address Book* that is output from the first activity. The second activity then formats and displays this data to the *User*. *Validation Rules* are used by both activities to control the storage and display of valid *Address Book* data.

The definitive source of information on IDEF0 is the FIPS183 standard document [1]. As is typical of standards documents, FIPS183 is dense and difficult to read. Refer to Wikipedia [2] or idef.com [3] for additional information on IDEF0.

A UML communication diagram will show the structure and behavior of a design. A UML communication diagram shows interactions between objects via a sequence of messages (i.e., method calls). Figure 12.3 shows the UML communication diagram for the Chap. 9 ABA Version B solution. The ABA_OOP_B_solution object, whose class contains the main method, will construct a ABA_OOP_B object and then call its go method. The messages that start at the ABA_OOP_B object back to itself represent the logic performed by the go method. The numbering of the messages, 1.1, 1.2, 1.3, 1.6, and 1.7 for the messages sent from ABA_OOP_B to itself, and 1.4 and 1.5 for messages from ABA_OOP_B to the TreeMap<String,ABA_OOP_contactData> object, indicate the order in which these methods are called by the go method.

A UML communication diagram also shows relationships between the classes represented by the objects shown in the model. For example, the line connecting the ABA_OOP_B and TreeMap<String,ABA_OOP_contactData> objects, along with

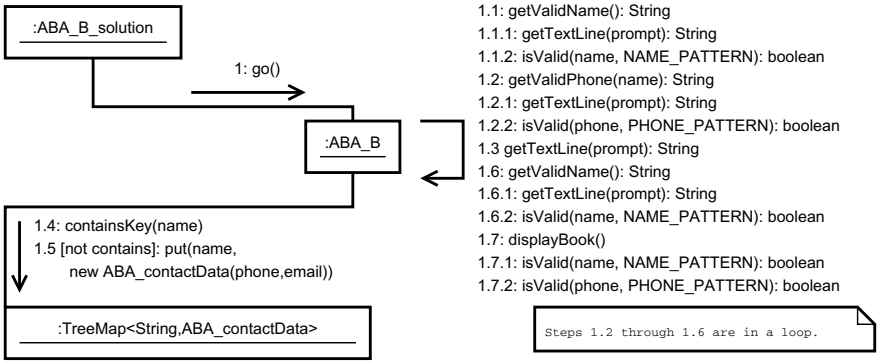


Fig. 12.3 UML communication diagram for Chap. 9 ABA version B

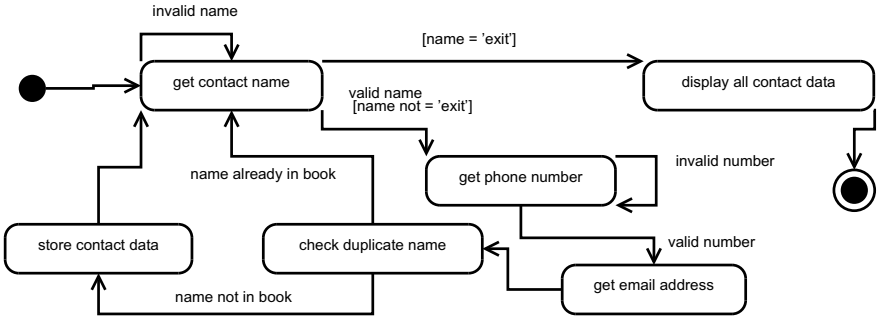


Fig. 12.4 Statechart for Chap. 9 ABA version B

the definition of the TreeMap, suggests that the ABA_OOP_B class is used as a container of many ABA_OOP_contactData objects.

12.2.3 OOD ABA Behavior Design Models

Figure 12.4 shows the statechart for the Chap. 9 ABA Version B solution. The statechart design model was introduced in Chap. 3 and used in Chaps. 6 and 9. Note that a statechart is also known as a *state diagram* or *state machine diagram* [4].

12.2.4 OOD Evaluate ABA Software Design

We'll use the six characteristics of a good software design described in Chap. 11 to evaluate the software design described above for the Chap. 9 ABA Version B solution.

12.2.4.1 Simplicity

In Chap. 11, simplicity is defined as *the amount and type of software elements needed to solve a problem*. We'll evaluate the ABA software design models described above based on this definition.

The UML class diagram in Fig. 12.1a contains only three classes. One of these classes contains the main method needed to run the Java program while another class is used as a container for a contact person's phone and email. There is one association and one composition relationship in this class diagram. From a class diagram perspective, this is a simple object-oriented design.

The UML package diagram in Fig. 12.1b shows one package containing the entire solution. From a package diagram perspective, this is a simple object-oriented design.

An IDEF0 function model includes five types of software elements: activity, input arrow, control arrow, output arrow, and mechanism arrow. Does the IDEF0 function model in Fig. 12.2 provide an accurate abstraction while minimizing the number of elements needed to express the design? With two activities and eight arrows showing inputs, controls, outputs, and mechanisms (ICOM), this IDEF0 function model describes two primary processing steps performed by the solution. In counting the ICOM arrows, note that both control arrows represent the same *Validation Rules*, both mechanism arrows represent the same User, and the *Address Book* arrow output from the first activity is then input into the second activity. Thus, there are only five distinct ICOM arrows in the diagram. Based on this brief analysis, this IDEF0 function model appears to be a simple abstraction of the solution.

The UML communication diagram in Fig. 12.3 shows the objects and messages involved in the Chap. 9 ABA Version B solution. The list of messages associated with the ABA_OOP_B object is rather extensive, and perhaps too detailed, resulting in this design diagram matching the details found in the code. Figure 12.5 shows a simpler diagram where the messages that are three-method-calls deep (e.g., 1.1.1, 1.1.2, 1.2.1, ...) have been removed. Removing this level of detail results in a design model that

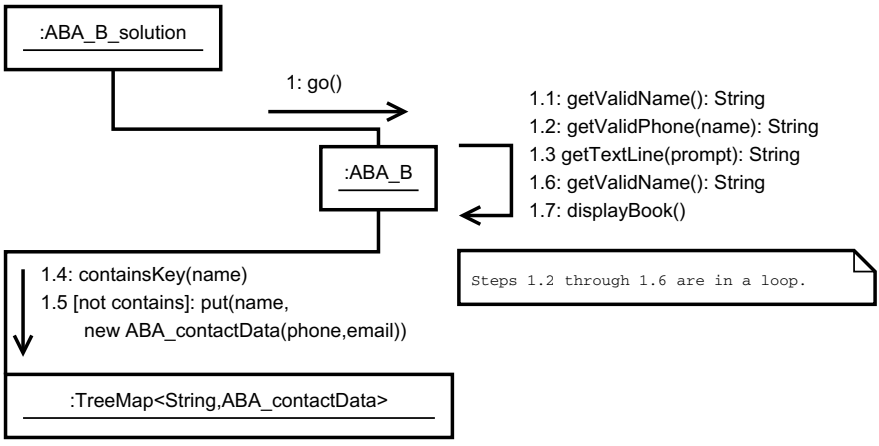


Fig. 12.5 Simpler UML communication diagram for Chap. 9 ABA version B

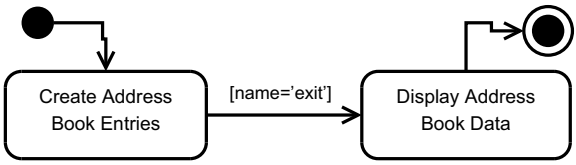


Fig. 12.6 More abstract statechart for Chap. 9 ABA version B

accurately reflects the code while abstracting away some of the details. Given the names of the messages (i.e., methods), this simpler design model clearly shows the overall design, where name and phone data are validated and stored in a TreeMap. The only vagueness in the 12.5 diagram is the message *1.3 getTextLine(prompt): String*, which does not clearly indicate the data value expected to be entered by the user. Note that creating another UML communication diagram that eliminates the second level of messages (e.g., 1.1, 1.2, ..., 1.7) to make the diagram even simpler would result in a diagram that does not convey any significant design information. The TreeMap object would be eliminated and the information about validating name and phone would no longer be expressed in the diagram.

A UML state machine diagram contains two types of software elements: states and transitions. Does the state machine in Fig. 12.4 provide an accurate abstraction while minimizing the number of elements needed to express the design? There are 6 states (8 if you count the initial and final states) and 11 transitions in the statechart, which shows the important behavior and user interactions of the ABA. Overall, this diagram seems fairly easy to understand. However, it does not use abstraction to hide many details. Figure 12.6 shows a state machine with only two states and three transitions. This diagram expresses a high-level design similar to the IDEF0 function model in Fig. 12.2. A benefit of the UML state machine modeling technique, when compared with the older statechart, is the option of creating substates (also known as sub-machines). In essence, we can create substates to represent a hierarchy of state machines. Figure 12.7 shows a substate diagram for the *Display Address Book Data* state that is shown in Fig. 12.6. The fact that a designer can create a hierarchy of state machines provides flexibility in how abstraction may be used to represent levels of software design. As shown in Figs. 12.6 and 12.7, a designer may choose to drill down on those states where it may be necessary to show a more detailed design.

12.2.4.2 Coupling

In Chap. 11, coupling is defined as *the degree to which each program module relies on other modules; the "connectedness" of a module to other modules; or the amount of interdependence between two or more modules*. We'll evaluate the ABA software design models described above based on this definition.

The UML class and package diagrams in Fig. 12.1a, b shows two relationships between the three classes. The association between the ABA_OOP_B_solution and ABA_OOP_B classes represents the main method constructing a ABA_OOP_B object and then calling it's go method. The composition relationship between the ABA_OOP_B and ABA_OOP_contactData classes represents the nonpersistent storage of address book data. These relationships represent the coupling in the Chap. 9 ABA Version B solution.

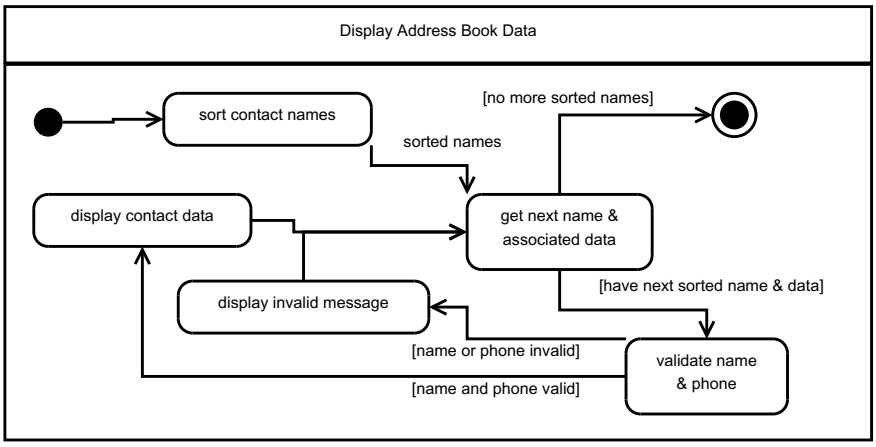


Fig. 12.7 Sub-Machine for Display Address Book Data for Chap. 9 ABA Version B

The IDEF0 function model in Fig. 12.2 shows one output flow (*Address Book*) from activity *Create Address Book* that flows as input into activity *Display Address Book*. This sharing of the address book data is the only coupling that exists between the two activities. Note that the control arrow *Validation Rules* and the mechanism *User* are associated with both activities. However, the *Validation Rules* are not transformed into the output, they are used to control the production of the output, and having the *User* associated with each activity simply denotes the role a person has when using the ABA.

The UML communication diagram in Fig. 12.5 shows object coupling that mirrors the class coupling expressed in the class and package diagrams. The UML communication diagram shows the association relationship between ABA_OOP_B_solution and ABA_OOP_B. The composition relationship between ABA_OOP_B and ABA_OOP_contactData is implied by the definition of the TreeMap data structure and the *1.5 [not contains] put(...)* message. Message numbers 1, 1.4, and 1.5 show message coupling between the three classes/objects, which represents a low form of coupling.

The UML state machine diagram in Fig. 12.6 shows coupling between the *Create Address Book Entries* and *Display Address Book Data* states. The exact nature of this coupling is not as evident as it is in the IDEF0 function model.

12.2.4.3 Cohesion

In Chap. 11, cohesion is defined as *the degree to which a software module is strongly related and focused in its responsibilities; a module that has a small, focused set of responsibilities and single-mindedness; or an attribute of a software unit that refers to the relatedness of its components*. We'll evaluate the ABA software design models described above based on this definition.

The UML class diagram in Fig. 12.1a can be used to assess the cohesion of each class since it shows the attributes and operations of each class. Looking at the ABA_OOP_B class, we see that it has two public attributes—book and console—that are used to store the address book data and obtain data from the user, respectively. The operations in this class clearly show some processing related to obtaining data from the user and validating this data. It also shows that this class is responsible for displaying the address book data. Essentially, the ABA_OOP_B class is responsible for all of the requirements stated in Chap. 9 for ABA Version B. Thus, this class has low cohesion. In contrast, the ABA_OOP_contactData class is responsible for containing data (phone and email) for one contact name. Given the attributes and operations in this class, this class is highly cohesive.

The UML package diagram in Fig. 12.1b does not show cohesion at the class level since attributes and operations are not being shown. However, we do see that all three classes are part of a single package. So we can ask, is this one package highly cohesive? Since all three classes combined represent the entire solution for the Chap. 9 ABA Version B, we could argue that this package is highly cohesive. We will revisit the notion of cohesion in a UML package diagram in Chap. 15.

For the IDEF0 function model in Fig. 12.2, we consider each activity to be a separate module. We need to determine if each IDEF0 activity has a singular focus (i.e., has strong cohesion). One way to address this is to identify the Chap. 9 ABA Version B requirements that apply to each of the IDEF0 activities. The following requirements are part of the *Create Address Book* activity.

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number.
- Use a simple text-based user interface to obtain the names and phone numbers.
- Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
- Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
- Prevent a duplicate name from being stored in the address book.

The *Create Address Book* activity implements all of the requirements stated for the Chap. 9 ABA Version B. We can react to this in one of two ways. We could argue that the IDEF0 function model is at such a high level of abstraction that it consolidates all of the requirements into a single activity. On the other hand, including user interactions, validation of data, and storage of valid data into one activity makes the activity have low cohesion. Given the above discussion on the cohesive properties shown in the class diagram, we conclude that we have a single IDEF0 activity that represents too much of the entire ABA.

Looking at the UML communication diagram in Fig. 12.5 results in an analysis of cohesion similar to the class diagram. The messages numbered 1.1 through 1.7 shows that the ABA_OOP_B object has methods that obtain data from the user,

validate this data, and displays the address book data. This supports the conclusion from the class diagram that the ABA_OOP_B class exhibits low cohesion. The two messages (i.e., 1.4 and 1.5) going to the TreeMap data structure shows that the ABA_OOP_contactData class has a singular focus of containing (phone and email) for one contact name. This also supports the conclusion from the class diagram that the ABA_OOP_contactData class is highly cohesive.

The UML state machine in Fig. 12.6 results in an analysis of cohesion similar to the IDEF0 function model. A single state—*Create Address Book Entries*—represents all of the requirements stated for the Chap. 9 ABA Version B. In this case, we can develop a sub-machine for this state that would have shown all of the states and transitions that fulfill the ABA requirements. This sub-machine would consist of states that are individually more cohesive since they represent a more detailed view of the ABA processing being performed. Thus, the hierarchy of state machines as illustrated in Figs. 12.6 and 12.7 uses abstraction to show a high-level machine while also allowing a more detailed view using a sub-machine whose states will likely be more cohesive.

12.2.4.4 Information Hiding

In Chap. 11, information hiding is defined as *having a component hide its implementation details (i.e., algorithms and data) from the other components*.

The UML class diagram shows that all of the attributes and operations have public visibility. This is a poor design since this means that any of these class members may be referenced by code in any other class. For example, code can be written to directly reference the TreeMap data structure. This could result in adding contact data that has not been validated. Similarly, since all of the operations are public, these could be called by code in any other class. None of the processing performed by these methods are hidden.

The UML package diagram, IDEF0 function model, UML communication diagram, and UML state machine design models cannot be evaluated for how well they hide information. This is because these modeling techniques do not directly show how information is used by the design elements.

12.2.4.5 Performance

In Chap. 11, performance is defined as *the analysis of an algorithm to determine its performance in terms of time (speed) and space (memory usage)*.

The UML class and UML communication diagrams both show that a TreeMap is used to store the address book data. As discussed in Chap. 6, this data structure will give us logarithmic time performance. The other design models do not show the data structures being used or algorithmic details to assess performance.

Since the UML state machine diagram shows interactions between the user and application, we could evaluate the user's experience using the state machine. However, evaluating the performance of a human–computer interaction (HCI) design has not yet been discussed. Chapters 17–22 will discuss ways to evaluate an HCI design.

12.2.4.6 Security

In Chap. 11, security is defined as *a set of technical controls intended to protect and defend information and information systems*. We'll evaluate the ABA software design models described above based on this definition.

The class diagram, IDEF0 function model, and UML communication diagram shown in Figs. 12.1a, 12.2, and 12.3, respectively, show data being validated before stored in the address book. Since the case study includes specific requirements about how to validate address book data, and since these design models show an implementation of these requirements, the Chap. 9 ABA Version B design satisfies two security criteria: validate input data and validate output data.

12.2.5 OOD ABA Software Design: Summary

A summary of the OOD design models used along with an evaluation of these models using characteristics for a good software design follows.

12.2.5.1 OOD ABA Design Models

The UML class and package diagrams are used to show the structure of a design. These types of models show the classes involved in the design and how these classes are related to each other. The class relationship types that have been used in our case study are association and composition.

Two modeling techniques show both structure and behavior. A software designer may create IDEF0 function models to show abstractions of the software based on high-level activities and the information flows (i.e., input, control, output, and mechanism) that influence these activities. The UML communication diagram technique shows the interactions of object instances via messages (i.e., method calls). A software designer can ignore certain detailed messages to convey the overall processing flow instead of showing all of the details (which can be observed in the code).

A statechart (or UML state machine) is used to show the behavior of a design. This behavior is represented by states and transitions, and is typically used by a software designer to show the important processing steps without regard to the elements that are responsible for implementing these steps.

12.2.5.2 OOD ABA Software Design Characteristics

The design models that illustrate the structure and behavior of the Chap. 9 ABA Version B solution exhibit the following design characteristics.

- Simple (good),
- Low or weak coupling (good),
- Low or weak cohesion (bad),
- Does not hide information (bad),
- Logarithmic time performance (good), and
- Does input and output validation (good).

12.3 OOD Top-Down Design Perspective

To reinforce the software design concepts covered so far, this section introduces a second case study that will be discussed top-down. Architects in the information technology industry, whether they focus solely on software or on both hardware and software, tend to apply a top-down design when decomposing a large system into high-level design elements (aka subsystems, components). We will first discuss the requirements of a second case study and then present a software design that meets these requirements. The software design is described using models to present abstractions without first discussing code. The intent of this second case study is to demonstrate the development of a software design based solely on the problem domain (and not on any existing implementation details).

12.3.1 OOD Personal Finances: A Second Case Study

The second case study will develop a personal finance software application that keeps track of accounts and transactions. The requirements of this application are as follows.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

12.3.2 OOD Personal Finances: Structure Design Models

The package diagram in Fig. 12.8 shows a single named package *PersonalFinance* and the classes within this package. This single package represents the classes that describe the scope of the personal finance problem domain.

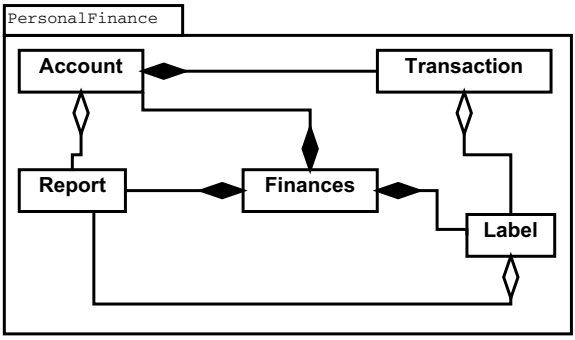


Fig. 12.8 Personal finance package diagram

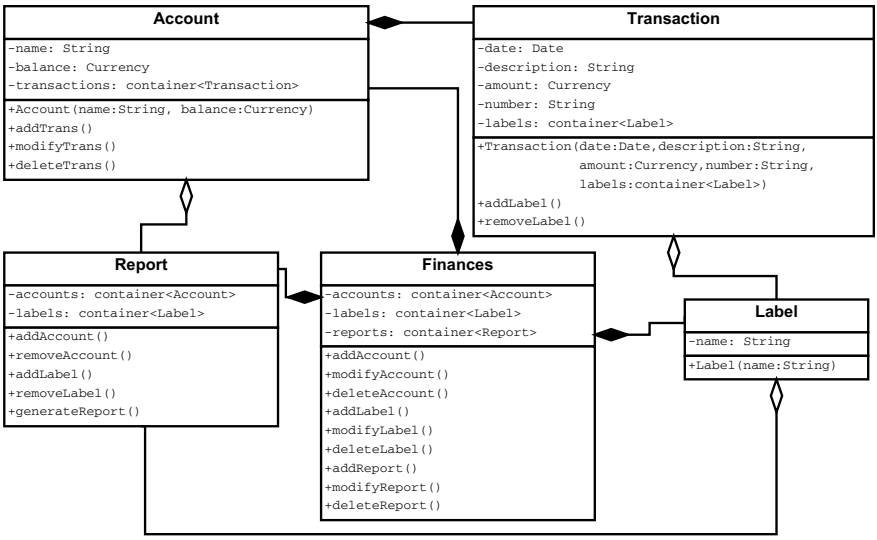


Fig. 12.9 Personal finance class diagram

The class diagram in Fig. 12.9 shows the attributes and public methods for each class shown in the package diagram. This class diagram is at a higher level of abstraction than what we anticipate needing to do with a detailed design or with the implementation. Examples of this abstraction include the following.

- Some attributes use the generic term *container* to represent a data structure that will need to be used to store multiple data values.
- None of the classes have *getter* and *setter* methods. For example, the *Transaction* class will need to have a method that returns each attribute value (e.g., getDate(), getDescription(), getAmount(), and getNumber()) and a method that sets each attribute value (e.g., setDate(date:Date), setDescription(desc:String), setAmount(amount:Currency), and setNumber(number:String)). The getter methods

would be needed when generating a report and the setter methods would be needed when a transaction is modified.
- The classes show the primary objects needed for the personal finance problem domain. At this high-level of abstraction, it is unclear whether these classes are responsible for a user interface or for persistent storage of personal finance data.

Another term that may be used to describe the class diagram in Fig. 12.9 is *domain model*. As just explained, the classes in this diagram represent the important aspects of the personal finance domain as delineated in the requirements listed above. As such, this class diagram represents knowledge about personal finance by showing the key concepts and their relationships to each other. As introduced in Chap. 1, software development processes (SDPs) describe a sequence of steps that, when performed, will produce software artifacts. As you may recall, the generic steps in an SDP include: gather needs, produce design, and implement design. The creation of the domain model class diagram is an example of a software model that accomplishes two things—it explains the needs/requirements and begins to show how the needs may be translated into a design. This illustrates the role of an architect in creating a high-level design, which may also be called an architecture. These high-level design models tend to convey needs/requirements while also serving as a starting point for creating more detailed design models. As we'll see in Chap. 15, while a domain model may represent a high-level view of a software design, these models tend to ignore important technical details that, if left out of more detailed designs, would result in an implementation that is not as robust and flexible in meeting future needs. While the term domain model has been used to describe the class diagram, the terms *business model* or *analysis model* may also be used to describe Fig. 12.9.

12.3.3 OOD Personal Finances: Structure and Behavior Design Models

A data-flow diagram (DFD) is a modeling technique that shows both the structure and behavior of a design. Figure 12.10 shows a high-level data-flow diagram for the personal finance case study. This figure illustrates the use of four notations within a DFD:

Process	A rounded-rectangle shape with a label that explains the processing being performed by this process. Figure 12.10 shows three processes—Account Management, Transaction Management, and Report Management.
Data store	A rectangle shape with a label that describes a persistent data store, e.g., database. Figure 12.10 shows one data store—Personal Finance Database.
External entity	A 3D-square that identifies an entity that exists outside the scope of the system being described. Figure 12.10 shows two external entities—User and Report.

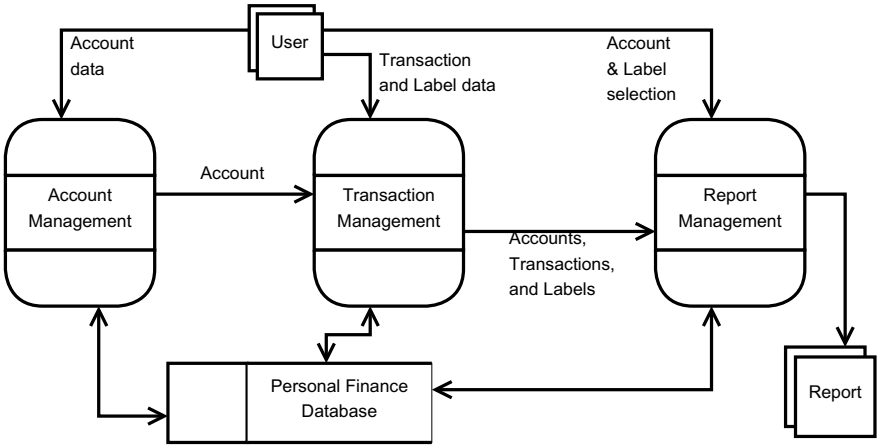


Fig. 12.10 Personal finance data-flow diagram

Data flow — A directed line that connects two other notations in the DFD, showing the flow of information between these two notations. A data flow may have an arrow on one or both ends and may include an optional label. Figure 12.10 shows five data flows each with a label and four data flows that are not labeled.

This DFD shows that Account Management creates/updates accounts which are then used by Transaction Management to record transactions within the account. Report Management would then create reports using the account, transaction, and label data. The data flows to/from the Personal Finance Database shows that all of the data created/updated by the three processes are stored in a persistent data store.

For more information about data-flow diagrams, refer to [5,6], or do a web search to find more recent articles.

The UML communication diagram in Fig. 12.11 shows the structure and behavior of the personal finance case study based on the domain model class diagram shown in Fig. 12.9. This communication diagram shows a typical flow of public method calls that express how the classes/objects interact with each other.

12.3.4 OOD Personal Finances: Behavior Design Models

The statechart diagram in Fig. 12.12 shows some of the user interactions with the personal finance system. This statechart was developed to emphasize the fact that transactions must be associated with an existing account.

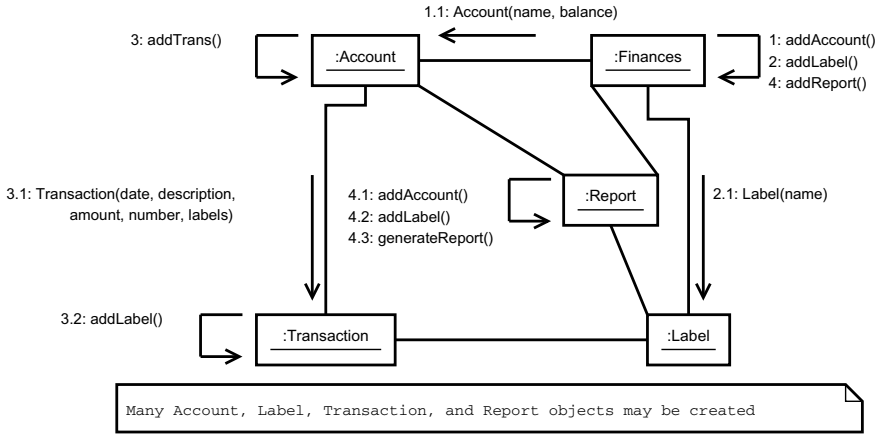


Fig. 12.11 Personal finance communication diagram

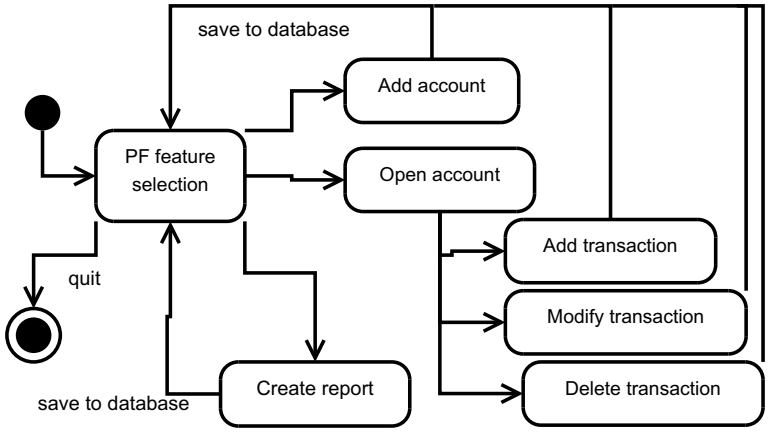


Fig. 12.12 Personal finance statechart

12.3.5 OOD Personal Finances: Summary

A summary of the OOD design models used along with an evaluation of these models using characteristics for a good software design follows.

12.3.5.1 Design Models

The design models shown in Figs. 12.8, 12.9, 12.10, 12.11 and 12.12 show a high-level design that represents an initial understanding of the needs/requirements and how they may be represented within the personal finance domain. Depending on what an architect wants to emphasize, some of these models may be omitted in the early stages of expressing knowledge of the domain. For example, if an understanding

of object-orientation is critical to the success of the personal finance software, then the package, class, and communication diagrams are important while the DFD and statechart may be omitted. On the other hand, showing interactions between the personal finance system and a user, or showing use of persistent storage, may be important. In these cases, the DFD and statechart would be important to develop.

12.3.5.2 Evaluation

An evaluation of the personal finance design, as shown in Figs. 12.8, 12.9, 12.10, 12.11, and 12.12, is briefly described below.

Simplicity Since we have described a high-level design, or architecture, our efforts should have resulted in models that accurately reflect the needs/requirements while also abstracting away the details implied by the requirements. The package, DFD, and statechart models are simple to read while the class and communication diagrams provide significantly more details that, perhaps, should not be included in a very high-level design.

Coupling The package, class, DFD, and communication models provide information to help us assess the coupling between the design elements. In the case of the package, class, and communication models, we see the interactions between the domain classes as described by the requirements. These couplings appear to be necessary given the list of requirements. The DFD shows an abstraction of the class diagram and emphasizes the three primary processes involved in the personal finance software system. Again, the interactions between these three processes appear to be necessary given the case study requirements.

Cohesion As discussed above for the class diagram, these classes represent the domain objects needed to design and implement personal finance software. If any of the classes include user interface or persistent data storage, then the class is responsible for too many distinctly different types of processing. Given this unknown, this high-level object-oriented design appears to have low cohesion (which is bad).

Information hiding Based on this high-level design, and given the lack of clarity about where the user interface and persistent storage design elements reside, one can argue this design effectively hides all implementation details.

Performance Without knowing more details about the data structures (i.e., containers) to be used, it is difficult to assess the performance of this high-level design. In addition, the *Personal Finance Database* shown on the DFD may have performance implications that cannot be assessed at this time.

Security The case study requirements do not state any need for information security. However, designers and implementers should always consider security even when this is not explicitly stated in the requirements. The high-level design models above do not address data input validation, data output validation, exception handling, fail-safe defaults, or type-safe languages.

12.4 OOD Post-conditions

The following should have been learned when completing this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You've created and/or modified design models that describe an object-oriented software design. To improve your learning of the topics presented in this chapter, you should create/modify design models using a bottom-up approach and also using a top-down approach. In a bottom-up approach, start with an implementation and develop design models that accurately reflect the code. In a top-down approach, start with requirements and develop high-level models that express these requirements.

Exercises

Discussion Questions

1. What changes would you make to the Chap. 9 ABA Version B design so that high cohesion is achieved while still having low coupling?
2. Describe a problem statement (i.e., scenario) and have students discuss use of the six characteristics for a good design. Are there situations where a balance (i.e., trade-off) needs to be done between two or more of the characteristics?

Hands-on Exercises

1. Using an existing code solution that you've developed, create design models that abstract away certain implementation details. Apply the characteristics for a good software design to your design models. How good or bad is your design?
2. Use your development of an application that you started in Chap. 3 for this exercise. Modify your design to improve your evaluation of the six characteristics for a good software design.
3. Select a sample problem domain from the list below, or choose a problem domain of interest to you, and develop a software design that describes structural and behavioral aspects of a potential solution.

Airline reservation and seat assignment An individual may reserve one or more flights where each flight has a from-airport with a departure date and time, and a to-airport with an arrival date and time. For each flight reservation, the individual may reserve one or more seats. Each flight reservation represents a one-way ticket. To complete the purchase of the flight reservations, the individual must provide the name of each passenger, a billing address and phone number, and credit card information.

Automated teller machine (ATM) An individual may deposit or withdraw funds from a bank account, transfer funds between two bank accounts, or obtain the balance of a bank account. An individual may perform as many transactions as they like, and may request a receipt that shows the results of each transaction. Before transactions can start, the individual must provide a valid bank card and pin number. The individual indicates when no more transactions are needed.

Bus transportation system A public transportation system needs to maintain their routes, times, drivers, and fares. This information may be different for weekdays, weekends, and for holidays.

Course-class enrollment An individual may register for one or more class sections. Each class section meets on one or more weekdays and times, has a classroom location, an instructor, and a course description. An individual may register for or drop a class section.

Digital library An individual may reserve or checkout items from an electronic library. The library contains books, movies, songs, and magazines. An individual may change a reservation and may extend their checkout due-date once.

Inventory and distribution control An organization keeps track of their product inventory and coordinates distribution of these products to customers. Each product has a unique identifier, description, dimensions, and weight.

Online retail shopping cart An individual may add and remove items from an electronic shopping cart. Each item in the shopping cart has a price and quantity purchased. To complete the purchase of the items in the shopping cart, the individual must identify the type of shipping requested, and provide their name, shipping address and phone number, billing address and phone number, and credit card information.

Personal calendar An individual wants to maintain their schedule on a calendar. The calendar may be used to show a single day, a week, or a month. The calendar allows the schedule of events to overlap each other.

Travel itinerary An individual needs to create a travel plan to visit N cities or destinations. The individual will identify travel time, distance, and route to get to each destination based on visiting the N locations in a specified order. The individual will also indicate the duration they will stay in each destination. An individual may have the need to create many distinct travel plans.

References

1. National Institute of Standards and Technology: Draft Federal Information Processing Standards Publication 183: Integration Definition for Function Modeling (IDEF0). NIST (1993). http://www.idef.com/wp-content/uploads/2016/02/idef0.pdf. Accessed 21 June 2017
2. Wikipedia.org: IDEF0. In: Wikipedia the free encyclopedia. Wikimedia Foundation (2017). https://en.wikipedia.org/wiki/IDEF0. Accessed 21 June 2017
3. Knowledge Based Systems: IDEF0 Function Modeling Method. Knowledge Based Systems, Inc (2017). http://www.idef.com/idefo-function_modeling_method/. Accessed 21 June 2017
4. Wikipedia.org: State diagram. In: Wikipedia the free encyclopedia. Wikimedia Foundation (2017). https://en.wikipedia.org/wiki/State_diagram. Accessed 23 June 2017
5. Gane C, Sarson T (1977) Structured systems analysis: tools and techniques. McDonnell Douglas Systems Integration Company
6. Yourdon E, Constantine L (1979) Structured design: fundamentals of a discipline of computer program and systems design. Prentice Hall, Upper Saddle River



SD Case Study: Transition to Software Design

13

The objective of this chapter is to apply the characteristics of a good software design to the case study and to introduce additional design models that may be used to represent different levels of design abstraction.

13.1 SD Preconditions

The following should be true prior to starting this chapter.

- You have been introduced to six software design characteristics—simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand five program design criteria—separation of concerns, design for reuse, design only what is needed, performance, and security.
- You have evaluated program code using these five criteria.
- You know that a hierarchy chart is used in structured solutions to illustrate the structure of your program design.
- You know that a Nassi–Shneiderman diagram and a statechart are models used to illustrate the behavior of your program design.

13.2 SD Transition to Software Design

Our strategy in learning more about how to express a software design is to introduce selected design models that will help us evaluate the ABA based on the six software design characteristics. We will evaluate Chap. 10 ABA Version B solution described in Sect. 10.2.2. Our choice of design models will consider ways to produce abstractions of the ABA that accurately represent its structure and/or behavior.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_13

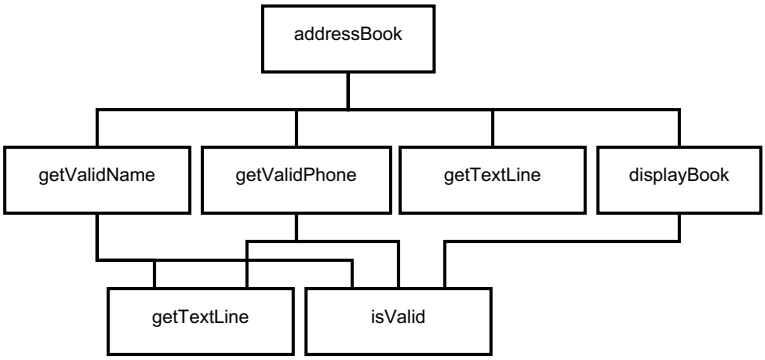


Fig. 13.1 Python Hierarchy chart for Chap. 10 ABA Version B

13.2.1 SD ABA Structure Design Models

The hierarchy chart for Chap. 10 ABA Version B solution is shown in Fig. 13.1. The hierarchy chart design model was introduced in Chap. 4 and used in Chaps. 7 and 10.

13.2.2 SD ABA Structure and Behavior Design Models

IDEF0 is a function modeling technique that shows both the structure and behavior of a design. Figure 13.2 shows two activities performed by Chap. 10 ABA Version B solution. The IDEF0 modeling technique uses four types of arrows:

- Input arrow: The data or objects that are transformed by the function into output. An input arrow points to the left side of an activity notation.
- Control arrow: The conditions required to produce correct output. Data or objects modeled as controls may be transformed by the function, creating output. A control arrow points to the topside of an activity notation.
- Output arrow: The data or objects produced by a function. An output arrow emanates from the right side of an activity notation.
- Mechanism arrow: The means used to perform a function. A mechanism arrow points to the bottom side of an activity notation.

In Fig. 13.2, *Contact Data* input by a *User* is used to create the memory-resident *Address Book* that is output from the first activity. The second activity then formats and displays this data to the *User*. *Validation Rules* are used by both activities to control the storage and display of valid *Address Book* data.

The definitive source of information on IDEF0 is the FIPS183 standard document [1]. As is typical of standards documents, FIPS183 is dense and difficult to read. Refer to Wikipedia [2] or idef.com [3] for additional information on IDEF0.

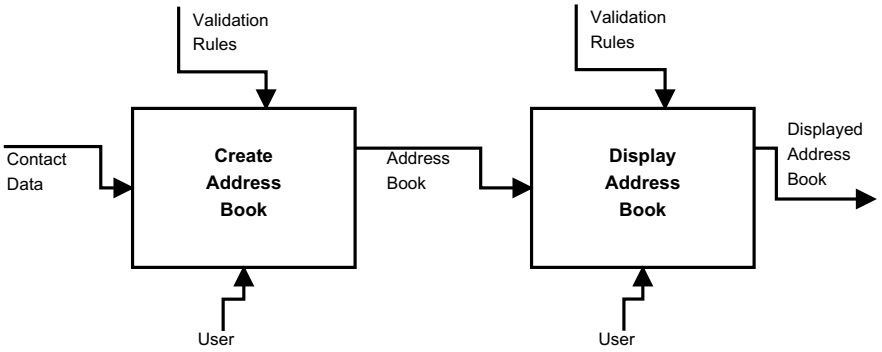


Fig. 13.2 IDEF0 function model for Chap. 10 ABA Version B

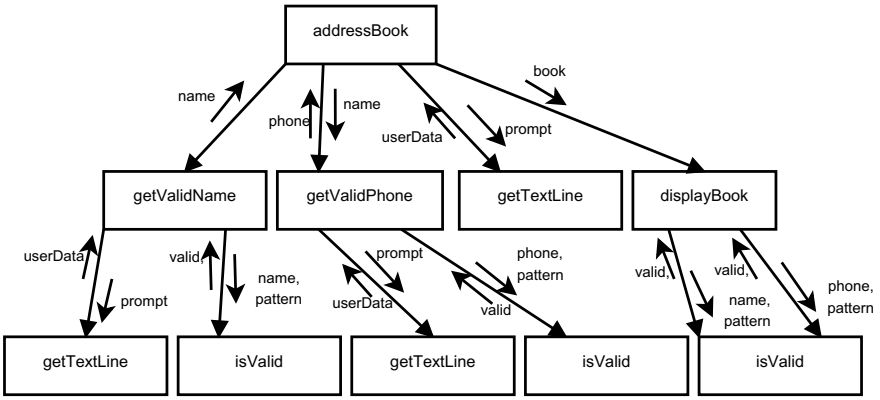


Fig. 13.3 Structure chart for Chap. 10 ABA Version B

A structure chart and structure diagram are two other models that will show the structure and behavior of a design. Figures 13.3 and 13.4 show these two diagrams for Chap. 10 ABA Version B solution.

A structure chart is similar to a hierarchy chart as it shows the order of function calls. In addition, a structure chart shows information flows associated with passing data to/from a function [4]. Each directed line that connects two rectangles/functions represents a function call. Each shorter directed line (not connected to anything) shows an information flow representing passing data via a function call or return statement. A structure chart may be used to show higher level abstractions instead of specific function names. In this way, a structure chart is used to show information flows between subcomponents or components of a larger system.

A structure diagram, introduced by Michael A. Jackson in 1975 [4], is also similar to a hierarchy chart as it shows a hierarchy of software elements. However, a structure diagram may include abstractions to help describe certain behaviors. For example, Fig. 13.4 has two rectangles—*process entry* and *add contact*—that do not

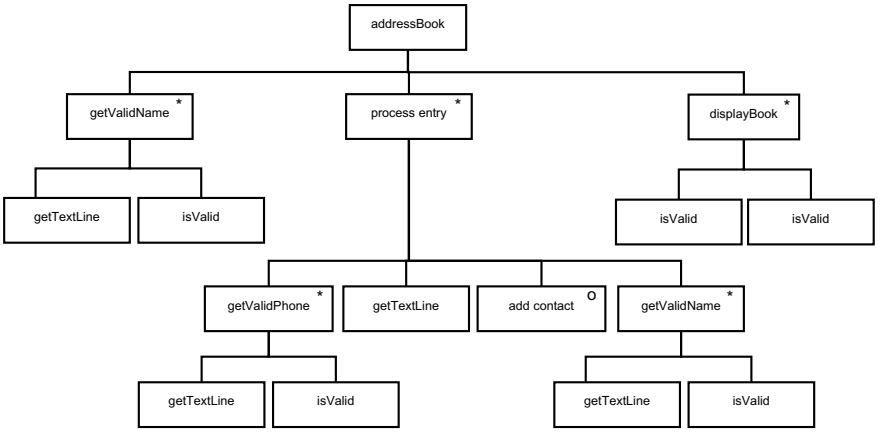


Fig. 13.4 Structure diagram for Chap. 10 ABA Version B

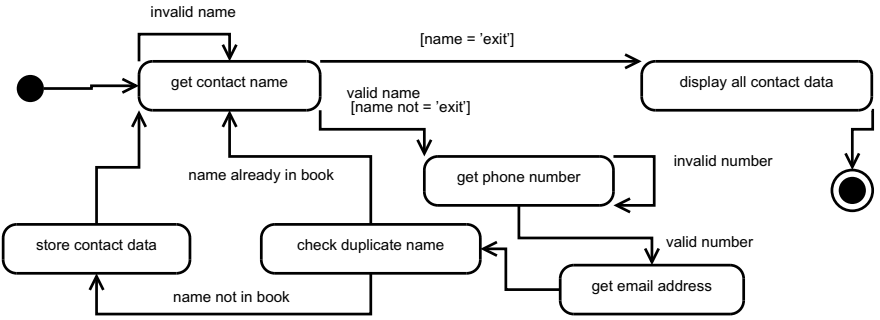


Fig. 13.5 Statechart for Chap. 10 ABA Version B

correspond to a function definition in the Python code. The *process entry* rectangle includes an asterisk (*) in the upper-right corner to indicate that the group of software elements below it in the hierarchy is performed iteratively. The *add contact* rectangle includes a lowercase "oh" (o) in the upper-right corner to indicate that a contact is added conditionally. Similar to a structure chart, a structure diagram may be used to show higher level abstractions. In this way, a structure diagram can show sequence, selection, and iteration between subcomponents or components of a larger system.

13.2.3 SD ABA Behavior Design Models

Figure 13.5 shows the statechart for Chap. 10 ABA Version B solution. The statechart design model was introduced in Chap. 3 and used in Chaps. 7 and 10. Note that a statechart is also known as a *state diagram* or a *state machine diagram* [5].

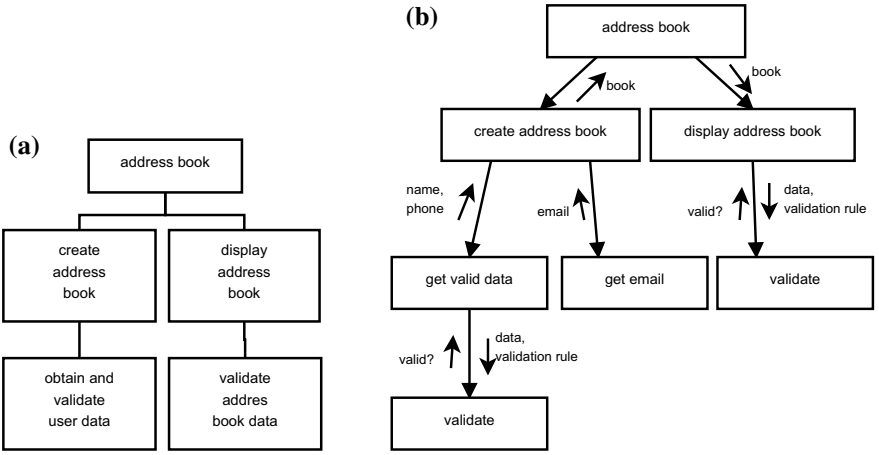


Fig. 13.6 (**a**) Simpler Hierarchy chart (**b**) Simpler structure chart

13.2.4 SD Evaluate ABA Software Design

We'll use the six characteristics of a good software design described in Chap. 11 to evaluate Chap. 10 ABA Version B solution.

13.2.4.1 Simplicity

In Chap. 11, simplicity is defined as *the amount and type of software elements needed to solve a problem*. We'll evaluate the ABA software design models described above based on this definition.

The hierarchy chart in Fig. 13.1 shows the sequence of function calls. This represents a program design, rather than a software design, since it does not abstract away any of the details found in the code. From the perspective of simplicity, creating a hierarchy chart to show all of the function calls would be time-consuming and complex for a larger solution. The hierarchy chart shown in Fig. 13.6a also reflects the processing of the solution but does so with only five design elements. The *address book* module at the top of the hierarchy represents the entire application while the two modules underneath it represent the two main components of the application. The two modules shown at the lowest level of the hierarchy provide more details about the two modules immediately above them while expressing important design information about the validation of data. To summarize, this hierarchy chart reduces the number of design elements while conveying the important aspects of the solution.

An IDEF0 function model includes five types of software elements: activity, input arrow, control arrow, output arrow, and mechanism arrow. Does the IDEF0 function model in Fig. 12.2 provide an accurate abstraction while minimizing the number of elements needed to express the design? With two activities and eight arrows showing inputs, controls, outputs, and mechanisms (ICOM), this IDEF0 function

model describes two primary processing steps performed by the solution. In counting the ICOM arrows, note that both control arrows represent the same *Validation Rules*, both mechanism arrows represent the same *User*, and the *Address Book* arrow output from the first activity is then input into the second activity. Thus, there are only five distinct ICOM arrows in the diagram. Based on this brief analysis, this IDEF0 function model appears to be a simple abstraction of the solution.

A structure chart shows information flows associated with passing data to/from a function. Like the discussion about the hierarchy chart in Fig. 13.1, the structure chart in Fig. 13.3 shows the program design. What we want is a structure chart that accurately portrays the design while abstracting away some of the details. The structure chart in Fig. 13.6b emphasizes the validation of name and phone while also showing that the address book also includes an email address. In comparing the two structure charts, Fig. 13.6b contains three fewer functions and ten fewer information flows. Relative to the structure chart in Fig. 13.3, the structure chart in Fig. 13.6b is a simpler design abstraction of the solution. Note that a structure chart could have shown the same five modules displayed in the hierarchy chart of Fig. 13.6a. Doing this would have eliminated two more modules and a few more information flows, making the structure chart even simpler. However, this simpler structure chart would have made it more difficult to express which data is validated (e.g., name and phone) and which is not (e.g., email). Since data validation is an important security criteria, we want to express this feature in our design models.

A structure diagram shows three types of software elements: sequence, selection, and iteration. The structure diagram in Fig. 13.4 shows the program design. Like the hierarchy chart and structure chart discussions above, we want to create a more abstract structure diagram that conveys important design information while eliminating those details that are deemed less important to document. Figure 13.7 shows a simple design abstraction, where specific Python function names have been replaced with a short phrase that describes the processing being done. Both the *create book entry* and *display book* notations include the asterisk symbol to denote iteration. The use of iteration in this structure diagram is informative as it reinforces our thinking of an address book containing many entries. Finally, like the structure chart discussed above, we could have created a much more abstract structure diagram similar to the hierarchy chart in Fig. 13.6a. Since we want to express data validation in our design models, a simpler structure diagram showing only five functions was not created.

A UML state machine diagram contains two types of software elements: states and transitions. Does the state machine in Fig. 13.5 provide an accurate abstraction while minimizing the number of elements needed to express the design? There are six states (eight if you count the initial and final states) and eleven transitions in the statechart, which show the important behavior and user interactions of the ABA. Overall, this diagram seems fairly easy to understand. However, it does not use abstraction to hide many details. Figure 13.8 shows a state machine with only two states and three transitions. This diagram expresses a high-level design similar to the IDEF0 function model in Fig. 13.2. A benefit of the UML state machine modeling technique, when compared with the older statechart, is the option of creating substates (also known as sub-machines). In essence, we can create substates to represent a hierarchy of state

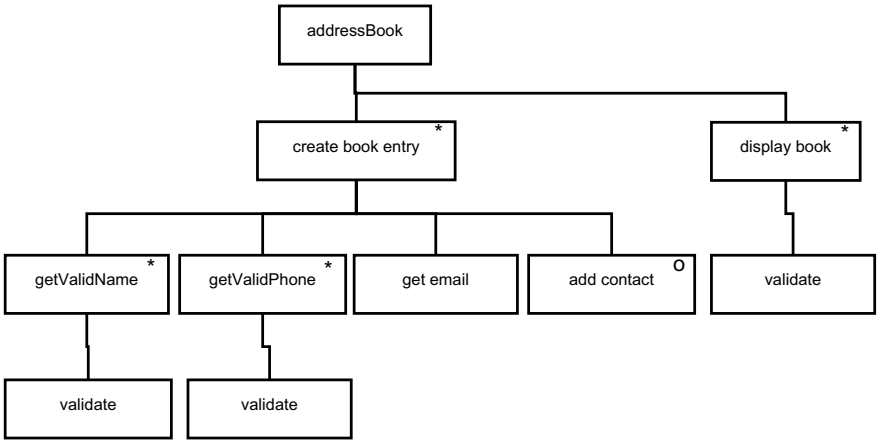
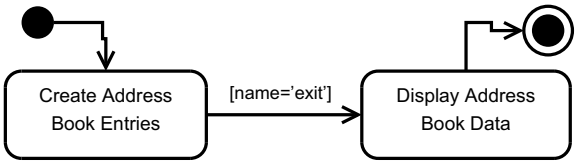


Fig. 13.7 Simpler structure diagram for Chap. 10 ABA Version B

Fig. 13.8 More abstract statechart for Chap. 10 ABA Version B



machines. Figure 13.9 shows a substate diagram for the *Display Address Book Data* state that is shown in Fig. 13.8. The fact that a designer can create a hierarchy of state machines provides flexibility in how abstraction may be used to represent levels of software design. As shown in Figs. 13.8 and 13.9, a designer may choose to drill down on those states where it may be necessary to show a more detailed design.

13.2.4.2 Coupling

In Chap. 11, coupling is defined as *the degree to which each program module relies on other modules; the "connectedness" of a module to other modules; or the amount of interdependence between two or more modules*. We'll evaluate the ABA software design models described above based on this definition.

A hierarchy chart shows coupling in terms of the order in which the modules are called and which modules use other modules. However, the degree in which modules depend on each other is not expressed in a hierarchy chart. Thus, we cannot ascertain the degree of coupling using Figs. 13.1 or 13.6a.

The IDEF0 function model in Fig. 13.2 shows one output flow (*Address Book*) from activity *Create Address Book* that flows as input into activity *Display Address Book*. This sharing of the address book data is the only coupling that exists between the two activities. Note that the control arrow *Validation Rules* and the mechanism *User* are associated with both activities. However, the *Validation Rules* are not trans-

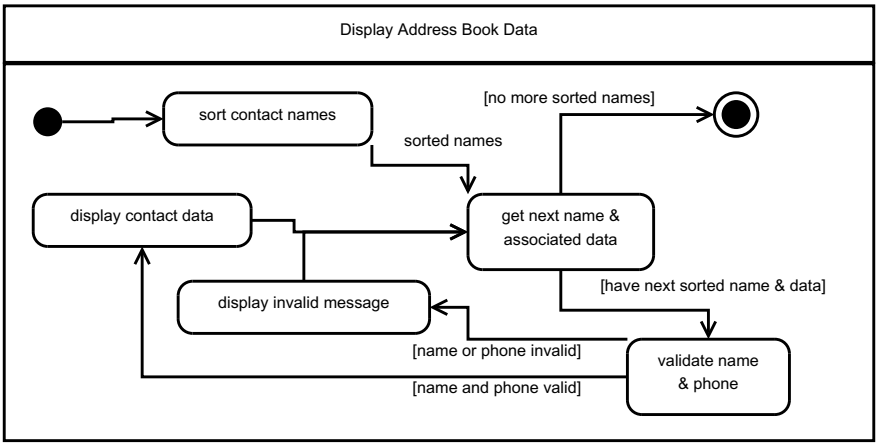


Fig. 13.9 Sub-machine for Display Address Book Data for Chap. 10 ABA Version B

formed into the output, they are used to control the production of the output, and having the *User* associated with each activity simply denotes the role a person has when using the ABA.

The structure chart in Fig. 13.6b shows coupling between the *create address book* and *displayBook* modules via the information flows labeled *book*. This coupling is the same as what is shown in the IDEF0 function model. The *get valid data* and *get email* modules provide address book data to the *create address book* module. The *validation* module validates a data value based on a validation rule. The coupling shown in this structure chart seems to be fairly low, particularly between the two modules shown immediately below the *address book* module.

The structure diagram in Fig. 13.7, like the hierarchy chart, only shows coupling in terms of the order in which the modules (i.e., components) are called. It does not show the degree to which these modules depend on each other.

The UML state machine diagram in Fig. 13.8 shows coupling between the *Create Address Book Entries* and *Display Address Book Data* states. The exact nature of this coupling is not as evident as it is in the IDEF0 function model.

13.2.4.3 Cohesion

In Chap. 11, cohesion is defined as the *degree to which a software module is strongly related and focused in its responsibilities; a module that has a small, focused set of responsibilities and single-mindedness; or an attribute of a software unit that refers to the relatedness of its components*. We'll evaluate the ABA software design models described above based on this definition.

For the hierarchy chart in Fig. 13.6a, *create address book* and *display address book* are the two modules that we evaluate for their cohesion. We need to determine if each of these modules has a singular focus (i.e., has strong cohesion). Below is a list of all of

the requirements stated for Chap. 10 ABA Version B, which are implemented entirely within the *create address book* module. Thus, the *create address book* module has low cohesion.

- Allow for entry and (nonpersistent) storage of people's names.
- Store for each person a single phone number.
- Prevent a duplicate name from being stored in the address book.
- Use a simple text-based user interface to obtain the names and phone numbers.
- Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
- Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.

We can do a similar analysis of cohesion for the IDEF0 function model, structure chart, and structure diagram, as shown in Figs. 13.2, 13.6b, and 13.7, respectively. Each of these design models has a *module* that is analogous to the *create address book* found in the hierarchy chart. In all of these design models, this module exhibits low (or weak) cohesion.

The UML state machine in 13.8 results in an analysis of cohesion similar to the hierarchy chart and IDEF0 function model. The state *Create Address Book Entries* represents all of the requirements stated for Chap. 10 ABA Version B. In this case, we can develop a sub-machine for this state that would have shown all of the states and transitions that fulfill the ABA requirements. This sub-machine would consist of states that are individually more cohesive since they represent a more detailed view of the ABA processing being performed. Thus, the hierarchy of state machines as illustrated in Figs. 13.8 and 13.9 uses abstraction to show a high-level machine while also allowing a more detailed view using a sub-machine whose states will likely be more cohesive.

13.2.4.4 Information Hiding

In Chap. 11, information hiding is defined as *having a component hide its implementation details (i.e., algorithms and data) from the other components*.

The IDEF0 function model and structure chart design models provide some evidence that the design does a good job of hiding information. Specifically, the IDEF0 *Create Address Book* activity produces an *Address Book* that is given to the *Display Address Book* activity. Similarly, the structure chart shows that the *create address book* module produces a *book* that is then given to the *displayBook* function. In both cases, the module responsible for displaying address book data has no idea how the address book was created.

The hierarchy chart, structure diagram, and statechart design models cannot be evaluated for how well they hide information. This is because these modeling techniques do not directly show how information is used by the design elements.

13.2.4.5 Performance

In Chap. 11, performance is defined as the *analysis of an algorithm to determine its performance in terms of time (speed) and space (memory usage)*.

None of the design models described above show a design abstraction that expresses enough detail about the algorithms and data structures being used. Thus, we cannot measure the performance of the design using any of these models.

Since the UML state machine diagram shows interactions between the user and application, we could evaluate the user's experience using the state machine. However, evaluating the performance of a human–computer interaction (HCI) design has not yet been discussed. Chapters 17 through 22 will discuss ways to evaluate an HCI design.

13.2.4.6 Security

In Chap. 11, security is defined as *a set of technical controls intended to protect and defend information and information systems*. We'll evaluate the ABA software design models described above based on this definition.

The hierarchy chart, IDEF0 function model, structure chart, and structure diagram, as shown in Figs. 13.6a, 13.2, 13.6b, and 13.7, respectively, all show that data is validated before being stored in the address book. Since the case study includes specific statements about how to validate address book data, and since these design models show an implementation of these requirements, the Chap. 10 ABA Version B design satisfies two security criteria: validate input data and validate output data.

13.2.5 SD ABA Software Design: Summary

A summary of the SD design models used along with an evaluation of these models using characteristics for a good software design follows.

13.2.5.1 SD ABA Design Models

A hierarchy chart is used to show the structure of a design. This type of model shows the hierarchy of function calls in a structured solution. A software designer may produce an abstraction of the function calls by identifying higher level modules that cooperate with each other in a hierarchical fashion.

Three modeling techniques show both structure and behavior. A software designer may create IDEF0 function models to show abstractions of the software based on high-level activities and the information flows (i.e., input, control, output, and mechanism) that influence these activities. The structure chart and structure diagram techniques are similar to a hierarchy chart, except that they include certain information about behavior. A software designer can use structure charts to show information flows between high-level modules that cooperate in a hierarchical fashion. Or, a software designer can use structure diagrams to show the sequence, selection, and iterative nature of the modules.

A statechart (or UML state machine) is used to show the behavior of a design. This behavior is represented by states and transitions, and is typically used by a software designer to show the important processing steps without regard to the elements that are responsible for implementing these steps.

13.2.5.2 SD ABA Software Design Characteristics

The design models that illustrate the structure and behavior of Chap. 10 ABA Version B solution exhibit the following design characteristics.

- Simple (good),
- Low or weak coupling (good),
- Low or weak cohesion (bad),
- Information hiding (unknown),
- Performance (unknown), and
- Does input and output validation (good).

13.3 SD Top-Down Design Perspective

To reinforce the software design concepts covered so far, this section introduces a second case study that will be discussed top-down. Architects in the information technology industry, whether they focus solely on software or on both hardware and software, tend to apply a top-down design when decomposing a large system into high-level design elements (aka subsystems, components). We will first discuss the requirements of a second case study and then present a software design that meets these requirements. The software design is described using models to present abstractions without first discussing code. The intent of this second case study is to demonstrate the development of a software design based solely on the problem domain (and not on any existing implementation details).

13.3.1 SD Personal Finances: A Second Case Study

The second case study will develop a personal finance software application that keeps track of accounts and transactions. The requirements of this application are as follows.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.

- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

13.3.2 SD Personal Finances: Structure Design Models

The hierarchy chart in Fig. 13.10 shows the personal finance application split into three functions. This represents the scope of the personal finance problem domain.

13.3.3 SD Personal Finances: Structure and Behavior Design Models

A Data-Flow Diagram (DFD) is a modeling technique that shows both the structure and behavior of a design. Figure 13.11 shows a high-level data-flow diagram for the personal finance case study. This figure illustrates the use of four notations within a DFD:

Process A rounded-rectangle shape with a label that explains the processing being performed by this process. Figure 13.11 shows three processes—Account Management, Transaction Management, and Report Management.

Data store A rectangle shape with a label that describes a persistent data store, e.g., database. Figure 13.11 shows one data store—Personal Finance Database.

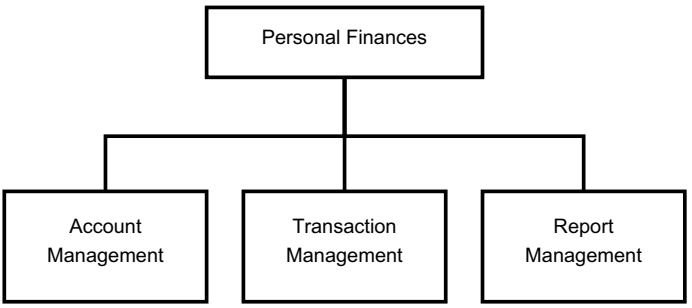


Fig. 13.10 Personal finance Hierarchy chart

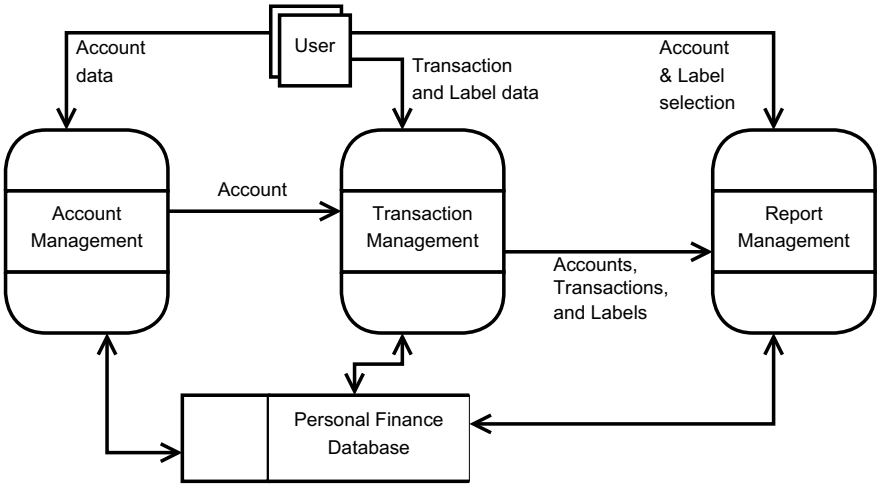


Fig. 13.11 Personal finance data-flow diagram

External entity A 3D-square that identifies an entity that exits outside the scope of the system being described. Figure 13.11 shows two external entities—User and Report.

Data flow A directed line that connects two other notations in the DFD, showing the flow of information between these two notations. A data flow may have an arrow on one or both ends and may include an optional label. Figure 13.11 shows five data flows each with a label and four data flows that are not labeled.

This DFD shows that Account Management creates/updates accounts which are then used by Transaction Management to record transactions within the account. Report Management would then create reports using the account, transaction, and label data. The data flows to/from the Personal Finance Database shows all the data created/updated by the three processes is stored in a persistent data store.

For more information about data-flow diagrams, refer to [6,7], or do a web search to find more recent articles.

The structure chart in Fig. 13.12 shows the flow of information between the three primary components, which appear immediately below the *Personal Finances* top-level module. It also shows more details about transactions and their association with an existing account.

13.3.4 SD Personal Finances: Behavior Design Models

The statechart diagram in Fig. 13.13 shows some of the user interactions with the personal finance system. This statechart was developed to emphasize the fact that transactions must be associated with an existing account.

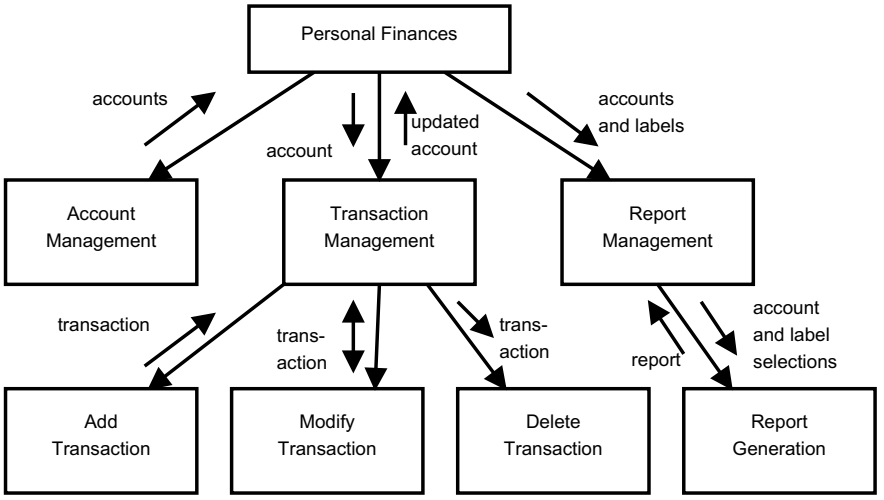


Fig. 13.12 Personal finance structure chart

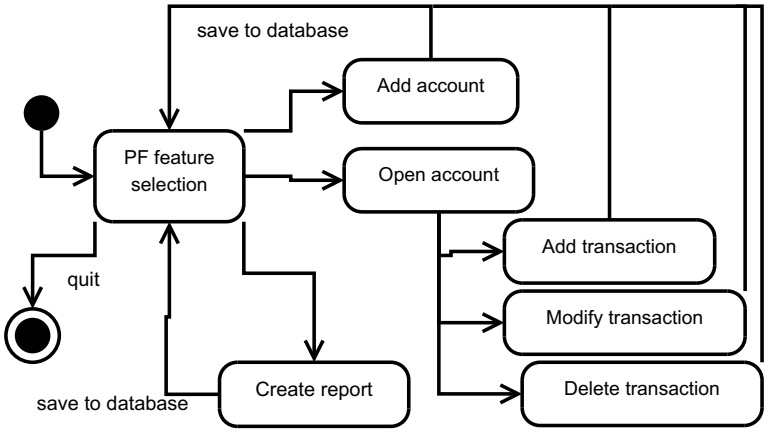


Fig. 13.13 Personal finance statechart

13.3.5 SD Personal Finances: Summary

A summary of the SD design models used along with an evaluation of these models using characteristics for a good software design follows.

13.3.5.1 Design Models

The design models shown in Figs. 13.10 through 13.13 show a high-level design that represents an initial understanding of the needs/requirements and how they may be represented within the personal finance domain. Depending on what an architect

wants to emphasize, some of these models may be omitted in the early stages of expressing knowledge of the domain. For example, if an understanding of the design structure is critical to the success of the personal finance software, then the hierarchy and structure chart diagrams are important while the DFD and statechart may be omitted. On the other hand, showing interactions between the personal finance system and a user, or showing use of persistent storage, may be important. In these cases, the DFD and statechart would be important to develop.

13.3.5.2 Evaluation

An evaluation of the personal finance design, as shown in the above four models, is briefly described below.

Simplicity Since we have described a high-level design, or architecture, our efforts should have resulted in models that accurately reflect the needs/requirements while also abstracting away the details implied by the requirements. The hierarchy, DFD, and statechart models are simple to read while the structure chart provides significantly more details that, perhaps, should not be included in a very high-level design.

Coupling The DFD and structure chart provide information to help us assess the coupling between the design elements. In the case of these two models, we see the interactions between the domain components as described by the requirements. These couplings appear to be necessary given the list of requirements. Both models emphasize the three primary processes involved in the personal finance software system.

Cohesion As discussed above for the hierarchy, DFD, and structure chart, the components expressed in these models represent the domain objects needed to design and implement personal finance software. If any of the components include user interface or persistent data storage, then the component is responsible for too many distinctly different types of processing. Given this unknown, this high-level structured design appears to have low cohesion (which is bad).

Information hiding Based on this high-level design, and given the lack of clarity about where the user interface and persistent storage design elements reside, one can argue that the design effectively hides all implementation details.

Performance Without knowing more details about the data structures to be used, it is difficult to assess the performance of this high-level design. In addition, the *Personal Finance Database* shown on the DFD may have performance implications that cannot be assessed at this time.

Security The case study requirements do not state any need for information security. However, designers and implementers should always consider security even when this is not explicitly stated in the requirements. The high-level design models above do not address data input validation, data output validation, exception handling, fail-safe defaults, or type-safe languages.

13.4 SD Post-conditions

The following should have been learned when completing this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You've created and/or modified design models that describe a structured software design. To improve your learning of the topics presented in this chapter, you should create/modify design models using a bottom-up approach and also using a top-down approach. In a bottom-up approach, start with an implementation and develop design models that accurately reflect the code. In a top-down approach, start with requirements and develop high-level models that express these requirements.

Exercises

Discussion Questions

1. What changes might you make to Chap. 10 ABA Version B design so that high cohesion is achieved while still having low coupling?
2. Describe a problem statement (i.e., scenario) and have students discuss use of the six characteristics for a good design. Are there situations where a balance (i.e., trade-off) needs to be done between two or more of the characteristics?

Hands-On Exercises

1. Using an existing code solution that you've developed, create design models that abstract away certain implementation details. Apply the characteristics for a good software design to your design models. How good or bad is your design?
2. Use your development of an application that you started in Chap. 4 for this exercise. Modify your design to improve your evaluation of the six characteristics for a good software design.

3. Select a sample problem domain from the list below, or choose a problem domain of interest to you, and develop a software design that describes structural and behavioral aspects of a potential solution.

Airline reservation and seat assignment An individual may reserve one or more flights where each flight has a from-airport with a departure date and time, and a to-airport with an arrival date and time. For each flight reservation, the individual may reserve one or more seats. Each flight reservation represents a one-way ticket. To complete the purchase of the flight reservations, the individual must provide the name of each passenger, a billing address and phone number, and credit card information.

Automated teller machine (ATM) An individual may deposit or withdraw funds from a bank account, transfer funds between two bank accounts, or obtain the balance of a bank account. An individual may perform as many transactions as they like, and may request a receipt that shows the results of each transaction. Before transactions can start, the individual must provide a valid bank card and pin number. The individual indicates when no more transactions are needed.

Bus transportation system A public transportation system needs to maintain their routes, times, drivers, and fares. This information may be different for weekdays, weekends, and for holidays.

Course-class enrollment An individual may register for one or more class sections. Each class section meets on one or more weekdays and times, has a classroom location, an instructor, and a course description. An individual may register for or drop a class section.

Digital library An individual may reserve or checkout items from an electronic library. The library contains books, movies, songs, and magazines. An individual may change a reservation and may extend their checkout due date once.

Inventory and distribution control An organization keeps track of its product inventory and coordinates distribution of these products to customers. Each product has a unique identifier, description, dimensions, and weight.

Online retail shopping cart An individual may add and remove items from an electronic shopping cart. Each item in the shopping cart has a price and quantity purchased. To complete the purchase of the items in the shopping cart, the individual must identify the type of shipping requested, and provide their name, shipping address and phone number, billing address and phone number, and credit card information.

Personal calendar An individual wants to maintain their schedule on a calendar. The calendar may be used to show a single day, a week, or a month. The calendar allows the schedule of events to overlap with each other.

Travel itinerary An individual needs to create a travel plan to visit N cities or destinations. The individual will identify travel time, distance, and route to get to each destination based on visiting the N locations in a specified order. The individual will also indicate the duration they will stay at each destination. An individual may have the need to create many distinct travel plans.

References

1. National Institute of Standards and Technology (NIST) (1993) Draft federal information processing standards publication 183: Integration definition for function modeling (IDEF0). http://www.idef.com/wp-content/uploads/2016/02/idef0.pdf. Accessed 21 June 2017
2. Wikipedia.org: IDEF0 (2017) In: Wikipedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/IDEF0. Accessed 21 June 2017
3. Knowledge Based Systems (2017) IDEF0 function modeling method. Knowledge Based Systems, Inc. http://www.idef.com/idefo-function_modeling_method/. Accessed 21 June 2017
4. Budgen D (2003) Software design. Addison Wesley, Boston
5. Wikipedia.org: State diagram (2017) In: Wikipedia the free encyclopedia. Wikimedia Foundation. https://en.wikipedia.org/wiki/State_diagram. Accessed 23 June 2017
6. Gane C, Sarson T (1977) Structured systems analysis: tools and techniques. McDonnell Douglas Systems Integration Company, Plano
7. Yourdon E, Constantine L (1979) Structured design: fundamentals of a discipline of computer program and systems design. Prentice Hall, Upper Saddle River



Introduction to Model-View-Controller

14

The objective of this chapter is to introduce the Model–View–Controller (MVC) architectural pattern.

14.1 Preconditions

The following should be true prior to starting this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

14.2 Concepts and Context

The case study software designs discussed in Chaps. 12 and 13 exhibit similar design characteristics. The OOD and SD designs have four characteristics in common suggesting the designs are good. These four design characteristics are simple, have low or weak coupling, have good (logarithmic) time performance, and they do input and output data validation. However, both the OOD and SD designs have low or weak cohesion, which is a bad design trait. This chapter introduces an architectural pattern called Model–View–Controller as a way to improve the cohesion of these designs while still maintaining the good characteristics these designs already exhibit.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_14

14.2.1 Introduction to Model–View–Controller (MVC)

Model–View–Controller (MVC) is a software architecture or framework that splits a software application into three design components.

- The *model component* is responsible for managing data. This may include interfacing with any persistent data storage and using memory-based data structures to store data during execution. The model component implements logic to allow the application to create, read, update, and delete (CRUD) the application data.
- The *view component* is responsible for providing an interface for user interactions, assuming the software requires a user interface.
- The *controller component* is responsible for the domain logic, also called business logic, associated with the software. This includes having the controller communicate with the view and model components. In essence, the controller component includes code that *glues* these three components together.

There are two clear benefits to using MVC. First, keeping the user interface separate from data management allows the technology for one of these components to be changed without impacting the technology used by the other component. For example, we can change the persistent data store from XML to a relational database without needing to change anything in the view (i.e., user interface) component. Likewise, the user interface can be changed from a Java-based graphical user interface to a mobile-based application without having to change anything in the model (i.e., data management) component. Second, using MVC results in having three components that are each more cohesive. As will be shown in Chaps. 15 and 16, each MVC component is more cohesive given the more focused set of responsibilities assigned to the component.

The MVC software architecture may be explained using the IDEF0 function model, as shown in Fig. 14.1. The *View* component sends data provided by the user to the *Controller*. The Controller may validate this data and/or apply domain rules to transform this input data before sending *domain data* to the Model component. Note the *UI data* output arrow from the View goes into the Controller as a control, to show this data is being validated, and as an input, to show this data is being transformed by domain rules. As a result of processing done by the Controller, domain data is sent to the *Model*. The Model may validate this data (as shown by the *domain data* control arrow) and/or treat this data as input (as shown by the *domain data* input arrow). The Model will send *model data* to the Controller, or optionally to the View for display to the user. When the model data is sent to the Controller, this data may be validated and/or transformed by applying domain rules, and then given back to the Model component or sent to the View for display to the user.

Figure 14.1 is a good example of the ICOM arrows in an IDEF0 function model providing too much information for the type of design abstraction we want to express. Essentially, the generic description of MVC using IDEF0 is difficult to read given the number of ICOM arrows used to express MVC. For this reason, a data-flow diagram (DFD) is shown in Fig. 14.2. A DFD includes notations that represent a process

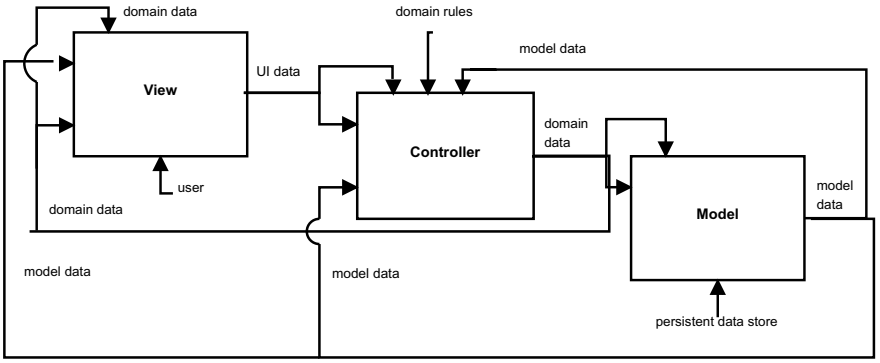


Fig. 14.1 Generic Model–View–Controller IDEF0 function model

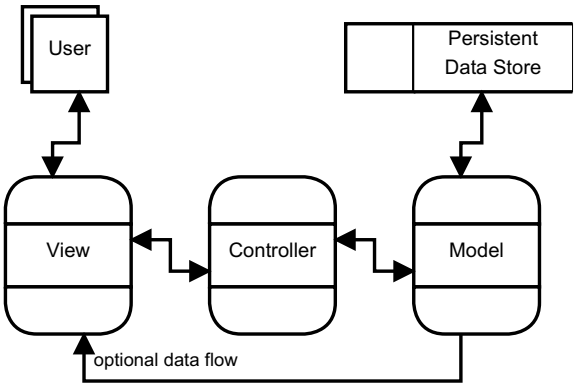


Fig. 14.2 Generic Model–View–Controller data-flow diagram

(e.g., see View, Controller, and Model), data flows (e.g., see the directed arrows that connect the other notations), external entities (e.g., see User), and data stores (e.g., see Persistent Data Store). Since a DFD data flow can represent any type of flow between components, it represents three of the ICOM arrows—input, control, and output. Thus, the DFD in Fig. 14.2 is significantly easier to read and more clearly shows the MVC architectural pattern.

Based on the two architecture diagrams shown in Figs. 14.1 and 14.2, the following identifies the key characteristics of the MVC architecture/framework.

- The user interacts only with the view component.
- A persistent data store is used only by the model component. The model is also responsible for using memory-based data structures for temporary data storage.
- The view and controller communicate with each other. Data may flow in either direction between these two components.
- The model and controller communicate with each other. Data may flow in either direction between these two components.
- Optionally, the model may send data directly to the view, but not vice versa.

The last bullet is worth a little discussion. Since the model is responsible for managing the data, it may be a good idea (e.g., for performance reasons) to have the model send data directly to the view for display to the user without involving the controller. On the other hand, any data coming from the view (which may have been modified by a user, perhaps one with malicious intent) must go through the controller before the model is allowed to update a data store. This places responsibility for validating user data in the controller. The validation done by the controller would be specific to the application domain being implemented. (It is also likely the model does some data validation based on the type of persistent data store being used.)

14.2.1.1 MVC in Distributed Applications

The ABA case study is a standalone software application, which means that it runs on a single computing device. Given this, the three components communicate with each other via function/method calls. How might the MVC description change if we are applying this architectural pattern to a distributed application?

Figure 14.3 shows a client–server (or 2-tier) architecture for MVC, where the client contains the view and a client-side controller while the server contains the model and server-side controller. The controller component is essentially split between the two tiers, resulting in data validation occurring on both the client and server devices. Why should validation be done on both tiers? First, it may be fairly easy for a user with malicious intent to create a client that communicates with the server. If the server were to assume that validation is done by the client, and thus the server does no validation, then the spoofed client could inject malicious data, do an SQL injection attack, or perform some similarly bad action that results in a less secure application. Second, if the client were to assume that validations are being done by the server, and thus the client does no validation, it may be possible for an attacker to spoof the server side of this application, resulting in the validation not being done. In this scenario, the user of the client may be providing sensitive data to the spoofed server.

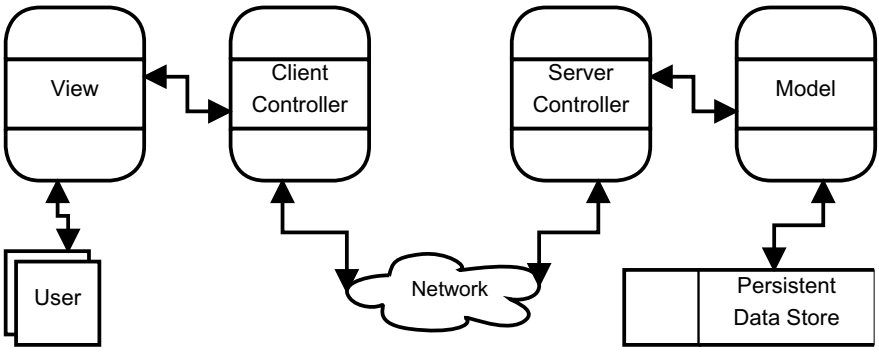


Fig. 14.3 Generic 2-tier Model–View–Controller data-flow diagram

We can extend this discussion to a 3-tier or n-tier application that uses the MVC architecture. Like the 2-tier discussion just presented, any component that communicates with another over a network should not assume the other component is doing validation. Validation needs to be done by each distributed component to avoid malicious users from taking advantage of your less-secure design.

14.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand Model–View–Controller is a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.

The following should have been learned when completing previous chapters.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as the process of generalizing a concept by removing details that are not needed to properly convey the design perspective being emphasized.

Exercises

Discussion Questions

1. What are the benefits of separating the user interface from the application data?
2. In a standalone application using MVC, which component should validate data and why?
3. Give an example of a type of application that would benefit from having the model component send data directly to the view?
4. Pick an application domain, identify requirements for this domain, and then assign each requirement to one or more of the MVC components.



OOD Case Study: Model–View–Controller

15

The objective of this chapter is to apply the Model–View–Controller architectural pattern to the case study using object-oriented design techniques.

15.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.

15.2 OOD: Transition to MVC

The case study software design discussed in Chap. 12 exhibits low (or weak) cohesion, which characterizes a poor design. This chapter will implement Model–View–Controller to improve the cohesion of the ABA while maintaining the good design

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_15

characteristics of simple, low coupling, logarithmic time performance, and doing input/output data validation.

One way to think about the implementation of Model–View–Controller is from a requirements perspective. We identify the component or components that need to be involved in order to satisfy each requirement. The following requirements have been implemented in the address book application.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Do not retrieve any information from the address book.

Implementing requirement 1 involves both the view (*allow for entry*) and model (*allow for nonpersistent storage*) components. Requirement 2 will be part of the model component while requirement 3 is associated with the view component. Requirements 4 and 5 describe the types of validation needed for the name and phone number values. Since this is a standalone application, i.e., it is not distributed across computing devices; this validation belongs to the controller component. (It is important to emphasize that any application whose components are distributed across computing devices should have validation requirements implemented in multiple components. While this duplicates much of the validation logic, it will greatly improve the defensive capabilities of the application in resisting malicious attacks via input channels.) Requirement 6 is another validation requirement, but this one can only be implemented in the model component. This is because the model is the only component that knows the type of data structure being used to non-persistently store the data. Finally, requirement 7 exists to simplify the design of the address book application by eliminating the need for a user interface to include user requests to display address book data. This last requirement affects the design of all three components by reducing the need for logic to display address book entries to the user.[1]

Given the above allocation of requirements to components, requirement 1 has been divided into two sub-requirements. Table 15.1 summarizes the allocation of requirements to components. The remaining sections in this chapter will refer to requirements 1a, 1b, and 2 through 6.

[1] While address book data is displayed once the user exits the application, this is done to verify the address book contained correct information based on the order in which data was entered by the user. As explained in prior chapters, this display of address book data is test code used to verify the correctness of the logic implementing requirements 1 through 6.

Table 15.1 Map requirements to MVC

Requirement	Model	View	Controller
1a. Allow for entry of a person's name		X	
1b. Allow for (nonpersistent) storage of people's names	X		
2. Store for each person a single phone number and a single email address	X		
3. Use a simple text-based user interface to obtain a name, phone number, and email address for each contact		X	
4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered			X
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered			X
6. Prevent a duplicate name from being stored in the address book	X		

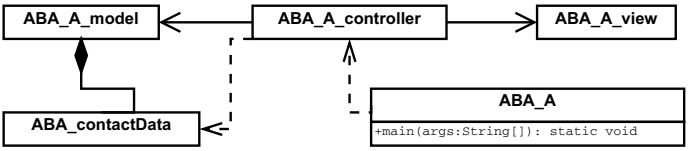


Fig. 15.1 Package diagram for MVC design ABA Version A

15.3 OOD: ABA MVC Design Version A

The Version A design of the first MVC implementation is described using design models and code snippets. The design is explained using a series of questions typically raised when first learning MVC.

15.3.1 OOD: In What Order Do MVC Objects Get Created?

The Version A design for the ABA using Model–View–Controller has five classes, as shown in the package diagram in Fig. 15.1. The model component consists of two classes: ABA_A_model and ABA_contactData. The view component has a single class named ABA_A_view. The controller component consists of the remaining two classes: ABA_A and ABA_A_controller.

The dependency relationship between ABA_A and ABA_A_controller exists because the main method creates an ABA_A_controller object and then uses this object to call the go() method. This is shown in Listing 15.1.

Listing 15.1 ABA MVC Design version A—main method

```
public class ABA_A {
    public static void main(String[] args)
    {
        ABA_A_controller aba = new ABA_A_controller();
        aba.go();
    }
}
```

The package diagram in Fig. 15.1 shows an association relationship between the ABA_A_controller class and two other classes: ABA_A_model and ABA_A_view. The default constructor in the ABA_A_controller class creates objects for these two classes, as shown in Listing 15.2.

Listing 15.2 ABA MVC Design version A—controller default constructor

```
public class ABA_A_controller
{
    private ABA_A_model model;
    private ABA_A_view view;

    public ABA_A_controller()
    {
        model = new  ABA_A_model();
        view = new ABA_A_view();
    }
```

The code in Listings 15.1 and 15.2 illustrates an important difference between dependency and association relationship types. An association relationship identifies a strong relationship between two classes. In the Version A design, the ABA_A_controller class has an instance variable for the two classes it is associated with (ABA_A_model and ABA_A_view). After the controller constructor has executed, any code in this class can communicate directly with the model or view objects. In contrast, a dependency relationship is a weaker relationship between two classes. In this design, the main method has a local variable that references the ABA_A_controller object. This object reference may only be used by code in the main method.

Another way to describe the difference between dependency and association relationship types is the impact a change would have on the related class. For the dependency relationship between ABA_A and ABA_A_controller, there are only two changes to the ABA_A_controller class that would force a change to the ABA_A code. These two changes are (1) the ABA_A_controller class name is changed or (2) the public go() method is changed by either changing its name or adding formal parameters. For the association relationships, a change to the ABA_A_model or ABA_A_view class name, or a change to any public method in these two classes, would result in making changes to the controller code.

The package diagram in Fig. 15.1 shows a composition relationship between the ABA_A_model and ABA_contactData classes. The default constructor method in

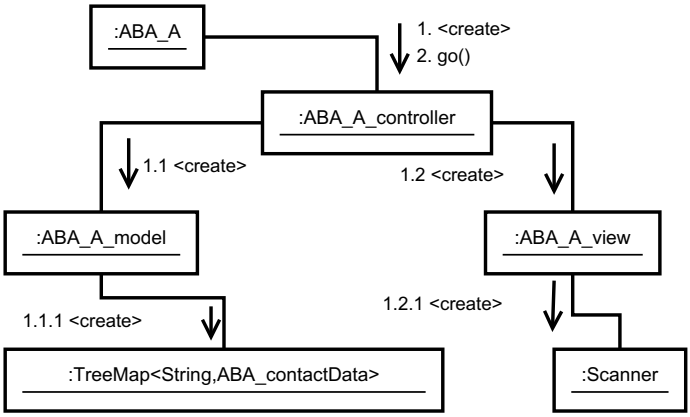


Fig. 15.2 Communication diagram—MVC design Version A—Start up

the ABA_A_model class creates a TreeMap data structure to store ABA_contactData objects. This is shown in Listing 15.3.

Listing 15.3 ABA MVC Design version A—model Startup Code

```
public class ABA_A_model {
    private TreeMap<String,ABA_contactData> book;
    private NavigableSet<String> keySet;
    private Iterator<String> iterKeySet;

    public ABA_A_model()
    {
        book = new TreeMap<String,ABA_contactData>();
        keySet = null;
    }
```

A composition relationship type identifies a whole-part relationship between a class that contains many instances of another class. In the Version A design, the ABA_A_model class has a book instance variable of type TreeMap representing a container. The book container is capable of storing many ABA_contactData objects.

The unified modeling language (UML) has two types of whole-part relationships: aggregation and composition. The notation for aggregation is a diamond symbol with no fill color. As shown in the package diagram in Fig. 15.1, composition uses a diamond symbol filled with black. This Version A design of the ABA uses composition because the ABA_contactData objects must be contained in the book data structure in order for these objects to exist. When the book container is destroyed, all of the ABA_contactData objects will also be destroyed.

The communication diagram in Fig. 15.2 shows the sequence of steps and the interaction of the MVC components in starting the ABA. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version A solution starts to execute.

Step 1: The main method creates an ABA_A_controller object.
Step 1.1: The ABA_A_controller constructor method creates an ABA_A_model object.
Step 1.1.1: The ABA_A_model constructor method creates a TreeMap data structure for nonpersistent storage of contact data, then initializes the keySet instance variable to null.
Step 1.2: The ABA_A_controller constructor method creates an ABA_A_view object.
Step 1.2.1: The ABA_A_view constructor method creates a Scanner object used to obtain user data.
Step 2: The main method calls the go method using the controller object.

The code for the ABA_A_view constructor method is shown in Listing 15.4.

Listing 15.4 ABA MVC Design version A—view Startup Code

```
public class ABA_A_view {
    private ABA_A_controller controller;
    private Scanner console;

    public ABA_A_view(ABA_A_controller controller)
    {
        this.controller = controller;
        console = new Scanner(System.in);
    }
```

15.3.2 OOD: How Do MVC Objects Communicate with Each Other?

The simple, and obvious, answer is by using object references to call public methods. More specifically, the ABA_A_controller class has two instance variables named model and view, as shown in Listing 15.2. These two instance variables contain object references to the ABA_A_model and ABA_A_view objects, respectively. Thus, the controller uses the model variable to call public methods defined in the ABA_A_model class and the view variable to call public methods defined in the ABA_A_view class.

The communication diagram in Fig. 15.3 shows the sequence of steps and the interaction of the MVC components as the user is creating contact information. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version A solution obtains contact information from the user.

Step 2: The main method calls the go method using the controller object.
Step 2.1: The go method obtains a valid contact name. This involves calling the private getValidName method, which will call view.getName() as many times as necessary to ensure a valid contact name is entered. As shown in Listing 15.5,

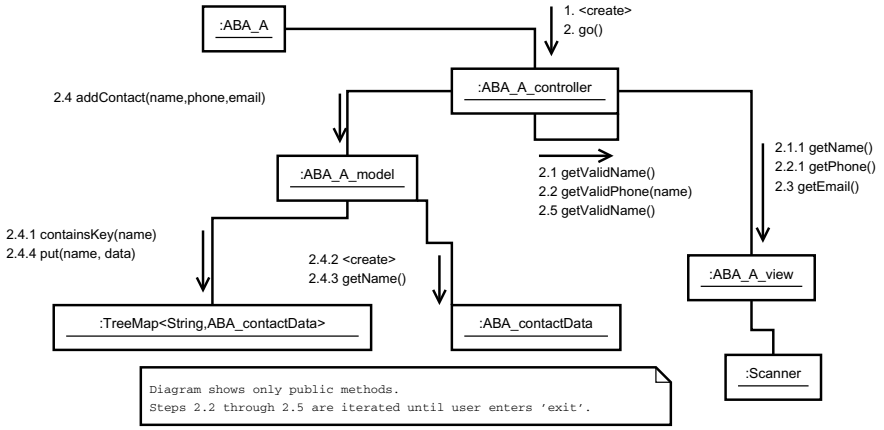


Fig. 15.3 Communication diagram—MVC design Version A

the special value "exit" is a valid contact name which allows the user to indicate they are done entering contact information. Steps 2.2 through 2.5 are performed as long as "exit" is not entered.

Step 2.2: The go method obtains a valid phone number. This involves calling the private getValidPhone method, which will call view.getPhone() as many times as necessary to ensure a valid phone number is entered.

Step 2.3: The go method obtains an email address. This involves calling view.getEmail() only once since there is no requirement to validate email addresses.[2]

Step 2.4: The go method tells the model component to store the contact information. This involves calling the model.addContact method to store the contact data in the book data structure.

Step 2.4.1: The addContact method first determines whether the contact name already exists in the book. When the contact name already exists in the data structure, the remaining 2.4.x steps are *not* performed.

Step 2.4.2: An ABA_contactData object is created.

Step 2.4.3: The contact name is retrieved from the ABA_contactData object.

Step 2.4.4: The name and ABA_contactData object are added to the book container.

[2]We'll discuss security more deeply in Chaps. 24 through 26. At this point in the book, our ABA case study is purposely small and simple so that we can focus on software design decisions and how they impact characteristics of a good software design. In general, software developers should ensure all data provided by an external entity is validated. Specific to the ABA, any serious contact management software should ensure that email addresses are valid and do not contain data that may lead to malicious behavior.

Step 2.5: The go method obtains a valid contact name. This involves calling view.getName() as many times as necessary to ensure a valid contact name is entered.

The code for the ABA_A_controller go method is shown in Listing 15.5.

Listing 15.5 ABA MVC Design version A—go() method

```
public void go() {
    String name, phone, email;
    name = getValidName();

    while (! name.equals("exit"))
    {
        phone = getValidPhone(name);
        email = view.getEmail(name);
        model.addContact(name, phone, email);
        name = getValidName();
    }
    displayBook();
}
```

The communication diagram in Fig. 15.4 shows what happens after the user enters "exit" to end the inputting of contact data. Specifically, the contents of the book are displayed. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version A solution displays contact data to the user.

Step 2.6: The go method calls the controller's displayBook method.
Steps 2.6.1 and 2.6.2: The displayBook method displays two messages by calling the view.displayMsg() method twice.

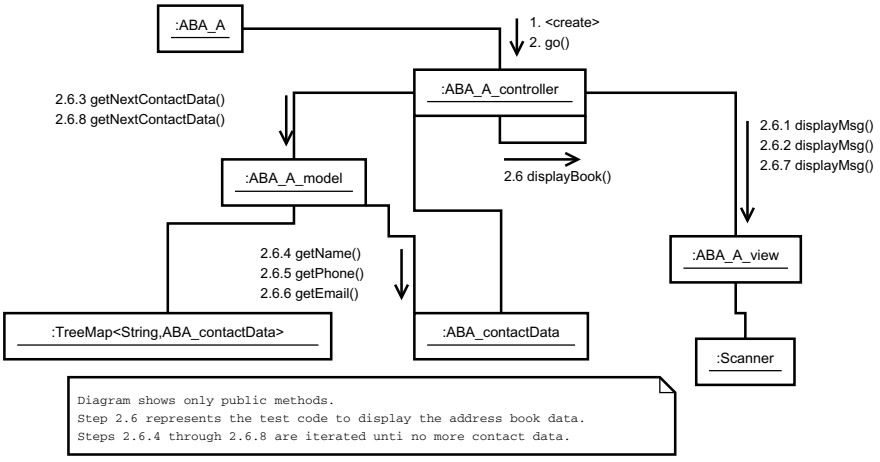


Fig. 15.4 Communication diagram—MVC design Version A—display book

Step 2.6.3: The displayBook method obtains the first contact data. This involves calling the model.getNextContactData method.

Step 2.6.4 through 2.6.6: The displayBook method obtains the contact's name, phone, and email. This involves calling the getName, getPhone, and getEmail methods in the ABA_contactData class.

Step 2.6.7: The displayBook method displays the contact data by calling the view.displayMsg method.

Step 2.6.8: The displayBook method obtains the next contact data. This involves calling the model.getNextContactData method.

15.3.3 OOD: ABA MVC Version A Software Design

The first ABA design using Model–View–Controller has five classes, as shown in Fig. 15.5. This diagram shows a class that contains the static main method, a class for each of the three MVC components, and a class (ABA_contactData) that contains

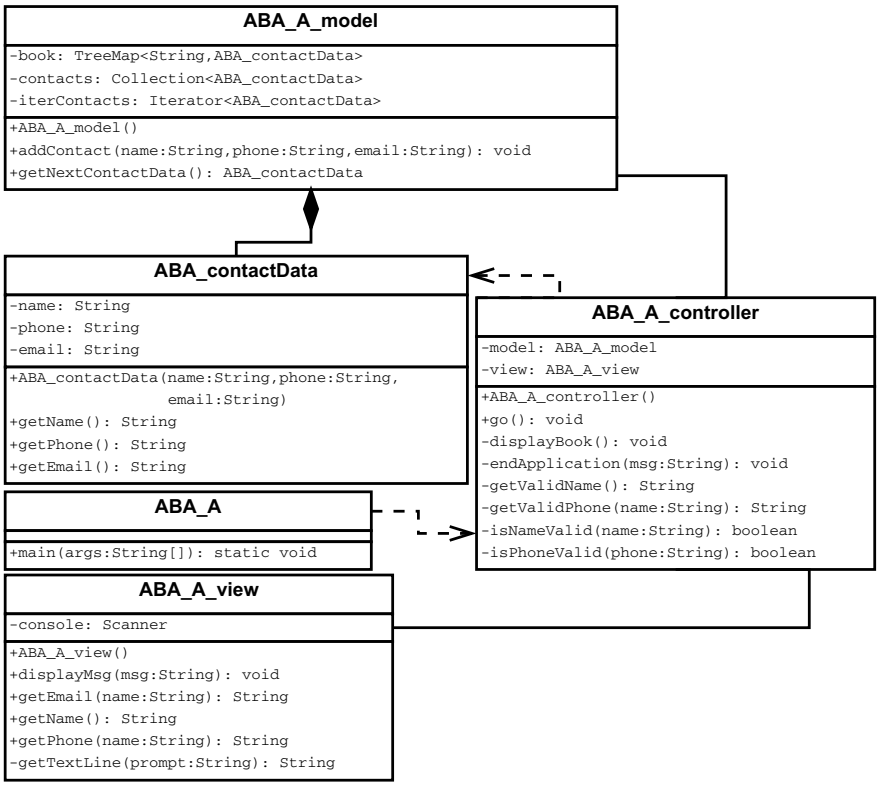


Fig. 15.5 Class diagram—MVC design Version A

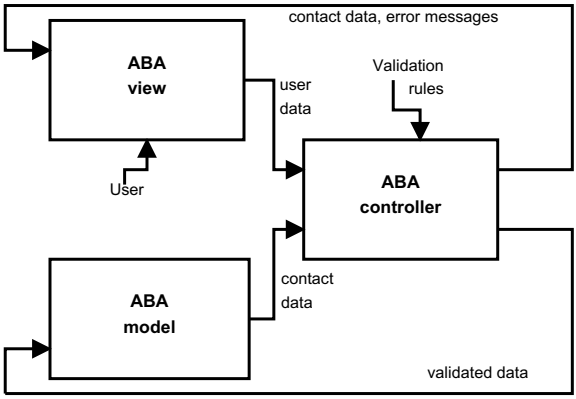


Fig. 15.6 IDEF0 function model—MVC design Version A

contact data for a person. While the controller class has a relationship with the model and view classes, the view and model classes have no relationship with each other. As described above, the model component consists of two classes (ABA_A_model and ABA_contactData), the view component is a single class (ABA_A_view), and the controller is two classes (ABA_A and ABA_A_controller).

The IDEF0 function model shown in Fig. 15.6 describes the primary flows of data between the three ABA components. It also shows the User interacting only with the ABA view component while the ABA controller component performs validation of the data.

The way the user interacts with the ABA Version A design is shown in Fig. 15.7. The user first must enter a valid name, or "exit" to display the address book and end the ABA. Once a valid name is entered, a valid phone number must be entered. Once a valid phone number is entered, any value is accepted for an email address. Only when the name is *not* already in the address book is this contact data added to the address book.

15.3.4 OOD: Evaluate ABA MVC Version A Software Design

The object-oriented design just described will now be evaluated using the six design criteria introduced in Chap. 11.

15.3.4.1 Simplicity

The controller component has two classes. The ABA_A class contains the main method that allows the ABA to execute. This results in the ABA_A class being dependent on the ABA_A_controller class, as described in Sect. 15.3.1. The ABA MVC Version A design uses the controller for any communication between the model and view components. This results in the ABA_A_controller class having an association relationship with the ABA_A_model and ABA_A_view classes. Finally,

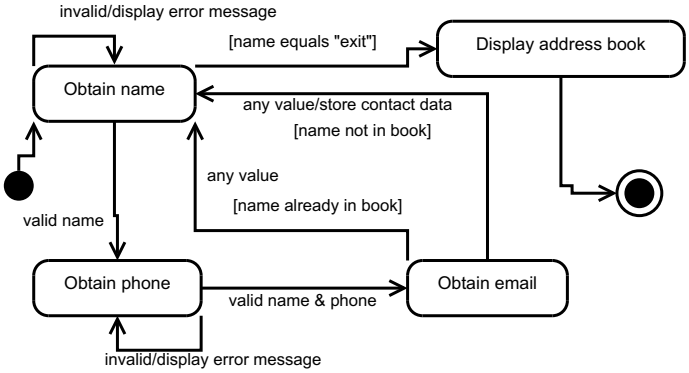


Fig. 15.7 State diagram—MVC design Version A

the controller component displays the contents of the address book via its private displaybook method. This method obtains ABA_contactData objects from the model component, resulting in the controller being dependent on the ABA_contactData class.

The model component has two classes. The ABA_A_model class represents the address book container via its TreeMap data structure. Since the TreeMap contains ABA_contactData objects, and these objects must be in the TreeMap in order to exist, a composition relationship exists between ABA_contactData and ABA_A_model classes.

The view component is a single class used to obtain data from the user and display information to the user.

There are a total of 5 classes, 9 instance variables, and 22 methods in the ABA MVC Version A solution. Of these, 7 instance variables and 20 methods are associated with implementing the seven requirements (1a, 1b, and 2 through 6) shown in Table 15.1. Only 2 instance variables and 2 methods are used as test code to verify the contents of the address book. This seems like a reasonable number of classes, instance variables, and methods for the number of requirements being implemented. The design of ABA_A appears to be simple to understand.

15.3.4.2 Coupling

The Model–View–Controller components described in the ABA MVC Version A design represent three distinct modules. The question is, to what degree does one of these modules rely on the other modules? We can address this question by considering four perspectives on how coupling between two modules can be expressed.

1. Look at the methods within a module that are called by other modules. Does the number of method calls seem excessive?

We can identify the public methods called by other components/modules by looking at the communication diagrams in Figs. 15.2, 15.3, and 15.4, and the class diagram in Fig. 15.5.

Controller public methods: The two public methods in the ABA_A_controller class are called by the main method. The other two components/modules do not call any controller public methods.

Model public methods: Two of the three public methods in the ABA_A_model class are called by the controller component/module to store contact data. The other public method in the ABA_A_model class and three of the four public methods in the ABA_A_contactData class are called by the controller component/module as part of the test code to verify the contents of the address book. The fourth public method in the ABA_A_contactData class is called by the ABA_A_model class to construct a contactData object. None of the model methods are called by the view component/module.

View public methods: All of the public methods are called by the controller. None of these methods are called by the model component.

Looking at coupling from the perspective of method calls shows no coupling between the model and view. The coupling between the controller and model is unidirectional; the controller calls methods within the model to either store or retrieve address book data. The coupling between the controller and view is also unidirectional, the controller calls methods within the view to obtain data from the user. The methods defined with the private access modifier can only be called from code within the same class. This ensures that no (method call) coupling exists between these private methods and another component.

2. Look at the data that is passed via each method call. Does the amount of data being passed between components via method calls seem too excessive or too complex? Looking at the class diagram in Fig. 15.5.

Controller public methods: The two methods in the ABA_A_controller class have no parameters. Since the main method is required to have a String[] argument, it is not considered relevant in terms of assessing coupling based on parameter passing.

Model public methods: Two of these methods have parameters. The addContact method has three parameters to pass the name, phone number, and email address for the address book entry to be stored. The constructor method ABA_contactData is passed the three data values to create an object that stores all of this data.

View public methods: Three of these methods have a parameter. The displayMsg function has a parameter to pass the message to be displayed to the user. The getEmail and getPhone methods have a name parameter so the view can ask for data for a specific contact person.

A parameter passing perspective shows very little coupling between the controller and model and between the controller and view. However, there is one method call from the controller to the view that does increase the coupling between these two components. The controller passes a specific error message when it calls the displayMsg method, which results in this message being displayed to the user. This error message is part of the user interface but is known by the controller component, increasing the coupling between these two components.

3. Look at the data that is returned via each method call. Does the amount of data being returned to another component seem too excessive or too complex?
   Looking at the class diagram in Fig. 15.5.

   Controller public methods: None of these methods return a data value.
   Model public methods: Four of these methods return a data value. The getName, getPhone, and getEmail methods return data values representing contact data while the getNextContactData method returns an ABA_contactData object.
   View public methods: Three of these methods return a String data value. The getEmail, getName, and getPhone methods return the email address, name, or phone number, respectively, entered by the user.

   A method return perspective shows no coupling between the model and view modules, coupling between the controller and model is unidirectional (model methods return data values to the controller but not vice versa), and coupling between the controller and view is also unidirectional (view methods return data values to the controller).
4. Look at the use of global data structures. Does one module define a global data structure that is referenced by another module?
   As seen in the class diagram in Fig. 15.5, all of the instance variables specified in each class are defined using the private access modifier (i.e., the dash in front of each attribute name indicates private). This ensures these data structures are accessible only by the methods specified within the class. The ABA_contactData class is constructed and stored within the model and given to the controller as the return value when the getNextContactData method is called. This class/object is used to give all of the contact data to the controller by encapsulating all of this data into a single class.

Given the above descriptions on coupling between the model, view, and controller components, the amount of coupling seems minimal and thus represents a good design.

15.3.4.3 Cohesion

The MVC components described in the design of ABA MVC Version A represent three distinct modules. The question is, to what degree does each of these modules contain methods that are strongly-related to each other and are single-minded in their purpose? We'll look at the purpose of each method and instance variable within the same module, and whether (or how much) these are related to each other.

Controller methods: With two exceptions, the controller methods either provide the processing flow of the application or implement data validation requirements. This matches the purpose of the controller component. The exception is the displayBook method, which is part of the test code to verify the contents of the address book.

The application processing flow implemented in the controller determines when the user has requested the application end by checking the name value entered to see if it matches "exit". This behavior requires the controller know how the user requests the application to end. This knowledge should reside in the view. One method within the controller is part of the test code that displays the contents of the address book. This method acts as an intermediary between the view—responsible for displaying address book data—and the model—which is responsible for storing this data.

The validation logic implemented in the controller includes the getValidName and getValidPhone methods passing a validation error message to the view for display to the user. This makes these two methods less cohesive with the rest of the controller since the controller is not responsible for communication with the user. The values passed to the validation methods and the method return values are all cohesive with the purpose of validating contact data.

Controller instance variables: The two private attributes allow the controller to call public methods in one of the other components.

Model methods: All of the model methods are designed to either store or retrieve address book data. This matches the purpose of the model component. The data used by these methods, whether this data comes from parameters or instance variables, all have to do with contact data. Likewise, the return values from these methods also have to do with contact data.

Model instance variables: The six private attributes in the model are all related to address book data. In particular, the *contacts: Collection* and *iterContacts: Iterator* instance variables are used by the test code to retrieve the address book data one entry at a time. Besides the constructor method, all the methods in the ABA_contactData class are *getter* methods. That is, once an ABA_contactData is constructed, the only thing the object allows is retrieval of its data.

View methods: All of the view methods are used to display information to and obtain data from the user. This matches the purpose of the view component. The data used by these methods, whether this data comes from parameters or instance variables, all have to do with the user interface. Likewise, the return values from these methods also have to do with data from the user.

View instance variables: The private attribute in the view allows this component to obtain data from the user.

The model component is highly cohesive, whereas the controller and view modules are less cohesive. As mentioned above, the controller validation logic gives the view an error message to be displayed, making the controller less cohesive. For the view, one could argue that the single class in the view is doing two distinctly different things—it is obtaining data from the user and it is displaying data to the user. Since the ABA is using a text-based user interface, the mechanisms used to obtain data (i.e., Scanner object) are different from the mechanisms to display data (i.e., System.out).

15.3.4.4 Information Hiding

Does each module in the MVC design Version A hide its implementation details from the other components? By implementation details, we mean the algorithms and data structures that are being used by the module.

Controller: The controller contains methods that either provide the processing flow of the application or implement the data validation requirements. None of the other components know how the controller performs validation.

Model: The model stores the address book data and determines if a name is already found in the address book. These data structures and algorithms are not known by the other modules.

The model includes a class—ABA_contactData—that is used to store one instance of contact data. Instances of this class are used by the model component and also passed to the controller component. Besides the constructor method, all the methods in the ABA_contactData class are *getter* methods. Since the MVC Version A implementation includes test code to display the contents of the address book, this sharing of contact data between model and controller is necessary since the controller must give the view the actual data to be displayed to the user.

View: The view displays information to and obtains data from the user. The way in which the view accomplishes this is not known by the other modules.

One other perspective on information hiding is to look to see how the interface to each component is defined. In this design, the list of public methods found in each class represents the interface to the component. The list of public methods is fairly typical given the list of requirements that we are addressing in our design. More about interfaces is discussed below.

15.3.4.5 Performance

The performance of the MVC Version A design is similar to the performance of the ABA solutions described in Chap. 6. Specifically, the use of a Java TreeMap gives us a worst case of $O(log_2 n)$ performance.

15.3.4.6 Security

The security of the MVC Version A design is similar to the security of the ABA solutions described in Chap. 9. Specifically, the design includes data input validation, data output validation, exception handling, and fail-safe defaults. In addition, Java is a type-safe language.

15.4 OOD: ABA MVC Design Version B

The design of MVC Version A of the ABA is okay, but can be improved. First, the coupling between the controller and view is too high. This is due to the (relatively) large number of view methods called by the controller, which results from the view methods being too specific regarding the input of user data. Fortunately, we already have the ABA_contactData class that contains all of the data for a single contact person. Why not have the view construct one of these objects each time it gets data from the user? The view can then give this object to the controller, which can then process this contact data.

Second, the cohesion of the controller can be improved. Specifically, the logic that compares the contact name to "exit" should really be in the user interface (view) component. To make the controller more cohesive, any user interface logic should be removed; the controller should only be responsible for the execution flow of the application and data validation.

The Version B design for the ABA using Model–View–Controller has seven classes, as shown in the package diagram in Fig. 15.8. The model and controller components have the same classes as the Version A design. The view component now has three classes: ABA_B_view, ABA_B_viewDisplayData, and ABA_B_viewObtainData.

The changes to the Version B package diagram, when compared to Version A, include the following: an association between the ABA_B_view and the two

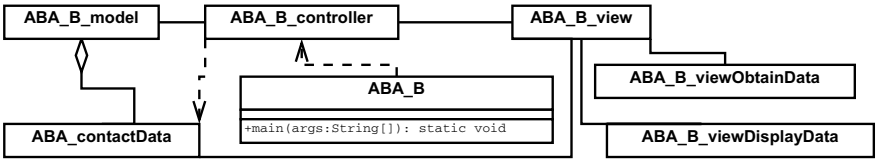


Fig. 15.8 Package diagram for MVC design ABA Version B

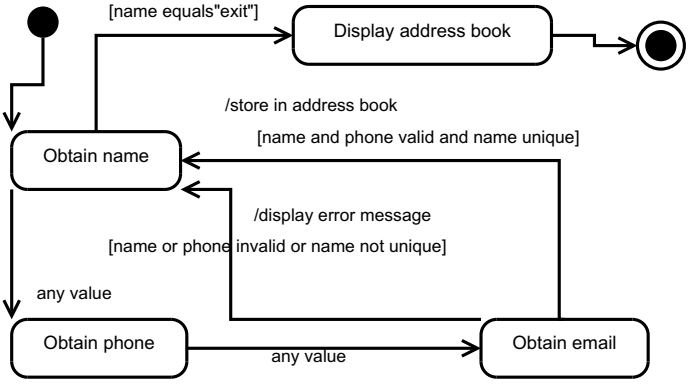


Fig. 15.9 State diagram—MVC design Version B

new classes—ABA_B_viewDisplayData and ABA_B_viewObtainData classes, an association between the ABA_B_view and ABA_contactData classes, and the composition relationship between ABA_B_model and ABA_B_contactData is now an aggregation relationship.

The association between the ABA_B_view and ABA_contactData classes is based on the view component creating an ABA_B_contactData object after obtaining the name, phone number, and email address from the user. The view returns this object to the controller, which will then validate the contact data. The fact that the view component creates an ABA_B_contactData object, which if valid, will be stored by the model in the TreeMap results in the relationship between the ABA_B_model and ABA_B_contactData being changed to an aggregation since the objects exist outside of the data structure.

To address the design flaws described above in Version A, the way the user interacts with the ABA has been changed in Version B. Figure 15.6 shows the IDEF0 function model for Version A. Since this is a high-level description of the Version A design, this function model also applies to Version B. However, the statechart in Fig. 15.9 illustrates significant changes to how the user interacts with the Version B ABA. The user enters a value for contact name, phone number, and email address. After entering these three values, a contact name value of "exit" causes the address book data to be displayed and the ABA ends. Otherwise, the contact name and phone number are validated. When these two values are valid, the contact data is added to the address book when the contact name is not already in the book. When either or both values are invalid, appropriate error messages are displayed. In either case, the user is prompted to enter the next contact person's name.

The second ABA design using Model–View–Controller has seven classes, as shown in Fig. 15.10. As described above for the Version B package diagram, the view component has two additional classes (ABA_B_viewDisplayData and ABA_B_viewObtainData) and creates ABA_contactData objects that are then stored in a TreeMap by the model component.

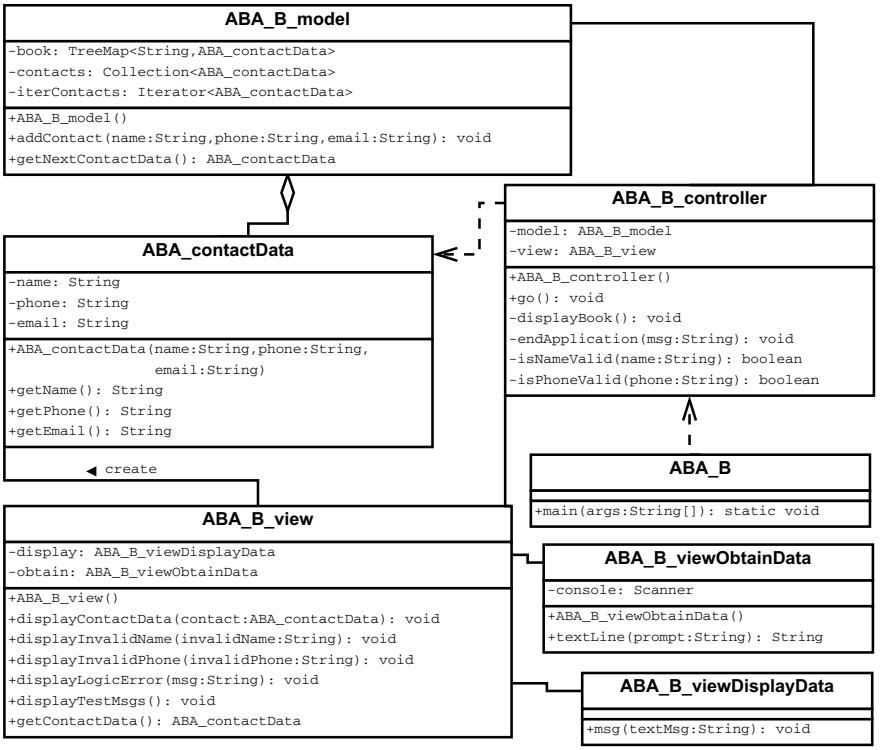


Fig. 15.10 Class diagram—MVC design Version B

15.4.1 OOD: In What Order Do MVC Objects Get Created?

The communication diagram in Fig. 15.11 shows the sequence of steps and the interaction of the MVC components in starting the ABA. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version B solution starts to execute.

Step 1: Same as MVC Version A design.
Step 1.1: Same as MVC Version A design.
Step 1.1.1: Same as MVC Version A design.
Step 1.2: Same as MVC Version A design.
Step 1.2.1: The ABA_A_view constructor method creates an ABA_A_viewDisplayData object used to display data to the user.

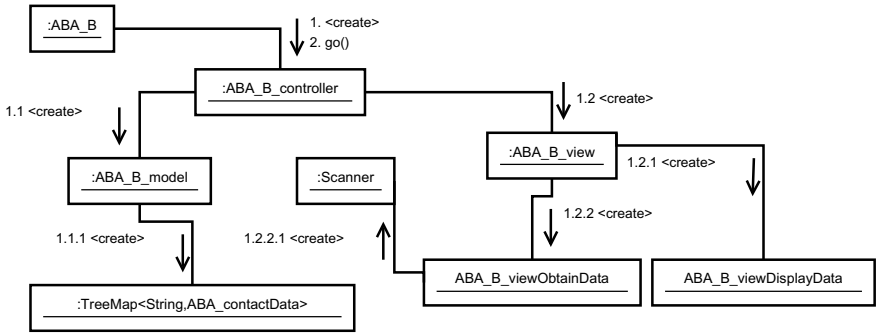


Fig. 15.11 Communication diagram—MVC design Version B—Start up

Step 1.2.2: The ABA_A_view constructor method creates an ABA_A_viewObtain Data object used to obtain user data.
Step 1.2.2.1: The ABA_A_viewObtainData constructor method creates a Scanner object used to obtain user data.
Step 2: Same as MVC Version A design.

The code for the ABA_A_view constructor method is shown in Listing 15.6.

Listing 15.6 ABA MVC Design version B—view Startup Code

```
public class ABA_B_view {
    private ABA_B_viewDisplayData display;
    private ABA_B_viewObtainData obtain;
    private final String EXIT = "exit";

    public ABA_B_view()
    {
        this.display = new ABA_B_viewDisplayData();
        this.obtain = new ABA_B_viewObtainData();
    }
```

15.4.2 OOD: How Do MVC Objects Communicate with Each Other?

The communication diagram in Fig. 15.12 shows the sequence of steps and the interaction of the MVC components as the user is creating contact information. The steps shown in this diagram are summarized below to reinforce the interactions between the MVC objects as the ABA Version B solution obtains contact information from the user.

Step 2: Same as MVC Version A design.
Step 2.1: The go method obtains contact data. This involves calling the public view.getContactData method, which will call the textLine method three times to

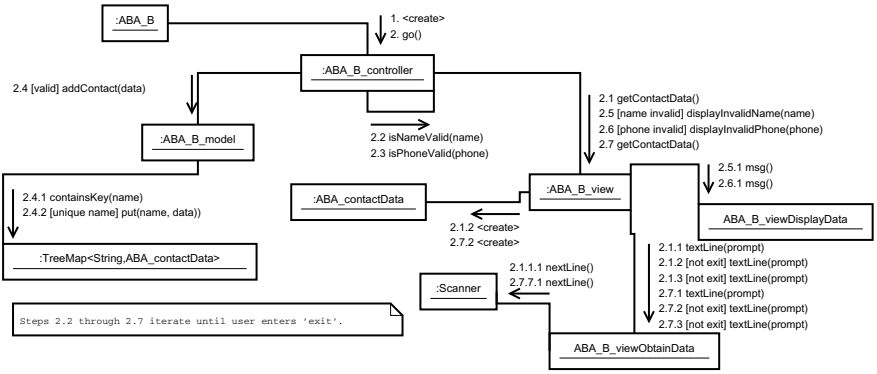


Fig. 15.12 Communication diagram—MVC design Version B

obtain a name, phone, and email address. As shown in Listing 15.7, the go method now iterates until the view component returns null instead of an ABA_contactData object. Listing 15.8 shows how the view component uses the special value "exit" to determine if the user is done entering contact data. Steps 2.2 through 2.7 are performed as long as the view does not return a null to the go method.

Step 2.2: The go method validates the contact name. This involves calling the private isNameValid method.

Step 2.3: The go method validates the phone number. This involves calling the private isPhoneValid method.

Step 2.4: When the contact data is valid, the go method tells the model component to store the contact information. This involves calling the model.addContact method to store the contact data in the book data structure.

Step 2.4.1: The addContact method first determines whether the contact name already exists in the book. When the contact name already exists in the data structure, Step 2.4.2 is *not* performed.

Step 2.4.2: The name and ABA_contactData object are added to the book container.

Step 2.5: When the contact name is invalid, the go method tells the view component to display an appropriate error message.

Step 2.6: When the phone number is invalid, the go method tells the view component to display an appropriate error message.

Step 2.7: The go method obtains contact data, as described by Step 2.1.

The code for the go method found in the ABA_A_controller class is shown in Listing 15.7. Of significance is the iteration to obtain contact data that stops when the view component returns null instead of an ABA_contactData object.

Listing 15.7 ABA MVC Design version B—go() method

```
public void go() {
    ABA_contactData data;
    boolean nameValid, phoneValid;
    data = view.getContactData();

    while (data != null)
    {
        nameValid = isNameValid(data.getName());
        phoneValid = isPhoneValid(data.getPhone());
        if (nameValid && phoneValid)
            model.addContact(data);
        else
        {
            if (! nameValid)
                view.displayInvalidName(data.getName());
            if (! phoneValid)
                view.displayInvalidPhone(data.getPhone());
        }
        data = view.getContactData();
    }
    displayBook();
}
```

The code for the getContactData method found in the ABA_A_view class is shown in Listing 15.8. This illustrates two important design choices: the view component is the only component that knows how the user enters contact data; and this method detects when the user is done entering data, as shown in the selection logic in the first if statement.

Listing 15.8 ABA MVC Design version B—getContactData() method

```
public ABA_contactData getContactData() {
    ABA_contactData userData;
    String name, phone, email;
    name = obtain.textLine("Enter contact name ('exit' to quit): ");
    if (name.equals(EXIT))
        userData = null;
    else
    {
        phone = obtain.textLine("Enter phone number for "+name+": ");
        email = obtain.textLine("Enter email address for "+name+": ");
        if (phone.equals(EXIT) || email.equals(EXIT))
            //One of these values may be EXIT if user enters CTRL-Z.
            userData = null;
        else
            userData = new ABA_contactData(name, phone, email);
    }
    return userData;
}
```

15.4.3 OOD: Evaluate ABA MVC Version B Software Design

The object-oriented design just described will now be evaluated using the six design criteria introduced in Chap. 11.

15.4.3.1 Simplicity

The design changes to Version B results in two new classes to the view component and an association relationship between the ABA_B_view and ABA_contactData classes.

There are a total of 7 classes, 11 instance variables, and 23 methods in the ABA MVC Version B solution. Of these, 7 instance variables and 20 methods are associated with implementing the seven requirements (1a, 1b, and 2 through 6) shown in Table 15.1. Only 2 instance variables and 3 methods are used as test code to verify the contents of the address book.

The changes made to improve the Version B design seem like a reasonable number of classes, instance variables, and methods for the number of requirements being implemented. The design of ABA_B appears to be simple to understand.

15.4.3.2 Coupling

The design changes to Version B adds an association relationship between the ABA_B_view and ABA_contactData classes, which reside in the view and model modules, respectively. We'll address this additional coupling between these two modules using the same four perspectives discussed in Sect. 15.3.4.

1. Look at the methods within a module that are called by other modules. Does the number of method calls seem excessive?
   The ABA_B_view class constructs an ABA_contactData object after obtaining the appropriate contact data from the user. This method call results in a coupling between the view and model components which did not exist in Version A.
2. Look at the data that is passed via each method call. Does the amount of data being passed between components via method calls seem too excessive or too complex?
   The construction of an ABA_contactData object results in the view module passing three data values representing the contact data entered by the user.
   The other significant change in Version B is the elimination of the generic displayMsg(String) public method and the addition of five public methods in the ABA_view class. In Version A, the controller would construct a String object representing the data or message to be displayed to the user. The controller would then call the displayMsg(String) method to display the data/message. This results in the Version A controller having significant knowledge regarding the format and content of what is being displayed to the user. In Version B, the five new public methods in the view component are designed for a single purpose. The displayContactdata(ABA_contactData), displayInvalidNameString, displayInvalid-

Phone(String), and displayLogicError(String) methods will format a message using the data contained in the formal parameter. The displayTestMsg() method has no formal parameters, resulting in this view method being completely responsible for formatting and displaying appropriate test messages. To conclude, the Version B view module now has *display* methods that are designed for specific purposes. They receive data from the controller and decide how to format this data for display to the user. Given this, we can say that the Version B controller has less information about how information is displayed to the user, reducing the coupling between these two modules.

3. Look at the data that is returned via each method call. Does the amount of data being returned to another component seem too excessive or too complex?
   The construction of an ABA_contactData object results in the view module having an object that was constructed by the model component.
4. Look at the use of global data structures. Does one module define a global data structure that is referenced by another module?
   No global data structures were added to the Version B design.

Even with the new association relationship between the view and model components, the amount of coupling in Version B is minimal and thus represents a good design.

15.4.3.3 Cohesion

Each of the MVC components is described below by identifying the changes in Version B and how these affect cohesion of the module.

Controller methods: Two methods in Version A are no longer in the ABA_B_controller class. Eliminating these two methods in Version B has no impact on the cohesiveness of the controller.
The application processing flow implemented in the Version B controller determines when the user has requested the application to end by testing the view.getContactData() method's return value. When this method returns null, the controller will end the application. Knowledge about how the user indicates an end to the ABA is now solely contained in the view component.
The validation logic implemented in the Version B controller now passes an invalid name or invalid phone number to the view module. The view is then responsible for displaying an appropriate error message to indicate the data value is invalid. In Version B, the controller is only responsible for performing validation logic; displaying an appropriate validation error message is now in the view (where it should be!).

Controller instance variables: Same as MVC Version A design.

Model methods: Same as MVC Version A design.

Model instance variables: Same as MVC Version A design.

View methods: The view component now has three classes. The public methods in the ABA_B_view class represent the interface used by the controller. While

the Version A view component had one public method responsible for displaying any data/message to the user, Version B has five public methods used to display data or messages. Each of these methods is responsible for a specific interaction with the user. The public methods in the two new classes are called only by the ABA_B_view class.

View instance variables: The private attributes in the three view classes are directly related to the purpose of this component.

The Version B controller component is more cohesive than Version A, largely due to the changes in how the controller interfaces with the view. The view component, even with its three classes, is more cohesive as it contains all of the knowledge about how the user interface is implemented.

15.4.3.4 Information Hiding

Each of the MVC components is described below by identifying the changes in Version B and how these affect information hiding within each module.

Controller: Same as MVC Version A design.

Model: The model continues to store ABA_contactData objects and determines if a name is already found in the address book. However, the Version B view component creates ABA_contactData objects and gives these to the controller for validation. This design change has a minor impact on information hiding since the view component now understands how to create an ABA_contactData object.

View: Same as MVC Version A design.

15.4.3.5 Performance

The performance of the MVC Version B design is similar to the performance of the ABA solutions described in Chap. 6. Specifically, the use of a Java TreeMap gives us a worst case of $O(log_2 n)$ performance.

15.4.3.6 Security

The security of the MVC Version B design is similar to the security of the ABA solutions described in Chap. 9. Specifically, the design includes data input validation, data output validation, exception handling, and fail-safe defaults. In addition, Java is a type-safe language.

15.5 OOD Top-Down Design Perspective

We'll continue to use the personal finances case study to reinforce Model–View–Controller via a top-down design approach.

15.5.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

Below, we will compare our top-down MVC software design with Chap. 12 designs, which focused on the domain of personal finances and ignored implementation details related to user interface and persistent storage.

15.5.2 OOD Personal Finances: Structure

The package diagram in Fig. 15.13 shows three named packages reflecting the use of Model–View–Controller. Comparing this with the package diagram in Fig. 12.8, we see domain classes identified in Chap. 12 in both the model and view packages in Fig. 15.13. Given the responsibility of the model—to store domain data—the model needs to understand the structure and relationships of the domain data. Likewise, the responsibility of the view—to provide interaction with a user—also suggests this package understands the types of data to be obtained from or displayed to a user.

One other item to note is the presence of the controllerValidate class in the controller package. While no requirements have been stated for validation of personal finance data, designers and programmers should validate data that comes from any external source. In the absence of requirements, developers should identify validation requirements based on known threats and common sense. Put another way, if a vulnerability is discovered in the software that is traced back to not doing data validation, is someone going to point to the lack of requirements as the source of this vulnerability? Likely not; the source of the vulnerability will be the developers that did not design, program, and test to ensure validation is done appropriately.

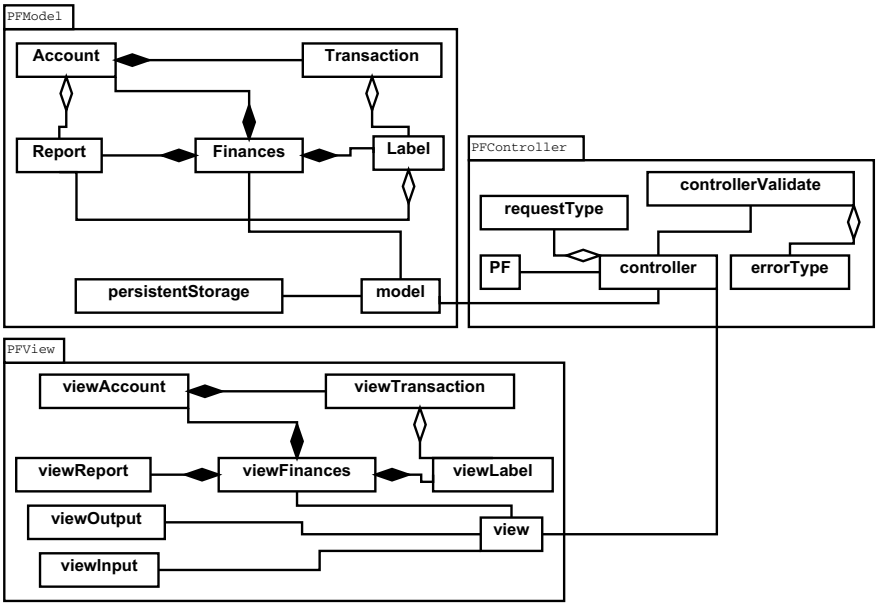


Fig. 15.13 Personal finance package diagram

Three class diagrams are shown below, one for each package. Figures 15.14, 15.15, and 15.16 show the attributes and public methods for each class in the model, view, and controller packages, respectively. These class diagrams are at a high level of abstraction; more work would be needed to translate these conceptual models into a detailed design or into an implementation. Examples of this abstraction include the following.

- Only one class identifies private methods (see the controllerValidate class). A more detailed design would likely show private methods in just about every class. As you know, private methods are used to implement common logic or detailed processing found within a single class.
- There are very few classes that have *getter* and *setter* methods. For example, the *Transaction* class will need to have a method that returns each attribute value (e.g., getDate(), getDescription(), getAmount() and getNumber()), and a method that sets each attribute value (e.g., setDate(date:Date), setDescription(desc:String), setAmount(amount:Currency), and setNumber(number:String)). The getter methods would be needed when generating a report and the setter methods would be needed when a transaction is modified.
- Some model classes use the generic term *container* to represent a data structure that will need to be used to store multiple data values.
- The model class persistentStorage is vague about the type of persistent storage technology used in the application. A generic PersistenceType represents the per-

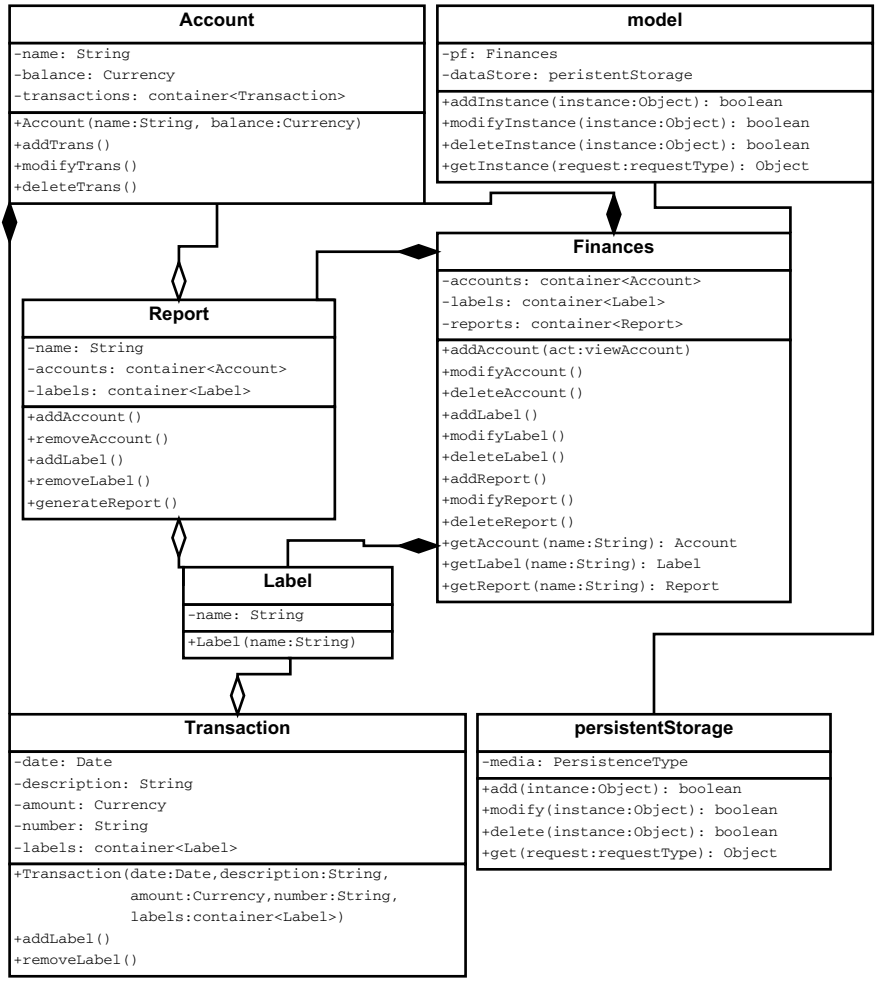


Fig. 15.14 Personal finance class diagram—model component

sistent data store, without having to specify, at this point in the design process, the type of data storage technology to be used. Once the persistent storage technology is selected, additional *helper classes* will likely be used to manage the details associated with the technology.

- The view classes representing the personal finance domain—viewAccount, viewFinances, viewLabel, viewReport, and viewTransaction—may be responsible for the user interface associated with each type of data. If this decision were made, then using these same classes as container objects passed to the controller would be a bad idea—a design should not allow the controller component to directly access user interface elements.

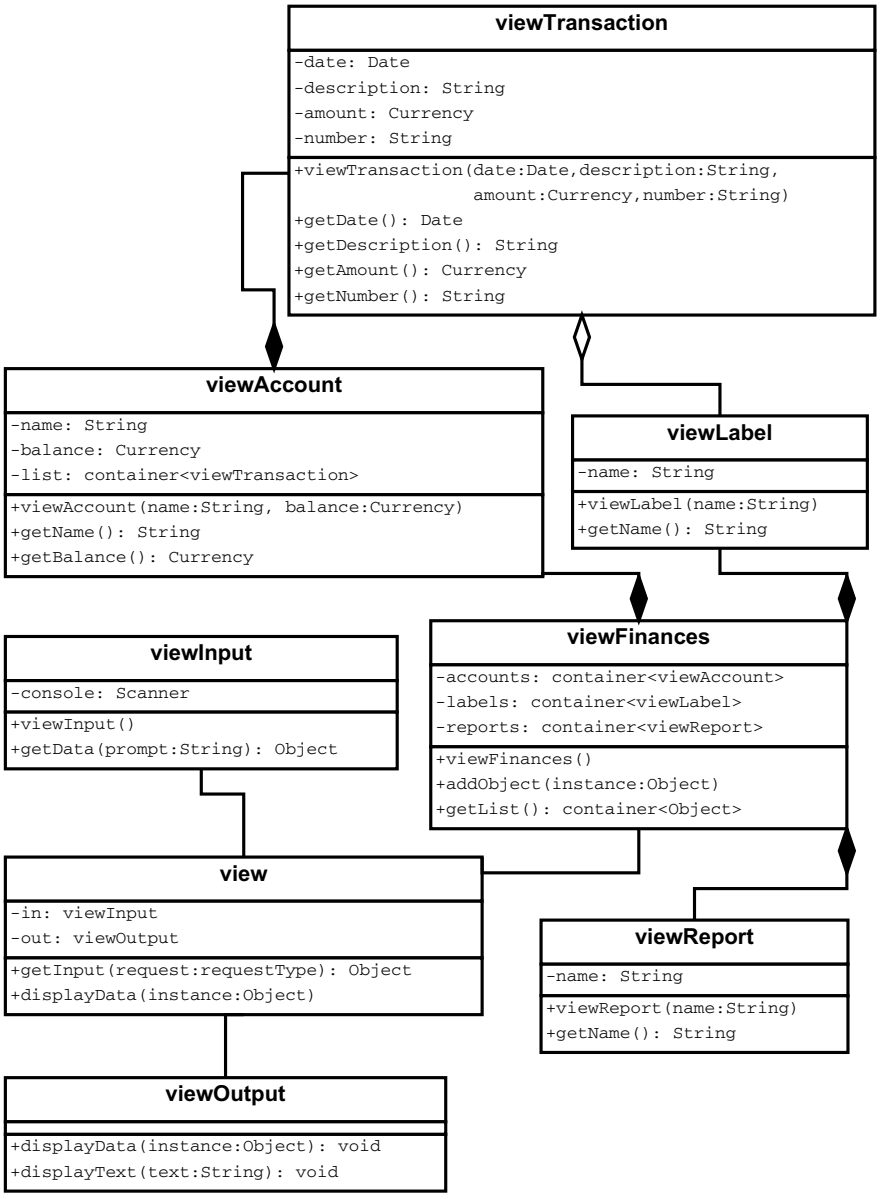


Fig. 15.15 Personal finance class diagram—view component

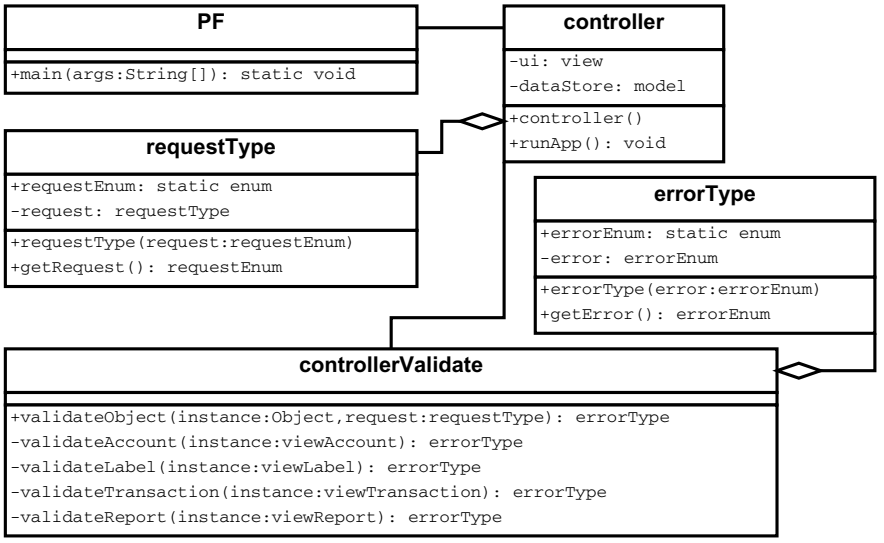


Fig. 15.16 Personal finance class diagram—controller component

- A more detailed design of the controller package would likely include additional *helper classes* to support the overall application flow of the personal finances software.
- The controller class would likely have many private methods that provide the detailed processing in support of the runApp method.

15.5.3 OOD Personal Finances: Structure and Behavior

The Data-Flow Diagram (DFD) used in Chap. 12 is shown in Fig. 15.17. This diagram is already at an appropriate level of abstraction. Modifying this diagram to show Model–View–Controller would not add any information that is not already being expressed in the package and class diagrams.

The UML communication diagram used in Chap. 12 is shown in Fig. 15.18. As you may recall, this diagram shows the basic interactions between the objects based on an understanding of the personal finance domain. In contrast, the UML communication diagram in Fig. 15.19 describes the object interactions associated with using Model–View–Controller. This communication diagram shows the interactions between objects to create a new account.

Figure 15.19 shows two messages labeled *1.1: getData(prompt)* and *1.2: getData(prompt)*. These result in the user entering the account name and balance. This user-supplied data is then used to create a viewAccount object (see Step 1.3), which is then given to the controller. Assuming the viewAccount object contains a valid account name and balance, the controller gives this object to the model via the *3: addInstance(viewAccount)* message. Within the model component, the viewAccount

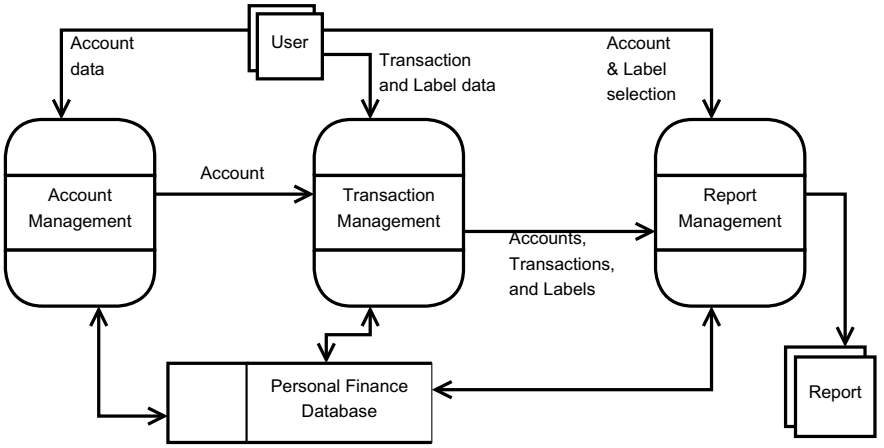


Fig. 15.17 Personal finance data-flow diagram (from Chap. 12)

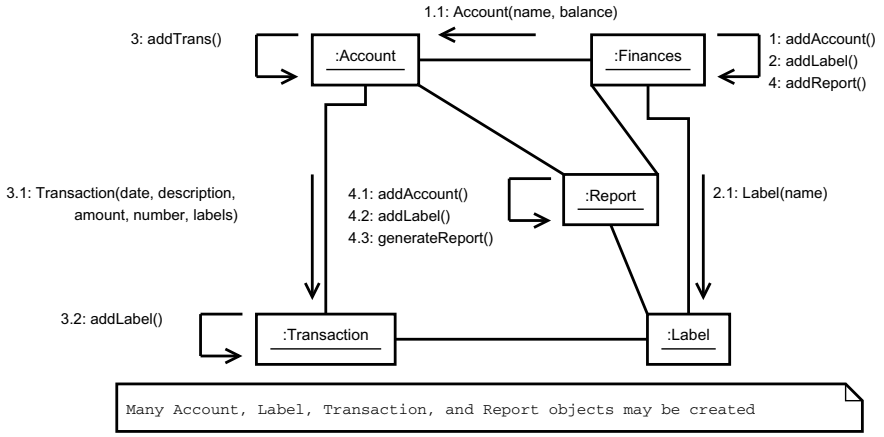


Fig. 15.18 Personal finance communication diagram (from Chap. 12)

object is used to add an account to the Finances container object via the *3.1: addAccount(viewAccount)* and *3.1.1: «create»* messages. After this, the account is added to persistentStorage via the message labeled 3.3.

15.5.4 OOD Personal Finances: Behavior

The UML statechart diagram described in Chap. 12 shows the user interactions with the personal finance application. Since changing the software design to use Model–View–Controller has no impact on the user interface design, there is no need to modify the statechart diagram. That is, the user interface behavior has not been changed even though the internal structure of the application code has gone through a dramatic change!

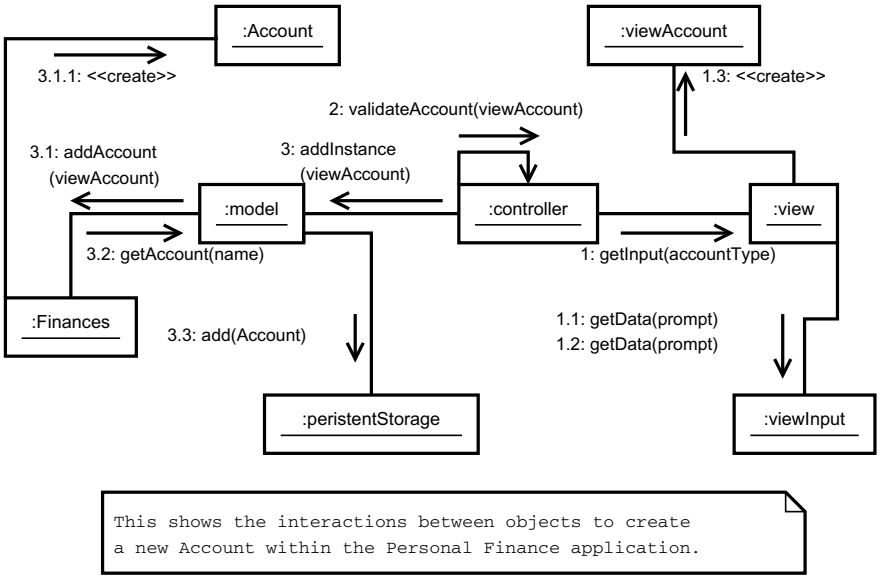


Fig. 15.19 Personal finance communication diagram

15.5.5 OOD Personal Finances: Summary

15.5.5.1 Design Models

The design models shown in Figs. 15.13, 15.14, 15.15, 15.16, and 15.19 show the impact Model–View–Controller has on the number and type of classes needed, and how these classes/objects interact with each other. As discussed in Chap. 12, a software architect may want/need to emphasize only certain aspects of the design. For example, if an understanding of object-orientation is critical to the success of the personal finance software, then the package, class, and communication diagrams are important while the DFD and statechart may be omitted. On the other hand, perhaps showing the interactions of the personal finance system with a user and persistent storage is important. In this case, the DFD, communication diagram, and statechart would be important to develop.

15.5.5.2 Evaluation

An evaluation of the personal finances design, as shown in Figs. 15.13, 15.14, 15.15, 15.16, and 15.19, is briefly described below.

Simplicity: Since we have described a high-level design, or architecture, our efforts should have resulted in models that accurately reflect the needs/requirements while also abstracting away the details implied by the requirements. The package, DFD, and statechart models are simple to read while the class and communication

diagrams provide significantly more details. Having a class diagram for each of the three packages (i.e., for the three MVC components) gives the appearance that this design provides too many details, perhaps making the class diagrams too hard to read and understand for such a high-level design. On the other hand, it is necessary to show the personal finance domain classes in both the view and model to illustrate an understanding of the application domain.

Coupling: The package, class, DFD, and communication models provide information to help us assess the coupling between the design elements. In the case of the package, class, and communication models, we see the interactions between the three MVC components are fairly clean—one class within each component is responsible for interacting with another component. This reduces the coupling between the MVC components. Coupling between classes residing in the same package/component is necessarily going to be a bit higher since these classes are cooperating with each other to satisfy the purpose of the component.

Cohesion: By using the Model–View–Controller architecture, the design models now show those classes responsible for the user interface, data storage, program flow, and domain knowledge. While there is some redundancy in the classes in the view and model to represent the personal finance domain, these classes are now more cohesive since they focus on only one aspect—data storage or user interface. This has clearly been improved when we compare this chapter's design with Chap. 12.

Information hiding: Based on the use of MVC, it is clear that the view has no idea how the model performs its responsibilities and vice versa. Similarly, while the controller has an interface with both the model and view, neither of these components knows how the controller performs its responsibilities (i.e., controls program flow and implements domain-specific processing).

Performance: Without knowing more details about the data structures (i.e., containers) to be used, it is difficult to assess the performance of this high-level design. In addition, the *Personal Finance Database* shown on the DFD may have performance implications that cannot be assessed at this time.

Security: The case study requirements do not state any need for validation of data. However, the controller component clearly shows that it will be doing validation of accounts, labels, reports, and transactions.

15.6 OO Language Features

Most object-oriented programming languages, including Java, allow a developer to create both interfaces and abstract classes. These two concepts are introduced in this chapter and then used in the case studies in future chapters.

Interface: A blueprint used to create a class definition. A Java interface may contain static constants and method signatures. In a class diagram, an interface is realized by another interface or a class.

abstract class: A class that cannot be used to construct an object. An abstract class may contain abstract and concrete methods, class and instance variables, and class and instance methods. In a class diagram, an abstract class is inherited by another class, which may be abstract or concrete.

15.6.1 A Simple Example

This example will use an interface and an abstract class to represent a small part of a software design for an autonomous car.

Listing 15.9 shows an interface that has one static constant and two method signatures.

Listing 15.9 Java Interface

```
public interface demoInterfaceCar {
    public static final double TOO_FAST_KPH = 160.9;

    //Methods in an interface are abstract w/out using the keyword.
    public boolean slowDown(double amount);
    public boolean speedUp(double amount);
}
```

Listing 15.10 shows a class that implements this interface. This implementation must define a concrete method for each of the method signatures found in the interface. As shown in Listing 15.10, a class implementing an interface may have members (i.e., instance variables and methods) not associated with the interface.

Listing 15.10 Use a Java Interface

```
public class demoUseInterfaceCar implements demoInterfaceCar {
    private int numberPassengers;
    private double speed;

    public demoUseInterfaceCar(int numberPassengers, double speed)
    {
        this.numberPassengers = numberPassengers;
        this.speed = speed;
    }

    public int getNumberPassengers()
    {
        return numberPassengers;
    }

    //Implement the methods declared in demoInterfaceCar
    public boolean slowDown(double amount)
    {
        System.out.println("slowDown amount:" + amount);
        return true;
    }
```

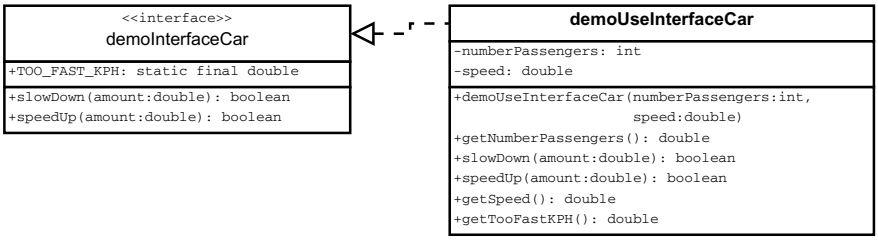


Fig. 15.20 Class diagram for demo interface

```
    public boolean speedUp(double amount)
    {
        System.out.println("speedUp amount:" + amount);
        return true;
    }

    public double getSpeed()
    {
        return speed;
    }
    public double getTooFastKPH()
    {
        return demoInterfaceCar.TOO_FAST_KPH;
    }
}
```

The class diagram in Fig. 15.20 shows the *realizes* relationship between the interface and the implementing class. A single class may implement many Java interfaces. This gives a designer flexibility in creating interfaces that provide for consistent implementation across many classes and packages.

Listing 15.11 shows an abstract class that has one instance variable, one static constant, two abstract methods, and one instance method.

Listing 15.11 Java Abstract Class

```
public abstract class demoAbstractCar
{
    private double speed;
    public static final double TOO_FAST_KPH = 160.9;

    //Abstract methods must be defined using the keyword.
    public abstract boolean slowDown(double amount);
    public abstract boolean speedUp(double amount);

    public double getSpeed()
    {
        return speed;
    }
}
```

Listing 15.12 shows a class that extends this abstract class. The subclass (or child class) must define a concrete method for each of the abstract methods found in the abstract class. As shown in Listing 15.12, an abstract class may have members not associated with the abstract class.

Listing 15.12 Use a Java Abstract Class

```
public class demoUseAbstractCar extends demoAbstractCar {
    private int numberPassengers;

    public demoUseAbstractCar(int numberPassengers,double speed)
    {
        this.numberPassengers = numberPassengers;
        super.speed = speed;
    }

    public int getNumberPassengers()
    {
        return numberPassengers;
    }

    //Implement the abstract methods declared in demoAbstractCar
    public boolean slowDown(double amount)
    {
        System.out.println("slowDown amount:" + amount);
        return true;
    }
    public boolean speedUp(double amount)
    {
        System.out.println("speedUp amount:" + amount);
        return true;
    }

    public double getTooFastKPH()
    {
        return demoAbstractCar.TOO_FAST_KPH;
    }
}
```

The class diagram in Fig. 15.21 shows the inheritance relationship between the abstract class and the subclass. A single class may extend a single (abstract or concrete) class.

15.6.1.1 Comparing Use of Interface and Abstract Class

The class diagrams in Figs. 15.20 and 15.21 result in the same list of publicly available methods. A programmer would instantiate a demoUseInterfaceCar or demoUseAbstractCar object to gain access to the methods (getNumberPassengers, slowDown, speedUp, getSpeed, and getTooFastKPH) representing the behavior of this automated car. The key differences in these two designs, which highlight the differences between a Java interface and a Java abstract class, are as follows.

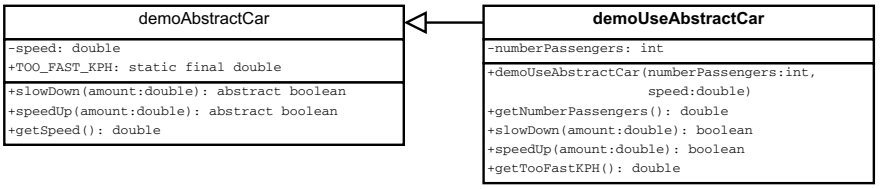


Fig. 15.21 Class diagram for demo abstract class

- The abstract class has a protected instance variable (speed) not found in the interface.
- The abstract class has a public method (getSpeed) not found in the interface.
- The class implementing the interface has a private instance variable (speed) not found in the class extending the abstract class.
- The class implementing the interface has a public method (getSpeed) not found in the class extending the abstract class.

Use an abstract class when you want to implement common behavior in a super class and also have no need to instantiate an object of the super class. Use an interface when you want to describe common behavior that should be included in many classes.

15.7 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes designing using a bottom-up and top-down approach.

Exercises

Discussion Questions

1. How is coupling between modules improved with a good implementation of MVC?
2. How is cohesion within a module improved with a good implementation of MVC?
3. How is information hiding improved with a good implementation of MVC?

Hands-On Exercises

1. Use an existing code solution that you've developed, develop design models that show how your solution could be redesigned to use MVC. Apply the software design criteria to your design models. How good or bad is your design?
2. Use your development of an application that you started in Chap. 3 for this exercise. Modify your design to use MVC and then evaluate your MVC design using the six software design criteria.
3. Continue Hands-on Exercise 3 from Chap. 12 by changing your software design to use MVC. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

    - Airline reservation and seat assignment,
    - Automated teller machine (ATM),
    - Bus transportation system,
    - Course-class enrollment,
    - Digital library,
    - Inventory and distribution control,
    - Online retail shopping cart,
    - Personal calendar, and
    - Travel itinerary.



SD Case Study: Model–View–Controller

16

The objective of this chapter is to apply the Model–View–Controller architectural pattern to the case study using structured design techniques.

16.1 SD: Preconditions

The following should be true prior to starting this chapter.

- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.

16.2 SD: Transition to MVC

The case study software design discussed in Chap. 13 exhibits low (or weak) cohesion, which characterizes a poor design. This chapter will implement Model–View–Controller to improve the cohesion of the ABA while maintaining the good design

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_16

characteristics of simple, low coupling, logarithmic time performance, and doing input/output data validation.

One way to think about the implementation of Model–View–Controller is from a requirements perspective. We identify the component or components that need to be involved in order to satisfy each requirement. The following requirements have been implemented in the address book application.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Do not retrieve any information from the address book.

Implementing requirement 1 involves both the view (*allow for entry*) and model (*allow for nonpersistent storage*) components. Requirement 2 will be part of the model component while requirement 3 is associated with the view component. Requirements 4 and 5 describe the types of validation needed for the name and phone number values. Since this is a standalone application, i.e., it is not distributed across computing devices, this validation belongs in the controller component. (It is important to emphasize that any application whose components are distributed across computing devices should have validation requirements implemented in multiple components. While this duplicates much of the validation logic, it will greatly improve the defensive capabilities of the application in resisting malicious attacks via input channels.) Requirement 6 is another validation requirement, but this one can only be implemented in the model component. This is because the model is the only component that knows the type of data structure being used to non-persistently store the data. Finally, requirement 7 exists to simplify the design of the address book application by eliminating the need for a user interface to include user requests to display address book data. This last requirement affects the design of all three components by reducing the need for logic to display address book entries to the user.[1]

Given the above allocation of requirements to components, requirement 1 has been divided into two sub-requirements. Table 16.1 summarizes the allocation of requirements to components. The remaining sections in this chapter will refer to requirements 1a, 1b, and 2 through 6.

[1] While address book data is displayed once the user exits the application, this is done to verify the address book containing the correct information based on the order in which data was entered by the user. As explained in prior chapters, this display of address book data is test code used to verify the correctness of the logic implementing requirements 1 through 6.

Table 16.1 Map requirements to MVC

Requirement	Model	View	Controller
1a. Allow for entry of a person's name		X	
1b. Allow for (nonpersistent) storage of people's names	X		
2. Store for each person a single phone number and a single email address	X		
3. Use a simple text-based user interface to obtain a name, phone number, and email address for each contact		X	
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered			X
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered			X
6. Prevent a duplicate name from being stored in the address book	X		

16.3 SD: ABA MVC Design Version A

The Version A design of the first MVC implementation is described using design models and code snippets. The design is explained by asking a question typically raised when first learning MVC.

16.3.1 SD: How Do MVC Components Communicate with Each Other?

The Version A design for the ABA using Model–View–Controller has four source code files representing the three MVC components. Listing 16.1 shows the code in the ABA_A.py source code file. This function starts the ABA by calling a function to initialize the controller component. Assuming this returns true, the controller's go function is called. This function drives the processing flow of the ABA, ending when the user enters "exit" for a contact name. At this point, the test code to display the contents of the address book is performed by calling the displayBook function in the controller.

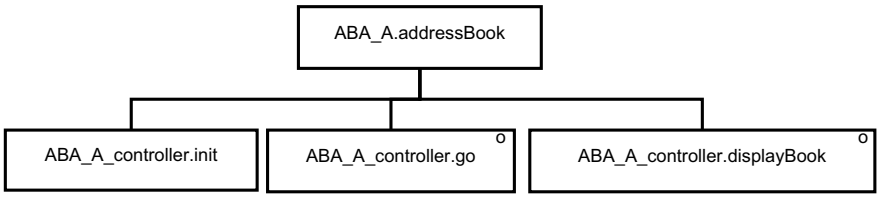


Fig. 16.1 Structure Diagram for MVC Design ABA Version A: one level of function calls

Listing 16.1 ABA MVC Design Version A—main method

```
import ABA_A_controller

def addressBook():
    if ABA_A_controller.init():
        ABA_A_controller.go()
        ABA_A_controller.displayBook()
```

Figure 16.1 shows a structure diagram representing the addressBook function. The lowercase "oh" (o) in the upper-right corner indicates that the go and displayBook function calls are conditionally executed.

Listing 16.2 shows the code for the controller's init function. This function calls the model's init function, which is shown in Listing 16.3. The model's init function defines three globals, including an empty dictionary representing the memory-based data structure used to store contact data. The ABA_A_model.init function always returns true.

Listing 16.2 ABA MVC Design Version A—ABA_A_controller.init() function

```
def init():
    return ABA_A_model.init()
```

Listing 16.3 ABA MVC Design Version A—ABA_A_model.init() function

```
def init():
    global book
    global sortedNames
    global iterSortedNames
    book = {}
    sortedNames = None
    iterSortedNames = None
    return True
```

Figure 16.2 shows a structure diagram with two levels of function calls. The asterisk (*) in the upper-right corner indicates that the functions called by the go and displayBook functions are inside an iteration.

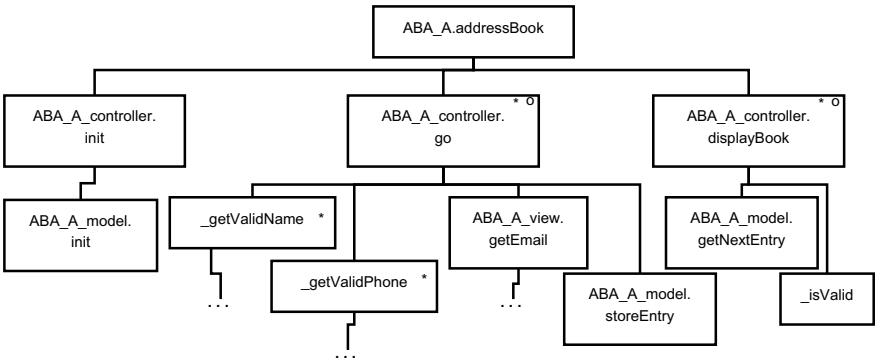


Fig. 16.2 Structure Diagram for MVC Design ABA Version A: two levels of function calls

Listing 16.4 shows the code for the controller's go function. This function calls the controller's _getValidName function, then iteratively calls other controller, view, and model functions until the user enters "exit" for a contact name. Figure 16.3 shows a structure chart with two levels of function calls. A structure chart shows the data passed to and returned from each function call. Combining the structure diagram in Fig. 16.2 and the structure chart in Fig. 16.3, we see the two controller functions—_getValidName and _getValidPhone—will iteratively ask the user for a data value until it is valid.

Listing 16.4 ABA MVC Design Version A—go function

```
def go():
    name = _getValidName()
    while name != "exit":
        phone = _getValidPhone(name)
        email = ABA_A_view.getEmail(name)
        ABA_A_model.storeEntry(name, (phone, email))
        name = _getValidName()
```

The steps below describe the general processing flow expressed in the controller's go function. These steps are also expressed in the structure diagram and structure chart in Figs. 16.2 and 16.3, respectively.

Step 1: The go function calls the _getValidName function. As shown in Listing 16.5 and Fig. 16.2, this *private* function will iterate until a valid contact name has been entered. The special value "exit" represents a valid contact name but is used by the controller to end the input of contact data. When the name entered is not "exit", steps 2 through 5 are performed.

Step 2: The _getValidPhone function is called. This will iterate until the user enters a valid phone number.

Step 3: The getEmail function in the view module is called to obtain an email address from the user.

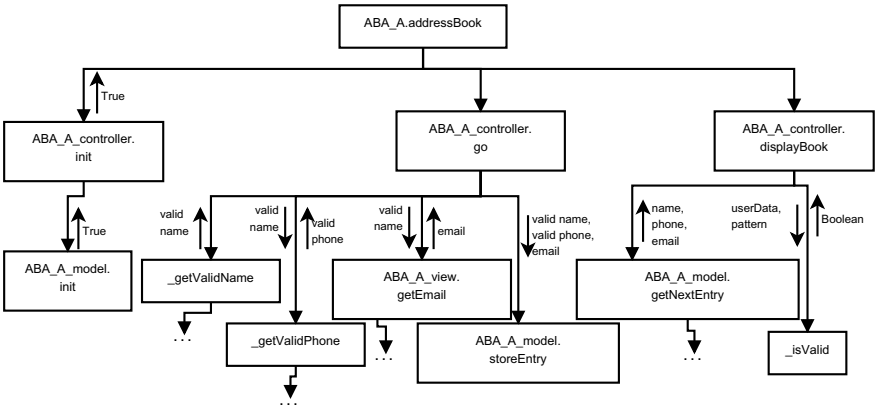


Fig. 16.3 Structure Chart for MVC Design ABA Version A: two levels of function calls

Step 4: The storeEntry function in the model component is called to store the contact data in the book global data structure.

Step 5: The _getValidName function is called to obtain another contact name.

Listing 16.5 ABA MVC Design Version A—_getValidName function

```
def _getValidName():
    #Continue asking for a contact name until valid data is entered.
    valid = False
    while not valid:
        name = ABA_A_view.getName()
        name = name.strip()
        valid = _isValid(name, RE_PATTERNS.NAME)
        if not valid:
            ABA_A_view.displayMessage("A contact name must contain " +
                "only uppercase and lowercase letters and spaces.")

    return name
```

The Python functions whose name begins with an underscore represent *private* functions in the module. This naming convention tells programmers that the only code that *should* call the function is code within the same source code file. While this naming convention is an important way to communicate the use of functions, all Python functions defined in a module may be called by code in any other module. In addition, global variables defined in one module are directly accessible from another module. For example, Listing 16.3 shows three global variables defined within the model component. These global variables may be directly accessed by the controller or view module by simply doing two things: (1) importing the ABA_A_model module and (2) writing a statement that includes the expression ABA_A_model.book.

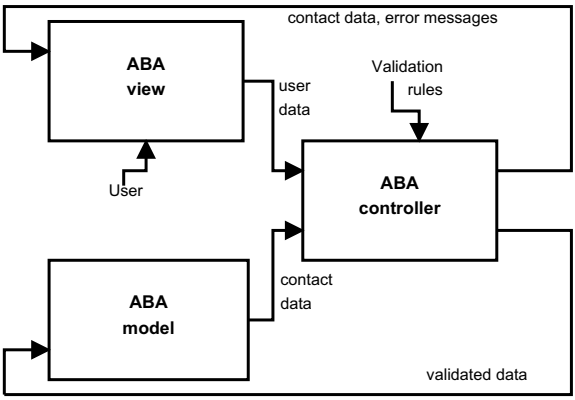


Fig. 16.4 IDEF0 function model—MVC design Version A

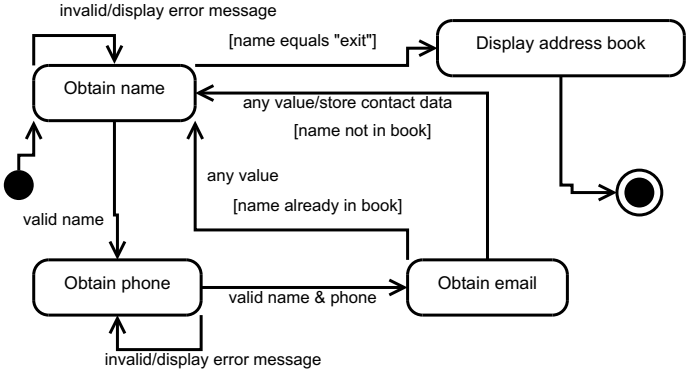


Fig. 16.5 State diagram—MVC design Version A

16.3.2 SD: ABA MVC Version A Software Design

The IDEF0 function model shown in Fig. 16.4 describes the primary flows of data between the three ABA components. It shows the User interacts only with the ABA view component while the ABA controller component performs validation of the data.

The way the user interacts with the ABA Version A design is shown in Fig. 16.5. The user first must enter a valid name, or "exit" to display the address book and end the ABA. Once a valid name is entered, a valid phone number must be entered. Once a valid phone number is entered, any value is accepted for an email address. Only when the name is *not* already in the address book is this contact data added to the address book.

16.3.3 SD: Evaluate ABA MVC Version A Software Design

The structured design just described will now be evaluated using the six design criteria introduced in Chap. 11.

16.3.3.1 Simplicity

The controller component has two source code files. The ABA_A.py source code file contains the addressBook function which calls functions defined in the ABA_A_controller.py source code file. The ABA_A_controller.py source code file has import statements for the model and view modules, allowing the controller to call any function in these other two modules.

The model and view components have no import statements, indicating that they are unable to call functions defined in other Python modules.

This design has a total of 4 source code files, 15 functions, and 3 global variables. Of these, all but 2 functions are associated with implementing the seven requirements (1a, 1b, and 2 through 6) shown in Table 16.1. The two test functions (displayBook and getNextEntry) are used to verify the contents of the address book. This seems like a reasonable number of source code files, functions, and global variables for the number of requirements being implemented. The design of Version A appears to be simple to understand.

16.3.3.2 Coupling

The Model–View–Controller components described in the ABA MVC Version A design represent three distinct modules.[2] The question is, to what degree does one of these modules rely on the other modules? We can address this question by considering four perspectives on how the coupling between two modules can be expressed.

1. Look at the functions within a module that are called by other modules. Does the number of function calls seem excessive?
   We can identify the functions called by other components/modules by looking at the three structure charts in Figs. 16.6, 16.7, and 16.8.

   Controller: The addressBook function is in the ABA_A.py source code file while six functions are in the ABA_A_controller.py source code file. We see from the three structure charts that the addressBook function calls the init, go, and displayBook controller functions. The controller's three *private* functions (i.e., functions whose names begin with an underscore) are all called by other controller functions.

[2] Use of the term *module* when discussing coupling and cohesion refers to one of the three MVC components. As a reminder, a Python source code file is also called a module. This chapter is assessing coupling and cohesion for MVC components. We could also assess coupling and cohesion for each source code file (i.e., Python source code file).

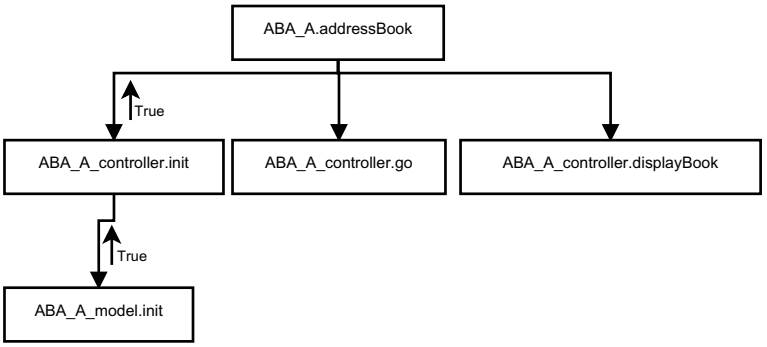


Fig. 16.6 Structure chart for MVC design ABA Version A: init function

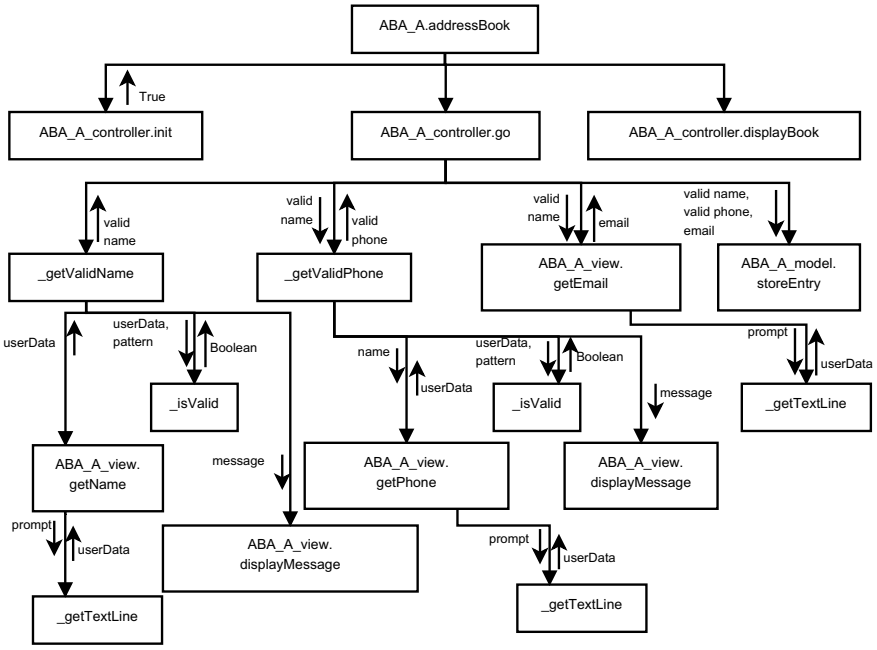


Fig. 16.7 Structure chart for MVC design ABA Version A: go function

Model: All three model functions are called by code in the controller. None of these functions are called by code in the view.

View: Five of these functions are called by the controller. The remaining function in the view is called only by other view functions.

Looking at coupling from the perspective of function calls shows no coupling between the model and view. The coupling between the controller and model is unidirectional; the controller calls functions within the model to either store

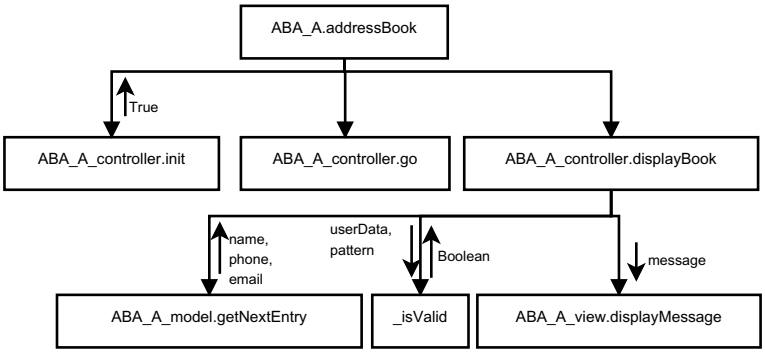


Fig. 16.8 Structure chart for MVC design ABA Version A: displayBook function

or retrieve address book data. The coupling between the controller and view is also unidirectional, the controller calls functions within the view to obtain data from the user or to display data to the user.

2. Look at the data that is passed via each function call. Does the amount of data being passed between components via function calls seem too excessive or too complex?
   Looking at the three structure charts in Figs. 16.6, 16.7, and 16.8.

   Controller: No controller function is called from the model or view component.
   Model: Two of these functions have no parameters. The third function storeEntry has two parameters used to give the model the contact data to be stored in the book data structure.
   View: Three of these functions have parameters. The getPhone and getEmail functions have a name parameter to allow the view to ask for data for a specific contact person. The displayMessage function has a parameter to pass the message to be displayed to the user.

   A parameter passing perspective shows very little coupling between the controller and model and between the controller and view. However, there is one view function called by the controller that increases coupling between these two components. The controller passes a specific message when it calls the displayMessage method, which results in this message being displayed to the user. This message is part of the user interface but is known by the controller component, increasing the coupling between these two components.

3. Look at the data that is returned via each function call. Does the amount of data being returned to another component seem too excessive or too complex?
   Looking at the three structure charts in Figs. 16.6, 16.7, and 16.8.

Controller: None of these functions are called by another component.
Model: The init function returns true to the controller, while getNextEntry returns the name, phone, and email data for a contact.
View: Three of these functions return a data value. The getName, getPhone, and getEmail functions return the name, phone number, or email address, respectively, entered by the user.

A function return perspective shows no coupling between the model and view modules, coupling between the controller and model is unidirectional (i.e., model functions return data values to the controller but not vice-versa), and coupling between the controller and view is also unidirectional (i.e., view functions return data values to the controller).

4. Look at the use of global data structures. Does one module define a global data structure that is referenced by another module?
   The three global variables used in the model component are not accessed by any other component.

Given the above descriptions on the coupling between the model, view, and controller components, the amount of coupling seems minimal and thus represents a good design.

16.3.3.3 Cohesion

The MVC components described in the design of ABA MVC Version A represent three distinct modules. The question is, to what degree does each of these modules contain functions that are strongly related to each other and are single-minded in their purpose? We'll look at the purpose of each function and variable within the same module, and whether (or how much) these are related to each other.

Controller functions: With three exceptions, the controller functions either provide the processing flow of the application or implement the data validation requirements. This matches the purpose of the controller component. The first exception is the displayBook function, which is part of the test code to verify the contents of the address book.
The second exception is the way in which the controller determines when the user has requested the application to end. This behavior, shown in Listing 16.4, requires the controller check the name value entered to see if it matches "exit". This knowledge should reside in the view.
The third exception is in the validation logic implemented in the controller. The *private* functions _getValidName and _getValidPhone pass an error message to the view for display to the user. This makes these two functions less cohesive with the rest of the controller since the controller is not responsible for communication with the user.

Controller variables: This module has two global variables, as shown in Listing 16.6. These two global variables contain the regular expressions for validating a contact name and a phone number. The local variables used within each function are directly related to the processing performed by the function.

Model functions: All of the model functions are designed to either store or retrieve address book data. This matches the purpose of the model component. The data used by these functions, whether this data comes from parameters or instance variables, is all related to contact data. Likewise, the return values from these functions are also related to contact data.

Model variables: This module has three global variables, as previously mentioned. The *global book* is a dictionary used to store contact data. The *global sortedNames* and *global iterSortedNames* are used by the getNextEntry test function to navigate through the book, one contact at a time.

View functions: All of the view functions are used to display information to and obtain data from the user. This matches the purpose of the view component. The data used by these methods, whether this data comes from parameters or instance variables, all has to do with the user interface. Likewise, the return values from these methods also have to do with data from the user.

View variables: This module has no global variables. The one local variable is directly related to the purpose of the view module.

The model component is highly cohesive, whereas the controller and view modules are less cohesive. As mentioned above, the controller validation logic gives the view an error message to be displayed, making the controller less cohesive. For the view, one could argue that the single module in the view is doing two distinctly different things—it is obtaining data from the user and it is displaying data to the user. Since the ABA is using a text-based user interface, the mechanisms used to obtain data (i.e., input function) are different from the mechanisms to display data (i.e., print function).

Listing 16.6 ABA MVC Design Version A—Controller global variables

```
import collections
import re

RE_patterns = collections.namedtuple("RE_patterns", "NAME  PHONE")
RE_PATTERNS = RE_patterns("^[ A−Za−z]+$", "^[0−9]+$")
```

16.3.3.4 Information Hiding

Does each module in the MVC Version A design hide its implementation details from the other components? By implementation details, we mean the algorithms and data structures being used by the module.

As mentioned above, when a source code file imports a Python module, it has direct access to all functions and global variables defined in the imported module. This inherently makes the use of Python more challenging when trying to implement a

design that supports information hiding. For this criteria to improve, we must rethink our use of source code files and the import statement.

16.3.3.5 Performance

The performance of the MVC Version A design is similar to the performance of the ABA solutions described in Chap. 7. Specifically, the use of a Python dictionary gives us constant time performance.

16.3.3.6 Security

The security of the MVC Version A design is similar to the security of the ABA solutions described in Chap. 10. Specifically, the design includes data input validation, data output validation, exception handling, and fail-safe defaults. In addition, Python is a type-safe language.

16.4 SD: ABA MVC Design Version B

The design of MVC Version A of the ABA is okay, but can be improved. First, the coupling between the controller and view is too high. This is due to the number of times the controller calls the view's displayMessage function. Each of these function calls passes a specific message to the view for display to the user. In these instances, the controller has too much knowledge about the user interface, which should be the responsibility of the view component.

Second, the cohesion of the controller can be improved. Specifically, the logic that compares the contact name to "exit" should really be in the user interface (view) component. To make the controller more cohesive, any user interface logic should be removed; the controller should only be responsible for the execution flow of the application and data validation.

Finally, the Version A design does a very poor job of information hiding. As already mentioned, Python allows any global variable or function to be directly used by another module, as long as the other module imports the source code file.

The Version B design for the ABA using Model–View–Controller has nine source code files, as listed in Table 16.2. This second MVC design has three source code files for the model, three for the controller, and three for the view.

To address the design flaws described above in Version A, the way the user interacts with the ABA has been changed in Version B. Figure 16.4 shows the IDEF0 function model for Version A. Since this is a high-level description of the Version A design, this function model also applies to the behavior of Version B. However, the statechart in Fig. 16.9 is significantly different from the Version A statechart shown in Fig. 16.5. In Version B, as long as the user does *not* enter "exit" for a contact name, all three data values—name, phone number, and email address—are entered by the user *before* validation is done. When "exit" is entered for a contact name, the address book data is displayed and the ABA ends. Otherwise, the contact name and phone number are

Table 16.2 ABA MVC design Version B—source code files

Source code file	Component?	Responsibilities
ABA_B.py	Controller	Starts the ABA
ABA_B_contact_data.py	Model	Stores a tuple containing data for one contact person. Has functions to allow the tuple to be created or retrieved
ABA_B_controller.py	Controller	Contains the overall processing flow for the ABA
ABA_B_controller_validation.py	Controller	Contains validation logic for the ABA
ABA_B_model.py	Model	Contains functions called by the controller
ABA_B_model_data.py	Model	Contains the address book data structure and functions that use the data structure
ABA_B_view.py	View	Contains functions called by the controller
ABA_B_view_input.py	View	Has one function used to obtain a data value from the user
ABA_B_view_output.py	View	Has one function used to display a message to the user

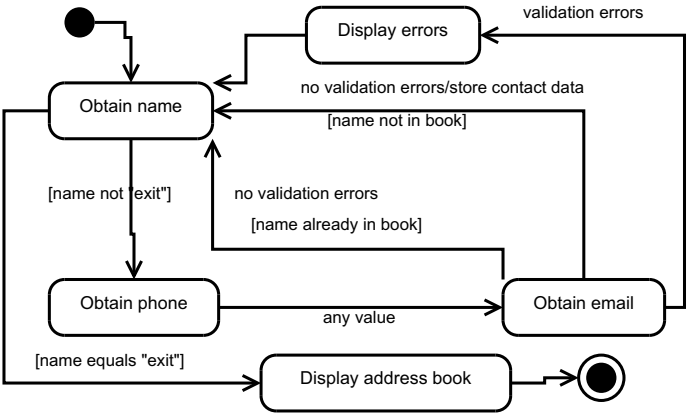


Fig. 16.9 State diagram—MVC design Version B

validated. When these two values are valid, the contact data is added to the address book when the contact name is not already in the book. When either or both values are invalid, appropriate error messages are displayed. In either case, the user is prompted to enter the next contact person's name.

16.4.1 SD: How Do MVC Objects Communicate with Each Other?

The Version A design for the ABA using Model–View–Controller had four source code files representing the three MVC components. The Version B design has over twice as many source code files! How has this improved the ABA design and its use of MVC?

First, the way the ABA starts in Version B is exactly like Version A. The ABA_B.py source code file calls the controller's init function. When this returns true, the controller's go function is called, which drives the processing flow of the ABA. When the go function ends, test code to display the contents of the address book is performed by calling the controller's displayBook function. This is shown in the structure diagram in Fig. 16.1.

However, the initialization of the model component includes more Python modules when compared to Version A. Listing 16.7 shows the ABA_B_controller.init function calling the ABA_B_model.init function. This controller function is exactly the same as Version A.

Listing 16.7 ABA MVC Design Version B—ABA_B_controller.init() function

```
import ABA_B_model

def init():
    return ABA_B_model.init()
```

Listing 16.8 shows the ABA_B_model.init function. Here is where Version B has changed. Now, the model's init function is calling the ABA_B_model_data.init function, which is shown in Listing 16.9. Since the ABA_B_controller.py source code file imports ABA_B_model, it can call any function and access any global variable defined in this module. However, the controller module cannot call functions nor access global variables in the ABA_B_model_data module since it does not import this source code file. The Version B design has now hidden the details (from the controller) of how the model component stores and retrieves contact data since all of these details are in the model_data module. In essence, the code in the ABA_B_model.py source code acts as an interface between the controller and the storage of data in the ABA.

Listing 16.8 ABA MVC Design Version B—ABA_B_model.init() function

```
import ABA_B_model_data

def init():
    return ABA_B_model_data.init()
```

Listing 16.9 ABA MVC Design Version B—ABA_B_model_data.init() function

```
def init():
    global book
    global sortedNames
    global iterSortedNames
    book = {}
```

```
    sortedNames = None
    iterSortedNames = None
    return True
```

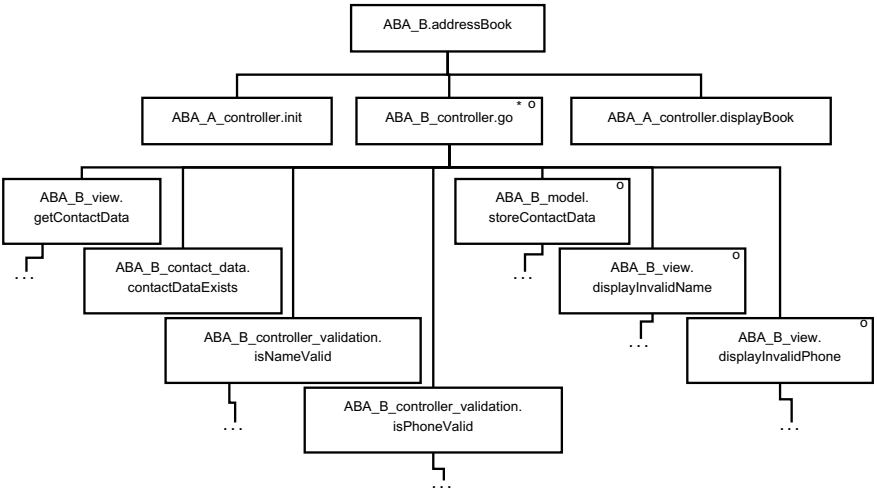


Fig. 16.10 Structure diagram for MVC design ABA Version B: go function two levels of function calls

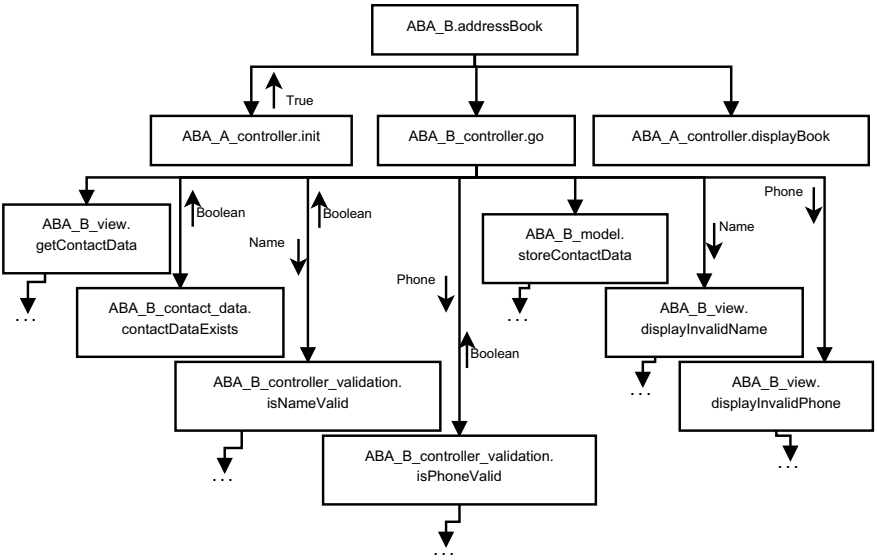


Fig. 16.11 Structure chart for MVC design ABA Version B: go function two levels of function calls

Figures 16.10 and 16.11 show a structure diagram and structure chart, respectively, with two levels of function calls for the main processing flow of the ABA, as implemented in the go function. Listing 16.10 shows the code for the controller's go function.

In Version B, the controller calls the view's getContactData function to obtain a name, phone number, and email address. The contactDataExists function in the ABA_B_contact_data module returns True when the global variable is not None. This global variable will be None when the user enters "exit" for a contact name. As long as the user has entered contact data, the name and phone number values will be validated using functions in the ABA_B_controller_validation module. When the name and phone number are valid, the storeContactData function is called in the model to store this tuple in the address book data structure. Otherwise, validation error messages are displayed to the user indicating which data values were incorrect.

Listing 16.10 ABA MVC Design Version B—go function

```
def go():
    ABA_B_view.getContactData()
    while ABA_B_contact_data.contactDataExists():
        nameValid = ABA_B_controller_validation.isNameValid(
                                ABA_B_contact_data.getName())
        phoneValid = ABA_B_controller_validation.isPhoneValid(
                                ABA_B_contact_data.getPhone())
        if nameValid and phoneValid:
            ABA_B_model.storeContactData()
        else:
            if not nameValid:
                ABA_B_view.displayInvalidName(
                        ABA_B_contact_data.getName())
            if not phoneValid:
                ABA_B_view.displayInvalidPhone(
                        ABA_B_contact_data.getPhone())
        ABA_B_view.getContactData()
```

Figure 16.12 shows the entire series of function calls for the getContactData function. When the user enters "exit" for the contact name, the noContactData function is called to set the global variable contactData in the ABA_B_contact_data module to None. When the user enters anything but "exit" for the contact name, the user is prompted to enter a phone number and email address. All three of these user-supplied data values are then used to create a tuple by calling the createContactDataTuple function also found in the ABA_B_contact_data module.

16.4.2 SD: Evaluate ABA MVC Version B Software Design

The structured design just described will now be evaluated using the six design criteria introduced in Chap. 11.

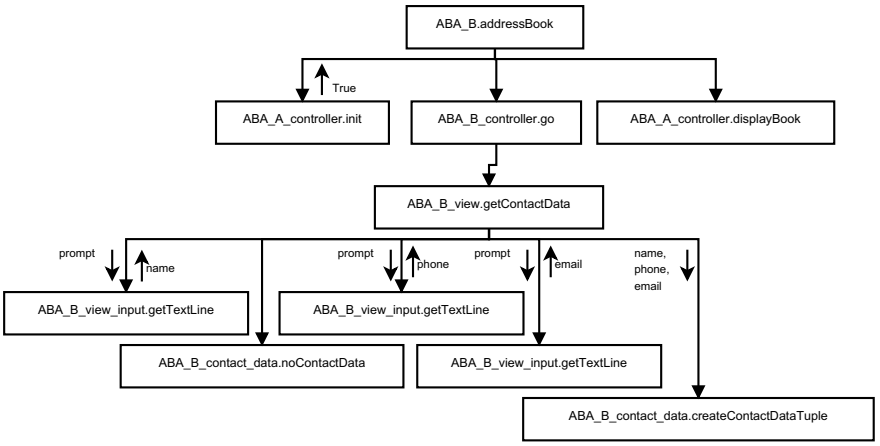


Fig. 16.12 Structure chart for MVC design ABA Version B: getContactData function

16.4.2.1 Simplicity

The controller component has three source code files. The ABA_B.py source code file contains the addressBook function which calls functions defined in the ABA_B_controller.py source code file. The ABA_B_controller.py source code file has import statements for the model and view modules, as well as the ABA_B_contact_data.py and ABA_B_controller_validation.py source code files.

The model component in Version B now has an import statement for the ABA_B_model_data source code file. The model_data module contains the global variables to store the contact data, functions used to manipulate this data structure, and imports the ABA_B_contact_data.py source code file.

The view component in Version B now has three import statements. The first import brings in the ABA_B_contact_data module, allowing the view to call functions to create a tuple of contact data. The second and third import statements bring in the view_input and view_output modules, which are used to obtain data from or display data to the user, respectively.

As mentioned above, this design has a total of nine source code files. These files contain a total of 29 functions and four global variables. Compared with Version A, we have approximately twice as many source code files and functions in Version B. All but five functions are associated with implementing the seven requirements (1a, 1b, and 2 through 6) shown in Table 16.1. Since the design and code is larger in Version B, this design solution is more complex than what we saw in Version A. If this design does not correct the issues Version A has with coupling, cohesion, and information hiding, then these changes are not worth keeping.

16.4.2.2 Coupling

The Model–View–Controller components described in the ABA MVC Version B design represent three distinct modules. The question is, to what degree does one of

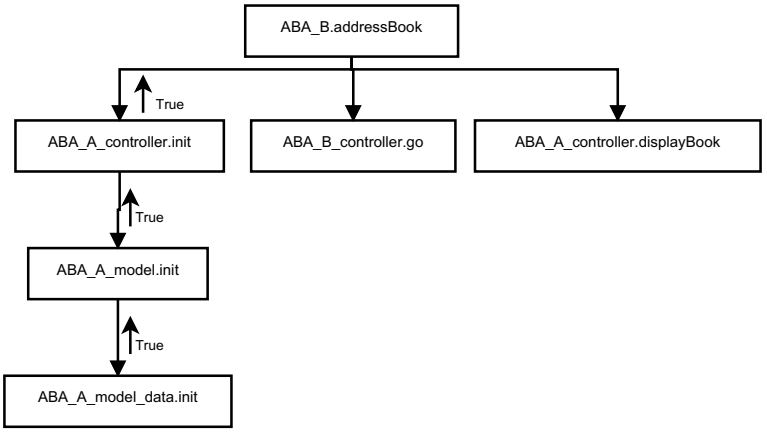


Fig. 16.13 Structure chart for MVC design ABA Version B: init function

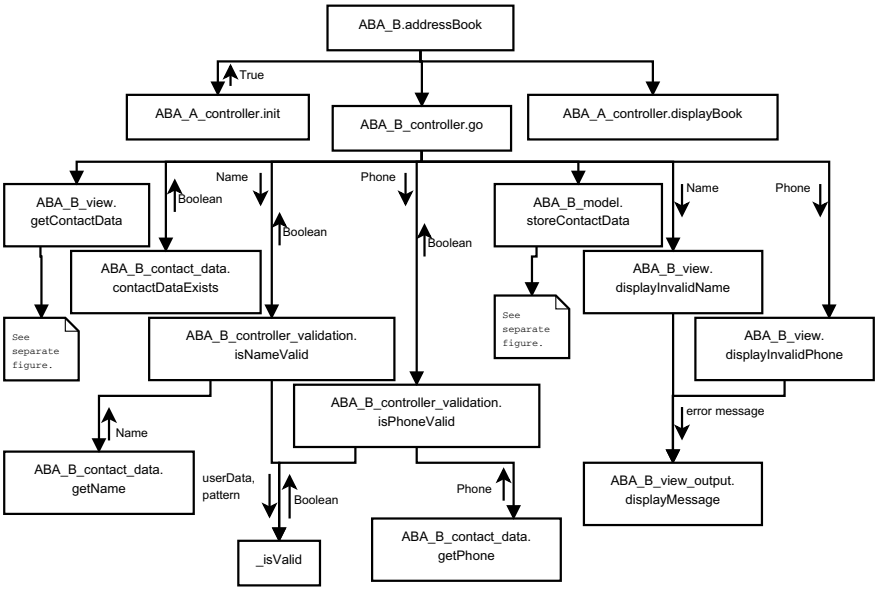


Fig. 16.14 Structure chart for MVC design ABA Version B: go function

these modules rely on the other modules? We can address this question by considering four perspectives on how the coupling between two modules can be expressed. We'll highlight the differences between the Version A and B designs.

1. Look at the functions within a module that are called by other modules. Does the number of function calls seem excessive?
   We can identify the functions called by other components/modules by looking at the four structure charts in Figs. 16.12, 16.13, 16.14, and 16.15.

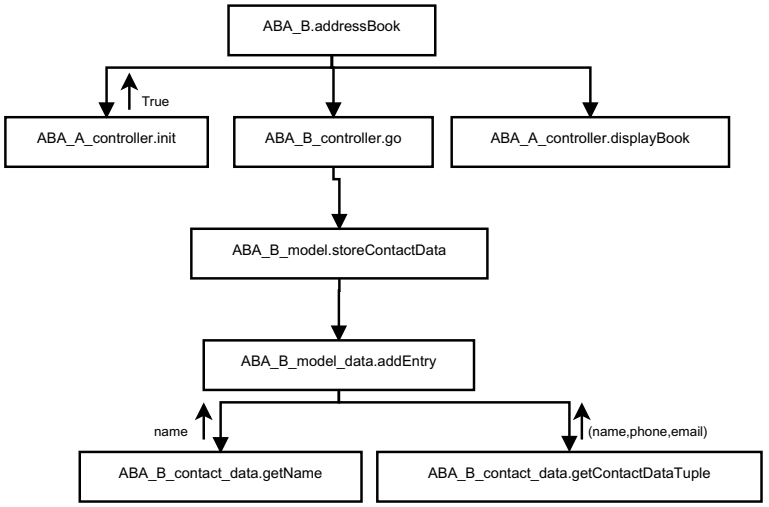


Fig. 16.15 Structure chart for MVC design ABA Version B: storeContactData function

Controller: The functions in the ABA_B_controller.py source code file call functions in the ABA_B_contact_data.py, ABA_B_model.py, and ABA_B_view.py source code files. The purpose of these three source code files is to provide functions for the controller component to call to initiate processing within the model or view component. The functions in the (ABA_B_controller_validation.py) source code file are only called from within the controller.

Model: All three functions in the ABA_B_model.py source code file are called by code in the controller. These three functions (init, getNextEntry, and storeContactData) are the only code that call functions defined in the ABA_B_model_data.py source code file. None of the functions in these two source code files are called by code in the view.

The third source code file in the model component (ABA_B_contact_data.py) contains seven functions and is responsible for storing and retrieving data for a single contact person. These functions are called by all three components. The view component calls the createContactDataTuple function to store the user-supplied data as a tuple in the global contactData variable. The view also calls the noContactData function to set the global variable to None, indicating the user has entered exit as a contact name. The controller component calls the contactDataExists, getName, and getPhone functions as part of doing validation, and calls setContactDataTuple as part of the displayBook test code. The model component calls getName and getContactDataTuple as part of the displayBook test function. If we eliminate the test code from our discussion, we see that the view component creates a contactData tuple while the controller and model components use the data stored in this tuple.

View: All five functions in the ABA_B_view.py source code file are called by code in the controller. The functions in the other two view source

code files (getTextLine in ABA_B_view_input.py and displayMessage in ABA_B_view_output.py) are called only by other view functions. None of the view functions are called by code in the model.

Looking at coupling from the perspective of function calls shows similar coupling to Version A, with one exception. As mentioned, the contact_data source code file is part of the model component. This contains functions that are called by the view component. Thus, we have coupling between the model and view components that did not exist in Version A.

2. Look at the data that is passed via each function call. Does the amount of data being passed between components via function calls seem too excessive or too complex?
Looking at the four structure charts in Figs. 16.12, 16.13, 16.14, and 16.15.

Controller: None of these functions are called by code residing in another component.
Model: The three functions called by the controller have no parameters. The createContactDataTuple function found in the contact_data source code file is called by the view. This function has three parameters representing the three user-supplied data values for a contact person.
View: Three of these functions have parameters. The displayInvalidName and displayInvalidPhone functions have a name and phone parameter, respectively, to allow the view to display a meaningful error message. The displayTestData test function has two parameters used to display a single contact person's data.

A parameter passing perspective shows very little coupling between the controller and model, and between the controller and view. Version B now has increased coupling between the model and view due to a contact_data function being called to save user-supplied data in a tuple. Version B has also eliminated the controller passing specific error messages to be displayed by the view, reducing coupling between the controller and view.

3. Look at the data that is returned via each function call. Does the amount of data being returned to another component seem too excessive or too complex?
Looking at the four structure charts in Figs. 16.12, 16.13, 16.14, and 16.15.

Controller: None of these functions are called by another component.
Model: The init function returns true to the controller. Three functions in the contact_data source code file return a value to the controller. The contactDataExists returns a Boolean to indicate whether user-supplied contact data currently exists. The getName and getPhone functions return a contact name or phone number, respectively.
View: None of the view functions return a value to the controller.

A function return perspective shows no coupling between the model and view modules, very low coupling between the controller and model, and no coupling between the controller and view.

4. Look at the use of global data structures. Does one module define a global data structure that is referenced by another module?

Controller: The controller has two global variables in the controller_validation source code file. This file is imported only by the controller source code file and thus is only available to use by code within the controller component.

Model: In Version B, the three global variables are now in a separate source code file (ABA_B_model_data). Since only the ABA_B_model code imports the model_data code, only the model component has access to these global variables. This design change reduces coupling when compared to Version A.

The ABA_B_contact_data source code file is new in Version B. This contains one global variable used to store contact data as a tuple. Three source code files (controller, model_data, and view) import the contact_data code. The functions in the contact_data source code file are designed to give access to the global variable without having to access it directly (i.e., by writing ABA_B_contact_data.contactData). As mentioned above, the view stores user-supplied contact data in the global variable while the controller and model components use other functions to obtain the data in the global variable.

View: Does not have any global variables.

If you eliminate the contact_data global variable, the coupling between the model, view, and controller components is reduced when compared with Version A. As just described, the functions in the contact_data module are called by all three components to either create a contact data tuple or to obtain contact data from this tuple. The use of the contact_data module does increase coupling between the three components. However, this design seems reasonable given the use of functions to control how the global variable ABA_B_contact_data.contactData is used.

16.4.2.3 Cohesion

The MVC components described in the design of ABA MVC Version B represent three distinct modules. The question is, to what degree does each of these modules contain functions that are strongly related to each other and are single-minded in their purpose? We'll look at the purpose of each function and variable within the same module, and whether (or how much) these are related to each other.

Controller functions: With one exception, the controller functions either provide the processing flow of the application or implement the data validation requirements. This matches the purpose of the controller component. The exception is

the displayBook function, which is part of the test code to verify the contents of the address book. Version B now has the view component recognizing when "exit" has been entered for a contact name and constructing an appropriate error message when a validation error occurs.

Controller variables: The two global variables containing regular expressions for validating a contact name and phone number are now in the ABA_B_controller_validation source code file, making use of these variables more cohesive with the validation functions.

Model functions: All of the model functions are designed to either store or retrieve address book data. This matches the purpose of the model component. The data used by these functions, whether this data comes from parameters, instance variables, or global variables, is all related to contact data. Likewise, the return values from these functions are also related to contact data.

Model variables: This module has four global variables, as previously mentioned. The *global book* is a dictionary used to store contact data. The *global sortedNames* and *global iterSortedNames* are used by the getNextEntry test function to navigate through the book one contact at a time. The *global contactData* is a tuple used to store an instance of contact data.

View functions: All of the view functions are used to display information to and obtain data from the user. This matches the purpose of the view component. The data used by these methods, whether this data comes from parameters, instance variables, or global variables, all has to do with the user interface. Likewise, the return values from these methods also have to do with data from the user.

View variables: This module has no global variables. The local variables are directly related to the purpose of the view module.

All three components are more cohesive when compared with Version A.

16.4.2.4 Information Hiding

Does each module in the MVC design Version B hide its implementation details from the other components? By implementation details, we mean the algorithms and data structures that are being used by the module.

In Version B, the controller hides the validation logic by putting this in a separate source code file imported only by the controller. Likewise, the view hides the functions used to input and output data to the user and the model hides the *global book* data structure and associated functions. This *hiding* is done by putting these functions and global variables in separate modules and importing these only as needed. All of this improves information hiding when compared to Version A.

As mentioned above, the contact_data module is imported by all three components. This gives each component the *potential* to directly access the *global contactData* variable. However, the design and implementation avoids doing this by calling functions defined in the contact_data module as a way to manipulate the global variable. This is also an improvement in information hiding when compared with Version A.

16.4.2.5 Performance

The performance of the MVC Version B design is the same as the performance of Version A. Specifically, the use of a Python dictionary gives us constant time performance.

16.4.2.6 Security

The security of the MVC Version B design is the same as the security of Version A. Specifically, the design includes data input validation, data output validation, exception handling, and fail-safe defaults. In addition, Python is a type-safe language.

16.5 SD Top-Down Design Perspective

We'll continue to use the personal finances case study to reinforce Model–View–Controller via a top-down design approach.

16.5.1 SD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 13, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

Below, we will compare our top-down MVC software design with Chap. 13 designs, which focused on the domain of personal finances and ignored implementation details related to user interface and persistent storage.

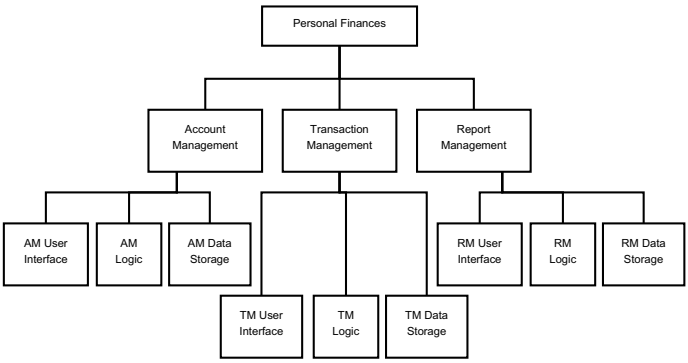


Fig. 16.16 Personal finance hierarchy chart using MVC

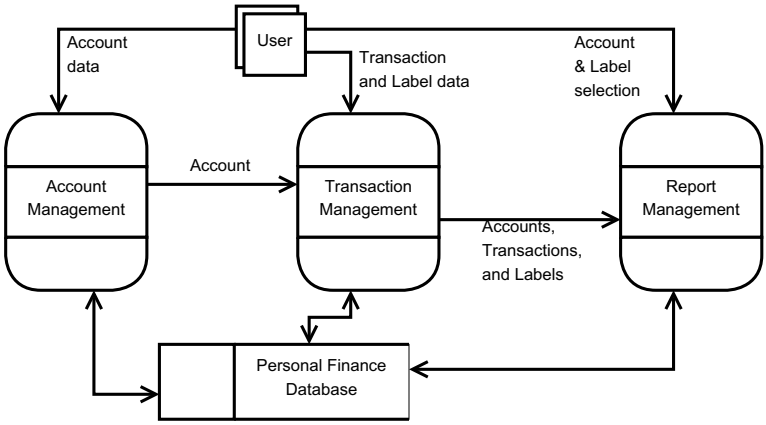


Fig. 16.17 Personal finance data-flow diagram (from Chap. 13)

16.5.2 SD Personal Finances: Structure

The hierarchy chart in Fig. 16.16 shows the personal finance application split into three domain functions. Each of these domain functions is further split into a function that supports a user interview (i.e., the view), processing logic (i.e., the controller), and data storage (i.e., the model).

16.5.3 SD Personal Finances: Structure and Behavior

The Data-Flow Diagram (DFD) used in Chap. 13 is shown in Fig. 16.17. This diagram is already at an appropriate level of abstraction. Modifying this diagram to show Model–View–Controller would not add any information that is not already being expressed in the hierarchy chart.

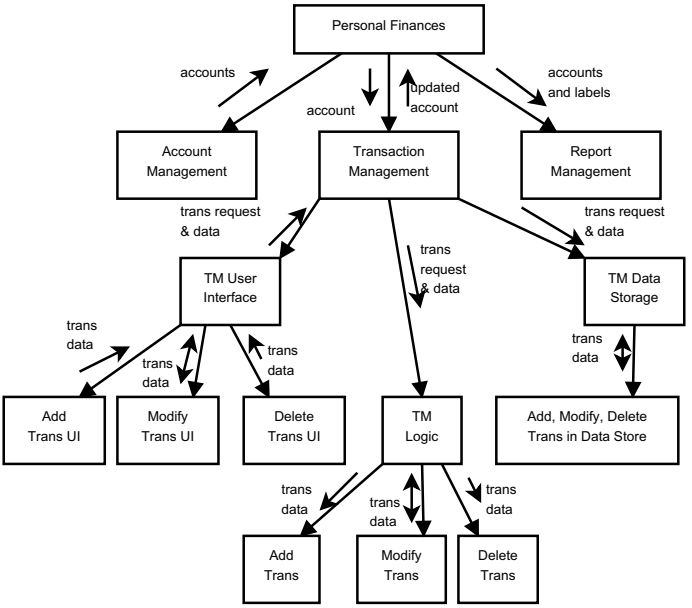


Fig. 16.18 Personal finance structure chart using MVC

The structure chart in Fig. 16.18 shows the flow of information for the Transaction Management function. This structure chart follows the design shown in the hierarchy chart in Fig. 16.16. The user interface functions for getting transaction data are all a part of the *TM User Interface* function, the controller functions are all a part of the *TM Logic* function, and the model functions are all a part of the *TM Data Storage* function.

The structure chart in Fig. 16.19 shows a different series of function calls within the Transaction Management function. In this design, the TM controller functions *Add Trans*, *Modify Trans*, and *Delete Trans* call user interface functions to obtain and/or display data then call a model function that will add, update, or delete a transaction from the data store.

These two structure charts show alternative designs for how the functions in the three MVC components may interact with each other. With a top-down design approach, you should consider alternative designs until you have enough details to settle on a design that best meets the requirements and design criteria.

16.5.4 SD Personal Finances: Behavior

The statechart diagram described in Chap. 13 shows the user interactions with the personal finance application. Since changing the software design to use Model–View–Controller has no impact on the user interface design, there is no need to

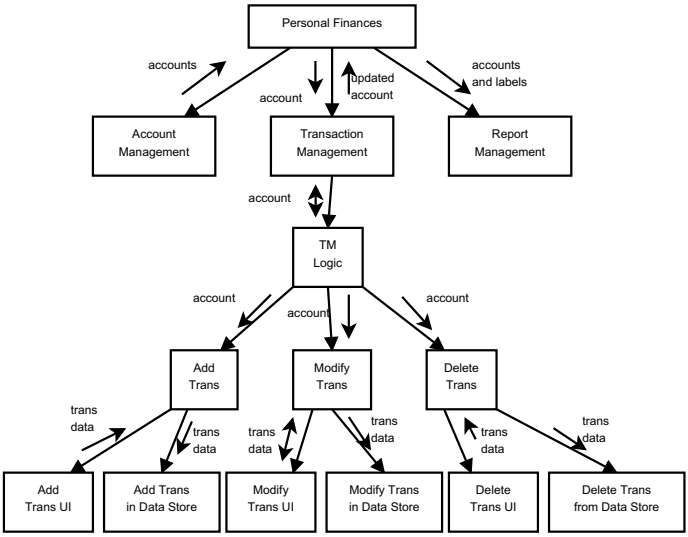


Fig. 16.19 Personal finance structure chart using MVC (alternative design)

modify the statechart diagram. That is, the user interface behavior has not been changed even though the internal structure of the application code has gone through a dramatic change!

16.5.5 SD Personal Finances: Summary

16.5.5.1 Design Models

The design models shown in Figs. 16.16, 16.18, and 16.19 show the impact Model–View–Controller has on the structure and interactions of the functions. As discussed in Chap. 13, a software architect may want/need to emphasize only certain aspects of the design. For example, if an understanding of the design structure is critical to the success of the personal finance software, then the hierarchy and structure charts are important while the DFD and statechart may be omitted. On the other hand, perhaps showing the interactions of the personal finance system with a user and persistent storage is important. In this case, the DFD and statechart would be important to develop.

16.5.5.2 Evaluation

An evaluation of the personal finances design, as shown in Figs. 16.16, 16.17, 16.18, and 16.19, is briefly described below.

Simplicity Since we have described a high-level design, or architecture, our efforts should have resulted in models that accurately reflect the needs/requirements while also abstracting away the details implied by the requirements. The hierarchy,

DFD, and statechart models are simple to read while the structure chart provides significantly more details that, perhaps, should not be included in a very high-level design.

Coupling The DFD and structure charts provide information to help us assess the coupling between the design elements. In the case of these two models, we see the interactions between the MVC and domain components as described by the requirements. These couplings appear to be necessary given the list of requirements. Both types of models emphasize the three primary processes involved in the personal finance software system.

Cohesion As discussed above for the hierarchy, DFD, and structure charts, the components expressed in these models represent the domain and MVC components needed to design and implement personal finance software. Given the use of MVC, we would expect the cohesion of each component to be high.

Information hiding Based on the use of MVC in this high-level design, we can expect information hiding to be effectively implemented between the three components.

Performance Without knowing more details about the data structures to be used, it is difficult to assess the performance of this high-level design. In addition, the *Personal Finance Database* shown on the DFD may have performance implications that cannot be assessed at this time.

Security The case study requirements do not state any need for information security. However, designers and implementers should always consider security even when this is not explicitly stated in the requirements. The high-level design models above do not address data input validation, data output validation, exception handling, fail-safe defaults, or type-safe languages.

16.6 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.

- You have created and/or modified models that describe a structured software design. This includes designing using a bottom-up and top-down approach.

Exercises

Discussion Questions

1. How is coupling between modules improved with a good implementation of MVC?
2. How is cohesion within a module improved with a good implementation of MVC?
3. How is information hiding improved with a good implementation of MVC?

Hands-on Exercises

1. Use an existing code solution that you've developed, develop design models that show how your solution could be redesigned to use MVC. Apply the software design criteria to your design models. How good or bad is your design?
2. Use your development of an application that you started in Chap. 4 for this exercise. Modify your design to use MVC and then evaluate your MVC design using the six software design criteria.
3. Continue Hands-on Exercise 3 from Chap. 13 by changing your software design to use MVC. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 13 for details on each domain.

    - Airline reservation and seat assignment,
    - Automated teller machine (ATM),
    - Bus transportation system,
    - Course-class enrollment,
    - Digital library,
    - Inventory and distribution control,
    - Online retail shopping cart,
    - Personal calendar, and
    - Travel itinerary.

Part III
Software Design Perspectives

This third part introduces five additional software design topics. Four of these five topics are covered independent of each other, allowing the learner to cover these topics in any order they choose. The design patterns topic is the exception; the design patterns chapters refer to chapters on HCI and secure design. A brief description of the topics and chapters is below.

- Human–computer Interaction Design
  Read Chaps. 17, 18 (OOD) and 19 (SD) to learn about a text-based user interface. Read Chaps. 17, 20, 21 (OOD), and 22 (SD) to learn about a graphical-based user interface.
- Quality Assurance
  Chapter 23 covers important quality assurance topics. To help you create better designs, you should read this chapter!
- Secure Design
  Chapter 24 covers important security design principles, while Chaps. 25 (OOD) and 26 (SD) apply these principles to the case studies.
- Design Patterns
  Chapter 27 covers design patterns, while Chaps. 28 (OOD) and 29 (SD) apply design patterns to the case studies. The OOD design patterns chapter refers to designs found in Chaps. 15, 18, 21, and 25. The SD design patterns chapter refers to designs found in Chaps. 16, 19, 22, and 26.
- Persistent Data Storage Design
  Chapter 30 covers data modeling and Chap. 31 introduces XML and relational databases, two common persistent storage technologies. Chapters 32 (OOD) and 33 (SD) apply these concepts to the case studies.



Introduction to Human–Computer Interaction (HCI) Design

17

The objective of this chapter is to introduce design concepts associated with human–computer interaction (HCI).

17.1 Preconditions

The following should be true prior to starting this chapter.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_17

17.2 Concepts and Context

Designing a user interface is hard, for several reasons. First, evaluating a user interface is subjective. For example, which of the following search web-page designs do you like better? [1]

- www.dogpile.com
- www.duckduckgo.com
- www.botw.org

The *dogpile* search page gives you some choices about the type of content you would like to search for—web, images, video, news, and shopping. The *duckduckgo* search page provides only a text box for entry of a search term—no search choices are provided. In contrast, the *botw* search page shows a directory structure on its search page. Do you like the minimalist design of duckduckgo? Or do you like the search choices presented on the dogpile page? Or do you like the option of searching through a directory structure as provided by *botw*?

Second, the term *user friendly* is vague but yet is often used to describe a good user interface. A person may look at a web page and conclude that *they really like this user interface because it is user friendly*. Another person may look at the same web page and conclude that *this user interface is not user friendly*. This term is used in a subjective manner and supports the popular saying: *beauty is in the eye of the beholder*.

Finally, different HCI technologies make it difficult to compare designs across platforms. Designing a user interface for a smartphone is different from designing a web page which is different from designing a software application that will run on a personal computer. Each of these types of applications uses different technologies, some of which (e.g., touch screen devices) have a significant impact on the HCI design. Even within the same set of technologies, there are differences. For example, designing a web page that is used for data entry (e.g., a web page form) is different from designing a web page that displays content to the user.

17.2.1 HCI Evaluation Criteria

What we need is a way to assess an HCI design using criteria that provides structure and consistency to this process. The following four criteria provide a way to assess HCI designs while also highlighting design aspects that are important for designers to think about as they develop a user interface [1].

- Efficiency—How does the HCI affect the productivity of the user?
- Learnability—How easy is it for a user to learn the HCI?

[1] The comparison of the following three web pages is based on their appearance on August 11, 2019.

- User satisfaction—How satisfied is the user with the HCI?
- Utility—Does the HCI provide useful and timely information?

17.2.1.1 Efficiency

Some questions to ponder that pertain to the efficiency of an HCI design.

1. What type of input device must be used to enter data?
   When entering text-based data, a keyboard is the traditional device but pointing devices and touch screen technology offer other options. When selecting an option from a list, a pointing device is the traditional device but keyboard (via function keys, hot keys, or arrow keys) and touch screen technology offer other options.
2. Within one screen/page, how much information is related to content versus data entry versus navigation?
   In the context of this question, content refers to static information while data entry and navigation require some form of user action/input.
3. What type of input device must be used to navigate?
   When the screen/page includes both text-based data entry and navigation options, hand movements between the keyboard and pointing device will slow down the user. In this scenario, a pointing device (e.g., mouse) tends to be slower than a function key or hot key that initiates the same action.
4. How much information needs to be displayed on one screen/page?
   Scrolling the screen/page because all of the information cannot be viewed (at the same time) on the display device will slow down the user. However, having multiple screens/pages so that scrolling is not necessary requires additional navigation actions from the user.

17.2.1.2 Learnability

Some questions to consider that pertain to the learnability of an HCI design.

1. Does the design use standard user interface objects and metaphors?
   With the creation of many different operating systems come many different user interface designs. The notion that a user interface standard exists and is used by all of the companies that produce an operating system is a pipe dream. Given this, there are some pseudo-standards that HCI designers should use. For example, the appearance and behavior of a text box for entry of text-based data or the appearance and behavior of a drop-down list should conform to the pseudo-standard that exists across all of the operating systems (and their mostly proprietary user interface platforms).
2. Does the user interface look different for different types of users?
   Some users of an application may have more authority and thus can see more information. A simple example of this is a payroll application at a company. A payroll clerk will not have the ability to see as much data as the payroll manager. Do these different views into the information get presented in a similar

way? That is, a payroll clerk gets promoted to payroll manager. Does this person need to relearn the entire system or is it a simple matter that some additional fields/information is now displayed to the individual?
3. Does the user interface require instructions on most screens/pages?
   Excessive amounts of instruction on a screen/page suggest that the HCI design is not easy to learn and use.
4. Is the user interface consistent across all of its screens/pages?
   Viewing and entering data, and navigating around an application, should be consistent across all of its screens/pages. For large organizations that have lots of custom software, every attempt should be made to make the user interface consistent across all of these applications.

17.2.1.3 User Satisfaction

Some questions to contemplate that pertain to the user satisfaction of an HCI design.

1. How do we balance design choices when a group of users with similar needs have different opinions about what they like and don't like in an HCI?
   A compromise between different opinions may produce the best design, or it may be worse than an HCI design that conforms to a majority (or minority) opinion.
2. How do we make design choices when the users are unknown?
   We may be designing version one of a software product. No users exist, so a user-centered design approach may not work.

17.2.1.4 Utility

Some questions to think about when considering the utility of an HCI design.

1. Does all of the information displayed pertain to the purpose of the screen/page?
   Information unrelated to the purpose of the screen/page may create too much clutter; the user cannot quickly find the pertinent information.
2. Do the series of screens/pages for an application have a logical flow to their use? Does the series of screens/pages correspond to the process steps being automated?
   Any mismatch between the HCI design and the underlying process being automated will make the application less useful.

17.2.1.5 A Simple Example

Let's say that a software application needs to (1) obtain a person's name and (2) display a list of people's names. Let's also assume that these two HCI needs would be designed into two separate screens/pages.

For the data entry of a person's name:

- Should they enter their full name into one text box? If yes, does the software expect this data to be entered in a specific format (e.g., *first middle last* versus *last, first middle*)?
- Should they enter their first name, middle name, and last name in separate text boxes? If yes, how should we sequence these three text boxes on the screen/page (e.g., *first middle last* versus *last first middle*)? Should the three text boxes be sequenced left-to-right or top-to-bottom within the screen/page?

For the display of a list of people's names:

- What format should be used to display the list of names (e.g., *first middle last* versus *last, first middle*)?
- Should the user have the option of sorting the list by first name, last name, or middle name?

For both the data entry and display:

- Does this application need to work in other countries/cultures? Instead of using *first middle last* as components of a name the application may need to use other terminology. Examples include *family-name given-name*, *surname given-name*, and the reverse of either of these examples.

17.2.2 Software and HCI Design Goals

When designing a software solution, the goal is to help a person or organization solve a problem. How well the software achieves this goal is largely dependent on the HCI design. Why? Because a user understands the capabilities and limitations of the software through the lens of its user interface. That is, the users' perception is that *the user interface is the software*.

Given the importance of the HCI design, what should someone tasked with developing a user interface think about as they consider HCI alternatives? An HCI designer should know the user, prevent user errors, optimize user abilities, and be consistent [2]. Each of these is discussed in more detail below.

17.2.2.1 Know the User

Knowing the user is challenging since this implies an understanding of the user's:

- True needs,
- Common activities, and
- Level of expertise in the problem domain and in information processing.

Understanding a user's true needs is hard since there may be different types of users where each type of user has slightly different needs. Effectively communicating user

needs may be challenging given the international nature of many software products, the differences in native languages, and the terminology being used may be specific to a business or industry. When the HCI designer has a different native language from the user, or does not understand the terminology being used by the business or industry, miscommunication of user needs will inevitably occur.

Understanding the user's common activities may be difficult when there are users that need to use the software for different purposes. How does the HCI design enable these different uses without adding unnecessary layers of complexity?

Understanding the user's level of expertise, whether this is related to the problem domain or in general information processing, may result in the need to design different types of HCIs to address these different levels of expertise.

17.2.2.2 Prevent User Errors

Whenever possible, a user interface should be designed to minimize possible user mistakes. For example, there may be a situation where there are only a few valid options the user may choose from. In this case, presenting these choices in a way that prevents the user from making an erroneous selection would be a good design choice.

For a text-based user interface, there may be a situation where there are only a few valid options. In this case, presenting a menu of choices and allowing the user to enter a letter or digit related to the menu choice being selected should minimize the possibility of the user making a mistake. Similarly for a graphical-based user interface, a small group of radio buttons or a drop-down list with fixed choices should minimize the possibility of user error.

When a user does make a mistake, the way in which the error message is displayed and how the error message is worded are additional areas where a design choice is important. An error message should guide the user to the source of the error without requiring lots of explanation.

17.2.2.3 Optimize User Abilities

Not all users of a software application are alike. Some may be more familiar with software systems than others. A power user is someone that has significant experience and knowledge using software systems. Designing a user interface that meets the needs of power users as well as individuals that have no experience using software systems or knowledge of the application domain is very challenging. For this reason, an HCI design should provide multiple ways to accomplish a task whenever this is reasonable. For example, shortcuts may be provided for power users while a wizard that provides a step-by-step walk through of a process could be used by less experienced users. Providing a help facility within a software application should be done so a user may obtain help based on their current context without making this facility too intrusive.

17.2.2.4 Be Consistent

Users get used to things appearing and working in certain ways. The use of metaphors (e.g., a trash can for deleting data) should be used in a consistent and common-sense manner. Colors should be used consistently to indicate similar types of information or processing. For example, reporting errors to a user should be done using a consistent color scheme. When multiple screens must be used within a software system, placing content that appears on each of these screens in the same relative location gives the user a quick way to find this content regardless of which screen they are using. Error messages should appear in the same relative location when multiple screens are being used. Having error messages displayed via a pop-up window in some cases and displayed within a screen in other cases may cause confusion.

17.2.3 HCI Design Steps

An overview of the design steps in developing a user interface design is now described [2].

- Gather user needs related to HCI.
- Develop and test HCI prototypes.
- Select HCI prototypes for further design and implementation.

The first two steps may be done with users in an interactive fashion. When combining the first two steps, these sessions should focus on the key interactions between the user and the software. When a large number of screens/pages are needed in the HCI design, particular attention should be spent discussing ways to navigate between these screens/pages. Depending on the types of prototyping tools you use, the selected prototypes may be translated quickly into a more detailed HCI design and then into an implementation. Another option is to simply use pencil and paper (or markers and whiteboard) to create various design layouts.

17.3 Post-conditions

The following should have been learned when completing this chapter.

- Evaluating an HCI design should include thinking about the design using at least four criteria:

1. Efficiency—How does the HCI affect the productivity of the user?
2. Learnability—How easy is it for a user to learn the HCI?
3. User satisfaction—How satisfied is the user with the HCI?
4. Utility—Does the HCI provide useful and timely information?

- An HCI designer should know the user, prevent user errors, optimize user abilities, and be consistent.
- User participation in the HCI design process is critical to developing a user interface that satisfies the four criteria listed above.

Exercises

Discussion Questions

1. Some people will prefer to use a website that offers many different types of content on a single page, while others may prefer using a website that contains information on a single topic. Can the four HCI evaluation criteria help explain these different preferences? If so, how?
2. Have you used a software application that reports errors in different ways? Has this inconsistency led to confusion about how you use the application? If so, explain.
3. Two HCI designs exist—one displays information while the other allows a user to enter lots of data. What differences might there be in the use of the four HCI evaluation criteria in designing these two types of HCI's?

Hands-on Exercises

1. Use an existing code solution that you've developed, evaluate its HCI design using the four evaluation criteria. How good or bad is your design?
2. Continue Hands-on Exercise 3 from Chaps. 12 or 13 by developing an HCI design and then evaluating your design using the HCI criteria described in this chapter. You may sketch possible HCI designs using paper and pencil or use an HCI prototyping tool if one is available. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chaps. 12 or 13 for details on each domain.

    - Airline reservation and seat assignment,
    - Automated teller machine (ATM),
    - Bus transportation system,
    - Course-class enrollment,
    - Digital library,

- Inventory and distribution control,
- Online retail shopping cart,
- Personal calendar, and
- Travel itinerary.

References

1. The Joint Task Force on Computing Curricula (2008) Computer science curriculum 2008: an interim report of CS 2001. ACM and IEEE Computer Society, New York
2. The Joint Task Force on Computing Curricula (2013) Computer science curricula 2013: curriculum guidelines for undergraduate degree programs in computer science. ACM and IEEE Computer Society, New York



18 OOD Case Study: Text-Based User Interface

The objective of this chapter is to apply the human–computer interaction (HCI) design concepts discussed in Chap. 17 to develop a better text-based HCI.

18.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand four HCI design criteria: efficiency, learnability, user satisfaction, and utility. You have evaluated user interface designs using these criteria.
- You understand that the process of creating an HCI design is critical for developing a user interface that satisfies the four criteria listed above. While developing an HCI design, the designer should know the user, prevent user errors, optimize user abilities, and be consistent.
- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_18

- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

18.2 OOD: Concepts and Context

Even though most modern software use a touch screen and/or graphics-based user interface (UI), there is still a need to use a more primitive text-based HCI. Examples of text-based user interfaces (TUI) include the *Terminal* window in Unix-like systems, the *Command Prompt* window in Windows-based systems, configuration of operating system components (e.g., BIOS settings), and many text-based games that were popular in the 1980s.

18.2.1 OOD: TUI Design Alternatives

There are a few common text-based UI approaches that may be combined to create a text-based software application. The alternative TUI design approaches include the following.

Guided Prompts: The user is prompted to enter a specific data value. This is a common design choice when the software needs to obtain data from the user. The ABA case study seen so far uses this approach.

Menus: A list of menu choices is displayed to the user followed by the user entering a letter or digit matching one of the menu choices. Once a valid menu choice has been entered, the associated processing is initiated. Many of the TUIs designed to configure an operating system component will utilize menus.

Commands: A command prompt is displayed to the user allowing the user to enter a command. Once a valid command has been entered, the associated processing is initiated. The Terminal window (in Unix-like systems) and the Command Prompt window (in Windows systems) are examples of this approach.

18.3 OOD: ABA TUI Designs

The current TUI design provides an inflexible interface to the user. The user must first create new entries in the address book. Only after exiting *data entry mode* does the user see the contents of the address book. This inflexible user interface design is shown in the state machine diagram from Chap. 15, shown in Fig. 18.1.

Two text-based UI designs are described below. Both provide the user with the flexibility to switch between creating new address book entries and displaying existing address book entries. The first design combines menus with guided prompts for

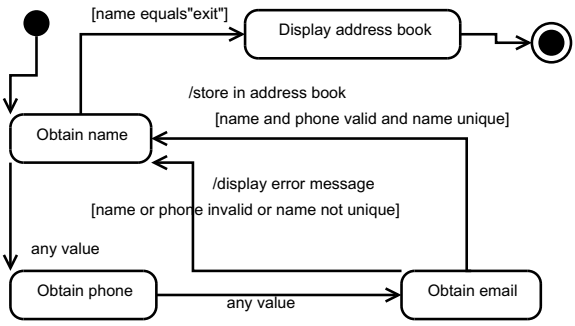


Fig. 18.1 State machine diagram—ABA Version B Chap. 15

data entry while the second design combines commands with guided prompts for data entry.

18.3.1 OOD: Menu and Guided Prompts

The sample user interactions shown in Listing 18.1 uses a menu to display the choices available to the user and guided prompts for entry of data values. This particular example shows the user creating a new address book entry, displaying the just created address book entry, and then exiting the application.

Listing 18.1 Sample ABA Menu & Data Entry Prompts

```
C   Create address book entry
D   Display address book entry
X   Exit
Choice: C

Enter contact name: Santa Claus
Enter phone number for Santa Claus: 3155559876
Enter email address for Santa Claus: santa@claus.org

C   Create address book entry
D   Display address book entry
X   Exit
Choice: d

Enter contact name: Santa Claus
The phone number for Santa Claus is 3155559876
The email address for Santa Claus is santa@claus.org

C   Create address book entry
D   Display address book entry
X   Exit
Choice: x
```

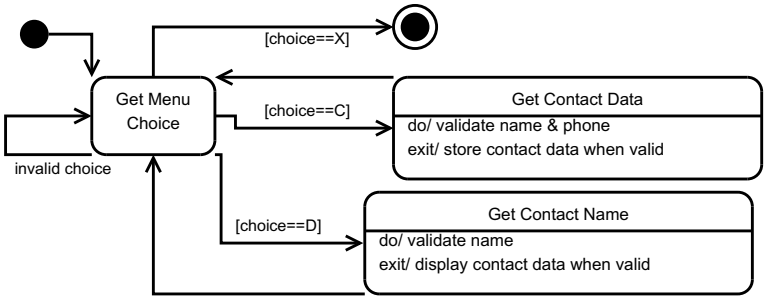


Fig. 18.2 State machine diagram—ABA menu and data entry prompts

The state machine diagram in Fig. 18.2 shows how the use of a menu provides more flexibility to the user in terms of switching between creating a new address book entry and displaying existing entries. The *Get Menu Choice* state will validate the menu choice entered by the user. When a valid menu choice is entered, the user is transitioned to a state that is appropriate for the valid menu choice. After the user has either done *Get Contact Data* or *Get Contact Name*, the user transitions back to the *Get Menu Choice* state to allow the user to decide what to do next.

18.3.2 OOD: Commands and Guided Prompts

The sample user interactions shown in Listing 18.2 uses a command prompt to obtain commands from the user and guided prompts for entry of data values. This particular example shows the user requesting help, creating a new address book entry, displaying the just created address book entry, and then exiting the application.

Listing 18.2 Sample ABA Commands and Data Entry Prompts

```
command> help
All commands are case insensitive.
Enter 'C' or 'create' to create an address book entry.
Enter 'D' or 'display' to display an address book entry.
Enter 'H' or 'help' to display this help text.
Enter 'X' or 'exit' to exit.

command> C
Enter contact name: Santa Claus
Enter phone number for Santa Claus: 3155559876
Enter email address for Santa Claus: santa@claus.org

command> display
Enter contact name: Santa Claus
The phone number for Santa Claus is 3155559876
The email address for Santa Claus is santa@claus.org

command> X
```

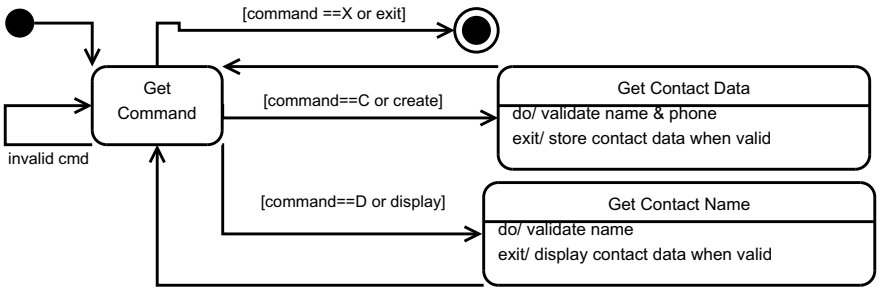


Fig. 18.3 State machine diagram—ABA command and data entry prompts

The state machine diagram shown in Fig. 18.3 shows how the use of a command prompt provides more flexibility to the user in terms of switching between creating a new address book entry and displaying existing entries. The *Get Command* state will validate the command entered by the user. When a valid command is entered, the user is transitioned to a state that is appropriate for the valid command. After the user has either done *Get Contact Data* or *Get Contact Name*, the user transitions back to the *Get Command* state to allow the user to decide what to do next. This design includes a command to display help information.

18.3.3 OOD: Evaluate ABA TUI Designs

To assess the three TUI designs—the Chap. 15 Version B design and the two alternative designs described above—we will focus on those aspects of the designs that differ from each other. The entry of contact data to create an address book, which is the same in all three designs, will be ignored in the descriptions below.

18.3.3.1 Efficiency

How does the HCI design affect the productivity of the user? Table 18.1 summarizes the differences in efficiency. The flexibility provided by the two designs presented in this chapter allows the user to more efficiently use the ABA. The entry of a menu choice or command does not adversely affect the assessment that the two designs introduced in this chapter are more efficient than the Chap. 15 design.

18.3.3.2 Learnability

How easy is it for a user to learn the HCI? Table 18.2 summarizes the differences in learnability. The Chap. 15 UI and the menu and guided prompts designs are easier to learn than the command and guided prompts design.

Table 18.1 Efficiency comparison

Chapter 15 UI	When a user wants to use the ABA to display existing contact information, they must enter "exit" as a contact name, as shown in Fig. 18.1
Menu and guided prompts	A user must enter a menu choice but can switch between creating and displaying contact data as often as they would like, as shown in Fig. 18.2
Commands and guided prompts	A user must enter a command but can switch between creating and displaying contact data as often as they would like, as shown in Fig. 18.3

Table 18.2 Learnability comparison

Chapter 15 UI	The sequential nature of this design, i.e., a user must first create address book data and explicitly "exit" before displaying address book data, makes this design easy to learn how to use
Menu and guided prompts	The menu choices self-document the options available to the user, making this design easy to learn how to use
Commands and guided prompts	A power user would know the commands and not need the help facility. Someone first learning or rarely using the ABA would likely need to display the help information a few times before remembering the commands

18.3.3.3 User Satisfaction

How satisfied is the user with the HCI? In order to assess this HCI design criteria, we would need to interview or survey users to obtain information on how satisfied they are with each of the design alternatives. Since the ABA has not been deployed as a software application that others may use, this criteria cannot be assessed.

18.3.3.4 Utility

Does the HCI provide useful and timely information? Table 18.3 summarizes the differences in utility. The flexibility provided by the two designs presented in this chapter allows the user to use the ABA in a manner that is more useful to them by providing timely access to information.

18.3.4 OOD: Commands Design Revisited

The TUI design that uses commands is particularly useful when many of your users are considered power users. In this case, we can extend the definition of the commands to include the appropriate address book data. The sample user interactions shown

Table 18.3 Utility comparison

Chapter 15 UI	The sequential nature of this design, i.e., a user must first create address book data and explicitly "exit" before displaying address book data, results in the ABA being of little use
Menu and guided prompts	The flexibility afforded a user in switching between creating and displaying contact data as often as they would like makes this design useful
Commands and guided prompts	The flexibility afforded a user in switching between creating and displaying contact data as often as they would like makes this design useful

in Listing 18.3 use more complex commands, allowing a command to optionally include user-supplied data typically obtained via the guided prompts.

Listing 18.3 Sample ABA Commands Revisited

```
command> help
All commands are case insensitive.
Command parameters in [square brackets] are optional.
Can use single (') or double quotes (") to delimit a contact name.
To create an address book entry: c ['name'] [phone] [email]
                                 create ['name'] [phone] [email]
To display an address book entry: d ['name']
                                  display ['name']
To display this help text: h
                           help
To exit: x OR exit

command> c 'Santa Claus' 3155559876 santa@claus.org

command> c "Jane Claus"
Enter phone number for Jane Claus: 3155551209
Enter email address for Jane Claus: jane@claus.org

command> display "Santa Claus"
The phone number for Santa Claus is 3155559876
The email address for Santa Claus is santa@claus.org

command> x
```

18.3.4.1 Evaluating Commands Design Revisited

The evaluation of this updated version of the Commands and Guided Prompts design is described below.

Efficiency: The efficiency of this updated commands design is superior to any of the other three designs since a *create* or *display* command may include all of

the necessary user-supplied address book data values. This eliminates the guided prompts for obtaining a name, phone number, and email address.

Learnability: This updated commands design is harder to learn than the other three designs since the commands are more complex. As shown when submitting a *help* command, the help information is much more complex than the earlier version of the commands design. As already mentioned, when you have users that are comfortable entering more complex commands (i.e., you have power users), this updated commands design may be your best alternative for implementation.

Utility: This updated commands design is just as useful and provides as timely information as the two designs presented earlier in this chapter.

18.4 OOD Case Study: TUI Design Details

A requirement has been added to the ABA, shown in the listing below in *italics*.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. *Allow for display of contact data in the address book.*

A text-based user interface requires the software to guide the user through the steps needed to complete a processing step. For example, the *Get Contact Data* state in the state machine diagrams in Figs. 18.2 and 18.3 forces the user to enter a contact name, then a phone number, and then an email address. The TUI design does not allow the entry of these values in any other order. This sequence of data entry steps is shown in Fig. 18.4 and illustrates the inflexible nature of text-based user interfaces where the software guides the user through the necessary user interactions. This diagram shows examples of using the *entry/* and *do/* actions within a state. The transition from the three named states each has a trigger that causes the transition to the next state.

The class diagram in Fig. 18.5 shows the changes to the ABA to implement the menu and guided prompts text-based user interface described above. The ABA_contactData class continues to have a relationship with all three MVC components, while the new ABA_A_viewRequest class only has a relationship with the view and controller components. The ABA_A_viewRequest class contains an enumerated type (REQUEST) which identifies the type of request the user has initiated

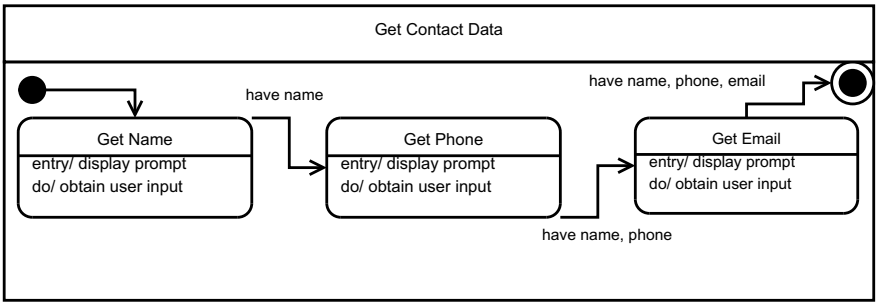


Fig. 18.4 State machine diagram—ABA Get Contact Data

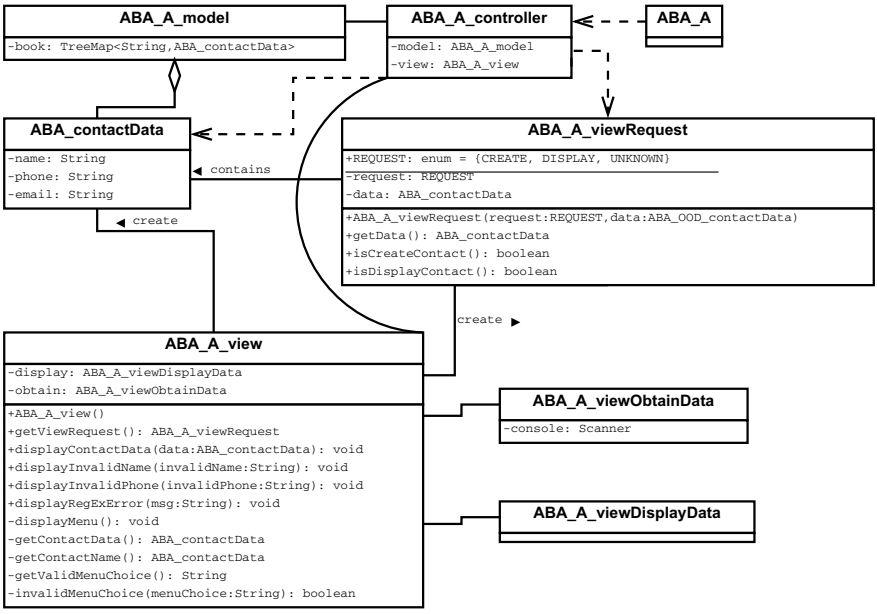


Fig. 18.5 Class diagram—ABA OOD Version A (menu and guided prompts)

via their menu selection. The getViewRequest method in the ABA_A_view class returns null to indicate the user has requested exiting the ABA.

As shown in the communication diagram in Fig. 18.6, the go() controller method now calls the getViewRequest() view method to get the user request and associated data. The getViewRequest() method displays the menu to allow the user to create a contact, display contact data, or exit the application. A ABA_OOD_A_viewRequest object is returned to the controller to indicate whether the user has requested creating a contact (request instance variable is REQUEST.CREATE) or displaying a contact (request instance variable is REQUEST.DISPLAY).

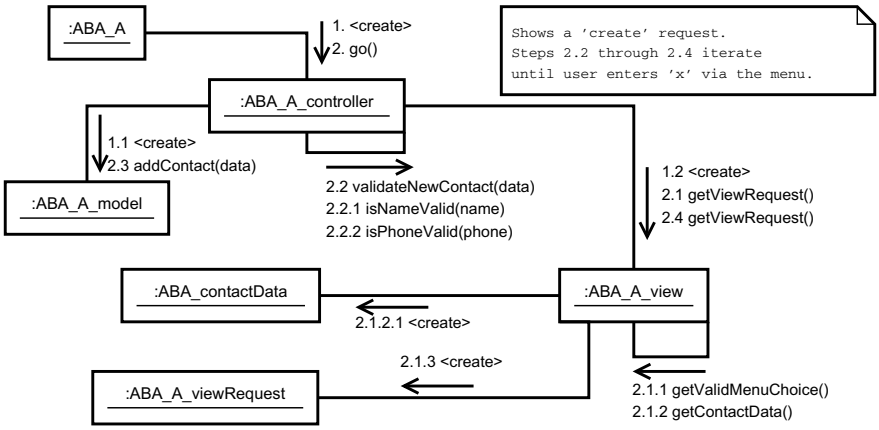


Fig. 18.6 Communication diagram—ABA OOD Version A (menu and guided prompts)

Figure 18.6 shows a create request, where the data instance variable in the ABA_A_viewRequest object contains the name, phone, and email values entered by the user. This data is first validated by the controller, and then if valid, given to the addContact model method to add this data to the address book. For a display request (not shown in the communication diagram), the data instance variable contains the name entered by the user. The name is first validated by the controller, and if valid, given to the getContact model method to obtain the contact data for this name. The ABA_contactData object returned by the model is then given to the view to be displayed.

18.5 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when creating a text-based user interface as part of a top-down design approach.

18.5.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.

- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

18.5.2 OOD Personal Finances: TUI Design Choices

The user interface design for the personal finances case study is potentially quite large. It needs to support user actions to create, read, update, and delete accounts, transactions, and reports. The number of user actions is large enough to warrant consideration of each text-based interface style discussed above.

A menu-based approach to providing a user with options seems like a reasonable approach. Table 18.4 shows two options for how the menu(s) could be designed. Option 1 shows a rather large menu listing all of the user actions. The user would enter a number (1 through 12) and then guided prompts would be used to obtain the data necessary for the option chosen. Option 2 has a much smaller main menu with just three options. Once the user enters a number (1 through 3), an appropriate submenu is displayed. The table shows the submenu for account management. Similar submenus would be displayed for transaction and report management. For this second menu option, efficiency could be improved by allowing the user to enter both the main menu and submenu choice at the `Enter choice:` prompt for the main menu. For example, guided prompts for updating a transaction would result from entering 2.3 at the main menu prompt.

A simple command-based user interface is shown in Listing 18.4. Having six commands seems like a reasonable design, one that could be learned fairly quickly. Four of these commands—create, delete, update, and view—would require a keyword be entered immediately after the command. The valid keywords would be account, transaction, and report to indicate which type of personal finances data entity is being referenced by the command.

Listing 18.4 Sample Personal Finances Commands and Data Entry Prompts

```
command> help
All commands are case insensitive
Enter 'C' or 'create' to create an account, transaction, or report
Enter 'D' or 'delete' to delete an account, transaction, or report
Enter 'H' or 'help' to display this help text
Enter 'U' or 'update' to update an account, transaction, or report
Enter 'V' or 'view' to view an account, transaction, or report
Enter 'X' or 'exit' to exit

command> C account
Enter account name: MyBank
Enter account type: savings
Enter starting balance: 100.00

command> delete transaction
Enter account name and type: MyBank credit-card
Enter transaction date: 03/23/2019
Enter transaction amount: 10.99
Do you want to delete the following transaction?
Date: 03/23/2019 Description: lunch Debit: 10.99
Enter Y or N: Y

command> X
```

Table 18.4 Personal finances menu design options

Menu Option 1	Menu Option 2 with submenu
1. Create account	1. Account management
2. View account	2. Transaction management
3. Update account	3. Report management
4. Delete account	X. Exit
5. Create transaction	Enter choice:
6. View transaction	
7. Update transaction	
8. Delete transaction	1. Create account
9. Create report	2. View account
10. View report	3. Update account
11. Update report	4. Delete account
12. Delete report	X. Exit
Enter choice:	Enter choice:

An example of a more complex command-based user interface is shown in Listing 18.5. The *KEY-VALUE-PAIR* shown in the help text represents a replacement for the

Table 18.5 Personal finances TUI design comparison

TUI Design	Efficiency	Learnability	Utility
Menu & Guided prompts	Maybe	Easy	Yes
Commands & Guided prompts	Yes	Moderate	Yes
Commands revisited	Yes	Hard	Yes

guided prompts used in the first two user interface designs described above. As shown in the two "create" examples, a KEY-VALUE-PAIR consists of a keyword (e.g., name, type, balance, account, desc, and debit) following by a colon and a data value (e.g., MyBank, savings, 100.00, lunch, and 10.99). This expanded version of a command-line interface would be much harder to learn when compared to the first two approaches.

Listing 18.5 Sample Personal Finances Commands Revisited

```
command> help create
All commands are case insensitive.
Command parameters in [square brackets] are optional.
Command parameters in {squiggly brackets} represent a grouping.
A grouping may be repeated using ellipses (i.e., ...).
Use single (') or double quotes (") to enclose data with spaces.
To create a data entity: c TYPE {KEY–VALUE–PAIR} ...
                         create TYPE {KEY–VALUE–PAIR} ...

command> C account name:MyBank type:savings balance:100.00

command> C transaction account:MyBank type:credit–card
         desc:lunch debit:10.99

command> X
```

18.5.3 OOD Personal Finances: Evaluate TUI Design Choices

Table 18.5 compares the three text-based UI design choices just presented. Assuming that the second option for the menu design is used, the ability to enter a number representing the choice from the main menu and submenu in one entry (e.g., 2.3) makes the menu design as efficient as the simple command-line option. The more complex command structure described in Listing 18.5 would be much harder for someone to learn, especially a user that occasionally uses the personal finances application.

Given everything discussed regarding a text-based user interface for personal finances, it seems reasonable to provide both the menu option 2 and simple command interfaces. Both of these approaches would use the same guided prompt design while allowing an occasional user the menu option 2 interface to efficiently use the

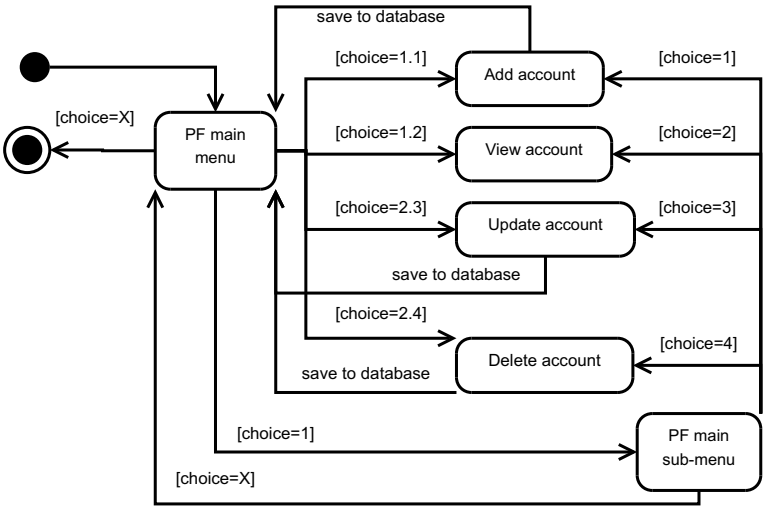


Fig. 18.7 State machine diagram—PF user interactions

application. While someone may argue that the more complex command structure as described in Listing 18.5 is the best choice for a power user, this user interface design includes design and implementation logic that is not usable by the other two approaches. Specifically, the parsing of a command must include logic for the KEY-VALUE-PAIR. This is unique to this approach. Combine this fact with learnability being hard for this design, and the conclusion drawn by the author is that the extra effort to design, build, and test the third design option is not worth the investment.

18.5.4 OOD Personal Finances: TUI Design Details

The statechart and UML class diagram in Figs. 18.7 and 18.8 show the modifications to the view component as a result of selecting a user interface design for the personal finances application.

The statechart shows the user interactions with the menu and submenu for an account. Similar behavior would exist for user interactions related to transactions and reports. Note the shortcut the user may elect to enter to get from the main menu directly to the guided prompts for adding, viewing, updating, or deleting an account. Entering *X* from the main menu would cause the personal finances application to end, while entering *X* from the submenu redisplays the main menu.

The UML class diagram contains a few significant changes when compared to the Chap. 15 class diagram shown in Fig. 15.15. First, an abstract class named viewUserData has been added to the design. Also note the inheritance relationships between viewAccount, viewTransaction, viewLabel, and viewReport and this new abstract class. This allows the view component to return user data related to any of these four data entities. This can be seen in the second significant change to the class

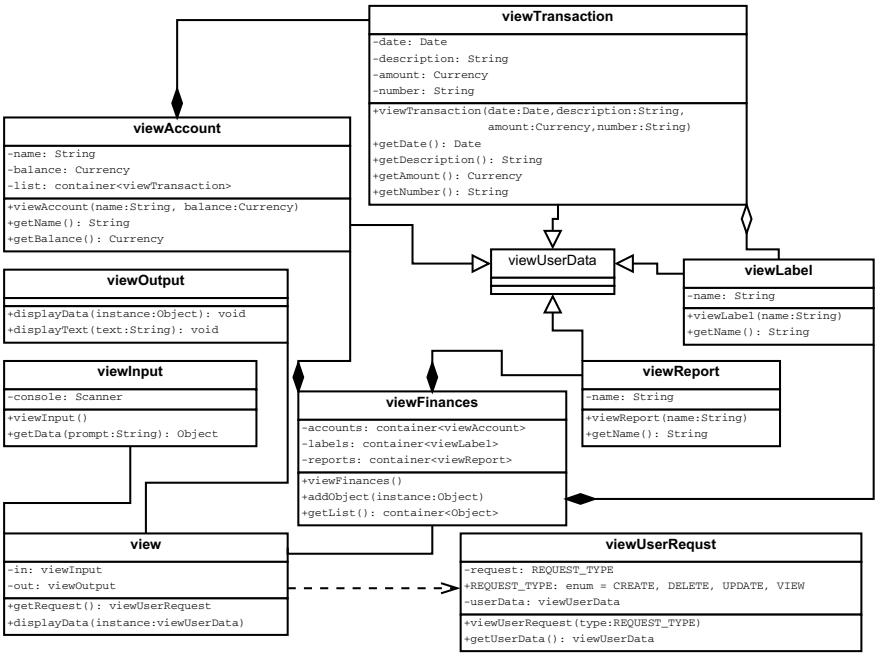


Fig. 18.8 Class diagram—PF view component

diagram, notably the getRequest public method in the view class. This method returns a viewUserRequest object that contains a viewUserData object. Along with the REQUEST_TYPE enumerated type defined in the viewUserRequest class, this allows the view to return a user request for creating, deleting, updating, or viewing either an account, transaction, label, or report.

18.6 Post-conditions

The following should have been learned when completing this chapter.

- A text-based UI should consider some combination of menus, commands, and guided prompts to provide a user with an efficient, useful, and easy to learn user interface.
- Using more complex commands in a TUI may be appropriate when many of your users are considered power users. Knowing your users is critical in determining whether this is a viable design option.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.

- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Discussion Questions

1. An ATM may not use a touch screen, instead it has buttons that get pushed to activate an option and uses a keypad to enter numbers. What type of text-based UI is this?
2. Can you think of any devices that use a text-based UI? If yes, what is the device and how would you describe this UI in terms of using prompts, menus, and/or commands? Evaluate this text-based UI using the HCI design criteria. Do you have any suggestions for improving this HCI design?

Hands-on Exercises

1. Use an existing code solution that you've developed, develop alternative text-based HCI design models that show ways in which a user could interact with your application. Apply the HCI design criteria to your design models. How good or bad is your design?
2. Use your development of an application that you started in Chap. 3 for this exercise. Modify your HCI design to use some combination of prompts, menus, and commands, and then evaluate your design using the HCI design criteria.

3. Continue hands-on exercise 3 from Chap. 12 by developing an HCI design using some combination of prompts, menus, and commands. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

   - Airline reservation and seat assignment
   - Automated teller machine (ATM)
   - Bus transportation system
   - Course-class enrollment
   - Digital library
   - Inventory and distribution control
   - Online retail shopping cart
   - Personal calendar
   - Travel itinerary



SD Case Study: Text-Based User Interface

19

The objective of this chapter is to apply the human–computer interaction (HCI) design concepts discussed in Chap. 17 to develop a better text-based HCI.

19.1 SD: Preconditions

The following should be true prior to starting this chapter.

- You understand four HCI design criteria: efficiency, learnability, user satisfaction, and utility. You have evaluated user interface designs using these criteria.
- You understand the process of creating an HCI design is critical to developing a user interface that satisfies the four criteria listed above. While developing an HCI design, the designer should: know the user, prevent user errors, optimize user abilities, and be consistent.
- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_19

- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

19.2 SD: Concepts and Context

Even though most modern software use a touch screen and/or graphics-based user interface (UI), there is still a need to use a more primitive text-based HCI. Examples of text-based user interfaces (TUI) include the *Terminal* window in Unix-like systems, the *Command Prompt* window in Windows-based systems, configuration of operating system components (e.g., BIOS settings), and many text-based games that were popular in the 1980's.

19.2.1 SD: TUI Design Alternatives

There are a few common text-based UI approaches that may be combined to create a text-based software application. The alternative TUI design approaches include the following.

Guided Prompts: The user is prompted to enter a specific data value. This is a common design choice when the software needs to obtain data from the user. The ABA case study seen so far uses this approach.

Menus: A list of menu choices is displayed to the user followed by the user entering a letter or digit matching one of the menu choices. Once a valid menu choice has been entered, the associated processing is initiated. Many of the TUIs designed to configure an operating system component will utilize menus.

Commands: A command prompt is displayed to the user allowing the user to enter a command. Once a valid command has been entered the associated processing is initiated. The Terminal window (in Unix-like systems) and the Command Prompt window (in Windows systems) are examples of this approach.

19.3 SD: ABA TUI Designs

The current TUI design provides an inflexible interface to the user. The user must first create new entries in the address book. Only after exiting *data entry mode* does the user see the contents of the address book. This inflexible user interface design is shown in the state machine diagram from Chap. 16, shown in Fig. 19.1.

Two text-based UI designs are described below. Both provide the user with the flexibility to switch between creating new address book entries and displaying existing address book entries. The first design combines menus with guided prompts for

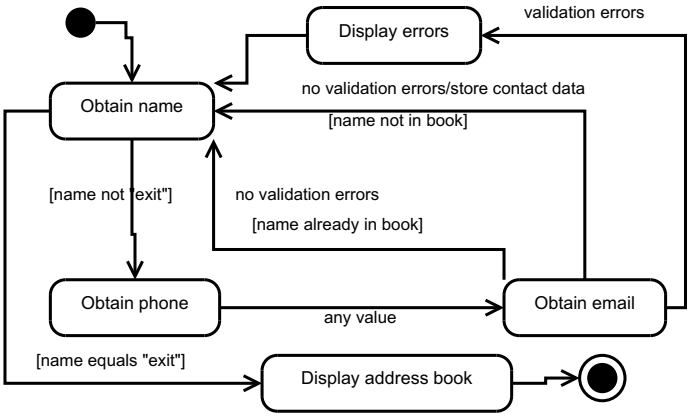


Fig. 19.1 State machine diagram—ABA Version B Chap. 16

data entry while the second design combines commands with guided prompts for data entry.

19.3.1 SD: Menu and Guided Prompts

The sample user interactions shown in Listing 19.1 uses a menu to display the choices available to the user and guided prompts for entry of data values. This particular example shows the user creating a new address book entry, displaying the just created address book entry, and then exiting the application.

Listing 19.1 Sample ABA Menu & Data Entry Prompts

```
C   Create address book entry
D   Display address book entry
X   Exit
Choice: C

Enter contact name: Santa Claus
Enter phone number for Santa Claus: 3155559876
Enter email address for Santa Claus: santa@claus.org

C   Create address book entry
D   Display address book entry
X   Exit
Choice: d

Enter contact name: Santa Claus
The phone number for Santa Claus is 3155559876
The email address for Santa Claus is santa@claus.org

C   Create address book entry
```

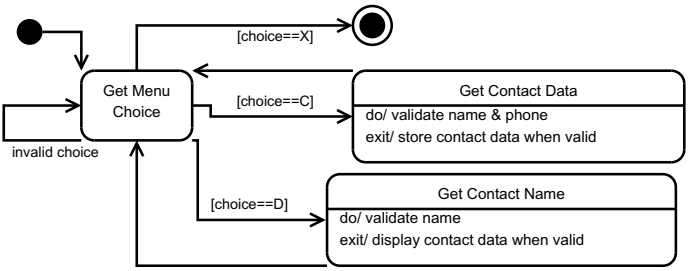


Fig. 19.2 State machine diagram—ABA menu and data entry prompts

```
D    Display address book entry
X    Exit
Choice: x
```

The state machine diagram in Fig. 19.2 shows how the use of a menu provides more flexibility to the user in terms of switching between creating a new address book entry and displaying existing entries. The *Get Menu Choice* state will validate the menu choice entered by the user. When a valid menu choice is entered, the user is transitioned to a state that is appropriate for the valid menu choice. After the user has either done *Get Contact Data* or *Get Contact Name*, the user transitions back to the *Get Menu Choice* state to allow the user to decide what to do next.

19.3.2 SD: Commands and Guided Prompts

The sample user interactions shown in Listing 19.2 uses a command prompt to obtain commands from the user and guided prompts for entry of data values. This particular example shows the user requesting help, creating a new address book entry, displaying the just created address book entry, and then exiting the application.

Listing 19.2 Sample ABA Commands & Data Entry Prompts

```
command> help
All commands are case insensitive.
Enter 'C' or 'create' to create an address book entry.
Enter 'D' or 'display' to display an address book entry.
Enter 'H' or 'help' to display this help text.
Enter 'X' or 'exit' to exit.

command> C
Enter contact name: Santa Claus
Enter phone number for Santa Claus: 3155559876
Enter email address for Santa Claus: santa@claus.org

command> display
Enter contact name: Santa Claus
The phone number for Santa Claus is 3155559876
```

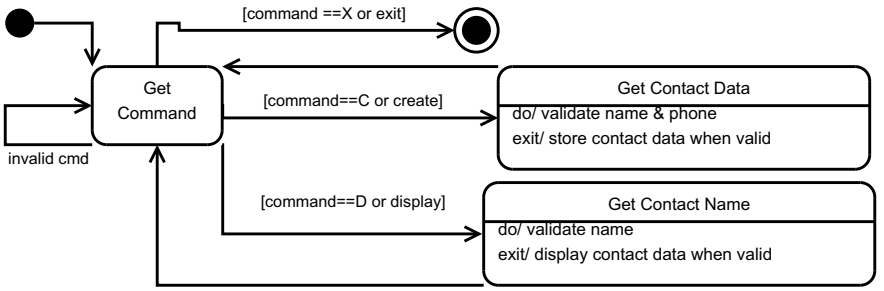


Fig. 19.3 State Machine Diagram—ABA command and data entry prompts

```
The email address for Santa Claus is santa@claus.org

command> X
```

The state machine diagram in Fig. 19.3 shows how the use of a command prompt provides more flexibility to the user in terms of switching between creating a new address book entry and displaying existing entries. The *Get Command* state will validate the command entered by the user. When a valid command is entered, the user is transitioned to a state that is appropriate for the valid command. After the user has either done *Get Contact Data* or *Get Contact Name*, the user transitions back to the *Get Command* state to allow the user to decide what to do next. This design includes a command to display help information.

19.3.3 SD: Evaluate ABA TUI Designs

To assess the three TUI designs—the Chap. 16 Version B design and the two alternative designs described above—we will focus on those aspects of the designs that differ from each other. The design elements that are similar, namely, the entry of contact data to create an address book, in all three designs will be ignored in the descriptions below.

19.3.3.1 Efficiency

How does the HCI design affect the productivity of the user? Table 19.1 summarizes the differences in efficiency. The flexibility provided by the two designs presented in this chapter allows the user to more efficiently use the ABA. The entry of a menu choice or command does not adversely affect the assessment that the two designs introduced in this chapter are more efficient than the Chap. 16 design.

Table 19.1 Efficiency comparison

Chapter 16 UI	When a user wants to use the ABA to display existing contact information, they must enter "exit" as a contact name, as shown in Fig. 19.1
Menu and guided prompts	A user must enter a menu choice but can switch between creating and displaying contact data as often as they would like, as shown in Fig. 19.2
Commands and guided prompts	A user must enter a command but can switch between creating and displaying contact data as often as they would like, as shown in Fig. 19.3

Table 19.2 Learnability comparison

Chapter 16 UI	The sequential nature of this design, i.e., a user must first create address book data and explicitly "exit" before displaying address book data, makes this design easy to learn how to use
Menu and guided prompts	The menu choices self-document the options available to the user, making this design easy to learn how to use
Commands and guided prompts	A power user would know the commands and not need the help facility. Someone first learning or rarely using the ABA would likely need to display the help information a few times before remembering the commands

19.3.3.2 Learnability

How easy is it for a user to learn the HCI? Table 19.2 summarizes the differences in learnability. The Chap. 16 UI and the menu and guided prompts designs are easier to learn than the command and guided prompts design.

19.3.3.3 User Satisfaction

How satisfied is the user with the HCI? In order to assess this HCI design criteria, we would need to interview or survey users to obtain information on how satisfied they are with each of the design alternatives. Since the ABA has not been deployed as a software application that others may use, this criteria cannot be assessed.

19.3.3.4 Utility

Does the HCI provide useful and timely information? Table 19.3 summarizes the differences in utility. The flexibility provided by the two designs presented in this chapter allows the user to use the ABA in a manner that is more useful to them by providing timely access to information.

Table 19.3 Utility comparison

Chapter 16 UI	The sequential nature of this design, i.e., a user must first create address book data and explicitly "exit" before displaying address book data, results in the ABA being of little use
Menu and guided prompts	The flexibility afforded a user in switching between creating and displaying contact data as often as they would like makes this design useful
Commands and guided prompts	The flexibility afforded a user in switching between creating and displaying contact data as often as they would like makes this design useful

19.3.4 SD: Commands Design Revisited

The TUI design that uses commands is particularly useful when many of your users are considered power users. In this case, we can extend the definition of the commands to include the appropriate address book data. The sample user interactions shown in Listing 19.3 use more complex commands, allowing a command to optionally include user-supplied data typically obtained via the guided prompts.

Listing 19.3 Sample ABA Commands Revisited

```
command> help
All commands are case insensitive.
Command parameters in [square brackets] are optional.
Can use single (') or double quotes (") to delimit a contact name.
To create an address book entry: c ['name'] [phone] [email]
                              create ['name'] [phone] [email]
To display an address book entry: d ['name']
                               display ['name']
To display this help text: h
                          help
To exit: x OR exit

command> c 'Santa Claus' 3155559876 santa@claus.org

command> c "Jane Claus"
Enter phone number for Jane Claus: 3155551209
Enter email address for Jane Claus: jane@claus.org

command> display "Santa Claus"
The phone number for Santa Claus is 3155559876
The email address for Santa Claus is santa@claus.org

command> x
```

19.3.4.1 Evaluating Commands Design Revisited

The evaluation of this updated version of the Commands and Guided Prompts design is described below.

Efficiency: The efficiency of this updated commands design is superior to any of the other three designs since a *create* or *display* command may include all of the necessary user-supplied address book data values. This eliminates the guided prompts for obtaining a name, phone number, and email address.

Learnability: This updated commands design is harder to learn than the other three designs since the commands are more complex. As shown when submitting a *help* command, the help information is much more complex than the earlier version of the commands' design. As already mentioned, when you have users that are comfortable entering more complex commands (i.e., you have power users), this updated command design may be your best alternative for implementation.

Utility: This updated commands design is just as useful and provides as timely information as the two designs are presented earlier in this chapter.

19.4 SD Case Study: TUI Design Details

A requirement has been added to the ABA, shown in the listing below in *italics*.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. *Allow for display of contact data in the address book.*

A text-based user interface requires the software to guide the user through the steps needed to complete a processing step. For example, the *Get Contact Data* state in the state machine diagrams in Figs. 19.2 and 19.3 forces the user to enter a contact name, then a phone number, and then an email address. The TUI design does not allow the entry of these values in any other order. This sequence of data entry steps is shown in Fig. 19.4 and illustrates the inflexible nature of text-based user interfaces where the software guides the user through the necessary user interactions. This diagram shows examples of using the *entry/* and *do/* actions within a state. The transition from the three named states each has a trigger that causes the transition to the next state.

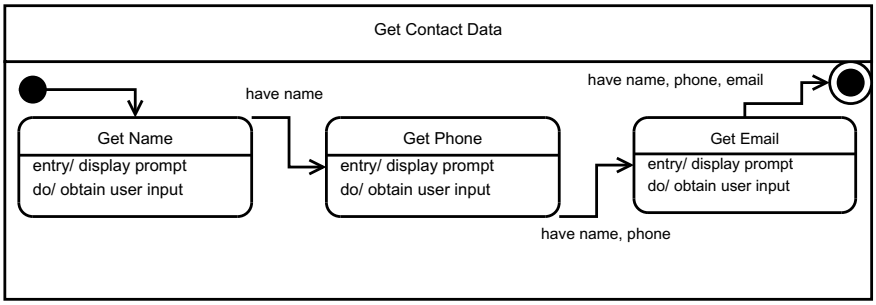


Fig. 19.4 State machine diagram—ABA Get Contact Data

Given the additional requirement to *allow for display of contact data in the address book*, as well as the HCI design change described above, the structure of the go function is quite different from the Chap. 16 Version B solution. The structure chart in Fig. 19.5 shows the structure of function calls resulting from the requirement and HCI design changes. The getViewRequest function displays a menu to allow a user to indicate whether they want to create a new contact, display an existing contact, or exit the application.

- Creating a new contact results in the getViewRequest function obtaining all three data values (i.e., name, phone, and email) from the user. In this scenario, the isCreateContact function would return true and the validateNewContact function is called. When this validation function returns true, the storeContactData function is called.
- Displaying an existing contact results in the getViewRequest function obtaining a contact name from the user. In this scenario, the isDisplayContact function would return true and the validateContactName function is called. When this validation function returns true, the getContact and displayContactData functions are called.
- Exiting the application results in the getViewRequest function obtaining no additional data from the user. In this scenario, the isExit function would return true, causing the controller's go function to end.

19.5 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when creating a text-based user interface as part of a top-down design approach.

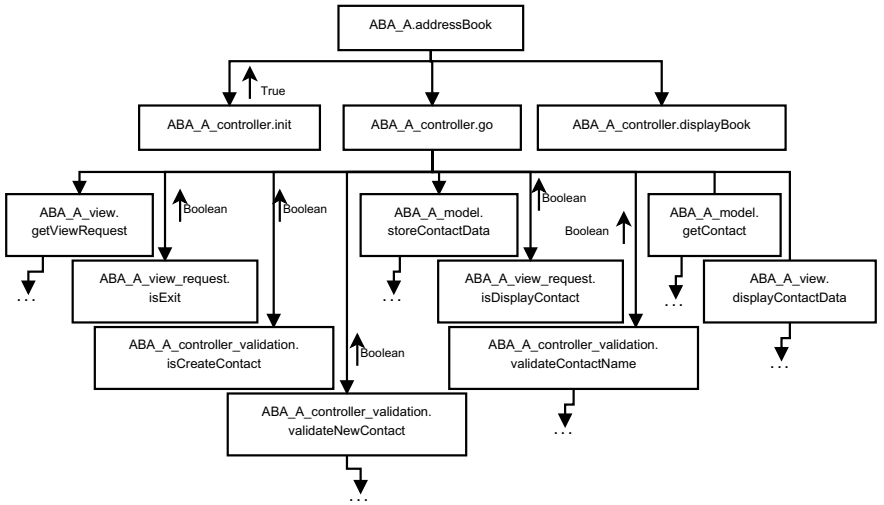


Fig. 19.5 Structure chart for MVC design ABA Version A: go function two levels of function calls

19.5.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 13, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

Table 19.4 Personal finances menu design options

Menu option 1	Menu option 2 with submenu
1. Create account	1. Account Management
2. View account	2. Transaction Management
3. Update account	3. Report Management
4. Delete account	X. Exit
5. Create transaction	Enter choice:
6. View transaction	
7. Update transaction	
8. Delete transaction	1. Create account
9. Create report	2. View account
10. View report	3. Update account
11. Update report	4. Delete account
12. Delete report	X. Exit
Enter choice:	Enter choice:

19.5.2 OOD Personal Finances: TUI Design Choices

The user interface design for the personal finances case study is potentially quite large. It needs to support user actions to create, read, update, and delete accounts, transactions, and reports. The number of user actions is large enough to warrant consideration of each text-based interface style discussed above.

A menu-based approach to providing a user with options seems like a reasonable approach. Table 19.4 shows two options for how the menu(s) could be designed. Option 1 shows a rather large menu listing all of the user actions. The user would enter a number (1 through 12) and then guided prompts would be used to obtain the data necessary for the option chosen. Option 2 has a much smaller main menu with just three options. Once the user enters a number (1 through 3), an appropriate submenu is displayed. The table shows the submenu for account management. Similar submenus would be displayed for transaction and report management. For this second menu option, efficiency could be improved by allowing the user to enter both the main menu and submenu choice at the *Enter choice:* prompt for the main menu. For example, guided prompts for updating a transaction would result from entering 2.3 at the main menu prompt.

A simple command-based user interface is shown in Listing 19.4. Having six commands seems like a reasonable design, one that could be learned fairly quickly. Four of these commands—create, delete, update, and view—would require a keyword be entered immediately after the command. The valid keywords would be account, transaction, and report, to indicate which type of personal finances data entity is being referenced by the command.

Listing 19.4 Sample Personal Finances Commands & Data Entry Prompts

```
command> help
All commands are case insensitive
Enter 'C' or 'create' to create an account, transaction, or report
Enter 'D' or 'delete' to delete an account, transaction, or report
Enter 'H' or 'help' to display this help text
Enter 'U' or 'update' to update an account, transaction, or report
Enter 'V' or 'view' to view an account, transaction, or report
Enter 'X' or 'exit' to exit

command> C account
Enter account name: MyBank
Enter account type: savings
Enter starting balance: 100.00

command> delete transaction
Enter account name and type: MyBank credit−card
Enter transaction date: 03/23/2019
Enter transaction amount: 10.99
Do you want to delete the following transaction?
Date: 03/23/2019 Description: lunch Debit: 10.99
Enter Y or N: Y

command> X
```

An example of a more complex command-based user interface is shown in Listing 19.5. The *KEY-VALUE-PAIR* shown in the help text represents a replacement for the guided prompts used in the first two user interface designs described above. As shown in the two 'create' examples, a KEY-VALUE-PAIR consists of a keyword (e.g., name, type, balance, account, desc, and debit) following by a colon and a data value (e.g., MyBank, savings, 100.00, lunch, and 10.99). This expanded version of a command-line interface would be much harder to learn when compared to the first two approaches.

Listing 19.5 Sample Personal Finances Commands Revisited

```
command> help create
All commands are case insensitive.
Command parameters in [square brackets] are optional.
Command parameters in {squiggly brackets} represent a grouping.
A grouping may be repeated using ellipses (i.e., ...).
Use single (') or double quotes (") to enclose data with spaces.
To create a data entity: c TYPE {KEY−VALUE−PAIR} ...
                         create TYPE {KEY−VALUE−PAIR} ...

command> C account name:MyBank type:savings balance:100.00

command> C transaction account:MyBank type:credit−card
      desc:lunch debit:10.99

command> X
```

Table 19.5 Personal Finances TUI Design Comparison

TUI Design	Efficiency	Learnability	Utility
Menu and guided prompts	Maybe	Easy	Yes
Commands & Guided Prompts	Yes	Moderate	Yes
Commands Revisited	Yes	Hard	Yes

19.5.3 OOD Personal Finances: Evaluate TUI Design Choices

Table 19.5 compares the three text-based UI design choices just presented. Assuming the second option for the menu design is used, the ability to enter a number representing the choice from the main menu and submenu in one entry (e.g., 2.3) makes the menu design as efficient as the simple command-line option. The more complex command structure described in Listing 19.5 would be much harder for someone to learn, especially a user that occasionally uses the personal finances application.

Given everything discussed regarding a text-based user interface for personal finances, it seems reasonable to provide both the menu option 2 and simple command interfaces. Both of these approaches would use the same guided prompt design while allowing an occasional user the menu option 2 interface to efficiently use the application. While someone may argue that the more complex command structure as described in Listing 19.5 is the best choice for a power user, this user interface design includes design and implementation logic that is not usable by the other two approaches. Specifically, the parsing of a command must include logic for the KEY-VALUE-PAIR. This is unique to this approach. Combine this fact with learnability being hard for this design, and the conclusion drawn by the author is that the extra effort to design, build, and test the third design option is not worth the investment.

19.5.4 SD Personal Finances: TUI Design Details

The statechart and structure chart diagrams in Figs. 19.6 and 19.7 shows the modifications to the view component as a result of selecting a user interface design for the personal finances application.

The statechart shows the user interactions with the menu and submenu for an account. Similar behavior would exist for user interactions related to transactions and reports. Note the shortcut the user may elect to enter to get from the main menu directly to the guided prompts for adding, viewing, updating, or deleting an account. Entering *X* from the main menu would cause the personal finances application to end, while entering *X* from the submenu redisplays the main menu.

The structure chart in Fig. 19.7 is the same as Fig. 16.18, it shows the flow of information for the Transaction Management function. The user interface logic to display the menu and submenus would reside in the *TM User Interface* function.

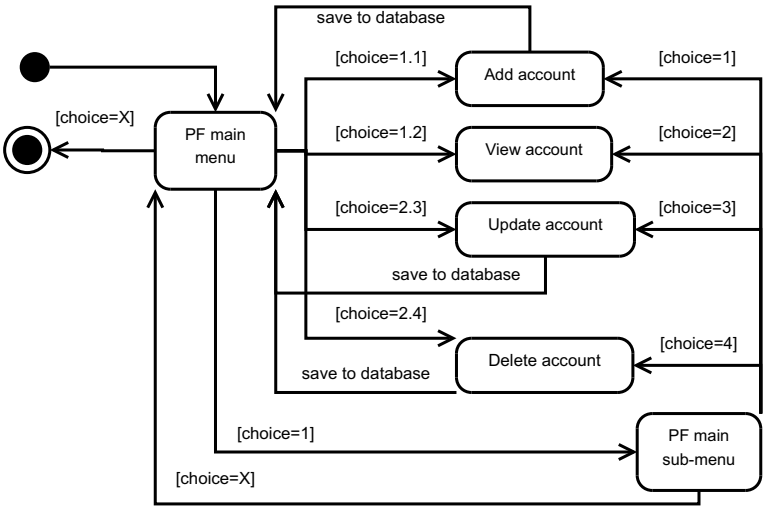


Fig. 19.6 State machine diagram—PF user interactions

The specific logic to prompt for data values would be in the *Add Trans UI*, *Modify Trans UI*, and *Delete Trans UI* functions. A similar function structure would exist for account management and report management.

Chapter 16 includes a second structure chart showing an alternative structure for functions. This structure chart, shown in Fig. 19.8, is not an appropriate structure when using menus. This is because the menu logic cannot be in the *TM Logic* function since this function is part of the controller and should not contain any view logic. Similarly, the *Add Trans*, *Modify trans*, and *Delete trans* functions are also part of the controller. The view functions for the menu logic are at the lowest level of the structure chart. However, the *Add Trans UI* and similar functions are not appropriate since at this point in the structure chart it is already known what action the user wants to perform, which is the purpose of the menus.

19.6 Post-conditions

The following should have been learned when completing this chapter.

- A text-based UI should consider some combination of menus, commands, and guided prompts to provide a user with an efficient, useful, and easy to learn user interface.
- Using more complex commands in a TUI may be appropriate when many of your users are considered power users. Knowing your users is critical in determining whether this is a viable design option.

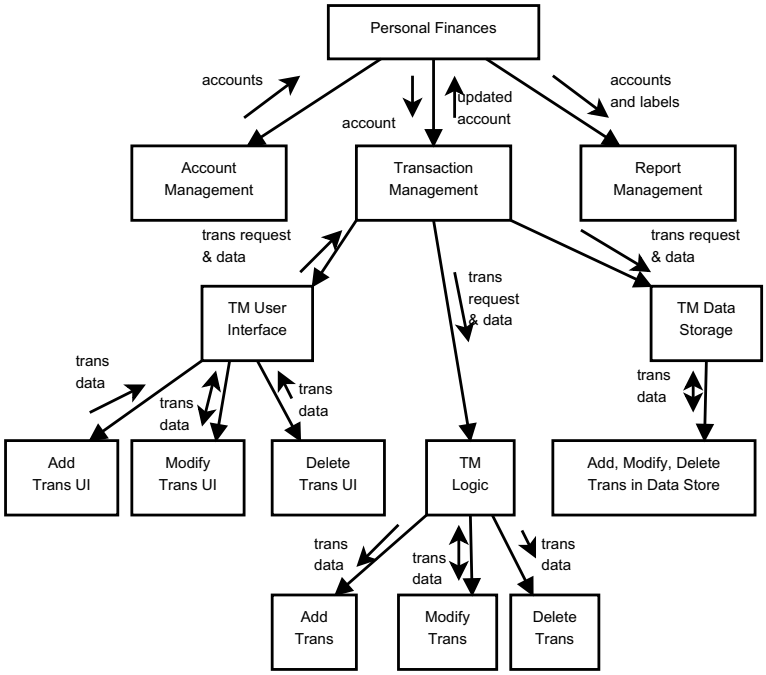


Fig. 19.7 Personal finance structure chart (from Chap. 16)

- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You have created and/or modified models that describe a structured software design. This includes thinking about design from the bottom-up and from the top-down.

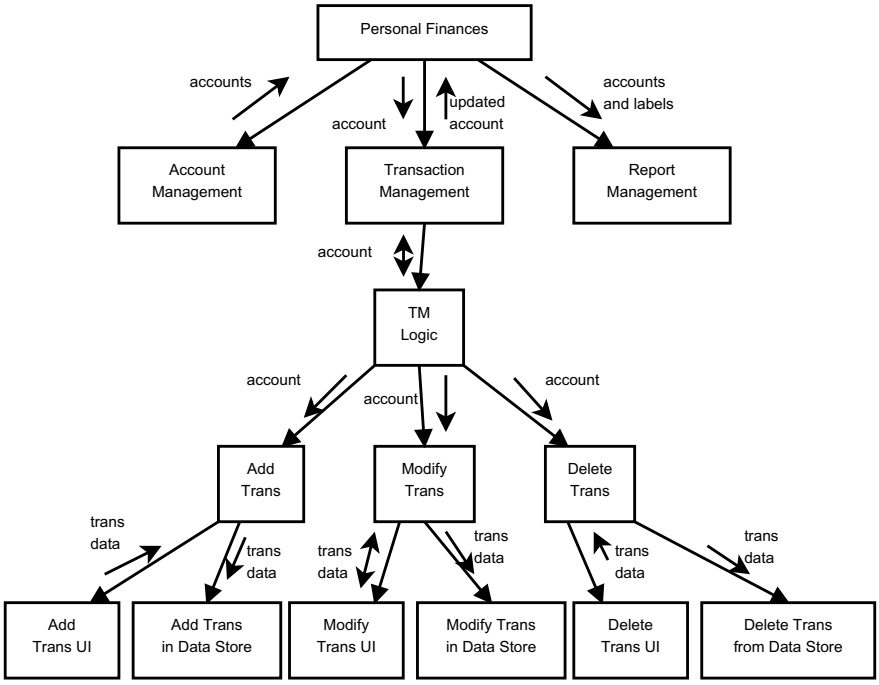


Fig. 19.8 Personal finance structure chart ver2 (from Chap. 16)

Exercises

Discussion Questions

1. An ATM may not use a touch screen, instead, it has buttons that get pushed to activate an option and uses a keypad to enter numbers. What type of text-based UI is this?
2. Can you think of any devices that use a text-based UI? If yes, what is the device and how would you describe this UI in terms of using prompts, menus, and/or commands? Evaluate this text-based UI using the HCI design criteria. Do you have any suggestions for improving this HCI design?

Hands-on Exercises

1. Use an existing code solution that you've developed, develop alternative text-based HCI design models that show ways in which a user could interact with your application. Apply the HCI design criteria to your design models. How good or bad is your design?

2. Use your development of an application that you started in Chap. 4 for this exercise. Modify your HCI design to use some combination of prompts, menus, and commands, and then evaluate your design using the HCI design criteria.
3. Continue hands-on exercise 3 from Chap. 13 by developing an HCI design using some combination of prompts, menus, and commands. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 13 for details on each domain.

   - Airline reservation and seat assignment
   - Automated teller machine (ATM)
   - Bus transportation system
   - Course-class enrollment
   - Digital library
   - Inventory and distribution control
   - Online retail shopping cart
   - Personal calendar
   - Travel itinerary



Model–View–Controller: TUI Versus GUI

20

The objective of this chapter is to compare and contrast two types of user interfaces and their impact on how the model–view–controller components interact with each other. This chapter also introduces some basic components found in a graphical user interface.

20.1 Preconditions

The following should be true prior to starting this chapter.

- You understand the four criteria used to evaluate a HCI design:

  1. Efficiency—How does the HCI affect the productivity of the user?
  2. Learnability—How easy is it for a user to learn the HCI?
  3. User satisfaction—How satisfied is the user with the HCI?
  4. Utility—Does the HCI provide useful and timely information?

- You understand the HCI design goals: know the user, prevent user errors, optimize user abilities, and be consistent.
- You appreciate the value user participation brings to the HCI design process.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_20

- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

20.2 MVC User Interface: Concepts and Context

When using the MVC architectural pattern, the controller and view components must interact with each other to support the application domain and user interface. The details of how these two components work together to support the application is largely dependent on whether the user interface is implemented as a text-based or graphical-based user interface.

20.2.1 Text-Based User Interface

Prior to Chap. 17, the ABA case study provided a text-based user interface (TUI) where the domain logic implemented in the controller enforced a specific sequence of user interactions. The user first had to enter a contact name or the word "exit". If "exit" was entered, the contents of the address book would be displayed and the application would end. Otherwise, the user would next enter a phone number and then an email address. Once these three data values were entered, in this specific order, they would be validated and either added to the address book or error messages would be displayed to the user.

In Chaps. 18 and 19, the ABA case study was modified to first display a menu. This gives the user the option of entering new contact data, displaying data for a person already stored in the address book, or exiting the application. When the user decides to create a new contact, the user must enter the three data values—name, phone number, and email address—*in this specific order*.

20.2.2 Graphical-Based User Interface

With a graphical user interface (GUI), the user controls what they do next. A GUI for the ABA case study will allow the user to decide whether to create a new contact, display data for an existing contact, or exit the application. In addition, when the user decides to create a new contact, a GUI design will likely allow a user to enter the three data values—name, phone number, and email address—*in any order they prefer*.

The other aspect of designing a GUI is the types of user controls a designer may use to enable user interactions. The list below identifies some of the common user interactions and the types of user controls used to support the interaction type.

User requests an action: A menu may identify various actions (e.g., File-Open, File-Save As), a push button represents a distinct action (e.g., Save, Cancel), or a hypertext link may initiate a particular action (e.g., show a different web page).

User selects a choice: A list box is typically used to display a list of items from which one may be selected, a small group of option buttons (also known as radio buttons) allows the user to select one option from the group, or a drop-down list allows the user to display the list to select an item.

User selects many choices: A list box may be used to allow the user to select multiple items, or a small group of check boxes allows the user to select as many options as they desire.

User enters data: A text box allows the user to enter a single line of text, a multi-line text box supports entry of large amounts of text, or a combo box (i.e., a text box that includes a drop-down list) allows a user to select from the drop-down list or enter data in the text box to find a matching item in the list.

20.2.3 Summary: TUI versus GUI

A text-based user interface requires the software to guide the user through the steps needed to complete a processing step. In terms of data entry, a TUI forces a user to enter data in a predetermined order. A graphical user interface allows the software to provide actions from which the user may select, and allows data entry to be done by the user (typically) in any order.

20.2.4 Impact on MVC Design

With a TUI, the controller component will call methods within the view component. These method calls allow the software to control choices the user has within the user interface. This includes sequencing the entry of data. Figure 20.1 shows a UML communication diagram for a generic text-based user interface. The controller component would call a method/function in the view to start the user interactions. The specific user interactions could be done within the view. As shown in this figure, the view would need to obtain user data in a specific sequence, then return the data as part of the generic method/function call getUserData.

With a GUI, the view component displays a window/page, the user chooses what to do based on the choices presented, and then the user initiates an action (via the GUI) that requires the view to communicate with the controller. Figure 20.2 shows a UML communication diagram for a generic graphical user interface. The view component is responsible for displaying the GUI and reacting when the user requests a specific action. Typically, the view component would create an object/data entity that is then

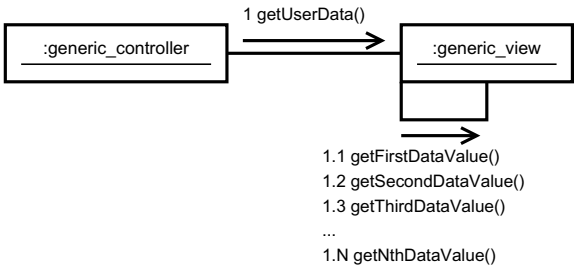


Fig. 20.1 Generic communication diagram—Text-based user interface

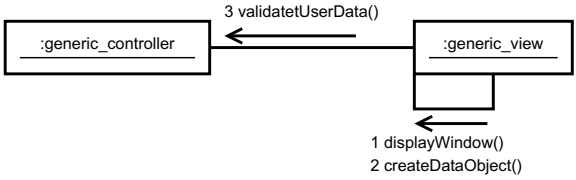


Fig. 20.2 Generic communication diagram—Graphical-based user interface

sent to the controller. This object/data entity would contain the data entered by the user and also identify the type of action the user requested. The controller would validate the data entered and decide what to do based on the user requested action and validation results.

20.2.5 MVC Summary: TUI Versus GUI

With a TUI, the controller controls the user interactions by calling view methods. The view will provide a particular sequence that the user must follow when using the application. With a GUI, the view controls the user interactions. The view will provide a graphical allowing the user to make choices about what to do next. The view will call a controller method to inform the controller what the user has done/requested.

20.3 GUI: More Concepts and Context

A graphical user interface (GUI) is the most common type of HCI in use in modern software. This includes support for touch screen devices (e.g., *smart* cell phones), web pages, and software applications that operate within desktop/laptop environments. In this chapter, we'll focus on GUI designs for desktop/laptop software applications.

A GUI design must integrate three different types of software elements.

1. Graphical components or objects.
   These components include GUI objects that contain other GUI objects (i.e., container objects) as well as basic GUI objects representing a distinct type of user

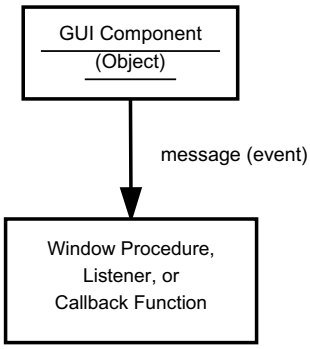


Fig. 20.3 Relationship between GUI objects and Listeners/Callback functions

interaction. The container GUI objects include windows, dialog boxes, and group boxes. The basic GUI objects include labels, buttons, text fields, menus, and list boxes. There are also GUI objects that combine two more basic objects. An example of this is a drop-down combo box, which combines a text box with a list box.

2. User events or messages.
   An event is an action, often a user action, that is responded to by the software. Examples of events are: a mouse button is clicked; a keyboard key is pressed; a graphical button object is clicked; a mouse pointer is moved; a mouse pointer is clicked-and-dragged; and a timer expires. A message is the mechanism used to convey the event to the application code.
3. Code that reacts to user events and manipulates the graphical components/objects. The code that reacts to events is known as window procedures (in C++), window listeners (in Java), and callback functions (in Python). Figure 20.3 shows a GUI object generating a message that is sent to the Window procedure, listener, or callback function. This message contains the event that has been triggered based on (typically) a user action on a GUI object. A window procedure, listener, or callback function *is idle* until an event triggers this logic to be executed. The logic we put in a procedure/listener/function responds to the event in an appropriate manner. This may include ignoring the event.

20.3.1 GUI Design Alternatives

A HCI designer will generally identify the types of GUI components needed and how these components will appear within a window container. The designer will also identify the (user) events that will need to be responded to. When multiple windows are needed, the designer must also determine which actions will cause another window to be displayed.

The commonly used GUI components and their associated events are predefined by a programming language library. For C++, you'll find GUI objects and events

in the Windows 32 API and in the Microsoft Foundation Classes. For Java, you'll find GUI objects and events in the Java API class hierarchy, specifically the java.awt (abstract windowing toolkit), javax.swing, and javafx packages. In Python, you'll find GUI objects and events in the graphical user interfaces with Tk/Tcl, specifically tkinter and related modules.

Some of the commonly used GUI components are now described, in alphabetical order.

20.3.1.1 Check Box

A check box is typically displayed as a small square with a description to the right of the square. A check box represents an option among a group of independent options. An example is the options (e.g., floor mats, CD player, power locks, power windows) when buying a new vehicle.

C++ Win32: Use the BS_CHECKBOX style of the Button class.
Java: Use the JCheckBox class.
Python: Use the tkinter.CheckButton class.

20.3.1.2 Label

A label is typically displayed as text with no borders. It represents static text (i.e., text that cannot be modified by the user) displayed in the window.

C++ Win32: Use the Static class.
Java: Use the JLabel class.
Python: Use the tkinter.Label class.

20.3.1.3 List Box

A list box is typically displayed within a rectangle with borders and a vertical scroll bar. It represents either a list of mutually exclusive options (like a group of radio buttons) or a list of independent options (like a group of check boxes).

C++ Win32: Use the Listbox class.
Java: Use the JList class.
Python: Use the tkinter.Listbox class.

20.3.1.4 Push Button

A push button is typically displayed as a rounded rectangle with a verb or verb phrase inside the rectangle. A push button denotes an action that will be performed when the button is activated by the user. Examples of push buttons are send (in an email application) and save (in a word processing application).

C++ Win32: Use the BS_PUSHBUTTON style of the Button class.
Java: Use the JButton class.
Python: Use the tkinter.Button class.

20.3.1.5 Radio Button

A radio button is typically displayed as a small circle with a description to the right of the circle. A radio button represents one option among a group of mutually exclusive options. An example is a list of color options for a new vehicle. Usually, a group of radio buttons is defined so only one can be selected at a given point in time.

C++ Win32: Use the BS_RADIOBUTTON style of the Button class.
Java: Use the JRadioButton class. Put these inside a ButtonGroup object to identify the group of buttons requiring mutually exclusive selection.
Python: Use the tkinter.Radiobutton class. Associate each Radiobutton object with the same variable to group the buttons requiring mutually exclusive selection.

20.3.1.6 Text Fields

A text field is typically displayed as a rectangle. It represents a place where a user can enter and modify text data found within the text field. Some text fields allow one line of text to be entered while another type of text field allows the user to enter multiple lines of text.

C++ Win32: Use the Edit class.
Java: Use the JTextField or JTextArea class for a single or multi-line text field, respectively.
Python: Use the tkinter.Entry class.

20.4 Post-conditions

The following should have been learned when completing this chapter.

- Using Model–View–Controller for a text-based user interface means the application controls the sequence of user interactions. This is done by having the controller component call view methods/functions.
- Using Model–View–Controller for a graphical user interface means the user controls the sequence of interactions. This is done by having the view component call controller methods/functions.

Exercises

Discussion Questions

1. What types of applications may benefit from having a text-based user interface?
2. What arguments could you make for an MVC TUI design behaving more like a GUI when the user enters commands?



OOD Case Study: Graphical-Based User Interface

21

The objective of this chapter is to apply the human–computer interaction (HCI) design concepts discussed in Chaps. 17 and 20 to the development of a graphical-based HCI.

21.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand the four criteria used to evaluate a HCI design:

1. Efficiency—How does the HCI affect the productivity of the user?
2. Learnability—How easy is it for a user to learn the HCI?
3. User satisfaction—How satisfied is the user with the HCI?
4. Utility—Does the HCI provide useful and timely information?

- You understand the HCI design goals: know the user, prevent user errors, optimize user abilities, and be consistent.
- You appreciate the value user participation brings to the HCI design process.
- Using Model–View–Controller for a text-based user interface means that the application controls the sequence of user interactions. This is done by having the controller component call view methods.
- Using Model–View–Controller for a graphical user interface means that the user controls the sequence of interactions. This is done by having the view component call controller methods.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_21

- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

21.2 OOD: ABA GUI Design

One GUI design is presented as an implementation of the ABA requirements, stated below for reference.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.

21.3 OOD: ABA GUI Version A

Two windows are used in this design, one lists all of the contact names while the other displays contact data for a single person. These two windows are shown in Figs. 21.1 and 21.2.

The statechart in Fig. 21.3 shows the actions the user may perform. First, the user will see contact names displayed in the list, representing contacts currently in the address book. Figure 21.1 lists four contact names. Second, the user can select an existing contact name and display the contact data for this person. Third, the user can click on the Create button to display the second window shown in Fig. 21.2. The user would enter appropriate data in the text fields and click OK to have the data validated. Valid contact data would result in the contact being added to the address book, assuming that the contact name does not already exist in the address book.

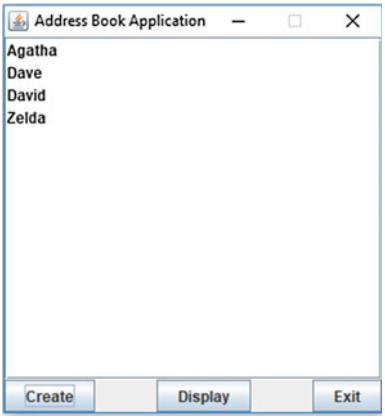


Fig. 21.1 ABA GUI Version A main window

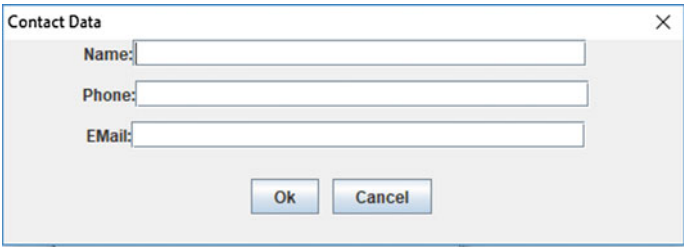


Fig. 21.2 ABA GUI Version A contact data

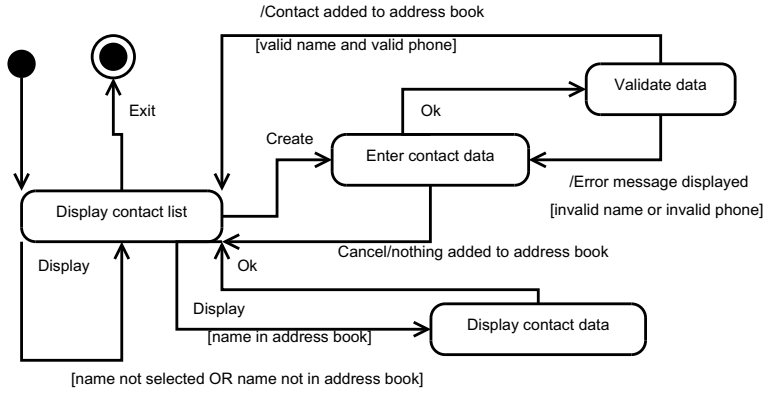


Fig. 21.3 ABA GUI Version A statechart

When the data is not valid, an error message is displayed and the user is allowed to correct the entered data. The user may then try to create the contact again. Finally, the user can exit the application.

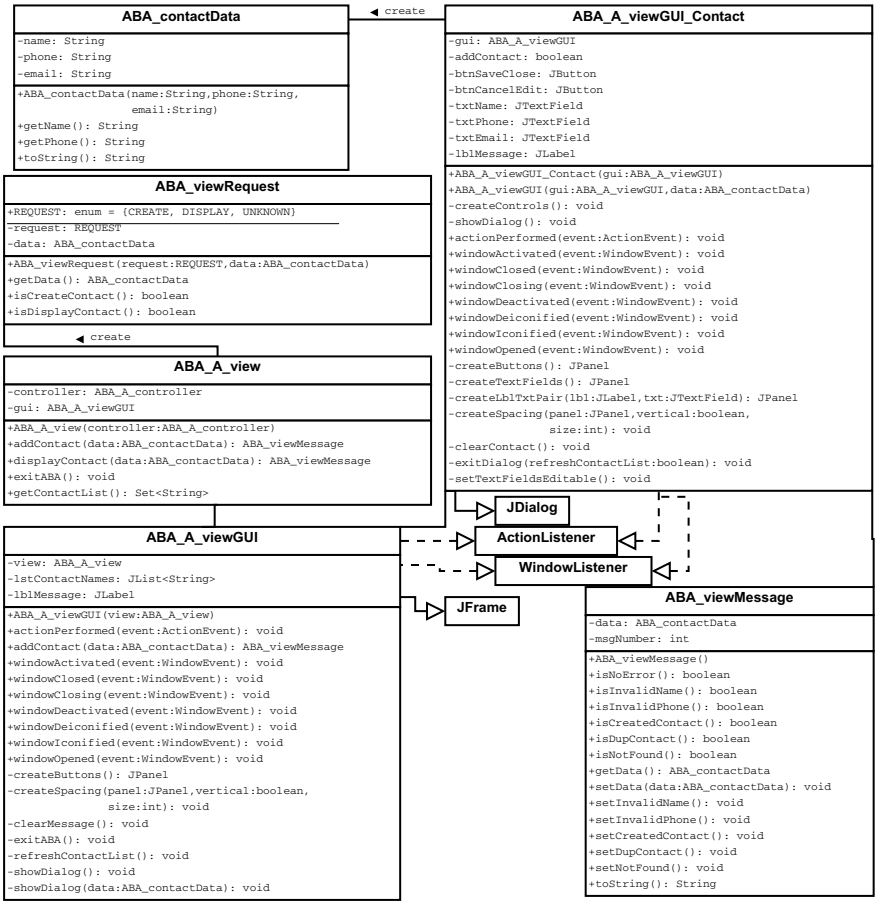


Fig. 21.4 ABA GUI Version A class diagram—view component

The classes that make up the view component are shown in Fig. 21.4. The viewGUI class contains the list of contact names along with buttons to create or display a specific contact. This class inherits from the JFrame Java API class and represents the main application window. The picture in Fig. 21.1 illustrates the appearance of this window. The viewGUI_Contact class contains text fields used to enter and display contact data for a single person. This class inherits from the JDialog Java API class indicating that it behaves like a pop-up window. More specifically, the viewGUI_Contact object is constructed as a *modal* dialog, meaning that while displayed, the user cannot interact with the application's main window. The picture in Fig. 21.2 illustrates the appearance of this dialog.

Figure 21.5 shows the UML communication diagram describing how the ABA Version A GUI application starts. The main method in ABA_A creates a ABA_A_ controller object (see step 1). This constructor method first constructs a ABA_A_model

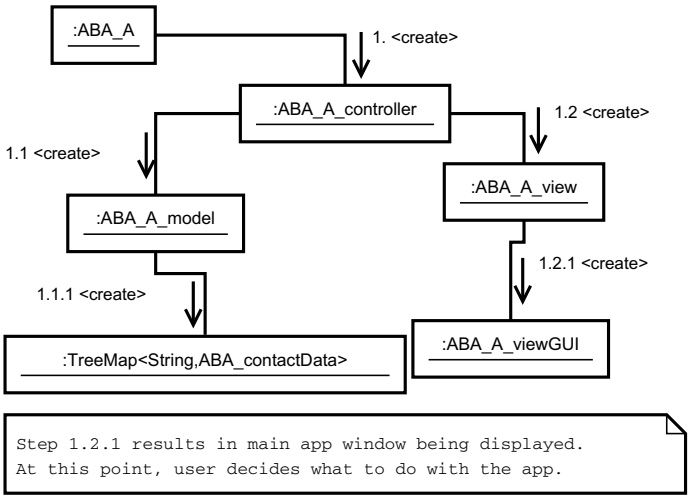


Fig. 21.5 ABA GUI Version A communication diagram—initializing ABA

object (step 1.1), then constructs a ABA_A_view object (step 1.2). The model constructor method creates a TreeMap object (step 1.1.1) used to store contact data. The view constructor method creates a ABA_A_viewGUI object (step 1.2.1) which is the main application window. At this point, the ABA is ready for use; the user determines what happens next! Note the absence of a go method in Fig. 21.5. As discussed in Chap. 20, this highlights one of the key differences between a text-based and graphical-based user interface; a GUI does not use a *driver* method while a TUI must have a method that drives the overall processing flow of the application.

One other detail about step 1.2.1 in the UML communication diagram shown in Fig. 21.5 is worth mentioning—the creation of the three buttons that appear on the main application window and how these are associated with a listener. Listing 21.1 shows portions of the ABA_A_viewGUI code illustrating how this is done in Java. The class implements two interfaces: ActionListener and WindowListener. A window listener deals with events sent to the JFrame, which includes the user clicking on the "X" button to close the application window. Listing 21.1 shows a WindowClosing method that will result in the ABA ending. The ABA_A_viewGUI code contains other WindowListener methods not shown in the Listing. An action listener is associated with individual user controls and requires the class to have an actionPerformed method (also partially shown in the Listing). The creation of the three push buttons shows a call to the addActionListener method passing *this*. Since *this* refers to the ABA_A_viewGUI object, these statements are indicating that any action on these buttons should be sent to the actionPerformed method defined in this class.

Listing 21.1 ABA MVC Design Version A - create JButton objects

```
public class ABA_A_viewGUI extends JFrame
        implements ActionListener, WindowListener
{
//not showing lots of code
   public void actionPerformed(ActionEvent event)
   {
      //not showing details of this method
   }
//not showing lots of code
   public void windowClosing(WindowEvent event) { exitABA(); }
//not showing lots of code
   JButton btnCreate = new JButton(CREATE_CONTACT);
   JButton btnDisplay = new JButton(DISPLAY_CONTACT);
   JButton btnExit = new JButton(EXIT_ABA);
   btnCreate.addActionListener(this);
   btnDisplay.addActionListener(this);
   btnExit.addActionListener(this);
//not showing lots of code
}
```

Figure 21.6 shows the UML communication diagram describing how the ABA Version A GUI creates a contact. This diagram assumes the user enters valid contact data to be added to the address book. The steps below describe the processing flow.

Step 1: The user clicks on the Create button on the main application window.
Step 1.1: The private showDialog method is called to display the pop-up dialog.
Step 1.1.1: An ABA_A_viewGUI_Contact object is created.
Step 1.1.1.1: The private createControls method is called to create the user controls in the dialog.
Step 1.1.1.2: The private showDialog method is called to show the dialog to the user.
Step 2: As noted, we assume the user enters valid contact data.
Step 3: We also assume the user clicks on the Save button within the dialog.
Step 3.1: The private createContact method is called.
Step 3.1.1: An ABA_contactData object is created containing the user-supplied data. The remaining steps refer to this object as data.
Step 3.1.2: The dialog tells the main application window (ABA_A_viewGUI) to add a contact using data.
Step 3.1.2.1: The main application window tells the view (ABA_A_view) to add a contact using data.
Step 3.1.2.1.1: The view creates an ABA_viewRequest object to communicate this user request to the controller. The remaining steps refer to this object as request.
Step 3.1.2.1.2: The view tells the controller ABA_A_controller to add a contact using request. The controller validates the user-supplied data, which is contained inside the request. In substep 1 the controller creates an ABA_viewMessage object to communicate the results of this request back to the view. The remaining steps

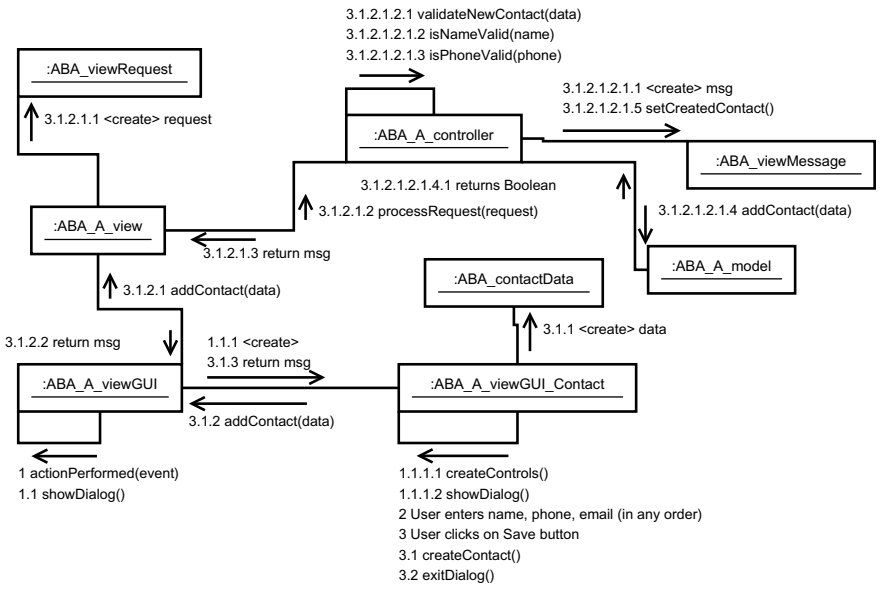


Fig. 21.6 ABA GUI Version A communication diagram—create a contact

refer to this object as msg. In substeps 2 and 2 the contact name and phone number are validated. Substep 4 adds the contact data to the address book; we assume the contact name is unique. In substep 5, the controller changes the state of the msg to indicate the contact was successfully added to the address book.

Step 3.1.2.1.3: The controller returns the msg to the view.

Step 3.1.2.2: The view returns the msg to the viewGUI.

Step 3.1.3: The viewGUI returns the msg to the viewGUI_Contact.

Step 3.2: The private method exitDialog is called to close the dialog and return the user to the main application window.

The processing flow to display existing contact data is similar. The viewGUI creates a new contactData object (aka data) containing the contact name associated with this display request. This data object is given to the view, which uses it to construct an ABA_viewRequest object (aka request). This request object is given to the controller, which then asks the model to retrieve the contact data for the user-supplied name. The controller creates an ABA_viewMessage object (aka msg) to communicate the results of this display request back to the view. This msg object is returned to the view, which returns it to the viewGUI (i.e, the main application window). The viewGUI then displays the viewGUI_Contact dialog showing the data retrieved by the model.

21.3.1 OOD: Evaluate Version A GUI Design

We'll now evaluate the Version A design using the HCI design criteria.

21.3.1.1 Efficiency

The display of existing contacts in the main application window allows the user to be more efficient in deciding whether a new contact needs to be created. Having the three buttons—Create, Display, and Exit—at the bottom of the window results in it taking a bit longer to activate one of these buttons. The design of the dialog box appears to be efficient, with two buttons when adding a contact but only one button when displaying existing contact data.

21.3.1.2 Learnability

The Version A GUI design is fairly easy to use. Two issues may cause a first-time user to be confused. First, it may take a moment before seeing the three buttons at the bottom of the main application window. Second, the large amount of space in the main application window with nothing being displayed may be initially confusing. On the positive side, the messages describing validation errors or success assist in the user's understanding of the application.

21.3.1.3 User Satisfaction

In order to assess this HCI design criteria, we would need to interview or survey users to obtain information on how satisfied they are with this GUI design. Since the ABA has not been deployed as a software application that others may use, this criteria cannot be assessed.

21.3.1.4 Utility

The user gets immediate feedback on each transaction they perform. Displaying a list of existing contacts is very useful to the user.

21.4 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when creating a graphics-based user interface as part of a top-down design approach.

21.4.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

21.4.2 OOD Personal Finances: GUI Design

The user interface design for the personal finances case study is potentially quite large. It needs to support user actions to create, read, update, and delete accounts, transactions, and reports. Ideally, we would like a single window to provide quick access to each of these user actions.

Figure 21.7 shows a GUI design for the main application window. Accounts and reports would be listed along the left edge of the window. The majority of the main window will either show transactions for the selected account (shown in Fig. 21.7) or the selected report. Creating, updating, and deleting transactions would be done via choices on the Transactions menu. Similarly, accounts, reports, and labels may be created, updated, and deleted using these choices from the respective menu. The Labels menu also allows the user to display the entire list of labels.

The statechart and UML class diagram in Figs. 21.8 and 21.9 show the modifications to the view component as a result of using a graphical user interface for the personal finances application.

The statechart shows the user interactions associated with opening an account and adding, deleting, or updating a transaction. Once the transactions are displayed, the user selects an appropriate choice from the Transaction menu to initiate processing. When the user selects a different account from the list, the list of transactions associated with this account is displayed.

The UML class diagram includes two GUI classes (viewFinancesGUI and viewTransGUI) representing the main application window and the list of transactions displayed within this window. The viewUserRequest class is used to send a user request to the controller. A request can obtain data for display in the GUI (e.g., REQUEST_TYPE is VIEW), or send data to the controller for validation and ultimately for storage via the model. The viewAccount, viewLabel, viewReport, and

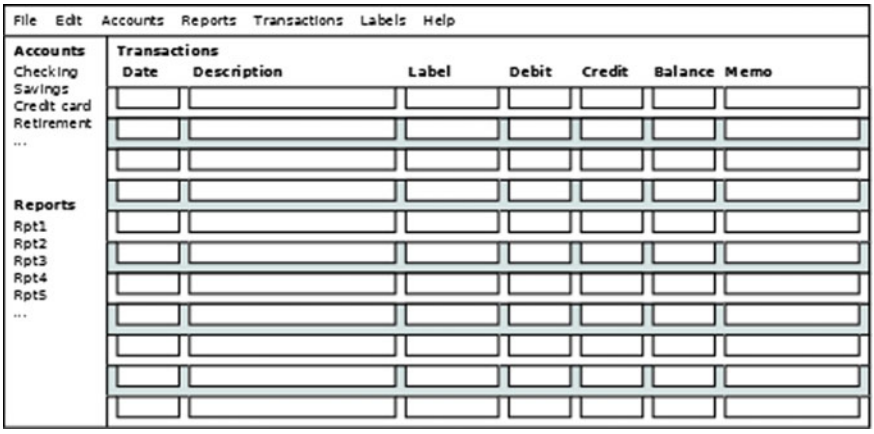


Fig. 21.7 Personal finances—GUI main window

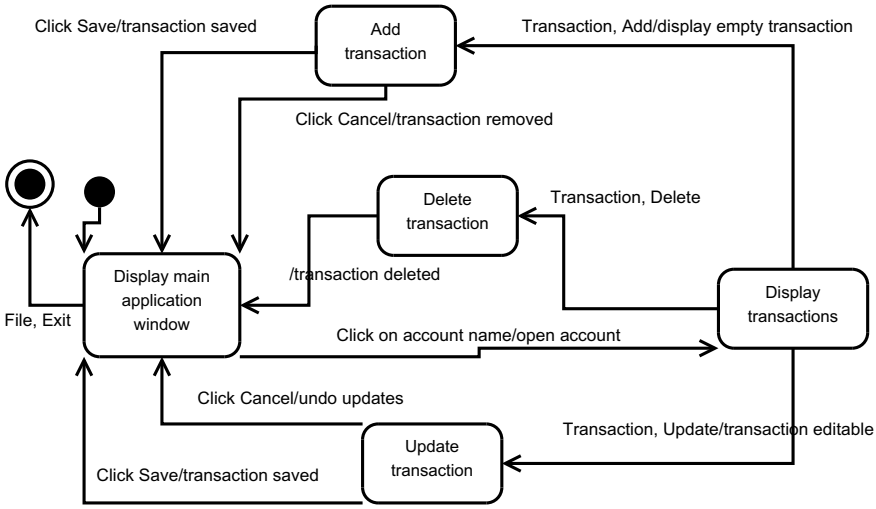


Fig. 21.8 State machine diagram—PF user interactions

viewTransaction classes are used to store a single instance of data either created by the user or obtained via the controller.

21.4.3 OOD Personal Finances: Evaluate GUI Design

With a single window supporting the application, efficiency, learnability, and utility may be assessed based on the design of this one window. As noted, the menu is used to create, update, or delete accounts, labels, reports, and transactions. Assuming

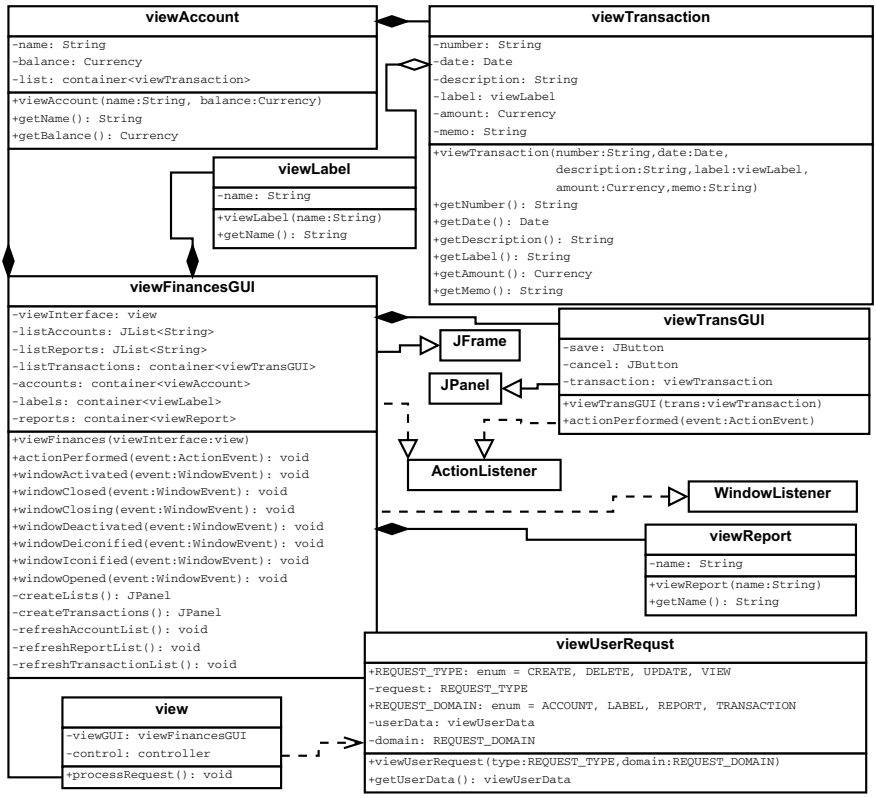


Fig. 21.9 Class diagram—PF view component

keyboard hot keys are associated with these menu choices, the user has the option of using a pointing device or the keyboard to activate an action. When an individual transaction is either created or updated, the GUI for that transaction will display save and cancel buttons. One of these buttons must be clicked to indicate the end of the create or update action.

From a learnability perspective, anyone familiar with a checking account transaction register, a small booklet used to record checking account transactions, will recognize the layout of transactions on the main application window. No attempt has been made to describe the GUI for creating, updating, or viewing a report. This is left as an exercise.

Given how similar the GUI design is to a paper-based checking account transaction register, the interface should provide the features necessary to support someone's personal finances.

21.5 OOD: Post-conditions

The following should have been learned when completing this chapter.

- A graphical user interface provides many alternatives for developing an HCI design. Creating a user interface design that is efficient, easy to learn, and useful is challenging because of the many choices one has for representing information in a graphical form.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Modify the GUI design and code for the ABA in any of the following ways.

   a. Allow a double-click on a contact name in the list to display the information for the selected contact.
   b. Move the buttons to the top of the main application window.
   c. Add hot keys to the buttons to create or display a contact.
   d. Change the JList to a JTable so the name, phone number, and email address can be displayed using three columns.

2. Modify the design for the personal finances application in any of the following ways.

   a. Show what a GUI would look like to create or update a report.
   b. Show what a GUI would look like to view a report.
   c. Update the class diagram based on your GUI designs for creating, updating, and viewing reports. That is, what is missing from the various classes to support these user actions?

3. Use an existing code solution that you've developed, develop alternative GUI-based HCI design models that show ways in which a user could interact with your application. Apply the HCI design criteria to your design models. How good or bad is your design?
4. Use your development of an application that you started in Chap. 3 for this exercise. Modify your HCI design to use some combination of graphical user controls, and then evaluate your design using the HCI design criteria.
5. Continue Hands-on Exercise 3 from Chap. 12 by developing an HCI design using some combination of graphical user controls. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

   - Airline reservation and seat assignment
   - Automated teller machine (ATM)
   - Bus transportation system
   - Course-class enrollment
   - Digital library
   - Inventory and distribution control
   - Online retail shopping cart
   - Personal calendar
   - Travel itinerary



SD Case Study: Graphical-Based User Interface

22

The objective of this chapter is to apply the human–computer Interaction (HCI) design concepts discussed in Chaps. 17 and 20 to the development of a graphical-based HCI.

22.1 SD: Preconditions

The following should be true prior to starting this chapter.

- You understand the four criteria used to evaluate a HCI design:

1. Efficiency—How does the HCI affect the productivity of the user?
2. Learnability—How easy is it for a user to learn the HCI?
3. User satisfaction—How satisfied is the user with the HCI?
4. Utility—Does the HCI provide useful and timely information?

- You understand the HCI design goals: know the user, prevent user errors, optimize user abilities, and be consistent.
- You appreciate the value user participation brings to the HCI design process.
- Using Model–View–Controller for a text-based user interface means that the application controls the sequence of user interactions. This is done by having the controller component call view methods.
- Using Model–View–Controller for a graphical user interface means that the user controls the sequence of interactions. This is done by having the view component call controller methods.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_22

- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

22.2 SD: ABA GUI Design

One GUI design is presented as an implementation of the ABA requirements, stated below for reference.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.

22.3 SD: ABA GUI Version A

Two windows are used in this design, one lists all of the contact names while the other displays contact data for a single person. These two windows are shown in Figs. 22.1 and 22.2.

The statechart in Fig. 22.3 shows the actions the user may perform. First, the user will see contact names displayed in the list, representing contacts currently in the address book. Figure 22.1 lists four contact names. Second, the user can select an existing contact name and display the contact data for this person. Third, the user can click on the Create button to display the second window shown in Fig. 22.2. The user would enter appropriate data in the text fields and click Ok to have the data validated. Valid contact data would result in the contact being added to the address book, assuming that the contact name does not already exist in the address book.

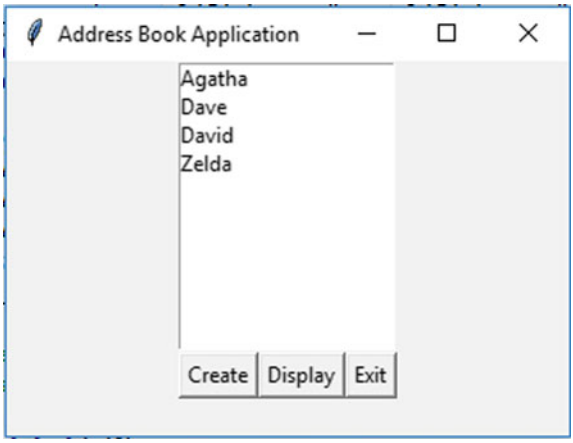


Fig. 22.1 ABA GUI Version A main window

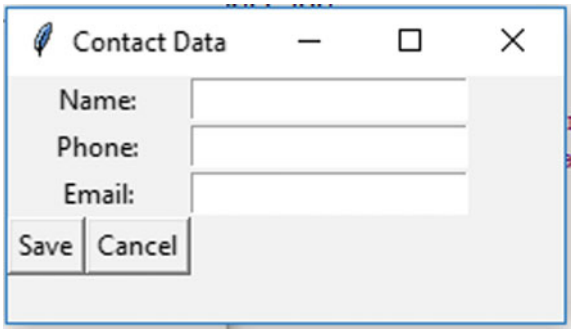


Fig. 22.2 ABA GUI Version A contact data

When the data is not valid, an error message is displayed and the user is allowed to correct the entered data. The user may then try to create the contact again. Finally, the user can exit the application.

The GUI solution for the ABA using Model–View–Controller has 11 source code files, as listed in Table 22.1. This MVC design has three source code files for the model, three for the controller, and five for the view. Since the ABA implements a GUI, the controller no longer has a go function to control the processing flow of the application. Instead, the initialization of the ABA ends when the main application window is displayed. At this point, the user dictates what happens next based on the choices presented to them in the GUI. The structure chart in Fig. 22.4 shows the function calls associated with initializing the ABA. The view component now has an init function which calls the createMainWindow function in the ABA_A_view_gui module. This function creates the user controls for the main window (i.e., a listbox, three buttons, and a label to display error messages) and then calls the update function in tkinter to display the main application window.

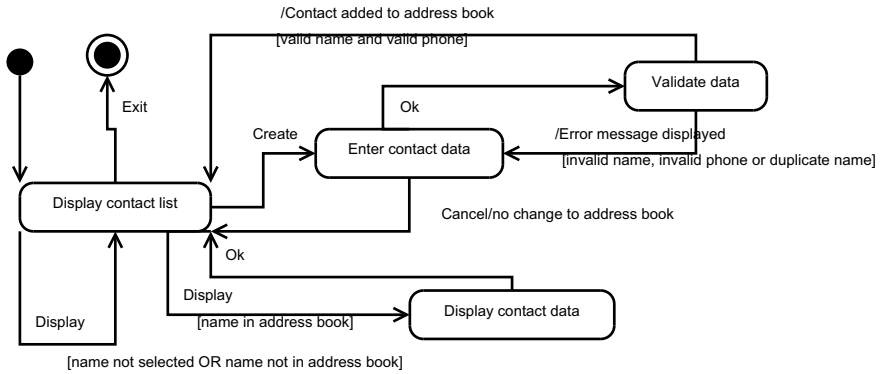


Fig. 22.3 ABA GUI Version A statechart

The structure charts in Figs. 22.5 and 22.6 show the user events and processing associated with creating a new contact. An explanation of the user events and associated processing for creating a contact is now described.

- The user clicks on the Create button on the main application window (see Fig. 22.1).
- The btnCreateCallBack function is called to display the pop-up dialog. The processing of this function is shown in Fig. 22.5.

  – The createDialog function is called to create the user controls (i.e., three labels with matching entry boxes, two buttons, and a label used to display an error message) and show the dialog window (see Fig. 22.2).
  – The wait_window function in tkinter is called to have the dialog window act as a *modal* window, meaning the user must close the dialog before they can interact with the main application window.

- The user enters data for a new contact and then clicks on the Save button on the dialog window.
- The btnSaveCallBack function is called to process this request to create a new contact. The processing of this function is shown in Fig. 22.6.

  – The data entered by the user is saved by calling the createContactDataTuple function.
  – The addContact function in the view module is called. This function gets the contact data, creates a user request, and then calls the controller's processRequest function to create this new contact person.
  – Not shown in Fig. 22.6 is the validation done by the controller, and if the contact data is valid, the controller calling the model's addContact function to add this new contact data to the address book.

Table 22.1 ABA MVC design GUI—Source code files

Source code file	Component?	Responsibilities
ABA_A.py	Controller	Starts the ABA
ABA_A_contact_data.py	Model	Stores a tuple containing data for one contact person. Has functions to allow the tuple to be created or retrieved
ABA_A_controller.py	Controller	Initializes the ABA, gets list of contact names for display in the main application window, and processes a user request (from the view)
ABA_A_controller_validation.py	Controller	Contains validation logic for the ABA
ABA_A_model.py	Model	Contains functions called by the controller
ABA_A_model_data.py	Model	Contains the address book data structure and functions that use the data structure
ABA_A_view.py	View	Contains one function called by the controller (init) and three functions called by other functions in the view
ABA_A_view_gui.py	View	The main application window
ABA_B_view_gui_contact.py	View	The dialog window showing data for one contact person
ABA_B_view_message.py	View	Identifies results from the controller processing a user request
ABA_B_view_request.py	View	Identifies the type of user request coming from the view (via user clicking on a button)

- When a new contact has been created (i.e., function isCreatedContact returns True), the list of contact names on the main application window is refreshed (so that this new contact's name is displayed in the list) and the dialog window is destroyed.
- When an error occurs trying to create a new contact, an error message is displayed in the dialog window. At this point, the user can correct the data based on the error message or click on the Cancel button. The Cancel button will destroy the dialog window without first creating a new contact.

The code used to create three buttons on the main application window is shown in Listing 22.1. Each button is added left justified in a Frame object, which acts as a container for the three buttons. Note that the Frame object has as its parent the main application window. When each button is created, the command operand identifies

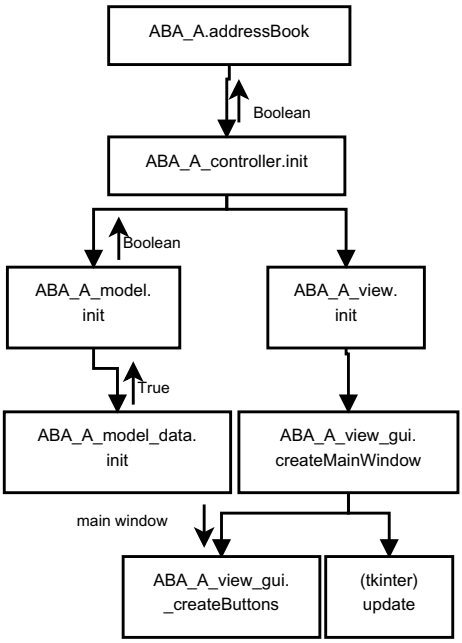


Fig. 22.4 Structure chart for MVC design ABA GUI: initialize

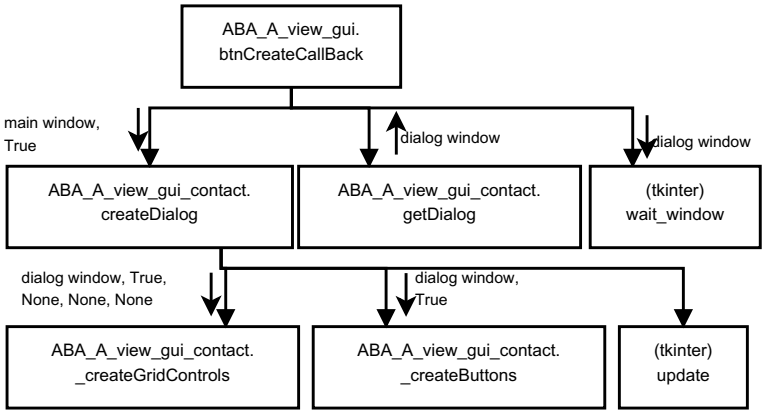


Fig. 22.5 Structure chart for MVC design ABA GUI: create contact (part 1)

the name of the *callback* function which is automatically called by tkinter when the user clicks on the button. The definition of the btnCreateCallBack function is also shown in this same source code listing. The structure chart in Fig. 22.5 represents the design of this callback function.

Listing 22.1 ABA MVC Design version A—create JButton objects

```
def _createButtons(window):
  frame = tk.Frame(window)
  btnCreate = tk.Button(frame, text='Create',
     command=btnCreateCallBack)
  btnCreate.pack(side='left')
  btnDisplay = tk.Button(frame, text='Display',
     command=btnDisplayCallBack)
  btnDisplay.pack(side='left')
  btnExit = tk.Button(frame, text='Exit', command=btnExitCallBack)
  btnExit.pack(side='left')
  frame.pack()

def btnCreateCallBack():
  global window
  lblMessage.config(text=NO_MESSAGE)
  #print("Create button was clicked")
  ABA_A_view_gui_contact.createDialog(window, True)
  window.wait_window(ABA_A_view_gui_contact.getDialog())
```

The processing flow to display existing contact data is similar. The btnDisplay-CallBack function in the view_gui module obtains the contact name selected in the listbox, sends a request to the controller to obtain contact data for the selected name, and then displays the dialog window showing the contact data. Like the create contact processing, errors may occur during the processing. Specifically, an error message is displayed when the user has not selected a name from the list of contacts.

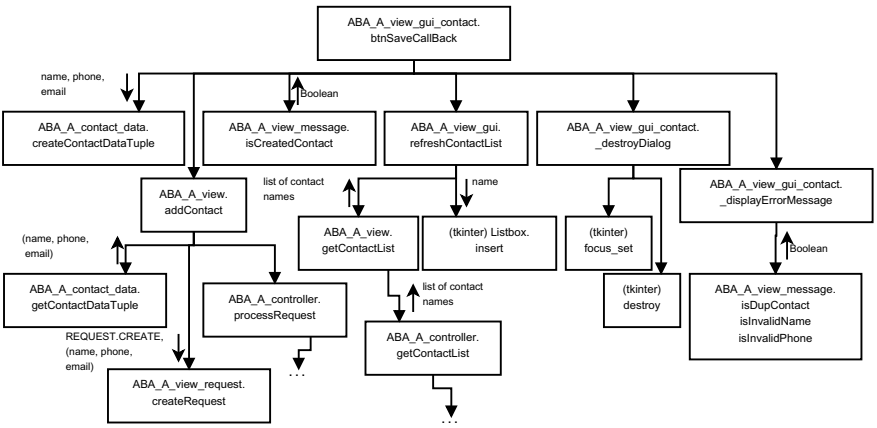


Fig. 22.6 Structure chart for MVC design ABA GUI: create contact (part 2)

22.3.1 SD: Evaluate Version A GUI Design

We'll now evaluate the Version A design using the HCI design criteria.

22.3.1.1 Efficiency

The display of existing contacts in the main application window allows the user to be more efficient in deciding whether a new contact needs to be created. Having the three buttons—Create, Display, and Exit—at the bottom of the window results in it taking a bit longer to activate one of these buttons. The design of the dialog box appears to be efficient, with two buttons when adding a contact but only one button when displaying existing contact data.

22.3.1.2 Learnability

The Version A GUI design is fairly easy to use. Two issues may cause a first-time user to be confused. First, it may take a moment before seeing the three buttons at the bottom of the main application window. Second, the large amount of space in the main application window with nothing being displayed may be initially confusing. On the positive side, the messages describing validation errors or success assist in the user's understanding of the application.

22.3.1.3 User Satisfaction

In order to assess this HCI design criteria, we would need to interview or survey users to obtain information on how satisfied they are with this GUI design. Since the ABA has not been deployed as a software application that others may use, this criteria cannot be assessed.

22.3.1.4 Utility

The user gets immediate feedback on each transaction they perform. Displaying a list of existing contacts is very useful to the user.

22.4 SD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when creating a graphics-based user interface as part of a top-down design approach.

22.4.1 SD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 13, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

22.4.2 SD Personal Finances: GUI Design

The user interface design for the personal finances case study is potentially quite large. It needs to support user actions to create, read, update, and delete accounts, transactions, and reports. Ideally, we would like a single window to provide quick access to each of these user actions.

Figure 22.7 shows a GUI design for the main application window. Accounts and reports would be listed along the left edge of the window. The majority of the main window will either show transactions for the selected account (shown in Fig. 22.7) or the selected report. Creating, updating, and deleting transactions would be done via choices on the Transactions menu. Similarly, accounts, reports, and labels may be created, updated, and deleted using these choices from the respective menu. The Labels menu also allows the user to display the entire list of labels.

The statechart Fig. 22.8 shows the modifications to the view component as a result of using a graphical user interface for the personal finances application. The statechart shows the user interactions associated with opening an account and adding, deleting, or updating a transaction. Once the transactions are displayed, the user selects an appropriate choice from the Transaction menu to initiate processing. When the user selects a different account from the list, the list of transactions associated with this account is displayed.

The structure chart in Fig. 22.9 shows the flow of information for the Transaction Management function. Since the user interface is a GUI, the view logic should drive the processing of the controller and model components. As shown in this structure chart, the Transaction Management UI (user interface) functions (e.g., Add Trans UI) will call a controller function (e.g., Add Trans), which would call a model function to store the data. A similar function structure would exist for account and report management.

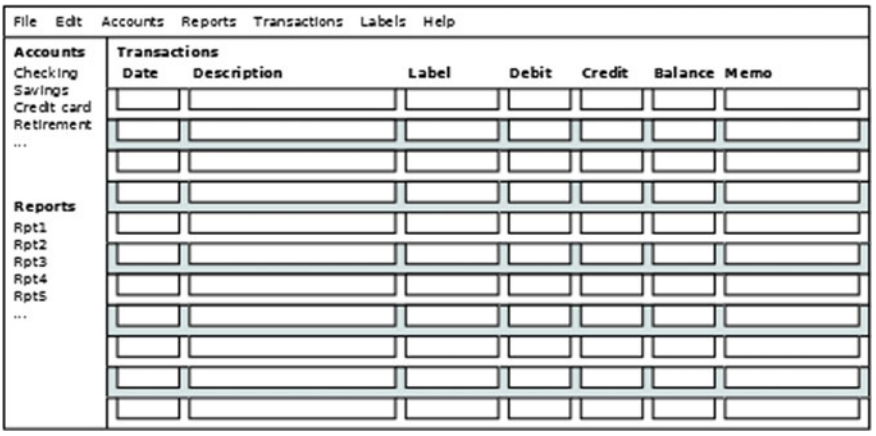


Fig. 22.7 Personal finances—GUI main window

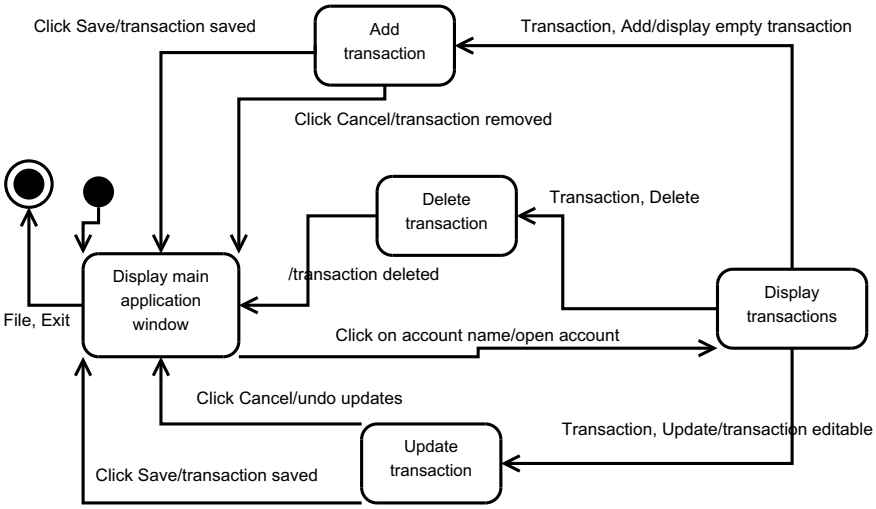


Fig. 22.8 State machine diagram—PF user interactions

22.4.3 SD Personal Finances: Evaluate GUI Design

With a single window supporting the application, efficiency, learnability, and utility may be assessed based on the design of this one window. As noted, the menu is used to create, update, or delete accounts, labels, reports, and transactions. Assuming keyboard hot keys are associated with these menu choices, the user has the option of using a pointing device or the keyboard to activate an action. When an individual transaction is either created or updated, the GUI for that transaction will display save and cancel buttons. One of these buttons must be clicked to indicate the end of the create or update action.

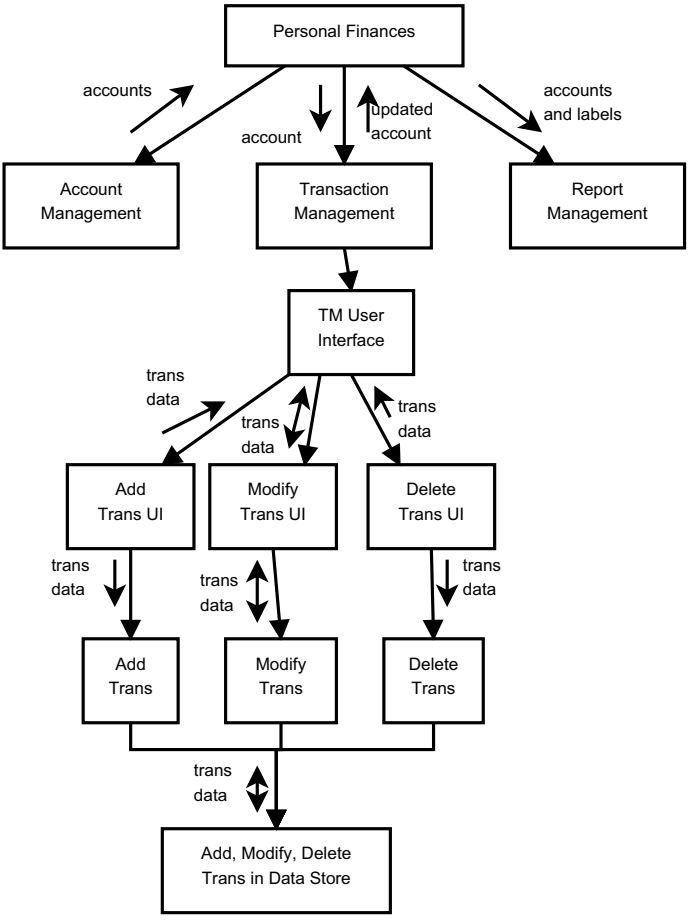


Fig. 22.9 Personal finance structure chart—GUI

From a learnability perspective, anyone familiar with a checking account transaction register, a small booklet used to record checking account transactions, will recognize the layout of transactions on the main application window. No attempt has been made to describe the GUI for creating, updating, or viewing a report. This is left as an exercise.

Given how similar the GUI design is to a paper-based checking account transaction register, the interface should provide the features necessary to support someone's personal finances.

22.5 SD: Post-conditions

The following should have been learned when completing this chapter.

- A graphical user interface provides many alternatives for developing an HCI design. Creating a user interface design that is efficient, easy to learn, and useful is challenging because of the many choices one has for representing information in a graphical form.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You have created and/or modified models that describe a structured software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-On Exercises

1. Modify the GUI design and code for the ABA in any of the following ways.

   a. Allow a double-click on a contact name in the list to display the information for the selected contact.

   b. Move the buttons to the top of the main application window.

   c. Add hot keys to the buttons to create or display a contact.

   d. Change the Listbox to a Grid of Labels so the name, phone number, and email address can be displayed using three columns. A Scrollbar will also be needed.

2. Modify the design for the personal finances application in any of the following ways.

   a. Show what a GUI would look like to create or update a report.

   b. Show what a GUI would look like to view a report.

3. Use an existing code solution that you've developed, develop alternative GUI-based HCI design models that show ways in which a user could interact with your application. Apply the HCI design criteria to your design models. How good or bad is your design?
4. Use your development of an application that you started in Chap. 4 for this exercise. Modify your HCI design to use some combination of graphical user controls, and then evaluate your design using the HCI design criteria.
5. Continue Hands-on Exercise 3 from Chap. 13 by developing an HCI design using some combination of graphical user controls. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 13 for details on each domain.

   - Airline reservation and seat assignment
   - Automated teller machine (ATM)
   - Bus transportation system
   - Course-class enrollment
   - Digital library
   - Inventory and distribution control
   - Online retail shopping cart
   - Personal calendar
   - Travel itinerary



Is Your Design Clear, Concise, and Complete?

23

The objective of this chapter is to discuss quality assurance (QA) methods that should be used to ensure your design is clear, concise, and complete.

23.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_23

23.2 Concepts and Context

The terms *quality* and *assurance* often get misused or are used interchangeably to represent the same concept. One way to remember the distinction between these two terms is to understand the history of the terms. The historical perspectives described below are based on the way in which these two terms are being used in this chapter. Quality has its roots in manufacturing, where the goal is to eliminate defects from being present in the final product. Assurance has its roots in security, where the goal is to produce a product that gives the user a degree of confidence that it is safe to use. These terms are now described from a software perspective.

Software Quality: Software is free of bugs/defects and meets user needs.
Software Assurance: Software is free of vulnerabilities and functions as intended.
Software Quality Assurance: Software is free of bugs/defects, free of vulnerabilities, meets user needs, and functions as intended.

While it may be impossible to prove that a piece of *meaningful* software contains no defects or no vulnerabilities, it is still appropriate to have as a goal the complete elimination of defects and vulnerabilities. The notion of proving the absence of defects (or vulnerabilities) is the holy grail of computer science. If some future includes the ability to mathematically prove that a piece of software, no matter how large or complex, contains no defects (or vulnerabilities), than the process of creating software is likely quite different from how we do it today.

While *meets user needs* and *functions as intended* sound like similar goals, these are actually quite different. Meeting user needs has to do with understanding the needs of the users, documenting these needs, and then translating these needs into a design and implementation. Throughout a software development process, the two significant challenges are (1) misinterpreting someone's needs, whether expressed verbally or in writing, and (2) understanding and dealing with the fact that user needs will change over time. In contrast, functions, as intended, is more of an engineering goal. If the requirements say the software must do X, Y, and Z, can we demonstrate that the software indeed does X, Y and Z? The challenge with this goal is to ensure that the software meets the requirements without also having unintended consequences. For example, perhaps the X, Y, and Z features are indeed in the software, but using these features in a certain way also allows the user to do Q, which is not a requirement. If feature Q happens to be a vulnerability then our unintended consequence has resulted in a less safe piece of software.

To do the best we can in ensuring our software quality assurance goals have been met, we need to think about the following four questions. The sections that follow will describe quality assurance methods which may be used to address one or more of these questions.

- What can we do to find as many bugs/defects as possible?
- What can we do to find as many vulnerabilities as possible?
- What can we do to determine whether user needs are being met?
- What can we do to determine whether the software functions as intended?

The thinking regarding the first two questions is perhaps obvious, but very important—the more defects and vulnerabilities we find and correct during software development, the fewer of these will exist once the software is deployed.

23.2.1 Software Quality Assurance

Software quality assurance is also described as a collection of verification and validation methods. These two terms are described using the following two questions [1].

1. Are we building the product right?
   This is called *software verification*. We are building the product right if the software meets the requirements and design specifications. We can (and should) apply verification methods throughout the entire software development process.
2. Are we building the right product?
   This is called *software validation*. We are building the right product if the software is meeting user needs and the requirements and design specifications are correct in the first place. We can (and should) apply validation methods throughout the entire software development process.

The remaining subsections introduce some of the common verification and validation methods.

23.2.2 Formal Review

A formal review may verify and/or validate a software artifact.[1] The formal review process described below is modeled after the Fagan inspection process [2].

A formal review is a process that involves a moderator, one or more inspectors, and an author. The moderator assigns a specific task to each inspector and gives each inspector the maximum amount of time to spend on the task. Each inspector looks at the artifact(s) under review and logs any defects that are found. An inspection meeting is then held which is facilitated by the moderator and attended by all inspectors and the author. During this meeting each inspector reads their list of defects out loud. There is no debate about a defect—if an inspector identifies something as a defect, then it's a defect. However, the author may ask questions to clarify their understanding of a defect. After the inspection meeting the author is tasked with addressing each defect. By *addressing each defect*, the author may correct the defect or provide an explanation of why it is not a defect. The moderator and author would then review the defect logs and the way in which the author addressed each defect.

[1] A software artifact is a document produced during a software development process; it contains descriptions of the software. An artifact may describe project plans, user needs, design decisions, code, tests, or some combination of these.

A few key points regarding this process are as follows:

- The moderator should have experience as an inspector and (ideally) in managing the inspection process.
- The entire inspection process is carefully managed from a time/resource perspective. The adage *time is money* is certainly applicable in this case. Software development projects usually have a budget. This budget would (hopefully) include costs (i.e., time and resources) to perform various quality assurance methods—including formal reviews. Giving each inspector the maximum amount of time to spend on the task they've been assigned allows the moderator to manage the costs of each formal review.
- The lack of any debate regarding the merits of a specific defect may seem odd. This will be addressed below as the other QA methods are being introduced.

The following sections provide a detailed description of the formal review process.

23.2.2.1 Formal Review: History

The process described below is based on Fagan's inspection process. Michael Fagan, while working for IBM in the 1970's, developed an inspection process that was used during the development of the S/360 operating system [2].

23.2.2.2 Formal Review: Roles and Process Steps

There are three types of participants in a formal review. Each participant adheres to one of the following roles.

Moderator: The leader of the inspection, manages the inspection process.
Tester: An inspector; someone that is assigned a specific role as a reviewer of artifacts.
Author: The primary author of the artifact(s) being reviewed or a representative from a group of authors.

The steps to be followed when doing a formal review are as follows:

1. Plan Inspection
2. Initial Meeting
3. Preparation
4. Inspection Meeting
5. Rework
6. Follow-up

23.2.2.3 Formal Review: Process Details

A detailed description of each step is provided below.

1. Plan Inspection
   The moderator performs this step, which involves

   a. Collecting the artifacts (i.e., documents) to be inspected.
   b. Collecting other artifacts that were used to develop the document(s) under inspection.
   c. Identifying and obtaining commitment from testers.
   d. Scheduling two meetings: the initial meeting and the inspection meeting.
   e. Assigning each tester a specific responsibility (see Sect. 23.2.2.4 for details).
   f. Indicating how much time each tester should take to inspect the artifact(s).

2. Initial Meeting
   The initial meeting is attended by the moderator, all testers, and the author. The moderator facilitates this meeting, which involves

   - Describing to each tester their responsibility and the time they have to complete their inspection.
   - Making available the artifact(s) to be inspected and any other related documents.

3. Preparation
   Each tester performs the following to prepare for the inspection meeting.

   a. Inspect artifact based on their responsibility (see Sect. 23.2.2.4 for details).
   b. Record each defect found in the artifact(s) being inspected on a defect log.
   c. Record the time taken to do the inspection.
   d. Submit their defect log to the moderator prior to the inspection meeting.

   The moderator will collect the defect log from each tester and make a copy of each defect log.
4. Inspection Meeting
   The inspection meeting is attended by the moderator, all testers, and the author. This meeting involves the following.

   - The moderator controls the meeting. There is *no arguing* during the meeting about whether something is a defect. If a tester says it's a defect, then it's a defect!
   - The moderator hands the original copy of each defect log to the appropriate tester.
   - The moderator hands a second copy of each defect log to the author.
   - One at a time, each tester reads aloud their defects, using their defect log. The author may ask questions to clarify their understanding of a defect. The author may update their defect log with additional notes to further document a defect.

5. Rework
   The author (or group of authors) will then *resolve* each defect. *Resolving* a defect may result in either

   - The artifact(s) being updated to correct the defect.
   - Or an explanation as to why this is not a defect.

   In either case, the defect resolution is recorded on the defect log.
6. Follow-up
   The moderator reviews the defect log updated by the author(s). This review is done to ensure that each defect has been *resolved*. The moderator may decide to schedule a second inspection if the number of defects found or the severity of the defects warrants a second review.

23.2.2.4 Formal Review: Tester Responsibilities

A tester has the following responsibilities.

- Find as many defects as possible!
- Focus on the particular type of inspection they've been assigned by the moderator. The types of inspections include the following.
  - Clear and Concise
    - Is the artifact easy to understand?
    - Are the diagrams and text written in a clear understandable manner?
    - Is the artifact verbose (overly wordy)?
  - Complete
    - Does the artifact include everything it needs to include?
    - Does the artifact include extraneous information?
  - Consistent
    - Is the artifact consistent with itself?
    - Is the artifact consistent with other artifacts in the project?
- For complete and consistent defect types
  - The tester may be given other documents that were used to develop the artifact under inspection.
  - The tester uses these other documents to check for completeness or consistency.

Table 23.1 Defect log: header

Project:
Artifact:
Author:
Date:

Table 23.2 Defect log: defect recording section

Number	Type	Inject phase	Removal phase	Fix time	Fix reference
Description:					

23.2.2.5 Formal Review: Example

Let's assume that a design document is to be inspected. The requirements document would have been used (as an input document) to develop the design. In this scenario, the moderator could assign tester responsibilities as follows:

- Tester 1 is assigned clear and concise. An example defect would be *the design artifact has a sentence (page 1, paragraph 3) that contains 60 words*.
- Tester 2 is assigned complete, and is given both design and requirements artifacts. An example defect would be *the design artifact is missing a design element for requirement 5.4*.
- Tester 3 is assigned consistent. An example defect would be *the design artifact uses terms* database *and* file *to mean the same thing. The design should consistently use one of these terms*.
- Tester 4 is assigned consistent, and is given both design and requirements artifacts. An example defect would be *the design artifact uses the term loan while the requirements artifact calls it a liability*.

23.2.2.6 Formal Review: Defect Log

A typical defect log has a header that identifies project, artifact, author, and date of the inspection. This is shown in Table 23.1.

Each defect is recorded in a section of the log that repeats itself to fill-out a page. Table 23.2 shows a typical defect recording section.

The information specified for each defect is now described. The tester would fill-out the Number, Type, Inject Phase, and Description fields during step 3 Preparation. The author would fill-out the Removal Phase, Fix Time, and Fix Reference during step 5 Rework.

Number: Enter the defect number. For each artifact, use a sequential number starting with one.

Type: Enter the defect type number from the list of defect types described in Table 23.3. Use your best judgment in selecting which type applies.

Inject Phase: Enter the SDP phase during which the defect was injected.

Removal Phase: Enter the SDP phase during which the defect was fixed (corrected).

Fix Time: Enter the time, in hours, that you took to find and fix the defect. This time is often an approximate value.

Fix Reference: If you or someone else injected this defect while fixing another defect, record the number of the improperly fixed defect. Leave blank when the fix is not associated with a prior defect incorrectly fixed.

Description: Write a succinct description of the defect that is clear enough to later remind someone of the error and how or why it was made.

Table 23.3 Formal review defect types

Type	Name	Description
10	Documentation	Design: does not conform to guidelines Code: comment not appropriate/correct, comment can be misunderstood
20	Syntax	Design: spelling or grammar mistake Code: spelling mistake, format of code inconsistent or hard to read
30	Build, Package	Code: change management not documented, library use not appropriate/incorrect, version control not documented
40	Assignment	Code: variable name not descriptive, using duplicate variable name leads to misunderstanding, variable scope not appropriate
50	Interface	Design: missing an interface design, logical error in interface design, technical error in interface design, HCI design is confusing or inefficient Code: function/method calls are too complex, I/O is too complex
60	Checking	Design: validation is missing, has logical errors, or has technical errors Code: error messages are vague/incorrect, missing validation checks
70	Data	Design: missing a logical or physical data design, error in logical data model, error in physical data model Code: data structure use not appropriate
80	Function	Design: missing a functional requirement, logical error found, technical error found Code: logic error found, use of pointers/references not correct, selection, iteration, or recursion errors
90	System	Design: architecture incorrect or missing a component Code: does not match design
100	Environment	Code: design, compile, test, or other support system problems

23.2.2.7 Formal Review: List of Defect Types

Table 23.3 identifies different types of defects that may be found in an artifact. This list is not intended to be complete but rather serve as a basis for developing a more complete list of defect types. An organization should develop their own defect types and use these consistently in their defect logs. This allows an organization to measure quality improvements for each defect type, and develop training based on defect types that consistently appear in their artifacts.

23.2.3 Informal Review

An informal review may verify and/or validate a software artifact. There are a few variations in how an informal review is conducted. The description below is intended to provide a sense for how an informal review is performed and begins to explain the differences between a formal and informal review.

In an informal review, each reviewer looks at the artifact(s) and identifies the defects they find. A review session is then held which is attended by all reviewers and the author. The reviewers and the author discuss each defect found by a reviewer. Consensus should be obtained about the list of defects that need to be corrected. After the review session, the author is tasked with fixing each identified defect.

One key point regarding an informal review

- I have participated in these types of informal reviews where there was no facilitator (aka moderator) for the discussion. Instead, the artifact author or one of the reviewers *took charge* of the meeting as an unofficial moderator. This can be an effective approach assuming that the individuals involved consider these discussions as constructive criticism of the author's work. However, I've seen situations where discussions become emotional as egos clash between the artifact author and one or more reviewers. In these situations, the effectiveness of the informal review is greatly diminished, and participants walk out of the session thinking it was a waste of their time.

The lesson I've learned from these experiences is to have an individual facilitate the informal review session without also being an author or reviewer. This allows the facilitator to (hopefully) remain objective during discussions.

23.2.4 Design/Code Walkthrough

A design or code walkthrough may verify and/or validate a software artifact. There are a few variations to how a walkthrough is conducted. The description below is intended to provide a sense for how a walkthrough is performed and begins to explain the differences between a formal review, an informal review, and a walkthrough.

In a design or code walkthrough, a meeting is held that is attended by all reviewers and the author. At this meeting, the author describes each artifact in a fair amount of

detail. As the author is walking through an artifact, the reviewers may comment on a particular aspect of the artifact that they feel could be improved. Consensus should be obtained about the parts of each artifact that should be modified/improved. After the meeting, the author is tasked with making each modification/improvement. The only substantive difference between a design walkthrough and a code walkthrough is the type of artifact being reviewed (i.e., review a structure chart or class diagram versus review source code).

Similar to the discussion regarding informal reviews, having the right person facilitate discussions during a walkthrough is an important consideration.

23.2.5 Customer Survey

A customer survey is typically done to validate a software artifact (i.e., does the software meet your needs?). However, a survey instrument could be created that verifies a software artifact (i.e., does the software satisfy the requirements?). A customer survey is generally completed anonymously and has more significance the larger the number of survey responses obtained.

Using a customer survey as a validation method tends to focus on one or more of the following.

- Assess the level of satisfaction from the user community.
- Elicit feedback on parts of system that are used and/or not used.
- Identify new or changing needs that the system must meet.

23.2.6 Software Testing

Software testing may verify and/or validate a software artifact and is typically applied only to source code artifacts (or only to an executable image produced from source code artifacts). While there are a handful of different types of software testing that may be performed, at the core of software testing is the notion of a test case. In essence, doing software testing involves developing test cases and then performing/executing each test case.

In its simplest form, a test case identifies the input data to be injected into the software and the expected results. When the documented expected results do not match the actual results produced by the software, a defect exists. This defect is either located in the software (i.e., we've found a bug) or in the test case (i.e., perhaps we did not accurately describe the expected results). Test code is often written as a way to implement test cases. In this case, the test code may automatically verify the results or may leave it up to visual inspection to confirm the results match what is expected.

Generally, there are two types of test cases. Note that a test case may satisfy both types described below.

White box: A single case that tests a specific control flow through a function/method, component, or program. Each test case is developed by knowing what the code does and how it does it.

Black box: A single case that tests one or more requirements or use cases. Each test case is developed by knowing the requirements (i.e., what the software is supposed to do).

Four different types of software testing are now described.

23.2.6.1 Unit Testing

Unit testing is generally done as the code is being developed. It's purpose is to verify that each function/method performs as it was designed. Thus, unit testing is performed on the smallest units of code—statements, functions, and methods. It is a verification method since it is looking at whether the design specifications were correctly implemented. It is *not* a validation method since unit testing does not focus on user needs.

When creating unit tests, each test case is written to test an individual function or method. That is, each function/method is tested independent of all other functions/methods. Each unit test case is a white box test case since it is written with knowledge of what the code is doing and how it is doing it. At a minimum, the number of unit test cases for a function/method should equal the number of distinct control flow paths through the function/method.[2]

23.2.6.2 Integration Testing

Integration testing is generally done as the code is being developed. It's purpose is to verify the design of each component/module that makes up the software application or system. Thus, integration testing is about testing the interactions between functions/methods that are combined to form a higher level design element (e.g., a component or module). For example, a Model–View–Controller design would have a set of integration test cases for the model component, another set of test cases for the view component, and a third set of test cases for the controller component. It is a verification method since it is looking at whether the design specifications were correctly implemented. It may also be a validation method if some of the cases are testing user needs. (Here, the user need may be expressed as a fundamental use case or requirement. By fundamental, I mean that the user need is so basic to the software that it is often implied by users to exist. For example, a user wanting to have a software program playing a card game would likely assume that there is a deck

[2]While the topic of creating white box test cases is beyond the scope of this book, an excellent resource to learn more about developing test cases is *The Art of Software Testing* by Glenford J. Myers. This book describes other types of software testing not included in this book and also discusses debugging, extreme testing, and testing Internet applications. The second edition of this book was published in 2004.

containing 52 cards, with 13 distinct card values in each suit and 4 distinct suits. The need for a deck having 52 cards may go unsaid by the user.)

When creating integration tests, each test case is written to test an interaction between two or more functions or methods. Most integration test cases are white box since they are written with knowledge of what the code is doing and how it is doing it. Some of the unit test cases may be used for integration testing, but it is expected that new cases are developed to specifically test the integration of functions/methods within a component or module. Black box test cases may also be developed during integration testing. These cases would address user's needs and represent the validation portion of the integration test.

23.2.6.3 System Testing

System testing is generally done at certain project milestones and after the code has been integrated. It's purpose is to verify the software meets requirements, as understood by developers. Thus, system testing is about testing all of the components/modules that make up the application or system. It is a verification method since it is looking at whether the design specifications were correctly implemented. It is also a validation method since it would now include the testing of use cases or requirements, both documented and implied.

When creating system tests, each test case is written to test a specific scenario or processing flow through the entire system. Some of the unit and integration test cases may be used for system testing, but it is expected that new cases are developed to test the entire system. New white box test cases would focus on the interaction of the components that make up the system while new black box test cases would focus on user needs.

23.2.6.4 Acceptance Testing

Acceptance testing is generally done by end-users. It's purpose is to validate the software meets their needs. Thus, acceptance testing is about testing those features the user community feels are most important or relevant.

Acceptance testing would involve the creation of all new test cases since this type of testing is most often performed by users. All of these test cases would be black box and are designed to validate a user's need. Thus, acceptance testing is a validation method but *not* a verification method.

23.2.7 Summary of QA Methods

Table 23.4 shows a summary of the non-testing QA methods, but does not include the customer survey method. This table shows the differences between the three types of *review* methods as described in this chapter. Please note, the description of these QA methods may be inconsistent with other sources, either due to terminology used or the culture within the information technology organization.

Table 23.4 Summary of non-testing methods

	Formal review	Informal review	Design/Code walkthrough
Well-defined process?	Yes	No	Maybe
Participants come prepared?[a]	Yes	Yes	No
Author describes artifact?[b]	No	No	Yes
Discussion/consensus on defects?	No	Yes	Yes
Example use as verification	Check design against requirements		
Example use as validation[c]	Check requirements against use cases		Check design against use cases

[a] When a participant comes prepared, they've identified and listed defects they found
[b] Author steps through artifact and explains its content
[c] Assumes use cases were developed by users

Table 23.5 Summary of testing methods

	Unit	Integration	System	Acceptance
Uses white box test cases?	Yes	Yes	Yes	No
Uses black box test cases?	No	Yes	Yes	Yes
Does verification?	Yes	Yes	Yes	No
Does validation?	No	Maybe	Likely	Yes
Done by programmer?	Yes	Likely	Maybe	No

Table 23.5 shows a summary of the testing QA methods. This table shows the differences between the four types of *testing* methods. These descriptions may be inconsistent with other sources, either due to terminology used or the culture within the information technology organization.

To conclude the goal of verification and validation methods are to

- Find as many defects as possible
- Find as many vulnerabilities as possible
- Ensure software meets user needs
- Ensure software functions as intended

23.3 Software QA in Software Design

Learning to design software is challenging for many reasons, one of which is the fact that we cannot test a design like we can test code. How do we know whether our design is any good if we cannot test it? The answer is hopefully obvious at this

point—the use of formal review, informal review, and design walkthrough allows us to assess the quality of our design based on the results obtained from using one of these QA methods.

23.3.1 Formal Review of ABA Software Design

The author strongly recommends the use of the formal review method when assessing the quality of a software design. The formality of this review method is a benefit, but just as important is the fact that this method avoids debate about what is good and not so good about a design. While most professional environments would result in a healthy discussion during an informal review or design walkthrough, the author has seen instances where these discussions get too personal. In these situations, an author's ego might prevent them from acknowledging the defects or arguing in defense of their design. Or, a reviewer might express their opinion too strongly resulting in a degradation in the professionalism of the review meeting. In contrast, an experienced moderator will facilitate the inspection meeting (which is part of the formal review process) to avoid a confrontational environment.

23.3.1.1 Formal Review of object-oriented design (OOD)

For the case study, we have a list of requirements and five design artifacts. For the OOD approach described in Chap. 15, the ABA Version B solution has a package diagram, a class diagram, a statechart, and two communication diagrams. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two inspectors are all that is needed.

1. One inspector would spend 30 min looking at the requirements and the design artifacts to ensure all of the requirements have been addressed in the design.
   For example, the two validation requirements should result in a design that continues to obtain user input until a valid value has been entered. The statechart should show the validation via states and transitions. Similarly, the OOD communication diagram should list a sequence of messages that shows validation being performed. When the inspector cannot confirm these requirements are in the design, a defect would be identified and documented on the defect log.
2. The second inspector would spend 30 min looking only at the design artifacts to ensure they are clear, concise, and consistent with each other.
   For the OOD approach, any method specified in the communication diagram should be listed in the correct class within the class diagram. Any deviation between these two diagrams results in a defect being identified and documented on the defect log. A defect resulting in a simple correction is a misspelling of a method name in one of the diagrams. A more complex defect might be when there are many methods specified in the communication diagram that does not appear anywhere in the class diagram.

23.3.1.2 Formal Review of structured design (SD)

For the case study, we have a list of requirements and nine design artifacts. For the SD approach described in Chap. 16, the ABA Version B solution has three structure diagrams, five structure charts, and a statechart. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two inspectors are all that is needed.

1. One inspector would spend 30 min looking at the requirements and the design artifacts to ensure all of the requirements have been addressed in the design.
   For example, the two validation requirements should result in a design that continues to obtain user input until a valid value has been entered. The statechart should show the validation via states and transitions. Similarly, the SD structure diagrams and structure charts should identify validation functions being called. When the inspector cannot confirm these requirements are in the design, a defect would be identified and documented on the defect log.
2. The second inspector would spend 30 min looking only at the design artifacts to ensure they are clear, concise, and consistent with each other.
   For the SD approach, a function specified in a structure diagram should also appear in a structure chart, as long as the two diagrams represent the same component or module within the design. Any deviation between these two types of diagrams would result in a defect being identified and documented on the defect log. A defect resulting in a simple correction is a misspelling of a function name in one of the diagrams. A more complex defect might be when there are many functions specified in the structure diagram that do not appear anywhere in the structure chart.

23.3.2 Design Walkthrough of ABA Software Design

A design walkthrough is an effective QA method when there are some experienced reviewers participating in the walkthrough. As indicated earlier, having a facilitator of the walkthrough session who is respected by all participants can help avoid disagreements from spilling over into unprofessional behavior.

For the OOD case study, we have a list of requirements and five object-oriented design artifacts. For the OOD approach described in Chap. 15, the ABA Version B solution has a package diagram, a class diagram, a statechart, and two communication diagrams. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two reviewers are all that is needed.

For the SD case study, we have a list of requirements and nine design artifacts. For the SD approach described in Chap. 16, the ABA Version B solution has three structure diagrams, five structure charts, and a statechart. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two reviewers are all that is needed.

Given the small scope of the case study, having two reviewers and the author participate in a walkthrough would be appropriate. One of the two reviewers should

have significant experience. When the reviewers are not familiar with the project, the author would likely begin by describing the requirements within scope of this design. As the author explains each design artifact, the reviewers would ask questions to clarify their understanding of the design. Any possible flaws discovered by a reviewer would be discussed and a determination would be made as to whether a defect has been found. When consensus is achieved, the author would record the defect so it can be corrected at a later point.

A challenge in doing a design walkthrough is the temptation to also discuss ways to correct a defect that has been found. This should be avoided during the design walkthrough session. Why? First, any discussion about correcting a defect may take a significant amount of time. This time is better spent focusing on identifying as many defects as possible. Second, discussing ways to correct a defect will distract the reviewers from their primary responsibility—to identify defects. When a reviewer has ideas about how to correct a design defect, this should be noted by the author. The author can then follow-up with the reviewer at a later time.

23.4 Software QA in Software Development

Learning to develop software is challenging for many reasons, one of which is the many different skills needed to effectively develop quality software. These skills include planning the work to be completed, analyzing the needs of the users, developing design alternatives and selecting a design to implement, implementing the selected design, testing your implementation, distributing the resulting software to the user community, and then supporting the software once it starts to be used.

The quality assurance methods described in this chapter may be used at many points during a software development project. Some examples of when to use these methods follows.

- Have experienced project managers do an informal review on the project plan. These reviewers may identify important milestones or tasks that need to be included in the plan.
- Have selected users do a formal review on the requirements document. The requirements may be assessed based on a high-level scope document previously produced or on a collection of use cases developed by the user community.
- Have other designers do a formal review on the design. These inspectors could assess the design based on the requirements document, based on the technical merits of the design, and based on whether the design is clear, concise, and complete.
- Have experienced designers do a design walkthrough. These reviewers may identify important technical shortcomings of the design or may think the design can be simplified.
- Have experienced testers do an informal review on the test plan/strategy and test cases. These reviewers may identify additional test cases designed to detect certain vulnerabilities in the software.

23.5 Post-conditions

The following should have been learned when completing this chapter.

- You understand the purpose of the following quality assurance methods—formal review, informal review, design/code walkthrough, unit testing, integration testing, system testing, and acceptance testing.
- You've assessed the quality of design artifacts using a formal review, informal review, or design walkthrough.

Exercises

Discussion Questions

1. Software quality assurance includes activities like formal reviews and design walkthroughs. How might these types of activities be used to improve the security of a design?
2. What are some reasons why a formal review is better than an informal review in discovering defects or vulnerabilities?
3. Is there a situation where an informal review may be better to use than a formal review?

Hands-on Exercises

1. Use an existing design solution you've developed and do a formal review to identify as many defects and vulnerabilities as possible. Improve your design based on the list of defects in the defect log.

References

1. Boehm B (1981) Software engineering economics. Prentice Hall, Upper Saddle River
2. Fagan M (1986) Advances in software inspections. IEEE Trans Softw Eng 12(7)



Software Design and Security

24

The objective of this chapter is to introduce information security as an important software design topic.

24.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_24

24.2 Concepts and Context

There are many security concepts and topics that fall within the information security domain. This chapter will briefly describe a number of design principles that researchers suggest will lead to a more secure software system.

24.2.1 Information Security Foundations

The three goals when developing information security are listed below. These three goals form the acronym CIA.

Confidentiality: Avoid unauthorized disclosure of information.
Integrity: Information has been altered only in authorized way.
Availability: Information is accessible and modifiable in a timely fashion by authorized entities.

One of the challenges with information security is how to balance these three goals when improving one goal often reduces another. For example, implementing more security mechanisms to improve confidentiality will adversely affect availability. Two-factor authentication is an effective way to increase confidence that a person is who they say they are, but this affects availability when the person's second factor (e.g., cell phone) is not working or lost.

Four additional concepts are also important to understand when you design security into a software product.

Anonymity: A property that certain records or transactions are not attributable to any individual.
Assurance: How trust is provided and managed in computer systems using policies, permissions, and protections.
Authenticity: Ability to determine whether statements, policies, and permissions issued by persons or systems are genuine.
Non-repudiation: User is responsible for their actions and should not be able to deny what they have done.

Anonymity and non-repudiation represent opposing views on whether identifying data is maintained by a system. A crisis prevention hotline would likely use a system which does not require the caller to identify themselves. In contrast, a financial system will likely record information about a user when they perform an action resulting in changes to the data.

24.2.2 Designing Information Security in Software

The first set of eight design principles come from an article written by Saltzer and Schroeder that was published in 1975 [1].

Economy of mechanism: Your design should be as small and as simple as possible. This allows quality assurance methods (e.g., formal reviews, design walkthroughs) to have the greatest chance of finding security vulnerabilities.

Fail-safe defaults: Your design should base access to data on permission rather than exclusion. The default situation should be to deny access. Only when the protection scheme identifies conditions to permit access should the data be accessible. In contrast, creating a scheme which describes the conditions for refusing access presents the wrong psychological perspective for a secure software design. To state this another way, the rules needed to express permission are likely to be simpler to understand than the rules needed to refuse permission.

Complete mediation: Each access to an object must be checked for authority. This implies that a foolproof method of identifying the source of each request must be devised and suggests improvements made for performance reasons, e.g., by remembering the result of a previous authority check, be examined skeptically. Care must be taken when a change in authority occurs, to ensure the remembered results are systematically updated.

Open design: Your design should not be a secret. This allows reviewers to comment on the security mechanisms being used while protecting the keys or passwords used by the mechanisms. You should assume your design is not a secret.

Separation of privilege: Requiring two (or more) keys to unlock a protection mechanism is more robust than allowing access after presenting a single key. This allows the two (or more) keys to be physically held by different individuals, organizations, or systems. Examples of this security design principle may be seen in the use of bank safe deposit boxes and in launching a nuclear weapon.

Least privilege: Grant users/systems only those access rights needed to perform their tasks. When software must access an information asset, it should ideally be granted this access only for the moments in time when it is using the information asset. This limits the damage resulting from an accident or error.

Least common mechanism: Minimize the number of mechanisms used by more than one user/system. Each security mechanism shared among most/all users or shared among systems, especially when shared variables are used, represents a potential information path that may unintentionally compromise security. Do not share objects and protection mechanisms, instead create separate instances for each user or system interface.

Psychological acceptability (make security usable): Design the human–computer interface (HCI) for ease of use, to promote correct use of the protection mechanism. A well designed HCI will match the protection mechanism to the user's mental image of their protection goals.

One of the eye-opening aspects of Saltzer and Schroeder's research is the publication year was 1975. This leads one to wonder …what has the software industry been doing for the past 40+ years? Why are there so many vulnerabilities in software systems and applications? Thinking about these eight principles, none of them seem all that challenging to design and implement. And yet the software industry seems to be playing catch-up each time we hear of another large data breach. One thing

is clear, the software industry needs to change some fundamental aspects of how software is developed in order to try to get ahead of the malicious attackers.

In 2013, Gary McGraw expanded on Saltzer and Schroeder's eight security design principles by adding five more [2].

Secure the weakest link: The suite of security mechanisms being used are only as good as the weakest security mechanism being used. The analogy often used is that a chain is only as strong as its weakest link. Likewise, a system is only as secure as its weakest security mechanism.

Defend in depth: Your design should include redundancy and layers of defense. This approach looks to manage security risks by using a diverse set of security mechanisms that provide redundant capabilities or are provided by different software layers.

Be reluctant to trust: Be skeptical of security protections not within your software system. The quote "trust but verify" often used to describe international agreements to limit nuclear weapons is a good motto to follow when it comes to placing your software within an operating environment. A cloud provider may claim to provide certain protections, but it is best to verify these as best you can.

Promote privacy: Your design needs to consider the types of personal information you are collecting from a user. Do you really need the information you are requesting? Should the personal information be encrypted? Does this data really need to be persistently stored?

Use your resources: Nobody knows everything about what a good software security design looks like. Talk to others about the design choices you are making. Have experts with different backgrounds review your design.

So what should the software industry do to address the large number of vulnerabilities found in software? It should, *no it must*, incorporate these 13 security design principles into the software development processes (SDP) it uses to design software. In addition, thinking about security should be an integral part of the entire process. Thus an SDP should describe tasks the development team should perform to think about security while developing a project plan, while documenting software needs, while developing a quality assurance plan, while designing the software, and while coding and testing the software.

24.2.3 Cryptographic Concepts

Cryptography is used to provide different security mechanisms. The following describes four uses of cryptography.

24.2.3.1 Cryptographic Systems

A cryptographic algorithm along with a key may be used to encrypt plaintext data into ciphertext, or to decrypt ciphertext back into plaintext. Two types of cryptographic algorithms are in use today.

Symmetric: The same key is used to encrypt plaintext and decrypt ciphertext.
Asymmetric: One key is used to encrypt plaintext and a different key is used to decrypt ciphertext.

Asymmetric algorithms use a key-pair with one key being public, i.e., known by everyone, and one key being private. The private key is (hopefully) known only by the individual or organization that owns the key-pair. For use as an encryption scheme, a system sending data to the individual or organization will encrypt their plaintext data using the public key. The individual or organization will receive the ciphertext and use their private key to decrypt, producing the original plaintext. Asymmetric encryption is also know as *public-key encryption*.

24.2.3.2 Digital Signatures

A digital signature uses public-key encryption to verify who sent the data. In this scenario, the sender will encrypt their plaintext data using their private key. Typically, the plaintext data includes some identifying information about the sender, which the receiver is expecting to see. The receiver will use the sender's public key to decrypt the data, then look for the *sender's* identifying information in the decrypted plaintext. Since only the keys in the key-pair may be used, and since in theory only the sender knows their private key, a digital signature may be used to verify who sent the data.

24.2.3.3 Cryptographic Hash Functions

A hash function computes a checksum on a data value. A checksum value contains a fixed number of bits, which is usually much smaller than the data value fed into the hash function. A cryptographic hash function is defined mathematically as a one-way function. A one-way hash function produces a checksum given a data value, but it is hard to recreate the data value if all you have is a checksum value. The checksum produced by a cryptographic hash function is commonly called a *message digest*. A cryptographic hash function is an effective way to store passwords without having to store the plaintext value.

24.2.3.4 Digital Certificates

A digital certificate is associated with an entity you want to communicate with. It is used to ensure a public key being used belongs to the entity you want to send data to. A digital certificate originates from a *certificate authority* and combines a public key with identifying information about the entity that owns the public key.

24.3 Use in Software Designs

A software design—structured and object-oriented—can use many of the modeling techniques discussed so far to express the inclusion of security principles into a design. In particular, using data-flow diagrams, IDEF0 function models, and UML state machine diagrams gives you the ability to express a design that adheres to many of the security principles just discussed. *In the following subsections, only those security design principles capable of being expressed by using the modeling technique are described.*

24.3.1 Data-Flow Diagram

A data-flow diagram (DFD) illustrates the structure and behavior of software by describing the processes, data flows, data stores, and external entities that are in the design.

Economy of mechanism: When a DFD expresses a design at a high level of abstraction, there are likely very few design elements in the diagram. This would give the impression the design is simple. In contrast, a DFD expressing significant design details is a good candidate to assess economy of mechanism. In this case, being able to express the details using a small number of DFD elements would represent a simple design.

Fail-safe defaults: A DFD showing detailed processing may express the use of permission rather than exclusion to access data. The challenge with data-flow diagrams is you only have input and output data flows, so some creativity may be needed to express fail-safe defaults.

Complete mediation: Like fail-safe defaults, a detailed DFD may show authorization being checked each time data is accessed.

Open design: Use various design models, including DFDs, to express and publicize your design.

Separation of privilege: Use processes, data flows and data stores to show a design requiring two (or more) keys to unlock a protection mechanism.

Defend in depth: Use DFDs to show redundancy and layers of defense in your design.

Be reluctant to trust: Use DFDs to show security protections which are outside the scope of your system (by using the external entities notation) and to show how you are skeptical of these security protections by having redundant security protections within your system.

Promote privacy: Use processes, data flows, and data stores to show the types of personal information being collected/stored and the security/privacy mechanisms designed to protect these data assets.

24.3.2 IDEF0 Function Model

An IDEF0 function model illustrates the structure and behavior of software by describing the functions and data flows in the design. With four types of data flows—inputs, controls, outputs, mechanisms—an IDEF0 function model can express a significant amount of design details. In particular, the controls data flows can express design constraints (e.g., password required) while the mechanism data flows can express the use of security mechanisms (e.g., AES, which stands for advanced encryption standard).

Economy of mechanism: When an IDEF0 function model expresses a design at a high level of abstraction, there are likely very few design elements in the diagram. This would give the impression the design is simple. In contrast, an IDEF0 function model expressing significant design details is a good candidate to assess economy of mechanism. In this case, being able to express the details using a small number of model elements would represent a simple design.

Fail-safe defaults: The use of control flows and mechanism flows may be used to express a design where access to data is based on permission.

Complete mediation: Like fail-safe defaults, a detailed IDEF0 function model may show authorization being checked each time data is accessed.

Open design: Use various design models, including IDEF0 function models, to express and publicize your design.

Separation of privilege Use functions and the four types of data flows to show a design requiring two (or more) keys to unlock a protection mechanism.

Least privilege: Use functions and the four types of data flows, particularly control and mechanism flows, to show a design which grants users/systems only those access rights needed to perform their tasks.

Defend in depth: Use IDEF0 function models to show redundancy and layers of defense in your design.

Be reluctant to trust: Use IDEF0 function models, in particular input, control and mechanism flows, to show security protections which are outside the scope of your system and to show how you are skeptical of these security protections by having redundant security protections within your system.

Promote privacy: Use functions and the four types of data flows to show the types of personal information being collected/stored and the security/privacy mechanisms designed to protect these data assets.

24.3.3 UML State Machine Diagram

A UML state machine illustrates the behavior of software by describing software states and transitions between those states. While a state is described using a name/label, it may also describe three actions—entry, do, and exit. The entry action occurs when transitioning into the state and the exit action occurs when transitioning out of the state. The do action is processing that is done while in the state. A transition is typically described using a trigger (i.e., what causes the transition to

occur) but may also include an action and a guard condition. Any action specified is performed when the transition occurs. When a guard condition is specified, it must be true in order for the transition to occur. These UML state machine features allow this modeling technique to represent a wide range of behavior.

Economy of mechanism: When a UML state machine expresses a design at a high level of abstraction, it is likely that there are very few design elements in the diagram. This would give the impression that the design is simple. A UML state machine that expresses significant design details is a good candidate to assess economy of mechanism. In this case, being able to express the details using a small number of model elements would represent a simple design.

Fail-safe defaults: The use of trigger and guard conditions on a transition may be used to express a design where access to data is based on permission.

Complete mediation: The use of states and transitions may be used to express a design which shows how each access to an object is dependent on first checking for authority.

Open design: Use various design models, including state machine diagrams, to express and publicize your design.

Separation of privilege: Use states and transitions to show a design requiring two (or more) keys to unlock a protection mechanism.

Least privilege: Use states and transitions to show a design that grants users/systems only those access rights needed to perform their tasks.

Defend in depth: Use state machine diagrams to show redundancy and layers of defense in your design.

24.3.4 Using Other Design Models

The security design principles not listed in the data-flow diagram, IDEF0 function model, and state machine diagram sections are listed below.

Least common mechanism: In an object-oriented design, a UML communication or UML sequence diagram will show when object instances are created. To support this principle, each security mechanism object would need to have a distinct instance created instead of sharing one instance for many users or for many system interfaces. In a structured design, use of local variables within any function supporting a security mechanism would ensure the variables are not shared across function calls. However, the SD design models discussed in this book do not express the use of local variables.

Psychological acceptability: None of the design models introduced in this book are useful in describing a human–computer interface (HCI) design. Determining whether an HCI design is easy to use and promotes correct use of a protection mechanism is challenging, especially when the mechanism is new to users.

Secure the weakest link: This requires someone knows which security mechanism being used is the weakest. Performing a formal review, informal review, or a design walkthrough (refer to Chap. 23) and including cybersecurity experts in the review

is a good way to identify the weakest mechanism. Once the weakest mechanism is identified, discussions can determine how to mitigate this risk.

Use your resources: Clearly, this principle has nothing to do with using design models. Instead, this principle is about people. Take advantage of the knowledge and experience found in your colleagues, friends, and professional network. Getting different perspectives on developing more secure software can help you and your project team produce more robust software.

24.4 Post-conditions

The following should have been learned when completing this chapter.

- There are three information security goals: confidentiality, integrity, and availability.
- There are four additional information security concepts which are important to understand: anonymity, assurance, authenticity, and non-repudiation.
- There are 13 security design principles researchers have identified as critical to think about and include when developing a software solution.

1. Economy of mechanism: simpler designs are easier to verify and validate.
2. Fail-safe defaults: allow access to data only when permission (i.e., authentication) has been verified.
3. Complete mediation: each access to data should be checked to ensure proper authority.
4. Open design: allow anyone to review your design.
5. Separation of privilege: use more than one key to unlock a protection mechanism. Store these keys in different locations.
6. Least privilege: give users/systems the minimum access rights necessary to perform their tasks.
7. Least common mechanism—minimize the number of protection mechanisms shared across users/systems.
8. Psychological acceptability: make security usable; align the user's mental image of their protection goals with your HCI design.
9. Secure the weakest link: spend more time on improving the weakest security mechanism within your system.
10. Defend in depth: design security into different software layers of your system; use redundant capabilities to provide a more robust set of security mechanisms.
11. Be reluctant to trust: be skeptical of security mechanisms provided by third-party software—trust but verify these mechanisms.
12. Promote privacy: your design needs to consider the types of personal information being collected and stored.

13. Use your resources: talk to others about your design choices; no one knows everything about good software security.

- Cryptographic concepts include: cryptographic systems, digital signatures, cryptographic hash functions, and digital certificates.

Exercises

Discussion Questions

1. Use an existing code solution you've developed and identify portions of the program design which correctly implements one or more of the 13 security design principles.
2. Use an existing software design you've developed and identify portions of it which correctly implements one or more of the 13 security design principles.
3. Use an existing software design you've developed and identify the design changes you would need to make to adhere to applicable security design principles not currently in the design.
4. Use an existing software design you've developed and identify the design changes you would need to make to adhere to the security design principles not correctly implemented.

Hands-on Exercises

1. Modify an existing code solution you've developed by implementing one or more of the 13 security design principles.
2. Modify an existing software design you've developed by adding one or more of the 13 security design principles.
3. Modify an existing software design you've developed by adding additional security design principles not correctly implemented or missing from the design.

References

1. Saltzer JH, Schroeder MD (1975) The Protection of Information in Computer Systems. Proc IEEE 63(9)
2. McGraw G (2013) Thirteen principles to ensure enterprise system security. Available via https://searchsecurity.techtarget.com/opinion/Thirteen-principles-to-ensure-enterprise-system-security. Cited 28 July 2015



OOD Case Study: More Security Requirements

25

The objective of this chapter is to apply the security design principles discussed in Chap. 24 to the development of the case studies.

25.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You have been introduced to the 13 security design principles that researchers have identified as critical to thinking about and including when developing a software solution. These principles are economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, psychological acceptability, secure the weakest link, defend in depth, be reluctant to trust, promote privacy, and use your resources.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that describe an object-oriented software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_25

25.2 OOD: ABA Security Design

We will add some requirements to our address book case study to demonstrate application of some of the security design principles described in the previous chapter. The new requirements are in *italics*.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person, a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.
8. *When the ABA starts, the user shall create a password. The password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character from the list ".!?@#$%&*_+−=".*
9. *Each request to create or display contact data shall first require the user to re-enter their password. Three incorrect entries of the user's password shall result in the ABA exiting.*
10. *The data shall be non-persistently stored in encrypted form. A symmetric cryptographic algorithm shall be used.*

Table 25.1 lists the 13 security design principles, maps these to the above requirements when applicable, and describes the design approach for the principle.

25.2.1 OOD: ABA Design Models

The statechart in Fig. 25.1 shows the user actions for this version of the ABA. First, the user enters a password to control access to the ABA data. This password is entered via a GUI dialog shown in Fig. 25.2, then validated against the password rules. Once the password is valid, a text-based menu is displayed to allow the user to create a new contact, display an existing contact, or exit the ABA. When the user chooses to create or display, the user must reenter their password before being allowed to complete their requested transaction. Figure 25.3 shows what this dialog window looks like. When the password is verified, the ABA continues by either prompting the user for the three data values (see the sub-machine diagram in Fig. 25.4 for Get Contact Data) or prompting the user for the contact name whose data should be displayed. When the user has a third bad attempt on verifying their password, the ABA will exit.

The high-level IDEF0 function model in Fig. 25.5 shows the various security controls and mechanisms used by this version of the ABA. Since the model component is responsible for storing the password and contact data, this component contains logic

Table 25.1 Map security design principles to requirements and design

Principle	Requirement	Design approach
Economy of mechanism		We want to develop design models which express enough details to be meaningful but also be simple to read and understand. Our design models cannot be too abstract as this would produce a design that would not show the security features
Fail-safe defaults	9	Keep design for creating and verifying a user's password separate from entry and display of contact data. In the design described below, a graphical user interface (GUI) is used to allow entry of the password while a text-based user interface (TUI) is used to obtain and display contact data
Complete mediation	9	This requirement exemplifies this principle; the user must enter a correct password before being allowed to create or display contact data
Open design		While the design is not open source, it is available to anyone with access to this book
Separation of privilege		No requirement identifies a need for this and the ABA design will not address this principle. An example of a security feature for this principle is the use of two-factor authentication
Least privilege	9	The design for requirement 9 should ensure that each request for data has been authorized and the password created by the user is only valid for the address book data created during a single application instance (since the data is non-persistently stored)
Least common mechanism	8 & 10	The design for creating and verifying a password will use the SHA-256 hash function, where SHA stands for Secure Hash Algorithm. The design for encrypting contact data will use the AES (Advanced Encryption Standard) symmetric key algorithm. The design of these two security mechanisms has no connection with each other
Psychological acceptability	8 & 9	Entry of a password uses a GUI user control to display each character entered as an asterisk. This is consistent with many applications using an asterisk to mask the input of each character entered as part of a password value
Secure the weakest link		The secret key generated and used to encrypt and decrypt contact data is stored in memory within the model component. A malicious user may be able to scan memory to find this key value, allowing this individual to decrypt and view/change contact data
Defend in depth	8, 9 & 10	Creating a valid password must be done before the ABA is used. Entering the same password must be done prior to creating or displaying contact data. Finally, the model component stores the phone number and email address in encrypted form. Only when the user requests display of this data is it decrypted
Be reluctant to trust	8 & 9	The requirement and design for creating a password will make it more difficult for someone to guess a password. The requirement and design for allowing only three invalid attempts at verifying a password ensures a malicious user cannot guess an unlimited number of times
Promote privacy	10	The design encrypts the phone number and email address for each contact person
Use your resources		Not applicable

for the SHA-256 hash function and AES symmetric encryption algorithm. However, the password and contact domain data are validated by the controller before being given to the model for storage. The view component also shows the use of a GUI dialog to obtain a password and text-based menus and prompts to obtain the user's request and associated data.

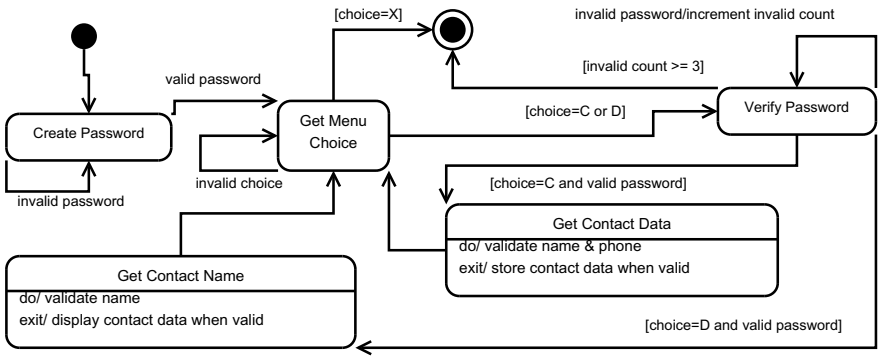


Fig. 25.1 ABA security design: statechart

Fig. 25.2 ABA security design: create password dialog

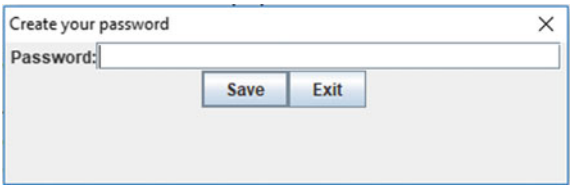
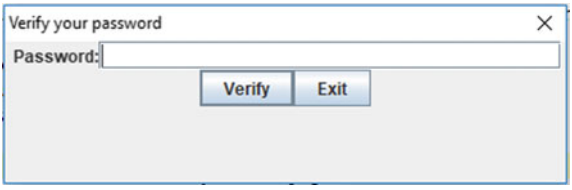


Fig. 25.3 ABA security design: verify password dialog



The package diagram in Fig. 25.6 shows the classes and relationship between the model, view, and controller components. The classes responsible for implementing security controls and mechanisms include the following.

ABA_A_viewPassword: Contains the plaintext password entered by the user via the GUI dialog. The model class is dependent on this class; this relationship is not shown on the package diagram.

ABA_A_viewGUI_Password: Represents the logic associated with the two password dialogs shown in Figs. 25.2 and 25.3.

ABA_A_modelCipherData: The model now stores instances of this object in a TreeMap. This object contains the contact name in plaintext and the phone and email in cipher(encrypted)text.

ABA_A_modelCrypto: Contains logic for the AES symmetric key algorithm.

ABA_A_modelGenKey: Generates a key used by the AES symmetric algorithm.

ABA_A_modelPassword: Generates and stores a message digest representing a valid password created by the user.

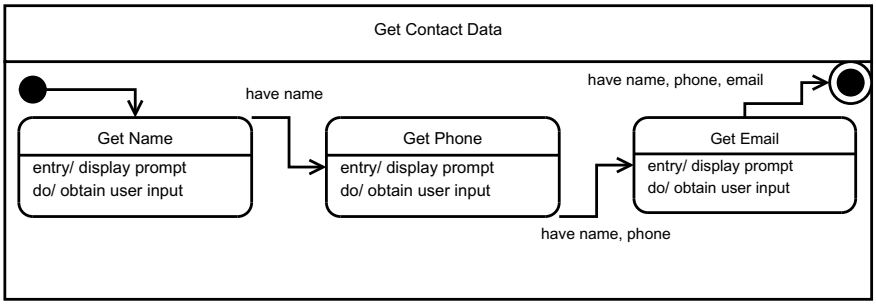


Fig. 25.4 ABA security design: sub-machine statechart for Get Contact Data

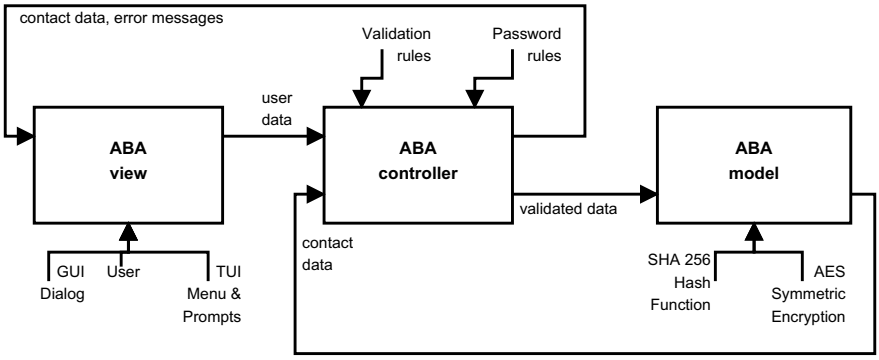


Fig. 25.5 ABA security design: IDEF0 function model

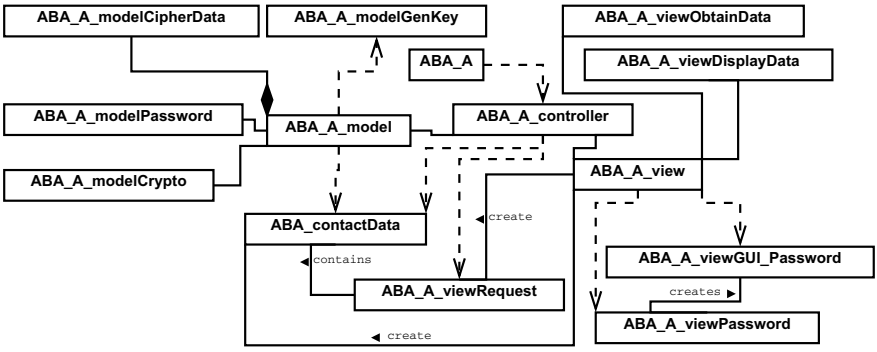


Fig. 25.6 ABA security design: package diagram

25.2.1.1 ABA Detailed Design and Implementation

This section provides more details on the design and implementation of the security controls and mechanisms used in this version of the ABA. We'll start by describing the changes to the view to support entry of a password. Figure 25.7 shows the classes in the view component. This class diagram shows attributes but hides methods for

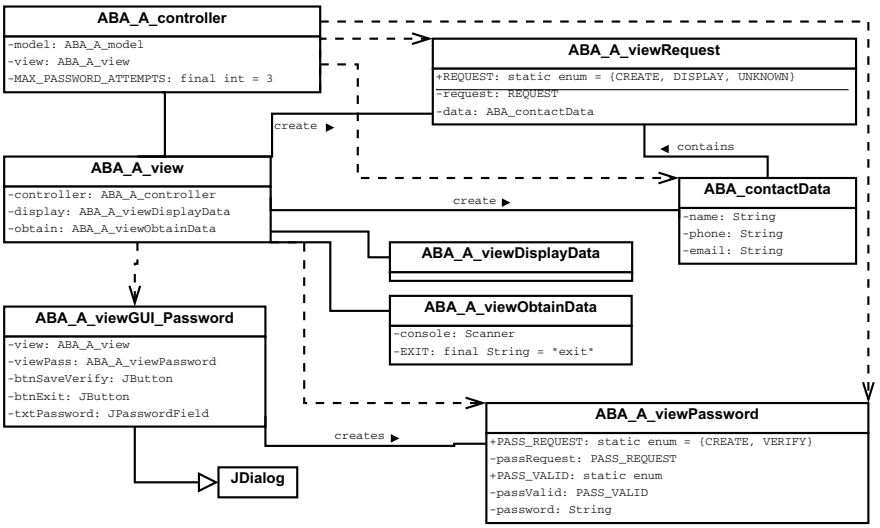


Fig. 25.7 ABA security design: class diagram view component

each class. The ABA_A_viewGUI_Password class inherits from the JDialog class, which is part of the Java swing package. This class uses a JPasswordField to hide the actual text being entered as a password. The ABA_A_viewPassword class acts as a container object passed to the controller for validation when creating a password, or for verification when verifying a password prior to the user creating or displaying contact data.

Listing 25.1 shows the creation of a JPasswordField user control. Note the method call setEchoChar('*'). This tells the control to display an asterisk each time the user enters a character.

Listing 25.1 ABA MVC Design version A - create JPasswordField

```
private JPanel createPasswordControls() {
  //Create JPanel that will contain the labels and text fields
  JPanel panel = new JPanel();
  panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

  txtPassword = new JPasswordField(TXT_WIDTH);
  txtPassword.setEchoChar('*');

  Dimension dim = txtPassword.getPreferredSize();
  txtPassword.setMaximumSize(dim);

  panel.add(createLblTxtPair(new JLabel(LBL_PASSWORD),
     txtPassword));

  return panel;
}
```

Two additional items are worth mentioning at this point. First, the ABA is a stand-alone application, wholly contained and executed on a single device. The ABA_A_viewPassword class contains the actual String value entered by the user for the password. An ABA_A_viewPassword object is created by the view and given to the controller for processing. Assuming that the user is creating their password, the controller validates it to ensure it adheres to the password rules (i.e., see requirement 8 above). When the password is valid, the controller gives the ABA_A_viewPassword object to the model. Only at this point is the plaintext password changed to a message digest. If the ABA were a distributed application, i.e., the view component runs on a client device and the model component runs on a server, then this design solution would be insufficient since the plaintext of the password would be traveling across a network from client to server. In this case, the client-side ABA application would contain the view and a partial controller implementation. The controller would validate the password (when it is being created), and then apply the hash function to produce a message digest. The message digest would be sent over the network to the server for storage.

Second, the SHA-256 hash function computes a message digest quickly. This can result in a successful dictionary attack if the password is made from a common word or phrase. In the case of the ABA, requirement 8 (*password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character*) makes a dictionary attack very unlikely to succeed. Regarding the speed of the SHA-256 algorithm [1], this processing starts by splitting the data to be hashed into 512-bit blocks. A password is likely going to contain fewer than 64 characters, resulting in the password being completely contained in one 512-bit block. The algorithm would then iterate 64 times to compute eight intermediate 32-bit values, which are then concatenated to produce the 256-bit message digest.

The class diagram for the model component is shown in Fig. 25.8. This class diagram shows attributes but hides methods for each class. The ABA_A_modelPassword class is used to transform the plaintext password value into a message digest. The private computeHash method, shown in Listing 25.2, shows the use of the Java MessageDigest class to create an instance associated with the "SHA-256" algorithm. This instance is then used to compute a 256-bit message digest using a `byte[]` representation of the plaintext password value.

Listing 25.2 ABA MVC Design version A - compute message digest

```
private byte[] computeHash(String pass)
{
  byte[] computedHashDigest = null;
  try
  {
    byte[] data = pass.getBytes();
    //use the SHA-256 algorithm
    MessageDigest md = MessageDigest.getInstance(SHA256);
    //Give hash function the byte data
    md.update(data);
    //Compute the hash digest
    computedHashDigest = md.digest();
```

```
    }
    catch (NoSuchAlgorithmException ex)
    {
      System.out.println("ABA_A_modelPassword: no such algorithm " +
          "exception in computeHash.");
      System.out.println(ex);
    }
    catch (Exception ex)
    {
      System.out.println("ABA_A_modelPassword: exception in " +
          "computeHash.");
      System.out.println(ex);
    }
    return computedHashDigest;
}
```

As noted above, the encryption and decryption of contact data is being done in the model component. The ABA_A_modelGenKey constructor method is used to generate a secret key used by the AES algorithm. Listing 25.3 shows the use of the KeyGenerator and SecureRandom Java API classes. The key variable is the only instance variable in this class. Not shown in the model class diagram (see Fig. 25.8), the public getKey method is called to obtain the SecretKey object.

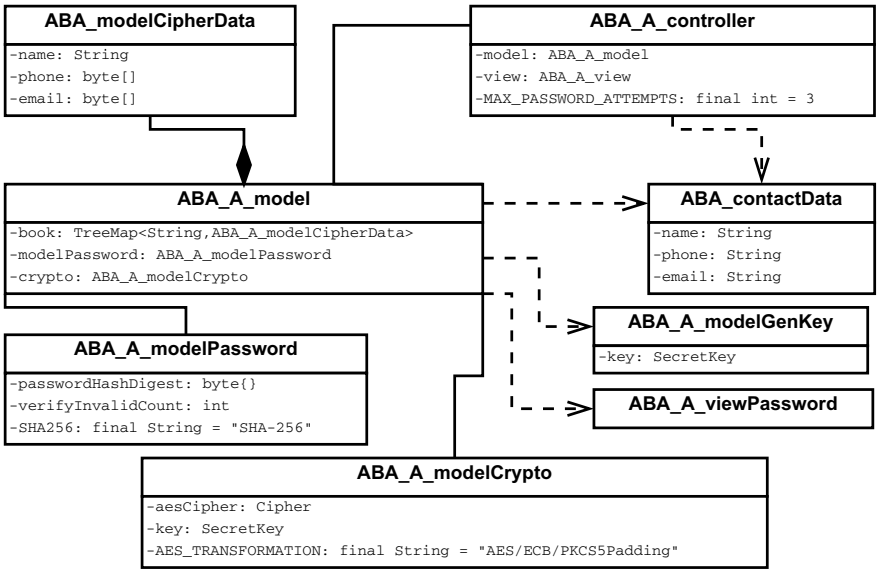


Fig. 25.8 ABA security design: class diagram model component

Listing 25.3 ABA MVC Design version A - generate secret key

```
public ABA_A_modelGenKey()
{
  final String AES = "AES";
  final String DSA_ALGORITHM_NAME = "SHA1PRNG";
  final String DSA_PROVIDER_NAME = "SUN";
  final byte[] MY_SEED_VALUE =
      "The ABA is a small but useful case study!".getBytes();

  try
  {
    //Get a key generator object for the AES algorithm
    KeyGenerator keyGen = KeyGenerator.getInstance(AES);
    //Do an algorithm-independent initialization of key generator
    SecureRandom random = SecureRandom.getInstance(
      DSA_ALGORITHM_NAME, DSA_PROVIDER_NAME);
    //Include a user supplied seed as part of this randomness.
    random.setSeed(MY_SEED_VALUE);
    //Initialize the key generator.
    keyGen.init(random);
    //Generate a secret key
    key = keyGen.generateKey();
  }
  catch (Exception ex)
  {
    System.out.println("demoSymK_Generate_Key: " + ex);
    key = null;
  }
}
```

The model constructor method in Listing 25.4 shows how an ABA_A_modelGen Key object is constructed and then used to obtain the secret key, which is used to construct an ABA_A_modelCrypto object. The ABA_A_modelCrypto object is then used to encrypt and decrypt the phone number and email address for contact. The encrypt method is shown in Listing 25.5. Also note in Listing 25.4, the TreeMap data structure now stores an ABA_A_modelCipherData object as the associated data value for a contact name, which is stored as a String object. As shown in the class diagram in Fig. 25.8, the phone and email instance variables in the ABA_A_modelCipherData are a byte[] data type, which is the type of data returned by the encrypt method.

Listing 25.4 ABA MVC Design version A - model constructor method

```
public ABA_A_model()
{
  this.book = new TreeMap<String,ABA_A_modelCipherData>();
  this.modelPassword = new ABA_A_modelPassword();
  //Create an AES symmetric key.
  ABA_A_modelGenKey genKey = new ABA_A_modelGenKey();
  //Create crypto object for AES..
  this.crypto = new ABA_A_modelCrypto(genKey.getKey());
}
```

Listing 25.5 ABA MVC Design version A - encrypt method

```
public byte[] encrypt(byte[] plaintext)
{
  byte[] ciphertext = null;
  try
  {
    //initialize cipher for encryption.
    aesCipher.init(Cipher.ENCRYPT_MODE, key);
    //encrypt plaintext.
    ciphertext = aesCipher.doFinal(plaintext);
  }
  catch (Exception ex)
  {
    System.out.println("ABA_A_modelCrypto.encrypt exception:");
    System.out.println(ex);
  }
  return ciphertext;
}
```

25.3 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when thinking about security design principles as part of a top-down design approach.

25.3.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below. Note the complete absence of security requirements.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.

- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

25.3.2 OOD Personal Finances: Security Design

Since this case study is storing sensitive financial information, we would expect the data to be persistently stored as ciphertext (i.e., in encrypted form). Likewise, we may want the user to provide evidence they are who they say they are. At the very least a pin number or password should be required. If we are concerned about the ease in which someone can guess the user's pin or password, we would design and implement rules regarding the length and content of a pin or password. We can also enforce a second factor of authentication, like sending a code to a cell phone and expecting the user to enter this code before they gain access to the personal finances application. These security controls and mechanisms may be sufficient when personal finances is a stand-alone application.

The security description above addresses the following security design principles.

Fail-safe defaults: Requiring a pin or password to gain access.
Separation of privilege: Use of a second factor for authentication.
Psychological acceptability (make security usable): We'll assume that entry of a pin or password is masked.
Promote privacy: Encrypting all of the data persistently stored.

If we assume that the personal finances application utilizes a cloud service for persistent storage of the data, this adds another layer of security concerns. In this case, the personal finances application is split between a client device (running a view component along with part of the controller) and a server device (running part of the controller and the model component). Our design should now include the encryption of data traveling between the client and server devices. This encryption should use a different algorithm, or at least a different key, than what is being used to encrypt the data for persistent storage. One approach would be to use the HTTPS protocol handshaking to establish a secret key between web browser and web server. This secret key would then be used during the duration of the current user session. Two benefits arise from this design solution. First, the handshaking between client and server within HTTPS is a well-established and tested protocol. It has shown remarkable resiliency to attack. Second, this guarantees that the secret key used for encrypting the data while in motion would be different from a secret key used to persistently store the data. This guarantee exists since the HTTPS handshaking generates a secret key each time a secured connection is established, while the secret key used for persistent storage would likely never change.

The security description for a distributed version of the personal finances application addresses the following additional security design principles.

Least common mechanism: Use of one secret key to encrypt data in motion while using a different secret key to encrypt data at rest.
Secure the weakest link: Encrypting data in motion and at rest tends to be overlooked in many of today's systems.
Defend in depth: Using encryption in different parts of the system.

Below are a few design ideas for some of the remaining security design principles.

Complete mediation: While it would be annoying to most users to always force entry of their pin/password before they can do something in the personal finances application, it may be appropriate to develop a compromise. The design could include a timer to compute the idle time between user actions. When the user is idle for more than N minutes, the software could force the user to reenter their pin/password. This idle detection mechanism could also use two-factor authentication.
Least privilege: Since the personal finances application is used by an individual, or perhaps by a few people using the same financial accounts, there is no reason to distinguish different types of users and their privilege levels.
Be reluctant to trust: If we assume the cloud service being used provides encryption services, this should be periodically tested by the team responsible for the personal finances application. If trust in the cloud provider cannot be verified, it's more secure to (redundantly) build encryption of persistent data into your design.

25.4 OOD: Post-conditions

The following should have been learned when completing this chapter.

- You understand how many of the 13 security design principles were applied to the case study software design.
- You've applied many of the 13 security design principles to a software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.

- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Discussion Questions

1. The HTTPS protocol uses an asymmetric algorithm to establish a secret key between the browser and server. Both the client and server then use the same secret key (i.e., using a symmetric algorithm) to encrypt and decrypt the data in motion. Why does the HTTPS protocol do this? Why not simply use an asymmetric algorithm to encrypt and decrypt the data in motion?
2. In the description of *be reluctant to trust* for the personal finances application, why should the team responsible for the application *periodically* verify the encryption being used by the cloud service? Why not verify this once and be done with it?
3. Can you identify a software application that implements *complete mediation*?
4. Describe a type of software application where *least privilege* would be important to design and implement.
5. The argument for having an *open design* is supported by the phrase *two heads are better than one*. Having many people looking at your design would likely result in finding more defects and vulnerabilities. Can you think of a scenario where *open design* would result in more security risks, not less?

Hands-on Exercises

1. Modify the design and code for the ABA.

    a. Remove the requirement that a user must enter their password each time they request to create or display contact data. Replace this with a design and implementation of the *idle detection mechanism* briefly described for the personal finances application.

2. Modify the design for the personal finances application.

a. Update the class diagrams from Chap. 15 to show how the security controls and mechanisms described in this chapter may be included in the personal finances object-oriented design.

3. Using an existing code solution you've developed, develop alternative security design models to show ways in which security design principles could be included in your application.
4. Use your development of an application you started in Chap. 3 for this exercise. Modify your design to use some security controls, and then evaluate your design using the 13 security design principles.
5. Continue Hands-on Exercise 3 from Chap. 12 by developing a design that satisfies many of the 13 security design principles. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

- Airline reservation and seat assignment
- Automated teller machine (ATM)
- Bus transportation system
- Course-class enrollment
- Digital library
- Inventory and distribution control
- Online retail shopping cart
- Personal calendar
- Travel itinerary

Reference

1. National Institute of Standards and Technology: Secure Hash Standard (SHS). NIST (2015) https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. Accessed June 26 2019



SD Case Study: More Security Requirements

26

The objective of this chapter is to apply the security design principles discussed in Chap. 24 to the development of the case studies.

26.1 SD: Preconditions

The following should be true prior to starting this chapter.

- You have been introduced to the 13 security design principles that researchers have identified as critical to thinking about and including when developing a software solution. These principles are: economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, psychological acceptability, secure the weakest link, defend in depth, be reluctant to trust, promote privacy, and use your resources.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that describe a structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_26

26.2 SD: ABA Security Design

We will add some requirements to our address book case study to demonstrate application of some of the security design principles described in the previous chapter. The new requirements are in *italics*.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person, a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.
8. *When the ABA starts, the user shall create a password. The password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character from the list ".!?@#$%&*_+-=".*
9. *Each request to create or display contact data shall first require the user to reenter their password. Three incorrect entries of the user's password shall result in the ABA exiting.*
10. *The data shall be non-persistently stored in encrypted form. A symmetric cryptographic algorithm shall be used.*

Table 26.1 lists the 13 security design principles, maps these to the above requirements when applicable, and describes the design approach for the principle.

26.2.1 SD: ABA Design Models

The statechart in Fig. 26.1 shows the user actions for this version of the ABA. First, the user enters a password to control access to the ABA data. This password is entered via a GUI dialog shown in Fig. 26.2, then validated against the password rules. Once the password is valid, a text-based menu is displayed to allow the user to create a new contact, display an existing contact, or exit the ABA. When the user chooses to create or display, the user must reenter their password before being allowed to complete their requested transaction. Figure 26.3 shows what this dialog window looks like. When the password is verified, the ABA continues by either prompting the user for the three data values (see the sub-machine diagram in Fig. 26.4 for Get Contact Data) or prompting the user for the contact name whose data should be displayed. When the user has a third bad attempt on verifying their password, the ABA will exit.

The high-level IDEF0 function model in Fig. 26.5 shows the various security controls and mechanisms used by this version of the ABA. Since the model component is

Table 26.1 Map security design principles to requirements and design

Principle	Requirement	Design approach
Economy of mechanism		We want to develop design models which express enough details to be meaningful but also be simple to read and understand. Our design models cannot be too abstract as this would produce a design that would not show the security features
Fail-safe defaults	9	Keep design for creating and verifying a user's password separate from entry and display of contact data. In the design described below, a graphical user interface (GUI) is used to allow entry of the password while a text-based user interface (TUI) is used to obtain and display contact data
Complete mediation	9	This requirement exemplifies this principle; the user must enter a correct password before being allowed to create or display contact data
Open design		While the design is not open source, it is available to anyone with access to this book
Separation of privilege		No requirement identifies a need for this and the ABA design will not address this principle. An example of a security feature for this principle is the use of two-factor authentication
Least privilege	9	The design for requirement 9 should ensure that each request for data has been authorized and the password created by the user is only valid for the address book data created during a single application instance (since the data is non-persistently stored)
Least common mechanism	8 & 10	The design for creating and verifying a password will use the SHA-256 hash function, where SHA stands for Secure Hash Algorithm. The design for encrypting contact data will use the AES (Advanced Encryption Standard) symmetric key algorithm. The design of these two security mechanisms has no connection with each other
Psychological acceptability	8 & 9	Entry of a password uses a GUI user control to display each character entered as an asterisk. This is consistent with many applications using an asterisk to mask the input of each character entered as part of a password value
Secure the weakest link		The secret key generated and used to encrypt and decrypt contact data is stored in memory within the model component. A malicious user may be able to scan memory to find this key value, allowing this individual to decrypt and view/change contact data
Defend in depth	8, 9 & 10	Creating a valid password must be done before the ABA is used. Entering the same password must be done prior to creating or displaying contact data. Finally, the model component stores the phone number and email address in encrypted form. Only when the user requests display of this data is it decrypted
Be reluctant to trust	8 & 9	The requirement and design for creating a password will make it more difficult for someone to guess a password. The requirement and design for allowing only three invalid attempts at verifying a password ensures a malicious user cannot guess an unlimited number of times
Promote privacy	10	The design encrypts the phone number and email address for each contact person
Use your resources		Not applicable

responsible for storing the password and contact data, this component contains logic for the SHA-256 hash function and AES symmetric encryption algorithm. However, the password and contact domain data are validated by the controller before being given to the model for storage. The view component also shows the use of a GUI dialog to obtain a password and text-based menus and prompts to obtain the user's request and associated data.

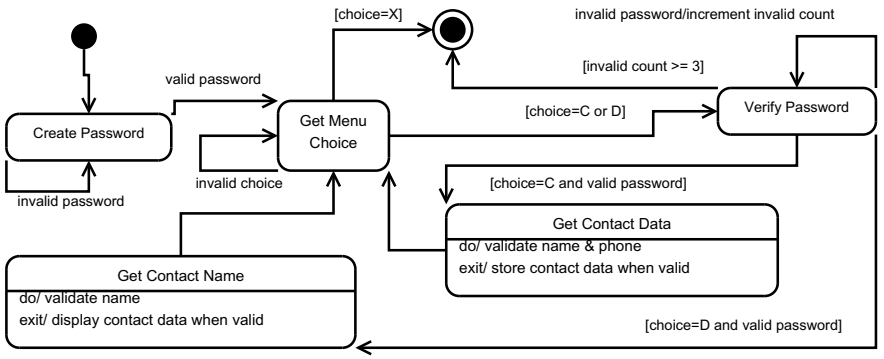


Fig. 26.1 ABA security design: statechart

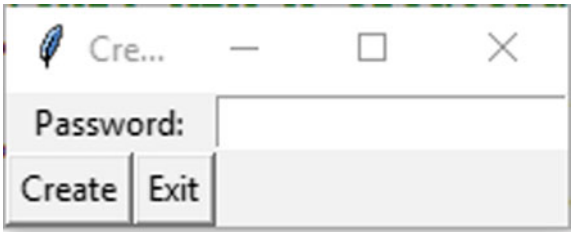


Fig. 26.2 ABA security design: create password dialog

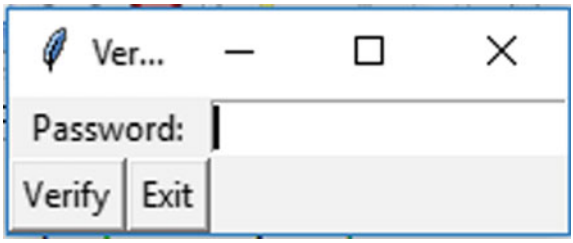


Fig. 26.3 ABA security design: verify password dialog

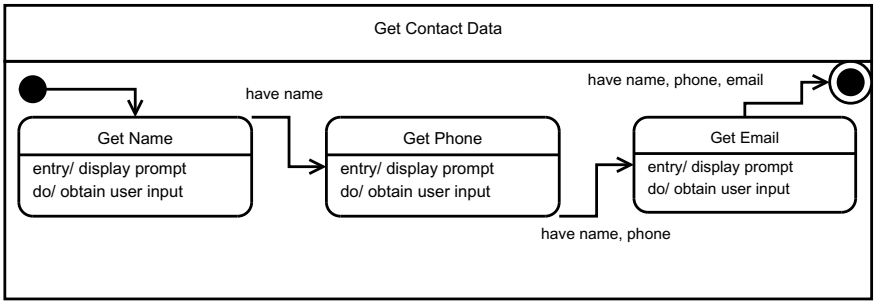


Fig. 26.4 ABA security design: sub-machine statechart for Get Contact Data

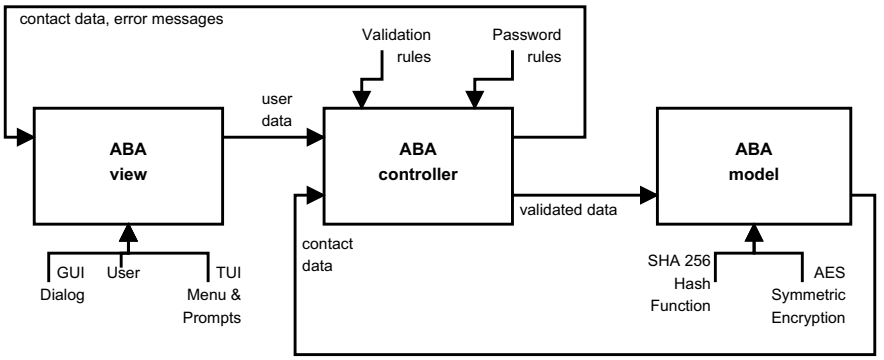


Fig. 26.5 ABA security design: IDEF0 function model

Table 26.2 ABA MVC design GUI—source code files with security controls

Source code file	Component?	Responsibilities
ABA_A_modelCipherData.py	Model	Stores a tuple containing data for one contact person. The phone and email data has been encrypted. Has functions to allow the tuple to be created or retrieved.
ABA_A_modelCrypto.py	Model	Uses AES to encrypt and decrypt contact data.
ABA_A_modelGenKey.py	Model	Generates a secret key used by modelCrypto.
ABA_A_modelPassword.py	Model	Uses SHA-256 to store the user password as a message digest.
ABA_B_viewGUI_Password.py	View	A GUI that allows the user to create or verify their password.
ABA_B_viewPassword.py	View	Identifies the type of password (i.e., create or verify) entered by the user and contains the password in plaintext form.

The solution for the ABA has 16 source code files. The source code files responsible for implementing security controls and mechanisms are listed in Table 26.2.

26.2.1.1 ABA Detailed Design and Implementation

This section provides more details on the design and implementation of the security controls and mechanisms used in this version of the ABA. We'll start by describing the changes to support creating the user's password. Figure 26.6 shows the functions involved with initializing the ABA. The result of these functions is the display of the dialog shown in Fig. 26.2. This dialog uses an Entry user control to allow the user to enter their password. The ABA_A_viewPassword module acts as a container object passed to the controller for validation when creating a password, or for verification when verifying a password prior to the user creating or displaying contact data.

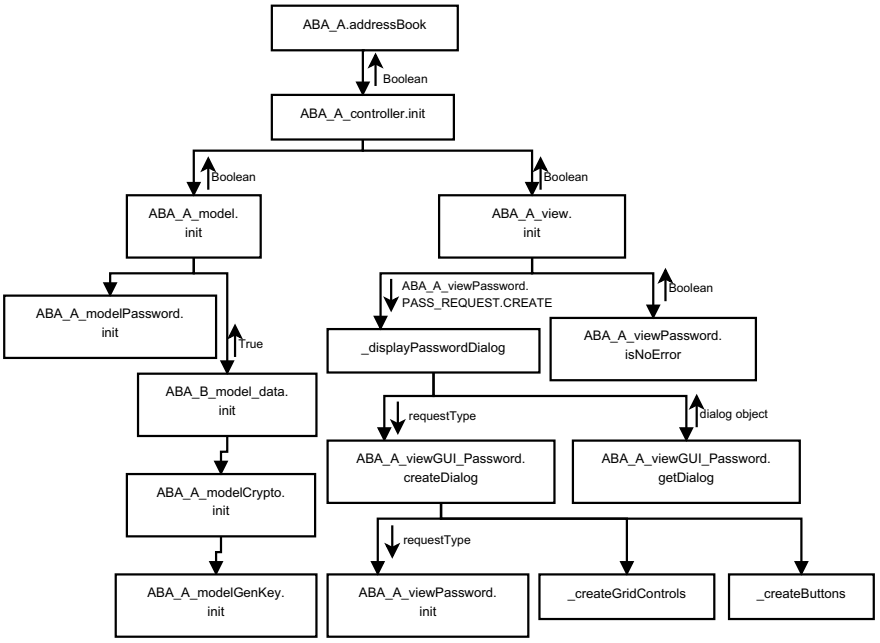


Fig. 26.6 ABA security design: structure chart for starting ABA

Listing 26.1 shows the creation of the Entry user control. Note the method call config(show="*"). This tells the user control to display an asterisk each time the user enters a character.

Listing 26.1 ABA MVC Design version A—create password Entry field

```
def _createGridControls(parent):
  global txtPassword
  tk.Label(parent, text="Password:").grid(row=0, column=0)
  txtPassword = tk.Entry(parent)
  #show asterisk for each character entered by user
  txtPassword.config(show="*")
  txtPassword.grid(row=0, column=1)
```

One additional item is worth mentioning at this point. The ABA is a stand-alone application, wholly contained and executed on a single device. The ABA_A_view Password module contains the actual string value entered by the user for the password. This module is used by the controller for processing. Assuming the user is creating their password, the controller validates it to ensure it adheres to the password rules (i.e., see requirement 8 above). When the password is valid, the model then uses the ABA_A_viewPassword module. Only at this point is the plaintext password changed to a message digest. If the ABA were a distributed application, i.e., the view component runs on a client device and the model component runs on a server, then this design solution would be insufficient since the plaintext of the password

would be traveling across a network from client to server. In this case, the client-side ABA application would contain the view and a partial controller implementation. The controller would validate the password (when it is being created), and then apply the hash function to produce a message digest. The message digest would be sent over the network to the server for storage.

The structure chart in Fig. 26.7 shows the functions used when creating a password and storing it as a message digest using a hash function. This shows that a plaintext password is used by both the view and controller components. Only when the model's setPassword function is called does the plaintext value get transformed into a message digest using the SHA-256 hash function.

The SHA-256 hash function computes a message digest quickly. This can result in a successful dictionary attack if the password is made from a common word or phrase. In the case of the ABA, requirement 8 (*password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character*) makes a dictionary attack very unlikely to succeed. Regarding the speed of the SHA-256 Algorithm [1], this processing starts by splitting the data to be hashed into 512-bit blocks. A password is likely going to contain fewer than 64 characters, resulting in the password being completely contained in one 512-bit block. The algorithm would then iterate 64 times to compute eight intermediate 32-bit values, which are then concatenated to produce the 256-bit message digest.

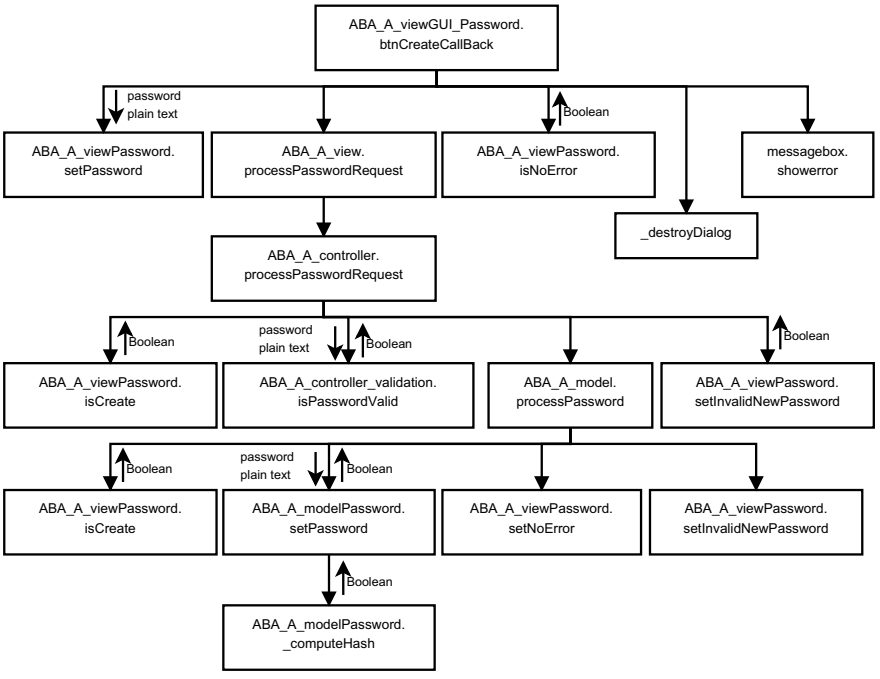


Fig. 26.7 ABA security design: structure chart for creating password

Listing 26.2 shows use of the sha256 function in the hashlib library, which is part of the cryptography libraries provided by Python.

Listing 26.2 ABA MVC Design version A—compute message digest

```
def _computeHash(passWord):
  passwordDigest = None
  hashAlg = hashlib.sha256()
  hashAlg.update(bytes(passWord, 'ascii'))
  passwordDigest = hashAlg.digest()
  return passwordDigest
```

The structure chart in Fig. 26.8 shows the use of encryption within the model component. The ABA_A_modelCipherData module contains encrypt and decrypt functions used for the phone and email. The structure chart shows the creation of a contact tuple (see the createContactDataTuple function) containing the contact name in plaintext and the phone and email in ciphertext.

As noted above, the encryption and decryption of contact data is being done in the model component. The ABA_A_modelGenKey.init function, shown in Fig. 26.6 structure chart, is used to generate a secret key used by the AES algorithm. Listing 26.3 shows this function, which uses the get_random_bytes function to generate a secret key. This function is part of the PyCryptodome library, which must be downloaded and installed once you have Python on your system [2,3]. The getKey function shown in the listing is called to obtain the secret key.

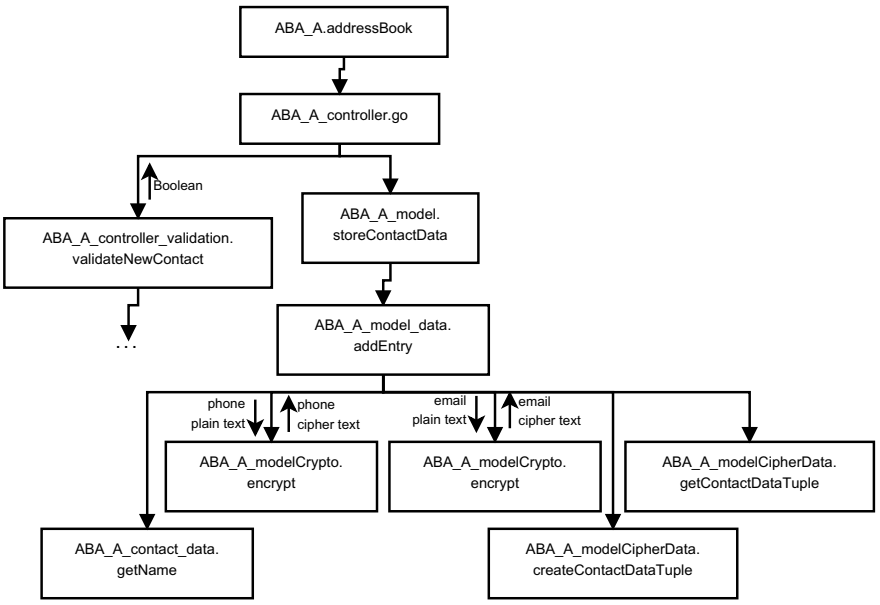


Fig. 26.8 ABA security design: structure chart for encrypting data

Listing 26.3 ABA MVC Design version A—generate secret key

```
from Cryptodome.Random import get_random_bytes

def init():
  global key
  key = get_random_bytes(16)

def getKey():
  global key
  return key
```

The ABA_A_modelCrypto module is used to encrypt and decrypt the phone number and email address for contact. The encrypt method is shown in Listing 26.4. Also note the comments in Listing 26.4, explaining how a plaintext value is encrypted and returned as a tuple pair containing the ciphertext and the initialization vector used to encrypt the data. The decrypt function shown in Listing 26.4 uses the initialization vector, saved in the tuple pair as part of the encrypt function, when creating an AES cipher instance. Also note the use of Cryptodome library [2,3] to use AES for encryption and decryption.

Listing 26.4 ABA MVC Design version A—encrypt function

```
from Cryptodome.Cipher import AES

#pre: have plaintext as a str object.
#post: returns tuple containing two bytes objects
#       [0] encrypted data
#       [1] initialization vector
def encrypt(plaintext):
  #Using Cipher FeedBack mode, turning AES into a stream cipher.
  #Each byte from plain text XOR'd with byte from key stream.
  aesCipher = AES.new(ABA_A_modelGenKey.getKey(), AES.MODE_CFB)
  #print('initVector=' + str(aesCipher.iv))
  ciphertext = aesCipher.encrypt(bytes(plaintext,'utf-8'))
  return (ciphertext, aesCipher.iv)

def decrypt(cipherTuple):
  plaintext = None
  if len(cipherTuple) == 2:
    #Using Cipher FeedBack mode, turning AES into a stream cipher.
    #Each byte from cipher text is XOR'd with byte from key stream.
    aesCipher = AES.new(ABA_A_modelGenKey.getKey(), AES.MODE_CFB,
      iv=cipherTuple[1])
    plaintext = aesCipher.decrypt(cipherTuple[0])
    #change plaintext from bytes object to str object
    plaintext = str(plaintext,encoding='utf-8')
  return plaintext
```

26.3 SD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when thinking about security design principles as part of a top-down design approach.

26.3.1 SD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 13, are listed below. Note the complete absence of security requirements.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

26.3.2 SD Personal Finances: Security Design

Since this case study is storing sensitive financial information, we would expect the data to be persistently stored as ciphertext (i.e., in encrypted form). Likewise, we may want the user to provide evidence that they are who they say they are. At the very least, a pin number or password should be required. If we are concerned about the ease in which someone can guess the user's pin or password, we would design and implement rules regarding the length and content of a pin or password. We can also enforce a second factor of authentication, like sending a code to a cell phone and expecting the user to enter this code before they gain access to the personal finances application. These security controls and mechanisms may be sufficient when personal finances is a stand-alone application.

The security description above addresses the following security design principles.

Fail-safe defaults: Requiring a pin or password to gain access.
Separation of privilege: Use of a second factor for authentication.

Psychological acceptability (make security usable): We'll assume that entry of a pin or password is masked.
Promote privacy: Encrypting all of the data persistently stored.

If we assume that the personal finances application utilizes a cloud service for persistent storage of the data, this adds another layer of security concerns. In this case, the personal finances application is split between a client device (running a view component along with part of the controller) and a server device (running part of the controller and the model component). Our design should now include the encryption of data traveling between the client and server devices. This encryption should use a different algorithm, or at least a different key, than what is being used to encrypt the data for persistent storage. One approach would be to use the HTTPS protocol handshaking to establish a secret key between web browser and web server. This secret key would then be used during the duration of the current user session. Two benefits arise from this design solution. First, the handshaking between client and server within HTTPS is a well-established and tested protocol. It has shown remarkable resiliency to attack. Second, this guarantees that the secret key used for encrypting the data while in motion would be different from a secret key used to persistently store the data. This guarantee exists since the HTTPS handshaking generates a secret key each time a secured connection is established, while the secret key used for persistent storage would likely never change.

The security description for a distributed version of the personal finances application addresses the following additional security design principles.

Least common mechanism: Use of one secret key to encrypt data in motion while using a different secret key to encrypt data at rest.
Secure the weakest link: Encrypting data in motion and at rest tends to be overlooked in many of today's systems.
Defend in depth: Using encryption in different parts of the system.

Below are a few design ideas for some of the remaining security design principles.

Complete mediation: While it would be annoying to most users to always force entry of their pin/password before they can do something in the personal finances application, it may be appropriate to develop a compromise. The design could include a timer to compute the idle time between user actions. When the user is idle for more than N minutes, the software could force the user to reenter their pin/password. This idle detection mechanism could also use two-factor authentication.
Least privilege: Since the personal finances application is used by an individual, or perhaps by a few people using the same financial accounts, there is no reason to distinguish different types of users and their privilege levels.
Be reluctant to trust: If we assume the cloud service being used provides encryption services, this should be periodically tested by the team responsible for the personal finances application. If trust in the cloud provider cannot be verified, it's more secure to (redundantly) build encryption of persistent data into your design.

26.4 SD: Post-conditions

The following should have been learned when completing this chapter.

- You understand how many of the 13 security design principles were applied to the case study software design.
- You've applied many of the 13 security design principles to a software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You have created and/or modified models that describe a structured software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Discussion Questions

1. The HTTPS protocol uses an asymmetric algorithm to establish a secret key between the browser and server. Both the client and server then use the same secret key (i.e., using a symmetric algorithm) to encrypt and decrypt the data in motion. Why does the HTTPS protocol do this? Why not simply use an asymmetric algorithm to encrypt and decrypt the data in motion?
2. In the description of *be reluctant to trust* for the personal finances application, why should the team responsible for the application *periodically* verify the encryption being used by the cloud service? Why not verify this once and be done with it?
3. Can you identify a software application that implements *complete mediation*?
4. Describe a type of software application where *least privilege* would be important to design and implement.

5. The argument for having an *open design* is supported by the phrase *two heads are better than one*. Having many people looking at your design would likely result in finding more defects and vulnerabilities. Can you think of a scenario where *open design* would result in more security risks, not less?

Hands-on Exercises

1. Modify the design and code for the ABA.

    a. Remove the requirement that a user must enter their password each time they request to create or display contact data. Replace this with a design and implementation of the *idle detection mechanism* briefly described for the personal finances application.

2. Modify the design for the personal finances application.

    a. Update the structure charts from Chap. 16 to show how the security controls and mechanisms described in this chapter may be included in the personal finances structured design.

3. Using an existing code solution you've developed, develop alternative security design models to show ways in which security design principles could be included in your application.
4. Use your development of an application you started in Chap. 4 for this exercise. Modify your design to use some security controls, and then evaluate your design using the 13 security design principles.
5. Continue Hands-on Exercise 3 from Chap. 13 by developing a design that satisfies many of the 13 security design principles. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 13 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)
    - Bus transportation system
    - Course-class enrollment
    - Digital library
    - Inventory and distribution control

- Online retail shopping cart
- Personal calendar
- Travel itinerary

References

1. National Institute of Standards and Technology: Secure Hash Standard (SHS) (2015) NIST. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. Accessed 26 June 2019
2. Python Software Foundation: pycryptodome 3.8.2. (2019). https://pypi.org/project/pycryptodome/. Accessed 28 June 2019
3. Python Software Foundation: Welcome to PyCryptodome's documentation. (2019). https://pycryptodome.readthedocs.io/en/latest/. Accessed 28 June 2019



Introduction to Design Patterns

27

The objective of this chapter is to introduce software design patterns.

27.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_27

27.2 Concepts and Context

Software design patterns became widely discussed in 1995 when a book titled *Design Patterns, Elements of Reusable Object-Oriented Software* was published [1]. The authors of this book were inspired by the importance of patterns in other disciplines. Most notably, a book by Christopher Alexander et al., which documented many patterns for the building architecture discipline [2].

A software design pattern is a description of a problem which occurs frequently in various contexts. It describes the *core of a solution* that can be implemented in many different ways while being true to the intent of the design pattern. The literature [1,3,4] promotes design patterns as a way to make your design more flexible. This flexibility would lead to an overall design that is easier to adapt as requirements change over time.

Software design patterns will be discussed using the perspectives presented in three books, all of which describe patterns using object-oriented design models.

1. The *Gang of Four* (GoF) book published in 1995, which started the discussion of using patterns to describe software designs.
2. A book by Craig Larman [3] published in 2002. As you may recall, a few of Larman's patterns were discussed in Chap. 11.
3. A book by Eduardo Fernandez-Buglioni [4] published in 2013. This book describes security patterns.

27.2.1 GoF Design Patterns

As mentioned, this book started the discussion on software design patterns. This book describes patterns using object-oriented models and code. The GoF [1] organized their patterns by their purpose. They identified three broad categories for their design patterns.

Creational: The purpose of these patterns is to create objects. They identified five creational patterns.

Structural: The purpose of these patterns is to deal with the composition of classes or objects. These patterns explain how classes or objects are related to each other. The GoF identified seven structural patterns.

Behavioral: The purpose of these patterns is to describe ways in which classes or objects interact with each other. The GoF identified 11 behavioral patterns.

27.2.1.1 Examples of GoF Software Design Patterns

A few of the 23 patterns described by the GoF include the following.

- Sample Creational Design Patterns

Singleton: Allows only one object instance to be created for the class. This design has a public method which provides access to this one object.

Abstract Factory: Provides an interface, using an abstract class, for creating object instances where objects are related without needing to specify their concrete classes. The abstract factory class contains method signatures for each type of object to be created. A concrete factory class will subclass the abstract factory class to implement the abstract methods. This results in instantiating objects using the appropriate concrete classes. Generally, an application using an abstract factory will only instantiate one of the concrete factory classes.

Factory Method: Provides an interface for creating an object where subclasses decide which class to instantiate. The interface contains a factory method that may include a parameter to indicate the type of object to create.

- Sample Structural Design Patterns

Facade: Provides a unified interface for a bunch of interfaces within a subsystem.

- Sample Behavioral Design Patterns

Command: Encapsulates a request as an object.

Iterator: Allows sequential access to elements (i.e., objects) within a container (or aggregate) object.

27.2.1.2 What Does a GoF Software Design Pattern Look Like?

The GoF template for a design pattern has the following sections.

- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Three GoF design patterns are briefly described below to illustrate how the above template is used in [1]. The first pattern is the creational pattern named Singleton. Table 27.1 provides a summary description for the template sections, while the class diagram in Fig. 27.1 and the Java code in Listing 27.1 provide implementation details.

Table 27.1 GoF example: Singleton pattern [1]

Section	Description
Intent	Ensure a class only has one instance, and provide a global point of access to it
Motivation	It is important that some classes only have exactly one instance. For example, only one file system or window manager provided by an OS. This pattern makes the class itself responsible for keeping track of its sole instance
Applicability	Use a Singleton pattern when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point; when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code
Structure	See the class diagram in Fig. 27.1
Participants	A Singleton class defines a protected constructor and a public static getInstance() operation. It also has a private attribute of type Singleton
Collaborations	Each client class will access the instance through the public getInstance() method
Consequences	Use of this design pattern results in controlled access to a sole instance; a reduced name space; a refinement of operations and representation; the possibility of extending this pattern to support a fixed number of instances besides one, i.e., modify the Singleton pattern if more than one instance becomes necessary
Implementation	See the class diagram in Fig. 27.1
Sample code	See Listing 27.1
Known uses	Access to a persistent data store may involve a single connection, e.g., to a database server
Related patterns	Many patterns can be implemented using the Singleton pattern. For example, Abstract Factory, Builder, and Prototype

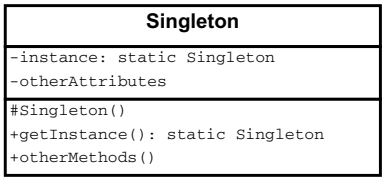


Fig. 27.1 GoF Singleton class diagram

Listing 27.1 GoF Singleton Sample Code

```
public class Singleton
{
  private static Singleton instance = null;
  //class may have other attributes!

  protected Singleton()
```

```
  {
    //initialize other attributes, if needed
  }
  public static Singleton getInstance()
  {
    if (instance == null)
      instance = new Singleton();
    return instance;
  }
  //class may have other operations!
}
```

The second GoF pattern is the structural pattern named Facade. Table 27.2 provides a summary description for the template sections, while the package diagram in Fig. 27.2 and the Java code in Listing 27.2 provide implementation details. The package diagram shows a Facade class providing a common interface to four classes (named One, Two, Three, and Four). The unnamed classes in the package diagram represent other classes likely to be part of the subsystem. The Java code in Listing 27.2 shows a Compiler class used to compile a source code file while hiding the details of the compiler subsystem.

Listing 27.2 GoF Facade Sample Code

```
public class Compiler //Facade to a compiler subsystem
{
    private Scanner scan;
    private Parser parser; //compiler subsystem class
    private ProgramNodeBuilder builder; //compiler subsystem class

    public Compiler()
    {
        builder = new ProgramNodeBuilder(...);
        parser = new Parser(...);
    }
    public void compile(File codeFile, BytecodeStream output)
    {
        scan = new Scanner(codeFile);
        parser.parse(scan, builder);
        CodeGenerator generator = new CodeGenerator(builder);
        generator.generate(output);
    }
}
```

The third pattern is the behavioral pattern named Iterator. Table 27.3 provides a summary description for the template sections, while the class diagram in Fig. 27.3 shows an Iterator interface being used to represent the common operations of an iterator, regardless of the underlying data structure (i.e., ConcreteAggregate) containing the data.

Table 27.2 GoF example: Facade pattern [1]

Section	Description
Intent	Provide a unified interface to a set of interfaces in a subsystem. A Facade defines a higher level interface that makes the subsystem easier to use
Motivation	Structuring a system into subsystems helps reduce complexity. We want to minimize communication and dependencies between subsystems
Applicability	Use the Facade pattern when you want to provide a simple interface to a complex subsystem; there are many dependencies between clients and the subsystem. A facade will decouple the clients from the subsystem
Structure	See the package diagram in Fig. 27.2
Participants	A Facade knows which subsystem classes are responsible for a request and delegates client requests to appropriate subsystem objects. Figure 27.2 shows four subsystem classes—One, Two, Three, Four—that are known by the Facade class. A request from another subsystem may be forwarded by the Facade to any of these four classes. Each subsystem class implements specific subsystem functionality; handling work assigned by the Facade object. These subsystem classes have no knowledge of the Facade class, i.e., they have no references to it
Collaborations	Each client communicates with the subsystem by sending requests to the Facade. The Facade forwards each request to an appropriate subsystem object. The clients using the Facade do not have access to subsystem objects
Consequences	Use of this design pattern shields clients from subsystem complexity and promotes weak coupling. However, this pattern does not prevent applications from directly using subsystem objects
Implementation	See the package diagram in Fig. 27.2
Sample code	See Listing 27.2
Known uses	When access to a complex subsystem needs to be simplified
Related patterns	Facade objects are often Singletons. An Abstract Factory may be used by Facade to provide an interface for creating subsystem objects. Finally, the Mediator design pattern is similar to Facade. However, the Mediator's purpose is to abstract arbitrary communication between colleague objects; it centralizes functionality that does not belong in any of the colleague classes. Another difference: a Mediator's colleagues are aware of and communicate with the Mediator object (instead of communicating with each other directly)

27.2.2 Larman Design Patterns

Like the GoF book, the Larman book describes patterns using object-oriented models and code. Larman introduces nine patterns characterized as general responsibility

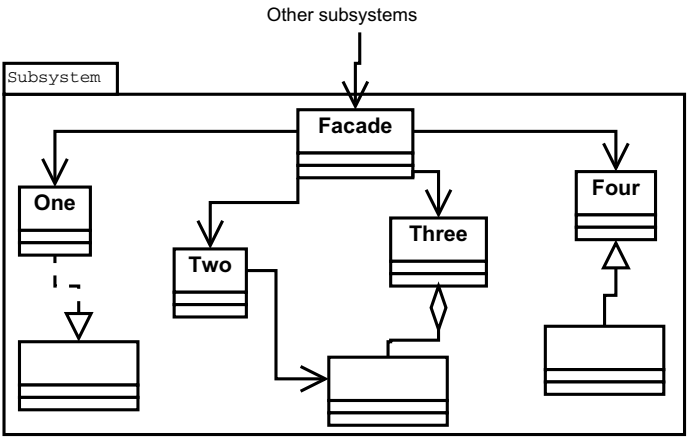


Fig. 27.2 GoF facade package diagram

assignment software patterns (GRASP) [3]. Two of his GRASP patterns—*Low Coupling* and *High Cohesion*—were described in Chap. 11.

27.2.2.1 Examples of Larman Software Design Patterns

Three of the nine patterns described by Larman include the following.

Creator: This is similar to the GoF Factory patterns. It describes a design solution where a class is responsible for creating object instances of another class.

Low Coupling: This pattern assigns responsibility in a way that decreases coupling between classes. See Table 11.2 for details on this GRASP pattern.

High Cohesion: This pattern assigns responsibility in a way that increases cohesion within a class. See Table 11.3 for details on this GRASP pattern.

27.2.2.2 What Does a Larman Software Design Pattern Look Like?

Larman uses the following template to describe his GRASP patterns.

- Problem
- Solution
- Discussion
- Contraindications
- Benefits
- Related patterns or principles

Larman's Creator pattern in GRASP is summarized in Table 27.4.

Table 27.3 GoF example: Iterator pattern [1]

Section	Description
Intent	Provide a way to access elements of an aggregate object sequentially without exposing its underlying representation
Motivation	An aggregate object should give you a way to access its elements without exposing its internal structure. For example, you may want to traverse the aggregate object in different ways, but you do not want to make the interface larger by adding operations for different traversals
Applicability	Use an Iterator pattern to access an aggregate object's contents without exposing its internal representation; support multiple traversals of aggregate objects; and to provide a uniform interface for traversing different aggregate structures
Structure	See the class diagram in Fig. 27.3
Participants	The class diagram in Fig. 27.3 shows two interfaces and two concrete classes. Iterator is an interface for accessing and traversing elements, while ConcreteIterator is a class that implements the Iterator interface. This class keeps track of the current position in traversal of the ConcreteAggregate. Aggregate is an interface for creating an Iterator object, while ConcreteAggregate is a class that implements the Iterator method to return an instance of the proper ConcreteIterator
Collaborations	A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal
Consequences	This design pattern supports variations in the traversal of an aggregate. It simplifies the aggregate interface and allows more than one traversal to be pending on an aggregate
Implementation	See the class diagram in Fig. 27.3
Sample code	Not provided
Known uses	Very common in object-oriented systems; most collections (i.e., data structures) provide one or more iterators. This pattern is implemented in the Java API; see the Interator interface and the Collection, List, Map, and Set interfaces for different types of aggregates
Related patterns	Iterators are often applied to recursive structures

27.2.3 Fernandez Design Patterns

The Fernandez book describes patterns for building more secure software. Instead of categorizing patterns like the GoF, Fernandez uses a matrix to classify his patterns [4]. The rows in his matrix represent different levels of architecture. These levels are application, operating system, distribution, transport, and network. The distribution, transport, and network levels refer to the application, transport, and network layers in the Internet Protocol Stack. The columns in the matrix are life cycle phases and include domain analysis and design. The column for the design life cycle phase is further split into purposes, e.g., filtering, access control, and authentication. The

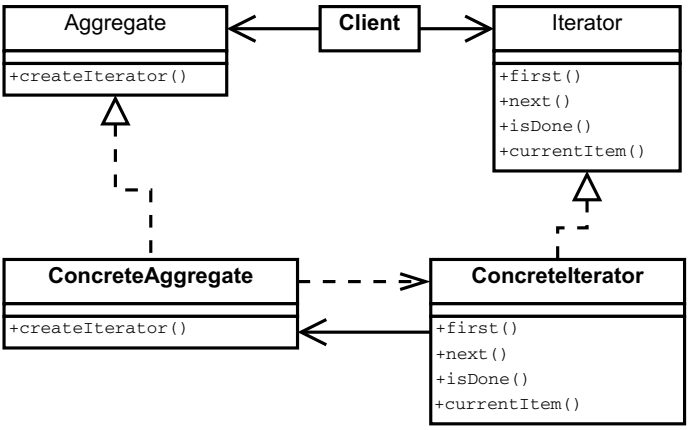


Fig. 27.3 GoF iterator class diagram

Table 27.4 Summary of GRASP Creator [3]

Solution	Assign class B the responsibility to create one or more instances of class A if any of the following are true • B aggregates, contains, or records A objects • B closely uses A objects • B has the data needed to initialize A objects
Problem	As we know, creating objects is a common activity in object-oriented systems. But who should be responsible for creating a new instance of some class?
Discussion	Look for class that needs a connection to created object. This connection may be reflected in a class diagram using any of these relationship types: • Aggregate aggregates Part • Container contains Content • Recorder records Recorded
Contraindications	When creation involves significant complexity, use the Factory pattern
Benefits	Lowers coupling, which implies lower maintenance costs
Related patterns or principles	Low Coupling, Factory, and Whole-Part

security patterns described by Fernandez are described using object-oriented design models.

27.2.3.1 Examples of Fernandez Software Design Patterns

Four of the patterns described by Fernandez include the following.

Table 27.5 Summary of symmetric encryption design pattern [4]

Example	Alice sends sensitive data to Bob. Evan can intercept this data and read the sensitive data
Context	Applications exchange sensitive information over insecure channels
Problem	Sensitive data may be read by unauthorized users while in transit (or at rest)
Solution	The sender transforms plaintext data into ciphertext using a secret key and then transmits the ciphertext over an insecure channel. The receiver transforms the ciphertext into plaintext data using the same secret key
Implementation	Both sender and receiver need to agree on the cryptography algorithm and secret key
Example resolved	Alice encrypts sensitive data then sends ciphertext to Bob. Evan can still intercept this data, but cannot read the sensitive data
Consequences	The key needs to be secret and must be shared between sender and receiver in secure manner. Selection of cryptographic algorithm
Known uses	The following cryptographic algorithms use symmetric encryption: GNuPG, OpenSSL, Java Cryptographic Extension, .NET framework, XML encryption, and Pretty Good Encryption (PGP)
See also	Other security patterns related to this include: secure channel communication pattern, asymmetric encryption, and patterns for key management

Symmetric Encryption: Describes the use of encryption to make a message unreadable unless you have the key. The same key is used to encrypt and decrypt the message.

Asymmetric Encryption: Describes the use of encryption to make a message unreadable unless you have the key. A public key is used to encrypt the message while a private key is used to decrypt the message.

Digital Signature with Hashing: Describes a way to allow the sender of a message to prove the message originated from them and not someone else. Also describes how the receiver of a message can verify the integrity of the message, i.e., that it has not been altered during transmission.

Secure Model–View–Controller: Describes Model–View–Controller components as tiers (or layers) of a system. Each tier enforces security applicable to the tier. Fernandez also describes a Secure Three-Tier Architecture pattern. This describes the three tiers using the terms presentation (i.e., view), business logic (i.e., controller), and data storage (i.e., model).

27.2.3.2 What Does a Fernandez Software Design Pattern Look Like?

Fernandez uses the following template to describe his security patterns.

Table 27.6 Summary of asymmetric encryption design pattern [4]

Example	Alice needs to send sensitive data to Bob, but they do not share a secret key. Evan can intercept this data and read the sensitive data
Context	Applications exchange sensitive information over insecure channels
Problem	Sensitive data may be read by unauthorized users while in transit (or at rest)
Solution	The sender transforms plaintext data into ciphertext using receiver's public key and then transmits the ciphertext over an insecure channel. The receiver transforms ciphertext into plaintext data using their private key
Implementation	Both sender and receiver need to agree on the cryptography algorithm, e.g., RSA
Example resolved	Alice looks up Bob's public key and uses it to encrypt sensitive data. Alice then sends the ciphertext to Bob. Evan can still intercept this data, but cannot read the sensitive data. Bob receives the ciphertext and uses his private key to decrypt the data
Consequences	Anyone can look up someone's public key. Selection of cryptographic algorithm and key length impacts performance and level of security
Known uses	The following cryptographic algorithms use symmetric encryption: GNuPG, Java Cryptographic Extension, .NET framework, XML encryption, and Pretty Good Encryption (PGP)
See also	Other security patterns related to this include secure channel communication pattern

- Example
- Context
- Problem
- Solution
- Implementation
- Example resolved
- Consequences
- Known uses
- See also

The four design patterns listed above are described in Tables 27.5 (Symmetric Encryption), 27.6 (Asymmetric Encryption), 27.7 (Digital Signature with Hashing), and 27.8 (Secure Model–View–Controller).

27.2.4 Summary of Design Pattern Templates

Table 27.9 shows a summary of the software design templates used by the three software design patterns books referenced in this chapter. As this table illustrates,

Table 27.7 Summary of digital signature with Hashing design pattern [4]

Example	Alice wants to send nonsensitive data to Bob and Bob wants to make sure the data came from Alice. Evan can intercept this data and modify it
Context	Applications exchange information over insecure channels and may need to confirm integrity and origin of the data
Problem	Need to authenticate the origin of the message (data)
Solution	The sender computes a message digest on the plaintext data using a hash function, transform the plaintext data into ciphertext using sender's private key, and then sends both the message digest and ciphertext. The receiver decrypts the ciphertext using the sender's public key, computes the message digest on decrypted ciphertext, and then compares the computed digest with digest received from sender
Implementation	Both sender and receiver need to agree on the cryptographic hash function (e.g., SHA-256) and cryptographic asymmetric algorithm (e.g., RSA)
Example resolved	Alice now uses an asymmetric algorithm and a hash function to send nonsensitive data to Bob. Bob verifies that his computed digest matches what Alice sent him. Evan can intercept this data, but cannot decrypt the data or use the hash digest
Consequences	Sender cannot deny that they sent the message (assuming their private key is only known by them)
Known uses	The following cryptographic algorithms use symmetric encryption: GNuPG, Java Cryptographic Extension, .NET framework, and XML signature
See also	Other security patterns related to this include secure channel communication pattern

there is no consensus regarding how software design patterns are described. Even so, design patterns offer a clear benefit when consistently used within an organization. First, it allows software developers to use a common set of terms to describe aspects of their design. Second, software design patterns are typically developed by experienced professionals, making them useful to software development teams with a range of experiences. Finally, design patterns allow a development team to quickly discuss and assess the use of different design approaches.

27.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand the ways in which a software design pattern is described.
- You understand the benefits of using software design patterns.

Table 27.8 Summary of secure Model–View–Controller pattern [4]

Example	The address book case study
Context	Applicable to stand-alone and distributed software systems in homogeneous or heterogeneous environments
Problem	Need to secure all tiers of a system, since having an insecure tier/layer invites attacks. Should provide services that are available through mostly transparent security features to address the confidentiality, integrity, and availability (CIA) of the system. For some systems, recording actions performed by each user (i.e., supporting non-repudiation) would also be an important consideration to address the risk of an attack from a legitimate user
Solution	Apply an appropriate security service to each layer/tier, e.g., use encryption on data sent between the layers. Design the human–computer interaction (view component) to require authentication and authorization of users. Design into the business layer (controller component) a unified access control model. Design into the storage layer (model component) use of encryption for sensitive data
Implementation	Define a global authorization model and select authentication approaches based on needs of the application/system. Also, need to select an encryption approach
Example resolved	See Chaps. 25 (OOD) or 26 (SD) for a security design for the ABA
Consequences	Centralized management of security services, including authorization constraints, authentication information, and logging repositories. Each layer should apply security restrictions, which are (hopefully) transparent to the user. Considerations for availability, non-repudiation, and performance of security services
Known uses	Web services and distributed applications
See also	Other security patterns related to this include secure channel communication pattern

Table 27.9 Summary of design pattern templates

GoF (Original)	Larman (GRASP)	Fernandez (Security)
Intent	Problem	Example
Motivation	Solution	Context
Applicability	Discussion	Problem
Structure	Contraindications	Solution
Participants	Benefits	Implementation
Collaborations	Related patterns/principles	Example resolved
Consequences		Consequences
Implementation		Known uses
Sample code		See also
Known uses		
Related patterns		

Exercises

Discussion Questions

1. Using an existing design solution you've developed, discuss the use of each pattern (listed below) in your design.

   a. Singleton
   b. Facade
   c. Creator

2. Using an existing design solution you've developed, discuss the use of each security design pattern (listed below) in your design.

   a. Symmetric Encryption
   b. Asymmetric Encryption
   c. Digital Signature with Hashing
   d. Secure Three-Tier Architecture

3. Using Table 27.9, discuss the similarities and differences in how design patterns are described by the GoF, Larman, and Fernandez.

Hands-on Exercises

1. Use an existing design solution you've developed, improve your design by adding one or more of the design patterns described in this chapter.

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston
2. Alexander C, Ishikawa S, Silverstein M (1977) A pattern language: towns, buildings, construction. Oxford University Press, Oxford
3. Larman C (2002) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall
4. Fernandez EB (2013) Security patterns in practice: designing secure architectures using software patterns, 1st edn. Wiley, New York



OOD Case Study: Design Patterns

28

The objective of this chapter is to apply design patterns introduced in Chap. 27 to the case studies.

28.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand the ways in which a software design pattern is described and you understand the benefits of using software design patterns.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_28

28.2 OOD: ABA Design Patterns

We will use the same list of requirements as stated in Chap. 18. These requirements are listed below. The last requirement, *in italics*, was added in Chap. 18.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person, a single phone number and a single email address.
3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. *Allow for display of contact data in the address book.*

28.2.1 OOD: ABA GoF Patterns

The design of the ABA described in this chapter includes the implementation of three GoF patterns. The package diagram in Fig. 28.1 shows a view component with an ABA_A_view class implementing the Facade pattern and an ABA_A_viewCommand class implementing the Command pattern [1].

The GoF Facade pattern [1] says a subsystem must provide a single interface to hide the complexities of the subsystem. The package diagram in Fig. 28.1 clearly shows the controller classes having an association only with the ABA_A_view class. The class diagram in Fig. 28.2 shows instance variables allowing this class to call public methods within other view classes and has public methods called by the controller.

The GoF Command pattern [1] in this design encapsulates a user request into an object used by both the view and controller components. The class diagram in Fig. 28.3 shows all the classes in the view component. The ABA_A_viewCommand

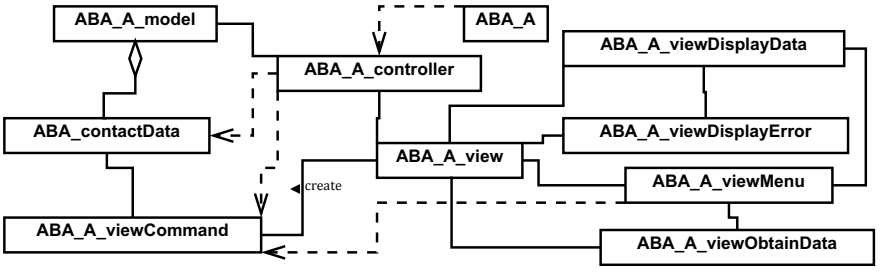


Fig. 28.1 Package diagram—ABA

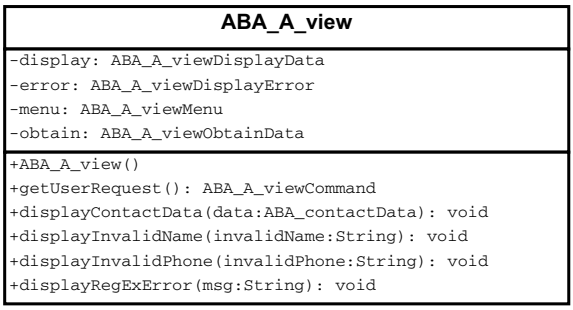


Fig. 28.2 Class diagram—ABA view class

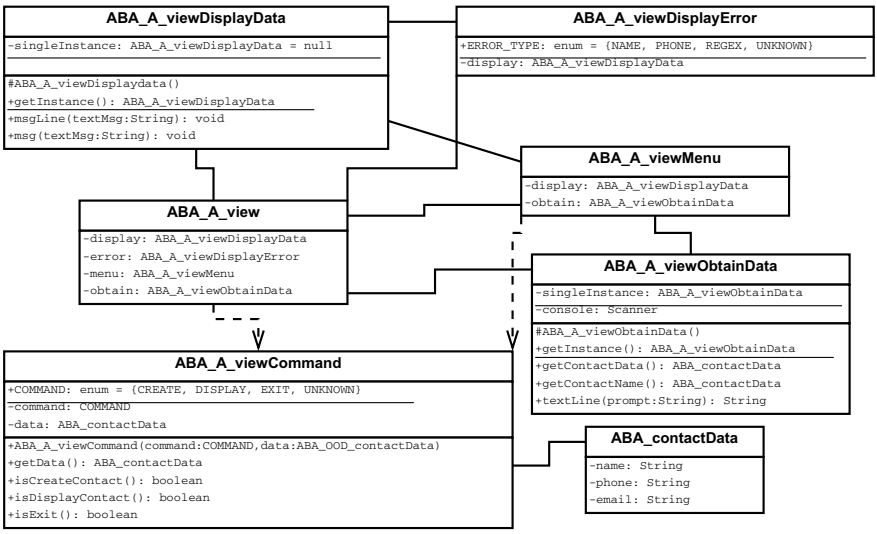


Fig. 28.3 Class diagram—ABA view component

class defines an enumerated data type to represent the three user requests: create, display, and exit. The data instance variable in this class is used to store the contact data associated with the user request. For a create command, the user enters all three values—name, phone, and email. For a display request, the user enters only a contact name. No additional user data is needed for the exit command.

The third GoF pattern used in this design is the Singleton pattern [1]. Two classes in the view component—ABA_A_viewDisplayData and ABA_A_viewObtainData—implement the Singleton pattern. The class diagram in Fig. 28.3 and Listing 28.1 shows the instance variables, constructor, and getInstance methods for the ABA_A_viewObtainData class. The private singleInstance variable is initialized to null and is used to store the single object instance. The constructor method in these classes is protected, while the public static getInstance method is used to return the single object instance, which is created in this method when the singleInstance vari-

able is null. Use of the Singleton pattern for these two classes makes sense. The ABA_A_viewDisplayData class is used by the view and viewDisplayError classes, while the ABA_A_viewObtainData class is used by the view and viewMenu classes.

Listing 28.1 ABA MVC Design—Singleton pattern

```
private static ABA_A_viewObtainData singleInstance = null;
private Scanner console;
private final String EXIT = "exit";

//pre: The single instance has not yet been constructed.
//post: singleInstance is now instantiated.
protected ABA_A_viewObtainData()
{
  console = new Scanner(System.in);
}

//pre: Someone needs the single instance for this class.
//post: Returns single object instance of this class.
public static ABA_A_viewObtainData getInstance()
{
  if (singleInstance == null)
    singleInstance = new ABA_A_viewObtainData();
  return singleInstance;
}
```

28.2.2 OOD: ABA Larman Patterns

The design of the ABA described in this chapter has about the same level of coupling between the controller and view as the solution described in Chap. 18, which was used as the staring point for this chapter's design. Larman's Low Coupling pattern [2] has been effectively implemented between the controller and view.

Cohesion within the view component has been improved (i.e., Larman's High Cohesion pattern [2] has improved the cohesion of classes in the view when compared to the Chap. 18 design). The specific changes to the view component in this chapter which increases cohesion are as follows.

ABA_A_viewMenu: This class is used to display and obtain a valid menu choice. This logic was part of the view class in the Chap. 18 design.

ABA_A_viewDisplayError: This class is used to display error messages. This logic was part of the view class in the Chap. 18 design.

ABA_A_view: The logic removed from this class, as described in the previous two items, has increased the cohesion of this class. As described above, the GoF Facade pattern has been implemented in this class. This resulted in creating two classes—viewMenu and viewDisplayError—just described.

28.2.3 OOD: ABA Fernandez Patterns

The design of the ABA described in this chapter does not include any security controls or mechanisms. We'll refer to the case study in Chap. 25 to discuss security patterns. First, Chap. 25 added three security requirements, listed below.

1. When the ABA starts, the user shall create a password. The password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character from the list ".!?@#$%&*_+-=".
2. Each request to create or display contact data shall first require the user to reenter their password. Three incorrect entries of the user's password shall result in the ABA exiting.
3. The data shall be non-persistently stored in encrypted form. A symmetric cryptographic algorithm shall be used.

These requirements resulted in a design which adheres to three security patterns described by Fernandez. The first pattern implemented in Chap. 25 is Authenticator [3]. The first two security requirements listed above describe the ABA password policy and when a user must authenticate. As described in Table 25.1, the security design principle complete mediation has been implemented since the user must reenter their password before being allowed to create or display a contact. The Authenticator pattern describes a solution which provides a single entry point into an application/system. In the case of the ABA design in Chap. 25, the user must create their password when the ABA starts. The user is not allowed to use the ABA until they've entered a valid password. In addition, the user must reenter their password each time they wish to create or display contact data.

The second pattern implemented in Chap. 25 is Symmetric Encryption [3]. The third security requirement listed above describes the ABA encryption policy when storing contact data. The encryption and decryption is done in the model component. Two classes shown in the model class diagram in Fig. 28.4 represent the implementation of the Symmetric Encryption pattern within a stand-alone application.

The ABA_A_modelGenKey constructor method generates a secret key used by the AES symmetric cryptographic algorithm. The ABA_A_modelCrypto class has encrypt and decrypt methods that will obtain the secret key via the getKey method. The model component receives an ABA_contactData object from the controller and creates an ABA_modelCipherData object by encrypting the phone number and email address. The model component stores ABA_modelCipherData objects in its memory-based data structure. When a user requests display of contact data, the contact name is given to the model in plaintext form. This is used to find the contact data in the data structure. When the name is found, the model component will call the decrypt method twice, resulting in an ABA_contactData object being created and given to the controller for display by the view.

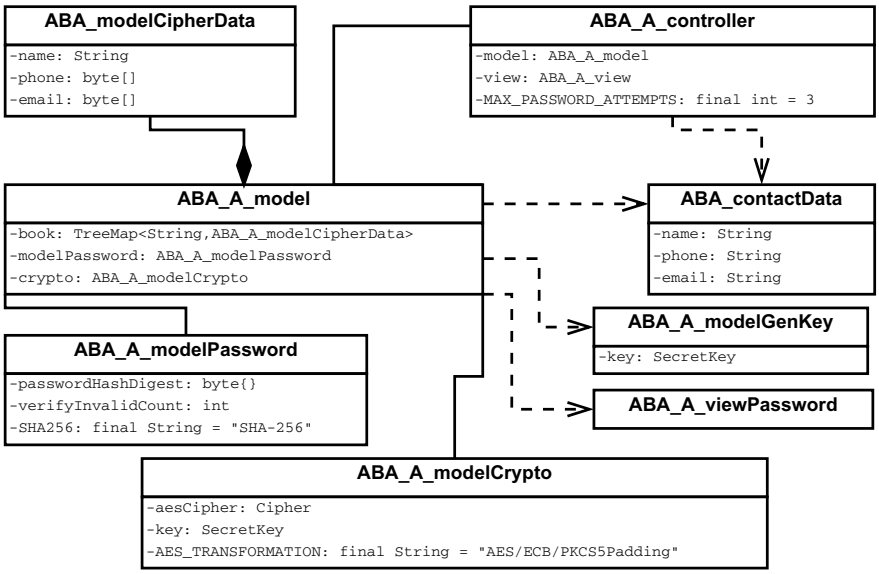


Fig. 28.4 ABA security design: Chap. 25 class diagram model component

The third pattern implemented in Chap. 25 is Secure Model–View–Controller [3]. Storing the user's password as a hash digest and storing the phone number and email address in encrypted form represents two significant security controls designed into the MVC framework.

28.3 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when using design patterns in a top-down design approach.

28.3.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.

- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

28.3.2 OOD Personal Finances: Design Patterns

Each subsection below briefly describes the use of design patterns (from [1–3]) for each object-oriented design discussed for the personal finances (PF) case study.

28.3.2.1 Design Patterns for Chap. 15 PF MVC

We'll describe possible design patterns based on the design described in Sect. 15.5. Based on the package diagram in Fig. 15.13, the following patterns could be used in the PF MVC design.

GoF Facade: The model and view classes should implement the GoF Facade pattern since classes in the controller component have an association relationship only with these two classes. That is, these two classes will hide the complexity of the other classes in the respective component.

GoF Singleton: The viewInput and viewOutput classes should implement the GoF Singleton pattern since we only need one instance of each of these to support the user interface.

Larman's Creator: The model component creates and owns many object instances representing different types of data stored in the PF application. This is expressed in the package diagram via the composition relationships between classes in this component. For example, the Finances class has a composition relationship with Account, Label, and Report classes, while the Account class has a composition relationship with the Transaction class. A composition relationship means the Finances class is responsible for creating and storing Account, Label, and Report objects while the Account class is responsible for creating and storing Transaction objects. Larman's description of his Creator pattern matches this description, resulting in this pattern being applicable to the PF MVC design.

GoF Factory: The number of different types of objects to be created and stored in the model component, as just described for Larman's Creator pattern, suggest we should use a GoF creational pattern to give our model component design more flexibility. The GoF Factory pattern could be used in this design. In this case, the

model component would have an abstract or concrete class with a factory method. This factory method would have a parameter indicating the type of object (e.g., Account, Label, Report, or Transaction) to create.

28.3.2.2 Design Patterns for Chap. 18 PF TUI

We'll describe possible design patterns based on the design described in Sect. 18.5. Based on the class diagram in Fig. 18.8, the following patterns could be used in the PF TUI design.

GoF Facade: The view class should implement the GoF Facade pattern since this class represents the interface to the entire view component.

GoF Singleton: The viewInput and viewOutput classes should implement the GoF Singleton pattern since we only need one instance of each of these to support the text-based user interface. The class diagram in Fig. 28.5 shows the design changes to these two classes.

GoF Command: The viewUserRequest class encapsulates a user action/request within the personal finances application. A viewUserRequest object is returned to the controller component (see definition of the getRequest method in the view class) for further processing of the user request.

Larman's Creator or GoF Factory: As described above for the Chap. 15 PF MVC design, either of these patterns could be used in the view component. The class diagram in Fig. 28.5 shows the viewUserData class as a superclass of the viewAccount, viewLabel, viewReport, and viewTransaction classes. This suggests use of the GoF Factory pattern as described above for the PF MVC design patterns.

28.3.2.3 Design Patterns for Chap. 21 PF GUI

We'll describe possible design patterns based on the design described in Sect. 21.4.1. Based on the class diagram in Fig. 21.9, the following patterns could be used in the PF GUI design.

GoF Facade: The view class should implement the GoF Facade pattern since this class represents the interface to the entire view component.

GoF Singleton: The viewFinancesGUI class should implement the GoF Singleton pattern since we only need one instance of this to support the graphical-based user interface. The class diagram in Fig. 28.6 shows the design changes to this class. Note that the viewTransGUI class should not implement the Singleton pattern since an Account will need to display many Transactions.

GoF Command: The viewUserRequest class encapsulates a user action/request within the personal finances application. A viewUserRequest object is given to the controller component (as part of the processRequest method in the view class) for further processing of the user request.

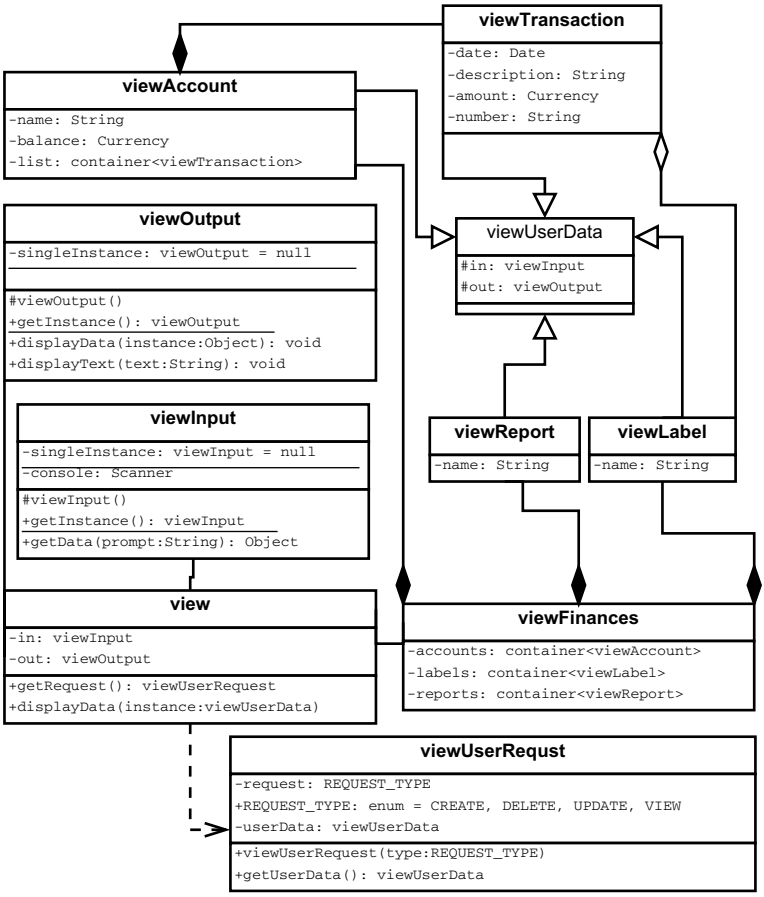


Fig. 28.5 PF Class Diagram: Chap. 18 updated with patterns

Larman's Creator or GoF Factory: As described above for the Chaps. 15 and 18 designs, either of these patterns could be used in the view component.

28.3.2.4 Design Patterns for Chap. 25 PF Security

We'll describe possible design patterns based on the design described in Sect. 25.3. Based on the security design description found in Sect. 25.3.2, the following security patterns described by Fernandez should be used in the PF security design.

Authenticator: Use of a pin or password and two-factor authentication. The description of a distributed version of the personal finances application would result in using two Fernandez patterns: Remote Authenticator/Authorizer and

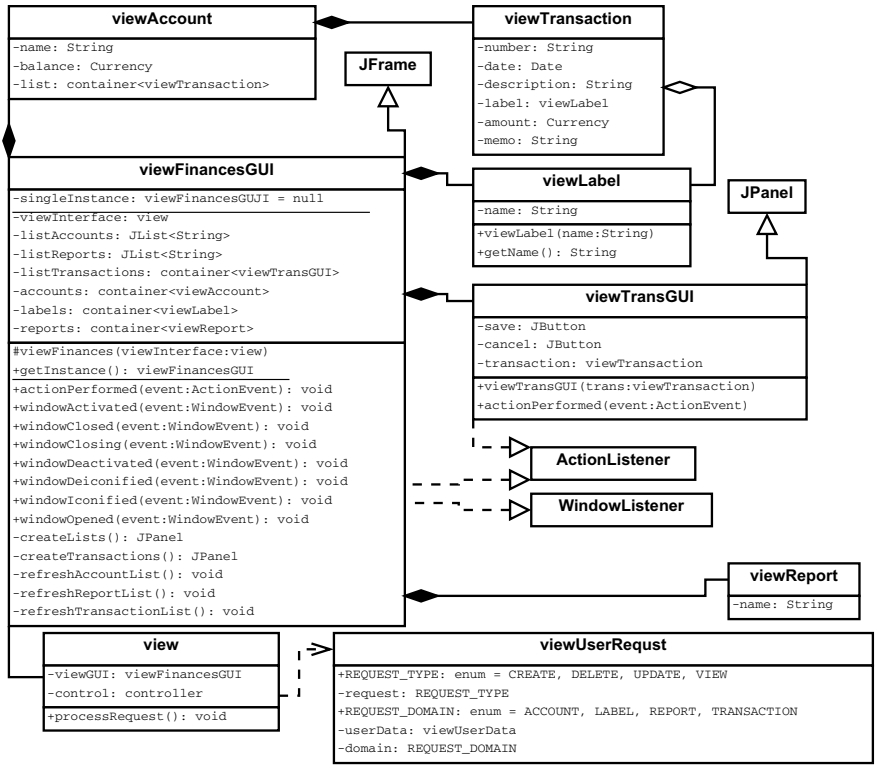


Fig. 28.6 PF Class Diagram: Chap. 21 updated with patterns

Credential. Both of these patterns address authentication in a distributed environment.

Symmetric Encryption: The encryption of data that is persistently stored.

Asymmetric Encryption: The description of a distributed version of the personal finances application suggests using asymmetric encryption between distributed PF components to securely agree to a secret key. This secret key is then used by a symmetric algorithm to encrypt and decrypt data in motion between distributed PF components.

Secure Model–View–Controller: The focus here is on ensuring that there are adequate security controls between the tiers (i.e., components) of the PF application.

Digital Signature with Hashing: The description of a distributed version of the PF application should include each PF component providing evidence proving that they are whom they say they are. We do not want a malicious client to gain access to server data, and we do not want a legitimate client interacting with a malicious server. Designing this pattern into the PF application should greatly reduce this risk.

28.4 OOD: Post-conditions

The following should have been learned when completing this chapter.

- You understand the ways in which a software design pattern is described and the benefits of using software design patterns.
- You've applied design patterns to a software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data-flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Use an existing code solution you've developed, develop design models that utilize one or more design patterns. Apply the six characteristics of a good design to your design models. How good or bad is your design?
2. Use your development of an application you started in Chap. 3 for this exercise. Modify your design to use some design patterns, and then evaluate your design using the six characteristics of a good design.
3. Continue Hands-on Exercise 3 from Chap. 12 by developing a design that utilizes one or more design patterns. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)
    - Bus transportation system

- Course-class enrollment
- Digital library
- Inventory and distribution control
- Online retail shopping cart
- Personal calendar
- Travel itinerary

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley
2. Larman C (2002) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall
3. Fernandez EB (2013) Security Patterns in Practice: Designing Secure Architectures Using Software Patterns, 1st edn. Wiley



SD Case Study: Design Patterns

29

The objective of this chapter is to apply design patterns introduced in Chap. 27 to the case studies.

29.1 SD: Preconditions

The following should be true prior to starting this chapter.

- You understand the ways in which a software design pattern is described and you understand the benefits of using software design patterns.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that describe a structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_29

29.2 SD: Design Patterns

As mentioned in Chap. 27, design patterns are described using object-oriented design models. This chapter will apply design patterns to structured code and will use structured design models and Python to express how these patterns are implemented.

29.2.1 SD Versus OOD

When developing an object-oriented solution, your focus is on identifying different types of *entities* to be represented in software. Each type of entity is defined using a software definition called a class. Each class contains instance variables and methods. A special type of method called a constructor is used to instantiate (i.e., create) objects when the software is executing. Thus, thinking about object-oriented design (OOD) is about defining classes and instantiating objects at runtime.

When developing a structured solution, your focus in on identifying data to be manipulated by functions defined in the software. Each type of data (or collection of related data elements) typically has a bunch of functions defined to represent the ways in which the data may be manipulated. The notion of instantiating data in structured design (SD) has little in common with instantiating objects in OOD.

Table 29.1 summarizes the difference between OOD and SD. Since design patterns are described using OOD models, this chapter will translate these models into corresponding SD models. One other comparison at this time: data structures in an OOD are generally used to store object instances, while data structures in a SD are used to store data values.

29.3 SD: ABA Design Patterns

We will use the same list of requirements as stated in Chap. 19. These requirements are listed below. The last requirement, *in italics*, was added in Chap. 19.

1. Allow for entry and (nonpersistent) storage of people's names.
2. Store for each person, a single phone number and a single email address.

Table 29.1 OOD versus SD

Paradigm	Data	Processing	Runtime
OOD	Instance variables defined in a class	Methods defined in a class	Instantiate objects, then use these objects to call methods
SD	Global variables defined in a module	Functions defined in a module	Call functions

3. Use a simple text-based user interface to obtain the contact data.
4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. *Allow for display of contact data in the address book.*

29.3.1 SD: ABA GoF Patterns

The design of the ABA described in this chapter includes the implementation of two GoF patterns. The structure chart in Fig. 29.1 shows the series of function calls in the view component, initiated by the controller, to obtain data associated with creating a new contact. This structure chart illustrates how the ABA_A_view module acts as an interface for the entire view component. Thus, the ABA_A_view module implements the Facade pattern [1].

The GoF Facade pattern [1] says that a subsystem will provide a single interface to hide the complexities of the subsystem. In this design, the ABA_A_view module in Listing 29.1 shows the function signatures called by the controller to initiate processing within the view component. The *import* statements at the top of Listing 29.1 identifies the other view modules called directly from the ABA_A_view module.

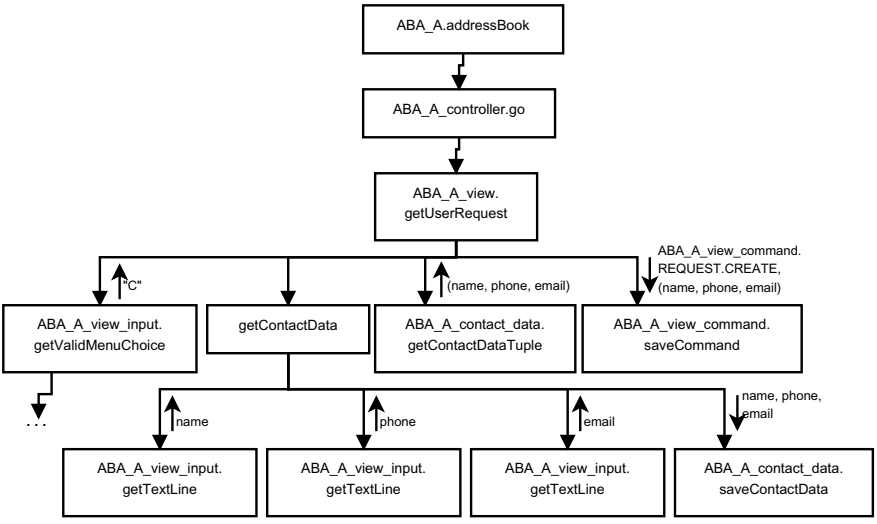


Fig. 29.1 Structure chart—ABA view create contact

Listing 29.1 ABA MVC Design—Facade pattern in ABA_A_view

```
import ABA_A_contact_data
import ABA_A_view_command
import ABA_A_view_input
import ABA_A_view_output

#pre: Display contact data for one person.
#post: One entry from address book has been displayed.
def displayContactData():

#pre: Display error message for invalid name.
#post: Error message reporting invalid name has been displayed.
def displayInvalidName(name):

#pre: Display error message for invalid phone.
#post: Error message reporting invalid phone has been displayed.
def displayInvalidPhone(phone):

#pre: User ready to enter contact data.
#post: ABA_A_contact_data now has a tuple or was set to None.
def getContactData():

#pre: Obtains from user a contact name for a single person.
#post: Returns object containing the user data entered.
def getContactName():

#pre: Get a request from the user.
#post: Returns the request entered by the user.
def getUserRequest():
```

The second GoF pattern used in this design is the Command pattern [1], which is used to encapsulate a user request into a module used by all three components—view, controller, and model. The structure chart in Fig. 29.1 shows the ABA_A_view_command.saveCommand function being called to store the type of command (i.e., ABA_A_view_command.REQUEST.CREATE) and the contact data tuple in the module. The structure chart in Fig. 29.2 shows the series of function calls in the controller component to validate new contact data and then call the ABA_A_model.storeContactData function when the contact data is valid. Note the use of functions in the ABA_A_view_command module by both the controller and model components.

Listing 29.2 shows function definitions and function signatures for the ABA_A_view_command module. It also shows a definition for an enumerated data type to represent the three user requests: create, display, and exit. The request global variable is used to store the request type and the contactData global variable is used to store a contact data tuple.

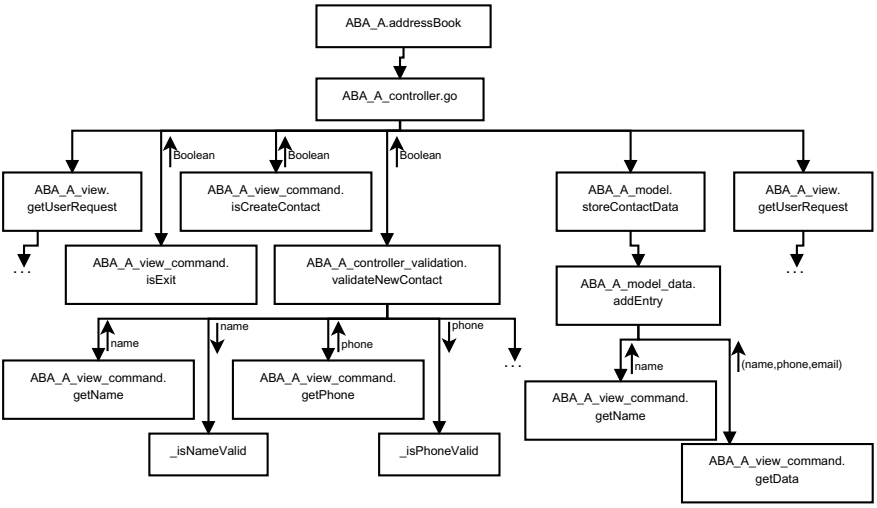


Fig. 29.2 Structure chart—ABA controller create contact

Listing 29.2 ABA MVC Design—Command pattern in ABA_A_view_command

```
import ABA_A_contact_data
from enum import Enum

#Create an enumeration to identify the type of request.
class REQUEST(Enum):
    CREATE = 1
    DISPLAY = 2
    EXIT = 3

#pre: Need to save a request to communicate with controller.
#post: Request has been saved.
def saveCommand(requestEnum, contact_data):
    global request
    global contactData
    request = requestEnum
    contactData = contact_data

#post: Returns the contact data.
def getData():
    global contactData
    return contactData

#post: returns phone number entered by user.
def getEmail():

#post: returns contact name entered by user.
def getName():
```

```
#post: returns email entered by user.
def getPhone():

#post: Returns Boolean indicating whether request is CREATE.
def isCreateContact():

#post: Returns Boolean indicating whether request is DISPLAY.
def isDisplayContact():

#post: Returns Boolean indicating whether request is EXIT.
def isExit():
```

29.3.1.1 ABA GoF Singleton Pattern

The GoF Singleton pattern is used in an OOD when only one object instance of a class should ever be created. Given the differences between OOD and SD, as described in Sect. 29.2.1 and Table 29.1, the Singleton pattern does not apply to structured design/programming.

In the OOD described in Chap. 28, the ABA_A_viewDisplayData and ABA_A_viewObtainData classes implement the Singleton pattern. These two classes provide logic to support the text-based user interface of the ABA. From an OOD perspective, there is no need to create many ABA_A_viewDisplayData or ABA_A_viewObtain Data objects, one object of each type is all that is needed. In fact, creating many of these objects negatively impacts memory usage since each of these objects reside in memory.

The SD being described in this chapter includes ABA_A_view_input and ABA_A_view_output modules, which provide similar functionality to the two classes just described. However, a Python module exists as a single instance. You cannot create multiple instances of a module.[1] The rest of this chapter will ignore the GoF Singleton pattern.

29.3.2 SD: ABA Larman Patterns

The design of the ABA described in this chapter has about the same level of coupling between the controller and view as the solution described in Chap. 19, which was used as the staring point for this chapter's design. Larman's Low Coupling pattern [2] has

[1] Saying that you cannot create multiple instances of a Python module assumes we are talking about a single-threaded stand-alone application. If we are designing a distributed application, then a module may be used on many different devices. In this case, there would be a module instance running on each device participating in the distributed application. Similarly, if the ABA were to use multiple threads in its design, the module (or modules) used by each thread would have multiple instances running. In this case, care must be taken to distinguish between global variables, which would be shared among all module thread instances, and threading local data, which would be distinct for each thread.

been effectively implemented between the controller and view. Similarly, cohesion of each module in the view component is the same as the solution in Chap. 19.

29.3.3 SD: ABA Fernandez Patterns

The design of the ABA described in Chap. 19 does not include any security controls or mechanisms. We'll refer to the case study in Chap. 26 to discuss security patterns. First, Chap. 26 added three security requirements, listed below.

1. When the ABA starts, the user shall create a password. The password must contain at least eight characters with at least one uppercase letter, one lowercase letter, one digit, and one special character from the list ".!?@#$%&*_+−=".
2. Each request to create or display contact data shall first require the user to reenter their password. Three incorrect entries of the user's password shall result in the ABA exiting.
3. The data shall be non-persistently stored in encrypted form. A symmetric cryptographic algorithm shall be used.

These requirements resulted in a design which adheres to three security patterns described by Fernandez. The first pattern implemented in Chap. 26 is Authenticator [3]. The first two security requirements listed above describe the ABA password policy and when a user must authenticate. As described in Table 26.1, the security design principle complete mediation has been implemented since the user must reenter their password before being allowed to create or display a contact. The Authenticator pattern describes a solution which provides a single entry point into an application/system. In the case of the ABA design in Chap. 26, the user must create their password when the ABA starts. The user is not allowed to use the ABA until they've entered a valid password. In addition, the user must reenter their password each time they wish to create or display contact data.

The second pattern implemented in Chap. 26 is Symmetric Encryption [3]. The third security requirement listed above describes the ABA encryption policy when storing contact data. The encryption and decryption is done in the model component. The structure chart in Fig. 29.3 shows the series of function calls for the implementation of the Symmetric Encryption pattern within a stand-alone application.

The ABA_A_modelGenKey.init function generates a secret key used by the AES symmetric cryptographic algorithm. The ABA_A_modelCrypto module has encrypt and decrypt methods which obtain the secret key via the ABA_A_modelGenKey.getKey function. The model component receives the contact data via the ABA_contact_data module and creates a tuple in the ABA_A_modelCipherData module containing the name in plaintext and the phone and email in ciphertext. The model component then stores the tuple in the ABA_A_modelCipherData module in its memory-based data structure. When a user requests display of contact data, the contact name is given to the model in plaintext form. This is used to find the contact data in the data structure. When the name is found, the model component will call the

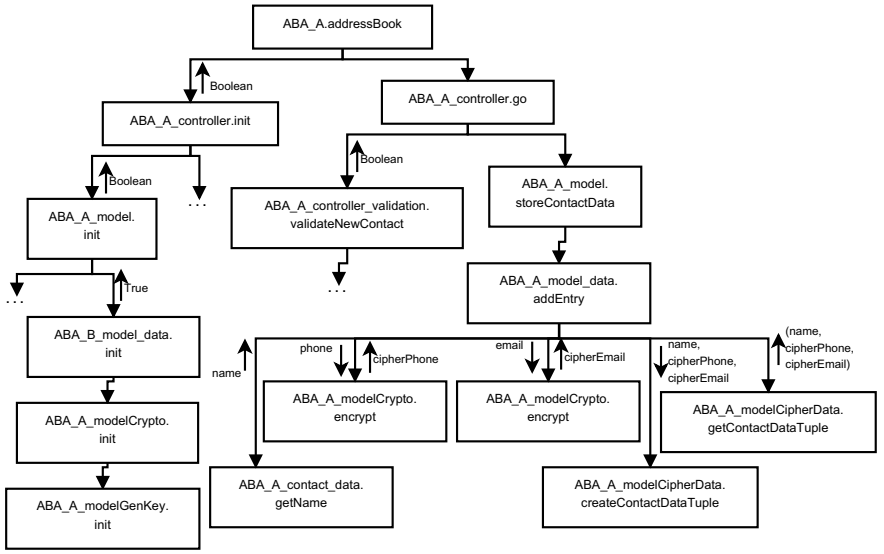


Fig. 29.3 ABA security design: structure chart for encrypting data

decrypt method twice, resulting in the ABA_A_contact_data module being updated with a tuple containing plaintext data values, which is then displayed to the user.

The third pattern implemented in Chap. 26 is Secure Model–View–Controller [3]. Storing the user's password as a hash digest and storing the phone number and email address in encrypted form represents two significant security controls designed into the MVC framework.

29.4 SD Top-Down Design Perspective

We'll use the personal finances case study to reinforce design choices when using design patterns in a top-down design approach.

29.4.1 SD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 13, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.

- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts, and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

29.4.2 SD Personal Finances: Design Patterns

Each subsection below briefly describes the use of design patterns (from [1–3]) for each structured design discussed for the personal finances (PF) case study.

29.4.2.1 Design Patterns for Chap. 16 PF MVC

We'll describe possible design patterns based on the design described in Sect. 16.5. Based on the hierarchy chart in Fig. 16.16 and the structure charts in Figs. 16.18 and 16.19, the following patterns could be used in the PF MVC design.

GoF Facade: The model and view components should each have a module implementing the GoF Facade pattern since functions in the controller component will need to invoke processing in both of these components. These two modules will hide the complexity of the other modules in the respective component.

29.4.2.2 Design Patterns for Chap. 19 PF TUI

We'll describe possible design patterns based on the design described in Sect. 19.5. Based on the structure chart in Fig. 19.7, the following patterns could be used in the PF TUI design.

GoF Facade: The view component should have a module implementing the GoF Facade pattern since we would want to hide the complexity of the view component from the controller. Similarly, the model component should have a module implementing the Facade pattern.

GoF Command: A module (e.g., viewUserRequest) should be designed to encapsulate a user action/request within the personal finances application. The viewUserRequest module would contain the user request along with any data associated with the request.

29.4.2.3 Design Patterns for Chap. 22 PF GUI

We'll describe possible design patterns based on the design described in Sect. 22.4.1. Based on the structure chart in Fig. 22.9, the following patterns could be used in the PF GUI design.

GoF Facade: The view component should have a module implementing the GoF Facade pattern since we would want to hide the complexity of the view component from the controller. Similarly, the model component should have a module implementing the Facade pattern.

GoF Command: A module (e.g., viewUserRequest) should be designed to encapsulate a user action/request within the personal finances application. The viewUserRequest module would contain the user request along with any data associated with the request.

29.4.2.4 Design Patterns for Chap. 26 PF Security

We'll describe possible design patterns based on the design described in Sect. 26.3. Based on the security design description found in Sect. 26.3.2, the following security patterns described by Fernandez should be used in the PF security design.

Authenticator: Use of a pin or password and two-factor authentication. The description of a distributed version of the personal finances application would result in using two Fernandez patterns: Remote Authenticator/Authorizer and Credential. Both of these patterns address authentication in a distributed environment.

Symmetric Encryption: The encryption of data that is persistently stored.

Asymmetric Encryption: The description of a distributed version of the personal finances application suggests using asymmetric encryption between distributed PF components to securely agree to a secret key. This secret key is then used by a symmetric algorithm to encrypt and decrypt data in motion between distributed PF components.

Secure Model–View–Controller: The focus here is on ensuring there are adequate security controls between the tiers (i.e., components) of the PF application.

Digital Signature with Hashing: The description of a distributed version of the PF application should include each PF component providing evidence proving that they are whom they say they are. We do not want a malicious client to gain access to server data, and we do not want a legitimate client interacting with a malicious server. Designing this pattern into the PF application should greatly reduce this risk.

29.5 SD: Post-conditions

The following should have been learned when completing this chapter.

- You understand the ways in which a software design pattern is described and the benefits of using software design patterns.
- You've applied design patterns to a software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You have created and/or modified models that describe a structured software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Use an existing code solution you've developed, develop design models that utilize one or more design patterns. Apply the six characteristics of a good design to your design models. How good or bad is your design?
2. Use your development of an application you started in Chap. 4 for this exercise. Modify your design to use some design patterns, and then evaluate your design using the six characteristics of a good design.
3. Continue Hands-on Exercise 3 from Chap. 13 by developing a design that utilizes one or more design patterns. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 13 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)

- Bus transportation system
- Course-class enrollment
- Digital library
- Inventory and distribution control
- Online retail shopping cart
- Personal calendar
- Travel itinerary

References

1. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Boston
2. Larman C (2002) Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process, 2nd edn. Prentice Hall, Upper Saddle River
3. Fernandez EB (2013) Security patterns in practice: designing secure architectures using software patterns, 1st edn. Wiley, Hoboken



Modeling Persistent Data

30

The objective of this chapter is to introduce different types of design models used to express the structure of data in persistent data stores.

30.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_30

30.2 Concepts and Context

Up to this point, the Address Book case study has used very simple data relationships and a memory-based data structure to non-persistently store the data. This chapter will discuss more complex data relationships and how these are represented in design models. Chapter 31 will then discuss two persistent storage technologies.

30.2.1 Externally Versus Internally Stored Data

Storing data internal to the software program requires the use of memory-based data structures. The ABA designs have been using data structures (e.g., lists, dictionaries, trees) to store nonpersistent data. In designing these solutions, our focus has been on the way in which the software needs to use the data and determine the type of data structure to use based on its performance characteristics.

Storing data in a persistent data store also requires an understanding of the way in which the software needs to access the stored data and the performance characteristics of the external file format. In addition, external file formats provide rules for how data elements are stored in relation to each other. Currently, our simple ABA case study has a name, along with a single phone number and email address, for each contact person. These simple data requirements present very few design options as we transition to a persistent data store (i.e., external file format). In Chaps. 32 and 33, we'll add more data requirements to the ABA, allowing us to explore more interesting data designs.

30.2.2 Logical Data Modeling

As we begin to focus on persistent data storage we need to think about data from two different perspectives: a logical view of the data and a physical view of the data. In a logical data model, we are focused on how the different data elements are related to each other. In a physical data model, we are concerned about how the data is physically stored within the external file format.

30.2.2.1 Entity Relationship Diagram

An entity relationship diagram (ERD) is a type of logical data model. An ERD is a high-level modeling technique that logically describes how different collections of data elements relate to each other. This modeling technique uses the notations shown in Fig. 30.1.

An *entity* represents a thing or object that consists of a collection of logically related data elements. The entity notation is a rectangle shape that has a noun or noun phrase to name the entity. A *relationship* describes how two entities are logically related to each other. The relationship notation is a diamond shape that has a verb or

verb phrase to describe the relationship. A *connector* is a line that connects an entity notation to a relationship notation.

30.2.2.2 Fully-Attributed Logical Data Model

A Fully-attributed logical data model (LDM) is another type of logical data model. A LDM is a more detailed modeling technique that logically shows how data elements are related to entities, and how entities are related to each other. This modeling technique uses the notations shown in Fig. 30.2.

The Entity, relationship, and connector notations are the same as the ERD. A LDM also includes notation to represent a *weak entity* and an *attribute*. A weak entity is similar to an entity, except a weak entity must include data from another entity in order to uniquely identify data instances of the weak entity. (The examples below will illustrate the differences between an entity and a weak entity.) The weak entity notation is a rectangle that has two lines that form its four sides. An *attribute* represents a single data element (i.e., data field) that is logically related to an entity. The attribute notation is an oval with a noun or noun phrase to name the attribute.

30.2.2.3 ERD and LDM Example

Let's say we want to model the data for a digital library that will store books and movies. We want to use this digital library to keep track of all of the titles we own. This digital library is intended for use by a single person.

The information we want to keep for a book might include the title, author, and publisher. The information we want to keep for a movie might include the title and director. To develop an ERD we need to think about these nouns (e.g., author, book, director, movie, publisher, and title) and determine which of these are entities and which would be attributes associated with an entity.

If we consider each of these nouns to be an entity we end up with an ERD similar to Fig. 30.3. Here the information for a book is represented in four entities. The book entity has a relationship with three other entities. Note the use of *cardinality* symbols (e.g., "1" and "*") for each relationship. The "written-by" relationship is read two ways: (1) *A book is written-by many authors*; and (2) *An author writes a book*. Below is a list of all of the data relationship rules for the ERD in Fig. 30.3.

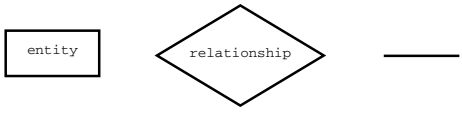


Fig. 30.1 Entity Relationship Diagram (ERD) notations

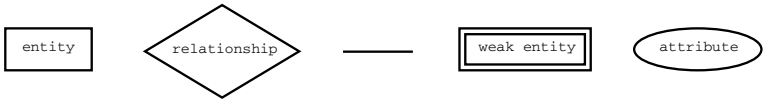


Fig. 30.2 Logical Data Model (LDM) notations

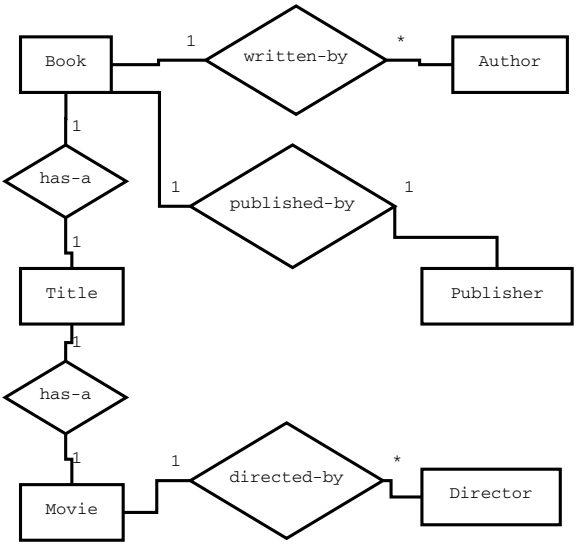


Fig. 30.3 Digital library ERD Version 1

- A book is written-by one or more authors. An author writes a book.
- A book is published by one publisher. A publisher publishes one book.
- A book has a title. A title is for one book.
- A movie is directed by one or more directors. A director directs a movie.
- A movie has a title. A title is for one movie.

There are (at least) two problems with this ERD. First, any additional information we want to store for a book or movie (e.g., year published) would result in another entity. This seems to contradict the purpose of an entity as a data object containing a collection of logically related data elements.

Second, what data attributes would we associate with each entity in the ERD? Figure 30.4 shows a LDM with an attribute associated with the author, title, publisher, and director entities. Note the book and movie entities do not have any attributes while the other three entities have only one attribute each. Again, this contradicts the purpose of an entity as a data object which contains a collection of logically related data elements.

Figure 30.5 shows a second attempt at an ERD for this digital library. This ERD does not show any relationship between the book and movie entities—we simply have a bunch of books and movies in the library where none of the books have any relationship to any of the movies. The other observation regarding the version 2 ERD is that all of the one-to-one relationships shown in the version 1 ERD have been eliminated, while the two one-to-many relationships are still shown in version 2. Below is a list of all of the data relationship rules for the ERD in Fig. 30.5.

- A book is written-by one or more authors. An author writes a book.
- A movie is directed by one or more directors. A director directs a movie.

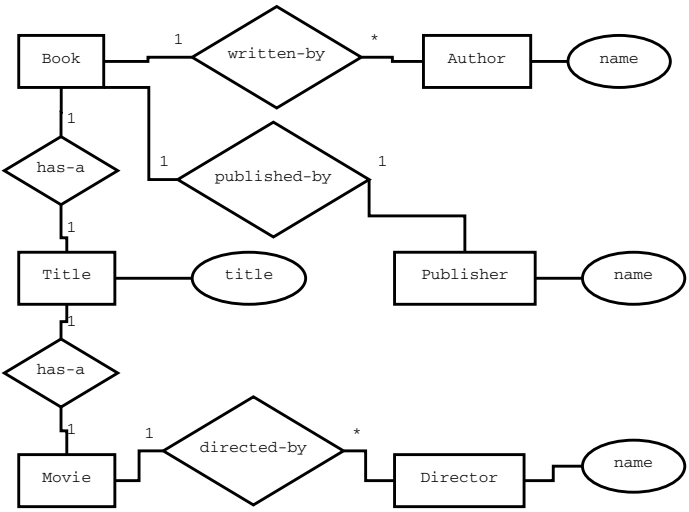
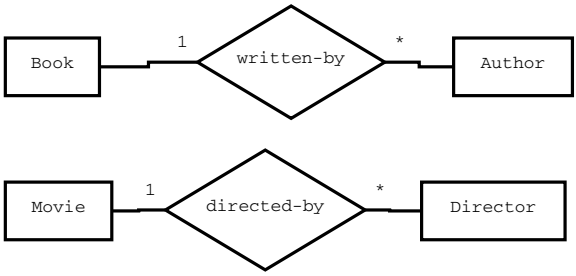


Fig. 30.4 Digital library LDM Version 1

Fig. 30.5 Digital library ERD Version 2



The LDM in Fig. 30.6 now makes more sense since each entity has a list of attributes associated with it. Each attribute represents a data value that helps to describe the entity. The isbn attribute is underlined for a book while the title and year published attributes are underlined for a movie. These attributes represent the *primary key* for the appropriate entity. That is, each book stored in the digital library will have a unique isbn value while each movie stored in the library will have a unique value for the combination of title and year published.

The version 2 LDM also shows the author and director as weak entities. This says an author of a book cannot be identified unless we know the specific book. From a data modeling perspective, the Author weak entity and its weak key attribute—name—requires the primary key value from the book instance (i.e., the isbn value) in order to correctly identify the authors for a book. A similar arrangement exists between movie and the director weak entity.

One last observation regarding Fig. 30.6—both the book and movie entities have attributes named title and year published. Can we simplify the LDM by showing one title and one year published attribute, with a relationship showing a connection from

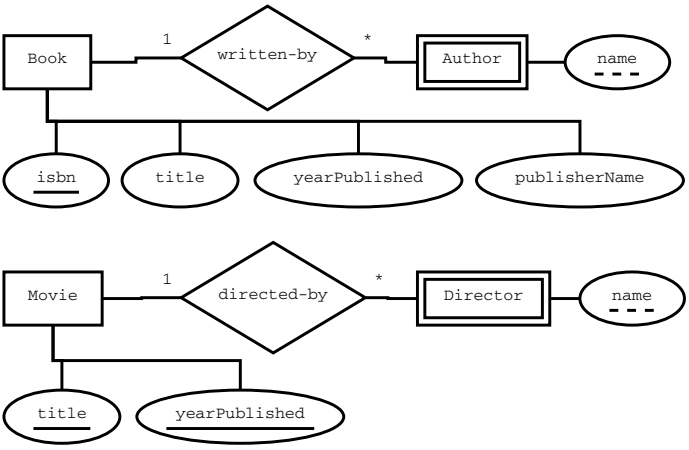
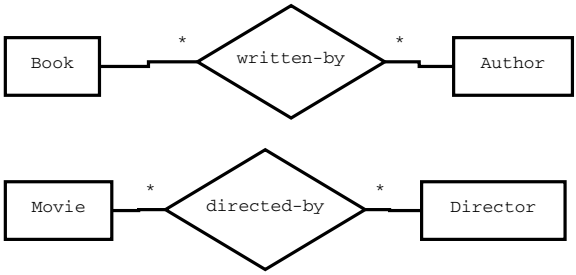


Fig. 30.6 Digital library LDM Version 2

Fig. 30.7 Digital library ERD Version 3



each of these attributes to both the book and movie entities? The answer is no, for two reasons. First, when an LDM shows one attribute being associated with two distinct entities, this is saying that the same attribute instance (i.e., value) is used in both entity instances. In the case of the digital library, it is not true that a title for a book will always exactly match the title for a movie. Second, the title and year published attributes associated with the movie entity are primary key attributes. However, these two attributes are not part of the primary key for the book entity.

Figure 30.7 shows a third attempt at an ERD for the digital library. The difference between the second and third ERDs are the cardinality rules. In version 3, we are showing a many-to-many relationship between book and author, and between movie and director. Below is a list of all of the data relationship rules for the ERD in Fig. 30.7.

- A book is written-by *one or more* authors. An author writes *one or more* books.
- A movie is directed by *one or more* directors. A director directs *one or more* movies.

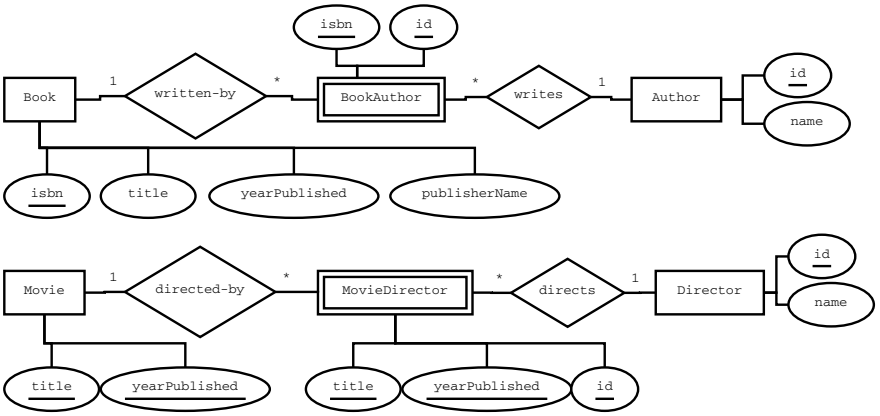


Fig. 30.8 Digital library LDM Version 3

The LDM in Fig. 30.8 has four relationships (written by, writes, directed by, and directs), four strong entities (book, author, movie, and director) and two weak entities (book author and movie director). Each weak entity and its two one-to-many relationships replace the corresponding many-to-many relationship shown in the ERD of Fig. 30.7. When converting a many-to-many relationship into two one-to-many relationships, the weak entity will contain, at a minimum, the primary key attributes from both strong entities they are related to. Below is a list of all of the data relationship rules for the LDM in Fig. 30.8. The first four rules are associated with cardinality and the last six rules describe the attributes associated with each entity.

- A book is written by *one or more* book authors. A book author writes *one* book.
- An author writes *one or more* book authors. A book author is written by *one* author.
- A movie is directed by *one or more* movie directors. A movie director directs *one* movie.
- A director directs *one or more* movie directors. A movie director is directed by *one* director.
- A book is uniquely identified by its isbn and is described by its title, year published, and publisher name.
- A book author is uniquely identified by its isbn and (author) id.
- An author is uniquely identified by its id (an integer value guaranteeing uniqueness) and is described by its name.
- A movie is uniquely identified by its title and year published.
- A movie director is uniquely identified by its title, year published, and (director) id.
- A director is uniquely identified by its id (an integer value guaranteeing uniqueness) and is described by its name.

Translating a many-to-many relationship in an ERD into two one-to-many relationships in an LDM is common practice when developing data models. This is done to make it easier to translate a logical data model into a physical data model based on the type of persistent storage technology to be used. This will be further discussed in Chap. 31.

30.2.3 Physical Data Modeling

Once we have a logical data model, the next step is to translate this into a physical representation. Unfortunately, there is not one approach to modeling the physical representation of data. This is because the type of physical storage to be used plays a significant role in how we decide to model the format of physical data. Two physical data formats—text-based XML files and relational databases—are described in detail in Chap. 31. A relatively simple physical data format is introduced in this chapter.

30.2.3.1 Text Files

When a text file is used to store data, we need to describe the order in which the data values are stored within the file. With this type of data storage, the most common approach is to store the data for a single entity instance on a line of text within the file. A common format for this type of text file is to use a delimiter character to separate each data value on a text line. Common character values to use as the delimiter are a comma or tab.

Listing 30.1 shows a text file where each logical data entity has its own *section* within the text file. The following observations are important:

- The special text lines {Book} and {Movie} are used to denote the start of text lines containing data for that logical data entity.
- Each data value is delimited with quotes, denoting a character (i.e., string) value. While an isbn value is a series of digits, we do not expect to do arithmetic on these values, and so we treat this data as characters.
- The data values on each text line are separated with commas.
- The book text lines list the author values last. This is because there may be many number of authors for a book.

Listing 30.1 Text File Example 1

```
{Book}
"isbn","title","edition","publisher","yearPublished","authors"
"0441172717","Dune","","Chilton Books","1965","Frank Herbert"
"0151008116","Life of Pi","","Harcourt","2001","Yann Martel"
"0137135599","Blown to Bits","Addison Wesley","2008","Hal
"0137135599","Abelson","Ken Ledeen","Harry Lewis"
{Movie}
"title","yearPublished","directors"
"The Shawshank Redemption","1994","Frank Darabont"
```

Listing 30.2 shows an alternative text file format where the name of a logical data entity is the first value found in each line of text.

Listing 30.2 Text File Example 2

```
"book","isbn","title","edition","publisher","yearPublished","authors"
"book","0441172717","Dune","","Chilton Books","1965","Frank Herbert"
"book","0151008116","Life of Pi","","Harcourt","2001","Yann Martel"
"book","0137135599","Blown to Bits","Addison Wesley","2008","Hal Abelson","Ken Ledeen","Harry Lewis"
"movie","title","yearPublished","directors"
"movie","The Shawshank Redemption","1994","Frank Darabont"
```

Both of these text file formats require the software code to be written with specific knowledge about the exact order of data values found on a text line. The code must know this order for each logical data entity whose data is stored in the text file. Another option would be to have a separate text file for each logical data entity. In this case, the text file would not need the special section text lines (that are shown in Listing 30.1) and each text line would be formatted with the same number of data values (like the three text lines shown inside the `Book` section).

30.2.4 Normalization

As a logical data model is created, one of our goals is to avoid unnecessary redundancy in the data elements. When redundant data elements exist in the physical format, any change to a data value would need to be updated in all of its redundant physical locations. Another goal when creating a logical data model is to keep the model as simple as possible while maintaining the logical relationships between the elements. To achieve these goals we apply a process called *normalization* [1]. This process analyzes data relations based on their functional dependencies and primary keys.

functional dependency: Denoted using the notation $A \rightarrow B, C$. This says that attributes B and C are functionally dependent on A. That is, a specific value for A will determine the values for B and C. For example, the functional dependency $studentID \rightarrow firstName, lastName, SSN$ says that once a specific student ID value is known, we can determine the first name, last name, and SSN for that student.

primary key: A primary key is one or more attributes that will uniquely identify each entity instance. Each attribute or set of attributes on the left side of a functional dependency are candidate primary keys.

relation: A text-based notation to express an entity and its associated attributes. For example, Student = (studentID, firstName, lastName, SSN) is a relation where *Student* is the name of the entity and the names inside the parentheses are the list of associated attributes. The underlined attribute(s) identify the primary key for the relation.

tuple: A text-based notation to express a list of attributes. For example, (studentID, firstName, lastName, SSN) is a tuple containing four attributes where one of the attributes (e.g., studentID) is the primary key.

The normalization process is described as a series of levels called forms.

30.2.4.1 First Normal Form (1NF)

A relation is in 1NF when *each attribute value is an atomic (or indivisible) value.* That is, a relation should have no multivalued attributes or nested relations [1].

For example, the following relation is not in 1NF.

```
Employee = (employeeID, employeeName, dependentName, address)
```

Why is this employee relation not in 1NF?

- The employee name is not atomic (e.g., first, middle, last).
- The dependent name is a multivalued attribute (i.e., an employee may have many dependents).
- The address is not atomic (e.g., street, city, state, zip).

30.2.4.2 Second Normal Form (2NF)

A relation is in 2NF when *it is in 1NF and each non-key attribute is fully functionally dependent on the primary key (PK).* That is, when a PK has multiple attributes, no non-PK attribute should be functionally dependent on a part of the PK [1].

For example, the following relation is not in 2NF.

```
EmployeeProject = (ssn, projID, hours, emplID, projName, projLocation)
```

First, this relation is in 1NF since each attribute is an atomic value. That is, each of the attributes listed in the relation are atomic and have a single value (for one employee project) instance. Why is this employee project relation not in 2NF?

- The functional dependency $ssn \rightarrow emplID$ shows a non-PK attribute (emplID) is functionally dependent on only a portion of the PK (ssn).
- The functional dependency $projID \rightarrow projName, projLocation$ shows two non-PK attributes (projName and projLocation) are functionally dependent on only a portion of the PK (projID).

One of the non-PK attributes—hours—is functionally dependent on the entire PK.

30.2.5 Third Normal Form (3NF)

A relation is in 3NF when *it is in 2NF and no non-PK attribute is transitively dependent on the PK*. That is, a relation should not have a non-key attribute functionally dependent on another non-key attribute [1].

For example, the following relation is not in 3NF.

```
EmployeeDepartment = (ssn, birthDate, deptID, deptName, deptMgrSSN)
```

First, this relation is in 2NF since the functional dependency below is true. $ssn \rightarrow birthDate, deptID, deptName, deptMgrSSN$ The reason why this is not in 3NF is that two of the attributes are transitively dependent on the PK as shown below:

- We can use the ssn to identify the attributes birth date and dept ID. Or, using functional dependency notation $ssn \rightarrow birthDate, deptID$. That is, knowing an employee's ssn allows us to get the employee's birth date and the department ID of where they work.
- We can use the dept ID to identify the attributes dept name and dept mgr ssn. Or, using functional dependency notation $deptID \rightarrow deptName, deptMgrSSN$. That is, knowing the department (deptID) allows us to get the name of the department and the department manager's ssn.
- Thus, each ssn (the PK) will identify a dept ID (a non-PK attribute), and each dept ID will identify two other non-PK attributes. This gives us a transitive dependency we want to avoid for 3NF.

30.2.6 Boyce-Codd Normal Form (BCNF)

A relation is in BCNF when *every non-trivial functional dependency in the relation is a dependency on a superkey*. That is, a relation is in BCNF if it eliminates all redundancy that can be discovered based on functional dependencies [1].

A *superkey* is any set of attributes that will uniquely identify each tuple in the relation. For example, the tuple (studentID, ssn, birthDate, firstName, lastName, graduationDate) has many valid functional dependencies, including:

- $studentID \rightarrow ssn, birthDate, firstName, lastName, graduationDate$
- $studentID, ssn \rightarrow birthDate, firstName, lastName, graduationDate$
- $studentID, ssn, birthDate \rightarrow firstName, lastName, graduationDate$
- $studentID, ssn, birthDate, firstName \rightarrow lastName, graduationDate$
- $studentID, ssn, birthDate, firstName, lastName \rightarrow graduationDate$
- $studentID, birthDate \rightarrow ssn, firstName, lastName, graduationDate$
- $studentID, birthDate, firstName \rightarrow ssn, lastName, graduationDate$
- etc.

All of these superkeys are candidate primary keys. The primary key is the minimal superkey (i.e., the minimal number of attributes in a relation that will uniquely identify each tuple in the relation). In this case, the first functional dependency in the above list shows the minimal superkey, resulting in the primary key being student ID.

For example, if we assume an employee can be on many projects, the following relation is not in BCNF.

```
EmployeeProject = (ssn, projID, projName, projHours)
```

The functional dependency $projID \rightarrow projName$ is true. But proj ID by itself is not a superkey (i.e., the combination of ssn and projID is the minimal superkey and the primary key).

30.2.7 Fourth Normal Form (4NF)

A relation is in 4NF when *every non-trivial multivalued dependency in the table is a dependency on a superkey*. To be in 4NF, a relation cannot have a multivalued dependency on a subset of the superkey [1].

For example, the following relation about a loan customer is not in 4NF.

```
LoanCustomer = (loanNumber, custID, custStreet, custCity)
```

First, this relation is in 3NF since no non-key attribute is functionally dependent on another non-key attribute. That is, all of the functional dependencies for this relation, listed below, show the non-key attributes are functionally dependent only on key attributes.

- $loanNumber, custID \rightarrow custStreet, custCity$
- $custID \rightarrow custStreet, custCity$

The second functional dependency above results in custStreet and custCity values being repeated for each customer loan, since only the custID is needed to uniquely identify these two attributes. This is why the above relation is not in 4NF.

30.2.7.1 Additional Levels of Normalization

Researchers have also defined a fifth (5NF) and a sixth normal form (6NF). These are not covered in this book as database designers tend to normalize up to either 3NF, BCNF, or 4NF. Furthermore, there are situations where a database designer will not normalize to 3NF to improve the performance of the database. This scenario is also outside the scope of this book.

30.2.8 Apply Normalization

We'll apply the normalization process to the digital library example presented earlier in this chapter.

Table 30.1 Relations and functional dependencies for LDM Example 2

`Book = (isbn, title, yearPublished, publisherName)`
`Author = (isbn, authorName)`
`Movie = (title, yearPublished)`
`Director = (title, yearPublished, directorName)`
• $isbn \rightarrow title, yearPublished, publisher$
• $isbn, authorName \rightarrow \varnothing$
• $title, yearPublished, directorName \rightarrow \varnothing$

30.2.8.1 Check 1NF

Is the ABA version 2 LDM in Fig. 30.6 in 1NF? i.e., does each attribute represent an atomic (or indivisible) value?

The relations and functional dependencies for the logical data model are shown in Table 30.1.

Each of the attributes now represents an atomic (or indivisible) value. (For this analysis, we are ignoring an author and director likely having a first and last name. We are considering their entire name to be atomic.) As shown in the functional dependencies, the multivalued author name and director name are part of the superkey for their respective relations. Given this, the logical data model in Fig. 30.6 is in 1NF.

30.2.8.2 Check 2NF

Is the ABA version 2 LDM in Fig. 30.6 in 2NF? i.e., are the relations in 1NF and is each non-key attribute fully functionally dependent on the primary key?

Given the relations and functional dependencies listed in Table 30.1:

- For a Book, knowing the isbn allows us to identify the title, yearPublished, publisher, and the authorNames for the book.
- For a Movie, knowing the title and year published allows us to identify the director names for the movie.

Thus, each of the non-key attributes is fully functionally dependent on the primary key. The logical data model in Fig. 30.6 is in 2NF.

30.2.8.3 Check 3NF

Is the ABA version 2 LDM in Fig. 30.6 in 3NF? i.e., are the relations in 2NF and is there no non-PK attribute that is transitively dependent on the PK?

Given the relations and functional dependencies listed in Table 30.1:

- Can a new edition of a book have the same isbn as a previous edition? No, each new edition has a different isbn value. So any change in the authors or publisher in this new edition will be identified given the new isbn value.

- Since the Movie and Director relations have no non-key attributes, these relations and their functional dependencies must be in 3NF.

Thus, each of the non-key attributes is fully functionally dependent on the primary key. The logical data model in Fig. 30.6 is in 3NF.

30.2.8.4 Check BCNF

Is the ABA version 2 LDM in Fig. 30.6 in BCNF? i.e., is every non-trivial functional dependency in a relation a dependency on a superkey?

Given the relations and functional dependencies listed in Table 30.1:

- Any functional dependency that specifies a superkey for the Book relation will have isbn as part of the superkey. Thus, all of the functional dependencies one could create in this relation are dependent on a superkey.
- Since the Movie and Director relations have no non-key attributes, these relations and their functional dependencies must be in BCNF.

Thus, each of the non-trivial functional dependencies is dependent on a superkey. The logical data model in Fig. 30.6 is in BCNF.

30.2.8.5 Check 4NF

Is the ABA version 2 LDM in Fig. 30.6 in 4NF? i.e., is every non-trivial multivalued dependency in a relation a dependency on a superkey?

Given the relations and functional dependencies listed in Table 30.1:

- The one multivalued dependency for the book relation has isbn and author as the superkey.
- All of the functional dependencies for the Movie relation have multivalued dependencies and the left side of each of these dependencies is a superkey.

Thus, each of the non-trivial multivalued dependencies is dependent on a superkey. The logical data model in Fig. 30.6 is in 4NF.

30.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating an LDM to a physical data model needs to consider the limitations of the physical storage format.

- You understand the process of normalization and have applied normalization forms to a data design.

Exercises

Discussion Questions

1. In your own words, explain each of the following concepts.

   a. ERD
   b. LDM
   c. Physical data model
   d. Normalization
   e. 1NF, 2NF, 3NF, BCNF, 4NF

Hands-on Exercises

1. Apply the normalization forms to the Digital Library version 3 LDM. Which level of normalization does this data model reach?
2. The Digital Library version 3 LDM includes the attribute publisherName associated with the Book entity. It is logically correct to say a publisher will publish many books. Given this, create a new ERD and LDM to show a Publisher entity along with its corresponding attributes and relationships.
3. Use an existing design solution you've developed, develop data models in preparation for persistently storing data.
4. Using your data models from the first hands-on exercise, evaluate your data models using the normalization forms. Which level of normalization does your data models reach?

Reference

1. Silberschatz A, Korth HF, Sudarshan S (2006) Database System Concepts, 5th edn. McGraw-Hill, New York



Persistent Data Storage

31

The objective of this chapter is to introduce text-based XML files and relational databases as two ways to persistently store data.

31.1 Preconditions

The following should be true prior to starting this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models.
- You understand the process of normalization and have applied normalization forms to data designs.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_31

Table 31.1 XML examples

<student>	<student>	<student>
Sally	<name>Sally</name>	<name>Sally</name>
1234567	<id>1234567</id>	<id>1122334</id>
Computer Science	<major>Computer Science</major>	<major />
</student>	</student>	</student>

31.2 XML Concepts

This section introduces XML as a way to store data persistently.

31.2.1 What is XML?

The acronym XML stands for *eXtensible Markup Language*. It is a tag-based language used to describe data elements and their relationships to each other. A tag-based language allows us to define a data element by embedding a tag-name inside angled brackets. For example, <student> denotes the start of data associated with a student while </student> denotes the end of data for a student. In between these two tags could be data values and/or other tags.

Table 31.1 shows three examples. The example on the left shows three data values inside a student tag. The example in the middle shows tags and data embedded inside a student tag. This example illustrates how XML represents relationships between data elements using a tree (or hierarchy) structure. The example on the right shows how a single tag may represent both a start and an end tag. Specifically, <major /> tag indicates this student does not have a major. Since tag names are programmer-defined, we have control over the data element names we decide to use. For example, we could use <firstName> instead of <name> and <studentID> instead of <id>. An important aspect of creating tag names is to make these names descriptive.

A very good resource to learn more about XML is at www.w3schools.com, a free web site resource initially developed by a Norwegian company that has tutorials for many of the popular web-based technologies [1].

31.2.2 XML Files

XML may be stored in a file using one of two formats: binary or text. A binary XML file is not readable by a text editor. Instead, special software must be written to read and update a binary XML file. Examples of binary XML files you've likely used are the document files created and maintained by the Open Office suite and the

Microsoft Office suite of software applications. This book does not discuss creating or updating binary XML files.

A text-based XML file is readable by a text editor. This allows us to view the tree structure of the XML tags. We can also modify the XML tree structure, the tag names, and the data associated with any of the tags. The rest of this section will discuss creating and updating text-based XML files.

Listing 31.1 shows a sample XML file for the digital library discussed in Chap. 30. This file contains three Book instances and one Movie instance.

Listing 31.1 Digital Library XML File Sample 1

```
<digital_library>
    <book>
        <isbn>0441172717</isbn>
        <title>Dune</title>
        <publisher>Chilton Books</publisher>
        <yearPublished>1965</yearPublished>
        <author>Frank Herbert</author>
    </book>
    <book>
        <isbn>0151008116</isbn>
        <title>Life of Pi</title>
        <publisher>Harcourt</publisher>
        <yearPublished>2001</yearPublished>
        <author>Yann Martel</author>
    </book>
    <book>
        <isbn>0137135599</isbn>
        <title>Blown to Bits</title>
        <publisher>Addison Wesley</publisher>
        <yearPublished>2008</yearPublished>
        <author>Hal Abelson</author>
        <author>Ken Ledeen</author>
        <author>Harry Lewis</author>
    </book>
    <movie>
        <title>The Shawshank Redemption</title>
        <yearPublished>1994</yearPublished>
        <director>Frank Darabont</director>
    </movie>
</digital_library>
```

31.2.3 Document Object Model (DOM)

A text-based XML file is parsed into a Document Object Model (DOM). The Document Object Model [2] is:

- A specification that defines the logical structure of HTML and XML documents.
- An application programming interface (API) for HTML and XML documents.

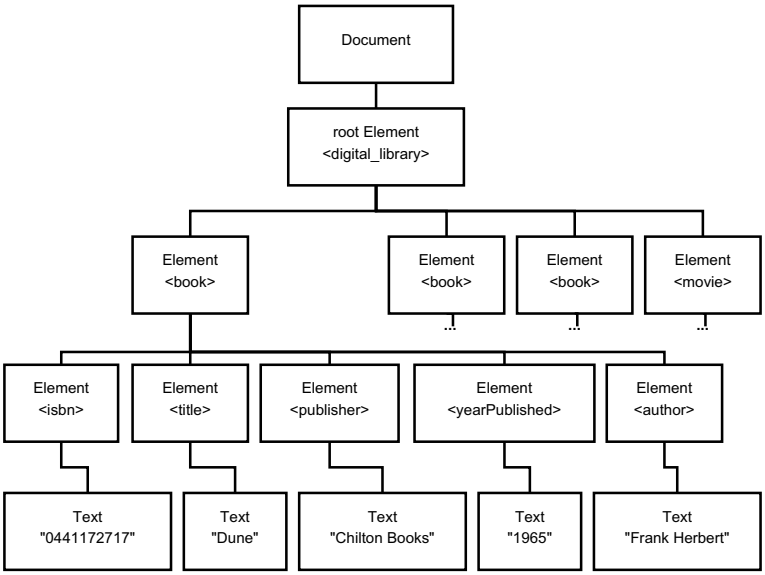


Fig. 31.1 DOM structure for XML Listing 31.1

- A World Wide Web Consortium (W3C) specification supported by all of the major web browsers.

Most of the popular programming languages, including Java and Python, provide an XML library that implements the W3C specification and associated API. The logical structure of a DOM is a tree of nodes representing the hierarchy of tags as expressed in a text-based XML file.

Figure 31.1 shows the DOM structure for the sample XML file in Listing 31.1. The hierarchy of nodes shown in this figure illustrates how an XML structure is translated into a DOM tree structure. At the top of the DOM is a Document object, which contains the entire hierarchy of nodes. Each of the XML tags is translated into an Element node, while the data values found in the XML file become Text nodes. The figure shows the entire tree structure for the first Book instance in the XML file. The other Book instances and the one Movie instance will have a similar structure.

The list below identifies the four most common types of objects found in a DOM.

Document: A container for the entire DOM tree structure and *conceptually* represents the root of the tree. A document will always contain one root Element, which represents the actual root of the tree.

Element: A start tag is translated into an Element node.

Node: A generic node type.

Text: The data associated with a start tag (i.e., Element node) is stored in a Text node.

The sample XML in Listing 31.1 uses indentation and whitespace characters to make it easy for people to read the XML text file. Listing 31.2 contains the same tags and data values as the first example but does not contain any newline or tab characters. Essentially, this text file contains one line of text by eliminating all of the whitespace characters which made the Sample 1 listing easier to read. (Important: for purposes of displaying the sample 2 XML file, newline characters were added to allow the content to be completely displayed inside the listing box.)

Listing 31.2 Digital Library XML File Sample 2

```
<digital_library><book><isbn>0441172717</isbn><title>Dune</title>
<publisher>Chilton Books</publisher><yearPublished>1965
</yearPublished><author>Frank Herbert</author></book><book><isbn>
0151008116</isbn><title>Life of Pi</title><publisher>Harcourt
</publisher><yearPublished>2001</yearPublished><author>Yann Martel
</author></book><book><isbn>0137135599</isbn><title>Blown to Bits
</title><publisher>Addison Wesley</publisher><yearPublished>2008
</yearPublished><author>Hal Abelson</author><author>Ken Ledeen
</author><author>Harry Lewis</author></book><movie><title>The
Shawshank Redemption</title><yearPublished>1994</yearPublished>
<director>Frank Darabont</director></movie></digital_library>
```

31.2.4 Java Examples

This section will introduce the use of Java to parse an XML file, to create a DOM, display the contents of a DOM, update a DOM, and to save a DOM back to an XML file.

31.2.4.1 Parse an XML File

The javax.xml.parsers package contains the classes and methods to parse an XML file to create a DOM. In Listing 31.3, the fileName parameter is the name of the XML file, including the file extension. When the DocumentBuilder parse method is successful, a Document object is returned. Any exception will cause null to be returned by the createDOM method.

Listing 31.3 Java createDom method

```
private Document createDOM(String filename)
{
    Document dom;
    DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance();
    try
    {
        DocumentBuilder db = dbf.newDocumentBuilder();
        dom = db.parse(filename);
    }
    catch (Exception ex)
```

```
    {
        System.out.println(ex);
        dom = null;
    }
    return dom;
}
```

31.2.4.2 Display a DOM

Since a DOM is a tree structure, we can display the entire DOM by moving through the tree structure using a recursive traversal algorithm. Our traversal will display information for the current node (wherever this is within the tree). It then recursively traverses the sub-tree for each child of the current node, moving through the child nodes from left-to-right.

Listing 31.4 shows the displayDOM and displayDOMhelper methods. The displayDOM method uses the Document object to obtain the root node for the DOM. It then calls the displayDOMhelper method, which implements the recursive traversal, displaying each node as it is reached. The indentLevel parameter for the recursive function keeps track of how much to indent each node as it's displayed to visually show the tree structure.

Listing 31.4 Java displayDom methods

```
private void displayDOM(Document dom)
{
    System.out.println("XML DOM node structure.\n" +
        "Each indentation is another level in the tree.");
    Element docElement = dom.getDocumentElement();
    displayDOMhelper(docElement, 0);
}

private void displayDOMhelper(Node element, int indentLevel)
{
    //display information for this element
    displayElement(element, indentLevel);

    //recursively display info for each child of this element
    if (element.hasChildNodes())
        displayDOMhelper(element.getFirstChild(), indentLevel + 1);

    //recursively display info for each sibling of this element
    Node nextSibling = element.getNextSibling();
    if (nextSibling != null)
        displayDOMhelper(nextSibling, indentLevel);
}
```

Listing 31.5 shows the display of the first Book instance from the first sample XML file shown in Listing 31.1. One of the interesting aspects associated with this DOM is it contains *extra* text nodes containing no data value. These are shown in Listing 31.5 as "type[Text],name[#text],value=[]". These nodes exist because of the whitespace characters (e.g., newline and tab characters) in the sample XML file

shown in Listing 31.1. The XML parser in Java treats white space characters as data, generating a Text node each time it finds whitespace characters.

Listing 31.5 Display of XML File Sample 1 using Java

```
Enter a file name (with extension): DigitalLibrary_1.xml
XML DOM node structure.
Each indentation is another level in the tree.
Displaying values for node type, name and value inside [].
type[Element],name[digital_library],value[]
  type[Text],name[#text],value[]
  type[Element],name[book],value[]
    type[Text],name[#text],value[]
    type[Element],name[isbn],value[]
      type[Text],name[#text],value[0441172717]
    type[Text],name[#text],value[]
    type[Element],name[title],value[]
      type[Text],name[#text],value[Dune]
    type[Text],name[#text],value[]
    type[Element],name[publisher],value[]
      type[Text],name[#text],value[Chilton Books]
    type[Text],name[#text],value[]
    type[Element],name[yearPublished],value[]
      type[Text],name[#text],value[1965]
    type[Text],name[#text],value[]
    type[Element],name[author],value[]
      type[Text],name[#text],value[Frank Herbert]
    type[Text],name[#text],value[]
  type[Text],name[#text],value[]
```

Listing 31.6 shows the display of the first Book instance from the second sample XML file shown in Listing 31.2. Since this XML file contains no whitespace characters, the display of the DOM shows no *extra* Text nodes.

Listing 31.6 Display of XML File Sample 2 using Java

```
Enter a file name (with extension): DigitalLibrary_2.xml
XML DOM node structure.
Each indentation is another level in the tree.
Displaying values for node type, name and value inside [].
type[Element],name[digital_library],value[]
  type[Element],name[book],value[]
    type[Element],name[isbn],value[]
      type[Text],name[#text],value[0441172717]
    type[Element],name[title],value[]
      type[Text],name[#text],value[Dune]
    type[Element],name[publisher],value[]
      type[Text],name[#text],value[Chilton Books]
    type[Element],name[yearPublished],value[]
      type[Text],name[#text],value[1965]
    type[Element],name[author],value[]
      type[Text],name[#text],value[Frank Herbert]
```

The entire Java program to display an XML file is found in displayDOM.java.

31.2.4.3 Update a DOM

A DOM tree structure can be updated by adding nodes at specific locations. The sample code in Listing 31.7 shows two methods. The changeDOM method calls getAuthorName to request the user to enter a new author name, calls addAuthorToFirstBook to add the entered name as a new author for the first book instance, and then saves the updated DOM back to an XML text file. We'll discuss saving the DOM to an XML file in the next section.

The addAuthorToFirstBook method gets a list of all book nodes in the DOM, saves the location of the first book node, creates a new Text node containing the user-supplied author name, creates a new Element node for an author tag, makes the new Text node a child of the new Element node, and then makes the new Element node a child of the first book node. Once this function has completed, the DOM structure has been modified by adding two additional nodes to the tree structure.

Listing 31.7 Java changeDom methods

```
private void changeDOM()
{
    String authorName = getAuthorName();
    addAuthorToFirstBook(authorName);
    saveDOMtoXML();
}

private void addAuthorToFirstBook(String authorName)
{
    //Get location of the first book node.
    NodeList nodes = dom.getElementsByTagName("book");
    if (nodes != null && nodes.getLength() > 0)
    {
        //The XML file has at least one book instance.
        Element firstBookNode = (Element)nodes.item(0);
        //Create text node to store the new author name.
        Text newText = dom.createTextNode(authorName);
        //Create element node for the new author.
        Element newElement = dom.createElement("author");
        //Add text node as child of the author element
        newElement.appendChild(newText);
        //Add author element to first book instance
        firstBookNode.appendChild(newElement);
    }
}
```

31.2.4.4 Save DOM to an XML File

Once a DOM structure has been updated, the updated DOM must be written to an XML file to persistently store the updated DOM. The code in Listing 31.8 shows the method to save a DOM to an XML file. This method creates an *output* file name, creates a Transformer object, creates a DOMSource object, creates a StreamResult object, and then uses the Transformer object to transform the DOMSource object (i.e., the DOM) into a StreamResult object (i.e., an output text file).

Listing 31.8 Java saveDOMtoXML method

```
private void saveDOMtoXML()
{
    try
    {
        String outputFileName =
            xmlFileName.substring(0,xmlFileName.length()-4) +
            "_OUTPUT" + xmlFileName.substring(xmlFileName.length()-4);
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();
        DOMSource source = new DOMSource(dom);
        StreamResult result = new StreamResult(
            new File(outputFileName));
        transformer.transform(source, result);
        System.out.println("Updated DOM written to " +
            outputFileName);
    }
    catch (Exception ex)
    {
        System.out.println(ex);
    }
}
```

31.2.5 Python Examples

This section will introduce the use of Python to parse an XML file, to create a DOM, display the contents of a DOM, update a DOM, and to save a DOM back to an XML file.

31.2.5.1 Parse an XML File

The xml.dom.minidom module contains the functions to parse an XML file to create a DOM. In Listing 31.9, the xmlFileName parameter is the name of the XML file, including the file extension. When the xml.dom.minicom parse function is successful, a Document object is returned. Any exception will cause None to be returned by the createDOM method.

Listing 31.9 Python createDom function

```
def createDOM(xmlFileName):
    try:
        dom = xml.dom.minidom.parse(xmlFileName)
    except Exception as ex:
        print(ex)
        dom = None
    return dom
```

31.2.5.2 Display a DOM

Since a DOM is a tree structure, we can display the entire DOM by moving through the tree structure using a recursive traversal algorithm. Our traversal will display information for the current node (wherever this is within the tree). It then recursively traverses the sub-tree for each child of the current node, moving through the child nodes from left-to-right.

Listing 31.10 shows the displayDOM and displayDOMhelper functions. The displayDOM function used the Document object to obtain the root node for the DOM. It then calls the displayDOMhelper function, which implements the recursive traversal, displaying each node as it is reached. The indentLevel parameter for the recursive function keeps track of how much to indent each node as it's displayed to visually show the tree structure.

Listing 31.10 Python displayDom functions

```
def displayDOM(dom):
    print("XML DOM node structure.", \
        "Each indentation another level in tree.")
    docElement = dom.documentElement
    displayDOMhelper(docElement, 0)

def displayDOMhelper(element, indentLevel):
    #display information for this element
    displayElement(element, indentLevel);

    #display information for each child of this element
    if element.hasChildNodes():
        displayDOMhelper(element.firstChild, indentLevel + 1)

    #display information for each sibling of this element
    nextSibling = element.nextSibling
    if (nextSibling != None):
        displayDOMhelper(nextSibling, indentLevel)
```

Listing 31.11 shows the display of the first Book instance from the first sample XML file shown in Listing 31.1. One of the interesting aspects associated with this DOM is it contains *extra* text nodes containing no data value. These are shown in Listing 31.11 as "type<Text>,name<#text>,value<>". These nodes exist because of the whitespace characters (e.g., newline and tab characters) in the sample XML file shown in Listing 31.1. The XML parser in Python treats whitespace characters as data, generating a Text node each time it finds whitespace characters.

Listing 31.11 Display of XML File Sample 1 using Python

```
Enter a file name (with extension): DigitalLibrary_1.xml
XML DOM node structure. Each indentation another level in tree.
type<Element>,name<digital_library>,value<>
  type<Text>,name<#text>,value<>
  type<Element>,name<book>,value<>
    type<Text>,name<#text>,value<>
    type<Element>,name<isbn>,value<>
```

```
      type<Text>,name<#text>,value<0441172717>
    type<Text>,name<#text>,value<>
    type<Element>,name<title>,value<>
      type<Text>,name<#text>,value<Dune>
    type<Text>,name<#text>,value<>
    type<Element>,name<publisher>,value<>
      type<Text>,name<#text>,value<Chilton Books>
    type<Text>,name<#text>,value<>
    type<Element>,name<yearPublished>,value<>
      type<Text>,name<#text>,value<1965>
    type<Text>,name<#text>,value<>
    type<Element>,name<author>,value<>
      type<Text>,name<#text>,value<Frank Herbert>
    type<Text>,name<#text>,value<>
  type<Text>,name<#text>,value<>
```

Listing 31.12 shows the display of the first Book instance from the second sample XML file shown in Listing 31.2. Since this XML file contains no whitespace characters, the display of the DOM shows no *extra* Text nodes.

Listing 31.12 Display of XML File Sample 2 using Python

```
Enter a file name (with extension): DigitalLibrary_1.xml
XML DOM node structure. Each indentation another level in tree.
type<Element>,name<digital_library>,value<>
  type<Element>,name<book>,value<>
    type<Element>,name<isbn>,value<>
      type<Text>,name<#text>,value<0441172717>
    type<Element>,name<title>,value<>
      type<Text>,name<#text>,value<Dune>
    type<Element>,name<publisher>,value<>
      type<Text>,name<#text>,value<Chilton Books>
    type<Element>,name<yearPublished>,value<>
      type<Text>,name<#text>,value<1965>
    type<Element>,name<author>,value<>
      type<Text>,name<#text>,value<Frank Herbert>
```

The entire Python program to display an XML file is found in displayDOM.py.

31.2.5.3 Update a DOM

A DOM tree structure can be updated by adding nodes at specific locations. The sample code in Listing 31.13 shows two functions. The changeDOM function calls getAuthorName to request the user to enter a new author name, calls addAuthorToFirstBook to add the entered name as a new author for the first book instance, and then saves the updated DOM back to an XML text file. We'll discuss saving the DOM to an XML file in the next section.

The addAuthorToFirstBook function gets a list of all book nodes in the DOM, saves the location of the first book node, creates a new Text node containing the user-supplied author name, creates a new Element node for an author tag, makes the

new Text node a child of the new Element node, and then makes the new Element node a child of the first book node. Once this function has completed, the DOM structure has been modified by adding two additional nodes to the tree structure.

Listing 31.13 Python changeDom functions

```
def changeDOM():
    authorName = getAuthorName()
    addAuthorToFirstBook(authorName)
    saveDOMtoXML()

def addAuthorToFirstBook(authorName):
    global dom
    #Get location of the first book node.
    nodes = dom.getElementsByTagName("book")
    if nodes != None and nodes.length > 0:
        #The XML file has at least one book instance.
        firstBookNode = nodes.item(0)
        #Create text node to store the new author name.
        newText = dom.createTextNode(authorName)
        #Create element node for the new author.
        newElement = dom.createElement("author")
        #Add text node as child of the author element
        newElement.appendChild(newText)
        #Add author element to first book instance
        firstBookNode.appendChild(newElement)
```

31.2.5.4 Save DOM to an XML File

Once a DOM structure has been updated, the updated DOM must be written to an XML file to persistently store the updated DOM. The code in Listing 31.14 shows the function that will save a DOM to an XML file. This function creates an *output* file name, creates an output file handle, and then uses the writexml method of the Document class to write the contents of the DOM to the output file handle (i.e., the output text file).

Listing 31.14 Python saveDOMtoXML function

```
def saveDOMtoXML():
    global xmlFileName
    global dom
    outputFileName = xmlFileName[0:len(xmlFileName)-4] +
            "_OUTPUT" + xmlFileName[len(xmlFileName)-4:]
    fileHandle = open(outputFileName,"wt")
    dom.writexml(fileHandle)
    fileHandle.close()
    print("Updated DOM written to", outputFileName)
```

Table 31.2 Rdb table example

StudentID	First	Middle	Last	Street1	City	State	Zip	Phone
12233345	Sue	B.	Doe	123 Doe St	Rome	NY	13440	3155559999
11933367	Joe		Doe	951 K St	Utica	NY	13502	3155551111

31.3 Relational Database Concepts

This section introduces Relational Database (Rdb) as a way to store data persistently.

31.3.1 What is a Relational Database?

A relational database (Rdb) is one of the most widely used persistent data storage formats currently in use. Conceptually, a Rdb is represented as a collection of tables, where each table is a physical manifestation of a logical data entity found on a LDM. A table is simply a two-dimensional matrix of rows and columns. Each row represents one record or data instance while each column represents a logical data attribute or a data field. For example, a table storing information about a student might have columns for student id, name (first, middle, last), address (street, city, state, zip), and phone number. Each of these columns represents a specific piece of information about a student, while the entire set of columns represents a collection of logically related information about a student. This collection of information would exist within one row of a student table and represents a student instance or student record. Table 31.2 shows two student instances (i..e., records). Each instance has nine distinct data values (i.e., fields, attributes).

The concept of a relational table as a bunch of rows and columns is simple to understand. We frequently see information presented to us in this manner. For example, voting results for an election, nutrition information for packaged food, and statistics for a sporting event. Another benefit of a relational database is that it is based on a few simple mathematical principles. Specifically, relational databases are based on relational algebra, which is a combination of predicate logic and set theory.

31.3.1.1 Predicate Logic

Predicate logic uses a few mathematical symbols to express quantifiers, logical operators, and variables. The quantifiers, logical operators, and variables are then used to define predicates.[1] The most common quantifiers are $\exists$ (there exists) and $\forall$ (for all). The most common logical operators are $\wedge$ (conjunction i.e, and), $\vee$ (disjunction,

[1] A simple definition of a predicate is a statement that may be true or false depending on the logic expressed and the value of the variables used.

i.e., or), $\Rightarrow$ (implication), and $\neg$ (negation i.e., not). Variables are usually denoted using lowercase letters. A few sample predicates are listed below.

- $\exists p \mid \text{knows}(p, \text{Java})$
  There exists a programmer that knows Java.
- $\forall p, \text{programmer}(p) \Rightarrow \text{knows}(p, language)$
  For all persons, if person is a programmer, then this person knows a programming language.
- $\forall p, \text{programmer}(p) \wedge (\text{knows}(p, \text{Java}) \vee \text{knows}(p, \text{Python})) \Rightarrow \text{benefit}(p, \text{learn (book)})$
  For all persons, if person is a programmer that knows Java or Python, then this person will benefit from reading this book.

31.3.1.2 Set Theory

Set theory uses a few mathematical symbols to express sets and set operators. A set is a collection of unique objects (or values). The contents of a set is often denoted using a list delimited by braces, for example, {1, 2, 3}. An empty set is denoted with the symbol $\emptyset$. The common set operators are $\cup$ (union), $\cap$ (intersection), $-$ (difference), $\times$ (Cartesian product), and power(S) (power set). A few examples are listed below.

- $\{1, 2, 3\} \cup \{1, 2, 4\}$ is $\{1, 2, 3, 4\}$
  If A and B are sets then A $\cup$ B is the set of all elements that are in A or B.
- $\{1, 2, 3\} \cap \{1, 2, 4\}$ is $\{1, 2\}$
  If A and B are sets then A $\cap$ B is the set of all elements that are in both A and B.
- $\{1, 2, 3\} - \{1, 2, 4\}$ is $\{3\}$
  If A and B are sets then A $-$ B is the set of all elements in A that are not in B.
- $\{1, 2, 3\} \times \{1, 2, 4\}$ is $\{(1, 1), (1, 2), (1, 4), (2, 1), (2, 2), (2, 4), (3, 1), (3, 2), (3, 4)\}$
  If A and B are sets then A $\times$ B is the set of all ordered pairs (a, b) where a is an element of A and b is an element of B.
- power($\{1, 2, 3\}$) is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
  If A is a set then power(A) is the set of all subsets of A.

Since we are using set theory to help explain how relational databases work, we use the term *relation* to mean a *set*.

31.3.2 Relational Database Model (aka: Physical Data Model)

Designing a relational database starts with creating an ERD (entity relationship diagram) and LDM (logical data model) to document the logical relationships between the data elements. This section talks about translating a LDM into a relational database model (i.e., a physical data model).

The entities and attributes in a LDM will be translated into tables and columns (i.e., fields) in a relational database model. The primary key attributes for each entity

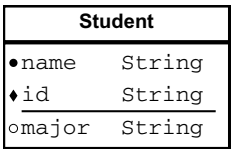


Fig. 31.2 Relational database model: one table

will become the primary key fields in the table. The fields defined in each table will include a data type. Relationships in a LDM describe the cardinality of the relationship, which would be one of the following:

1:1 A one-to-one relationship between two data entities. Each instance of entity A is related to one instance of entity B and vice versa. For example, a relationship between person and student is 1:1. A person may be a student and a student is a person.

1:M A one-to-many relationship between two data entities. Each instance of entity A is related to many instances of entity B but each instance of entity B is related to one instance of entity A. For example, a relationship between student and term gpa is 1:M. A student may have many term-gpas but a term-gpa is for one student.

M:N A many-to-many relationship between two data entities. Each instance of entity A is related to many instances of entity B and each instance of entity B is related to many instances of entity A. For example, a relationship between student and course is M:N. A student may take many courses and a course may be taken by many students.

A simple relational database model is shown in Fig. 31.2. This shows a database with a single table called Student with three fields. All three fields contain character string data and the id field is the primary key. The small letter "oh" in front of the major field indicates this field is optional; a major value may not be provided (i.e., this field may be empty or what relational databases call a null value). The other two fields must contain a non-null value.

A more interesting example is to translate the logical data model for the digital library described in Chap. 30 into a relational database model. Figure 31.3 shows the Version 3 logical data model for the digital library. Figure 31.4 is the corresponding relational database model. Note the following about this translation from a LDM to a PDM:

- Each logical data entity has been translated to a relational table.
- Each logical data attribute has been translated into a field. Each field includes its data type.
- Each primary key attribute has been translated into a primary key field. For example, see isbn in Book and title and yearPublished in Movie.

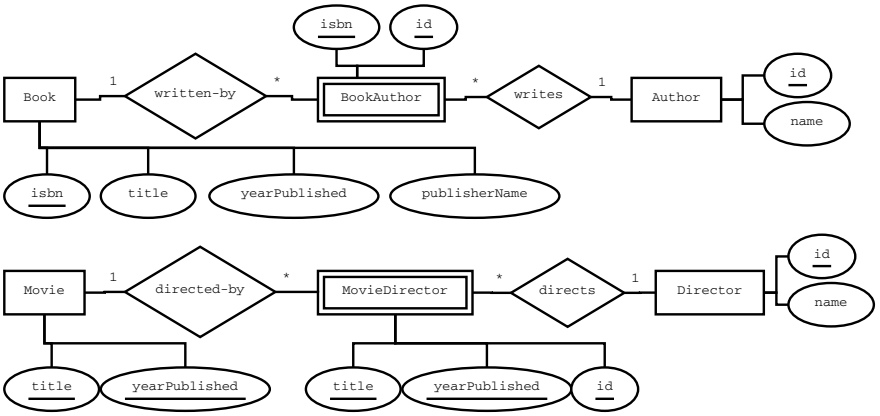


Fig. 31.3 Digital library LDM Version 3 (from Chap. 30)

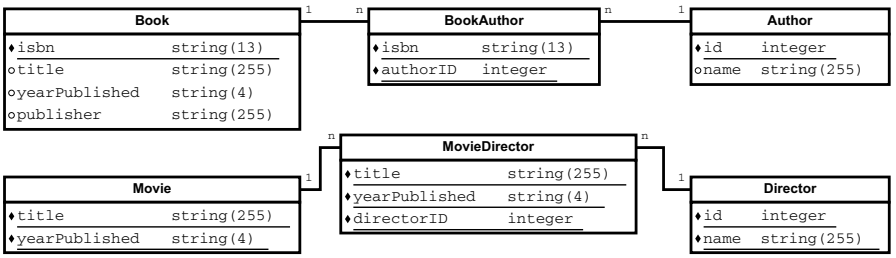


Fig. 31.4 Digital library relational database model Version 3

31.3.2.1 Translating LDM to PDM

There are two issues when translating a logical data model into a relational database model. The logical data model in Fig. 31.5 will be used to illustrate these two issues, while the relational database model in Fig. 31.6 shows the results of resolving the two issues described below.

1. How does a weak entity in a logical data model get translated into a relational database model?
   The LDM in Fig. 31.5 has a weak entity called TermGPA with a weak key of the term. When this weak entity gets translated into a table in the relational database model, the primary key from the related strong entity must be included in the table. Figure 31.6 shows table TermGPA with three fields while the corresponding entity only has two attributes. The studentID will have the same value as the corresponding id field from the related Student table. This is combined with the term field to form the primary key for the TermGPA. That is, combining the values from studentID and term will uniquely identify each row in the TermGPA table. The studentID field in the TermGPA table is also called a *foreign key* field. This term applies to any field in a table where this field is a primary key in another

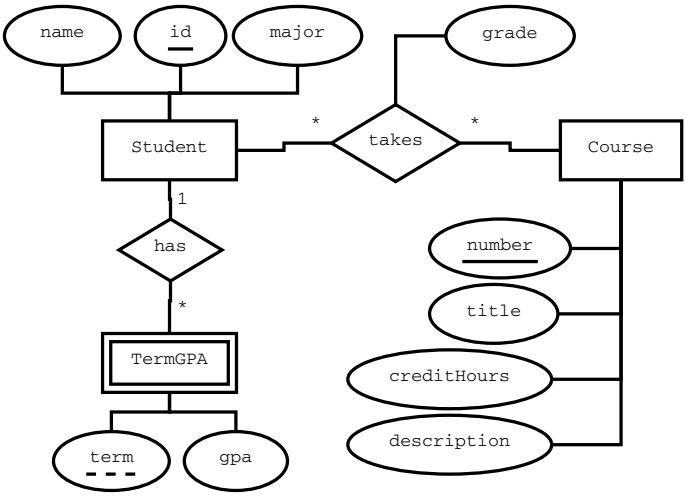


Fig. 31.5 Student-GPA-Course LDM

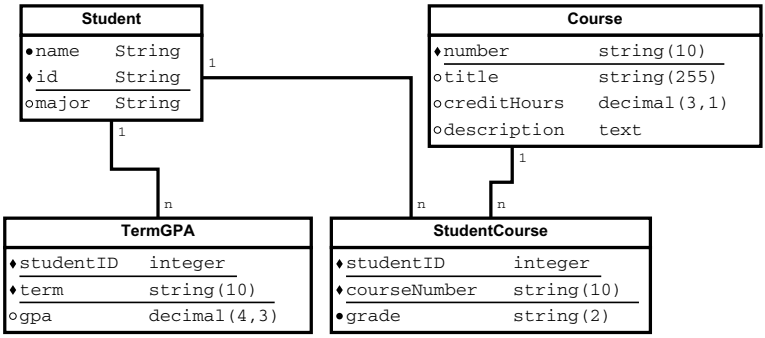


Fig. 31.6 Student-GPA-Course PDM

table. Why do we care about foreign keys? Remember the normalization process described in Chap. 30? One of the stated goals of the normalization process is to eliminate redundant data in our logical data models. It looks like the concept of a foreign key, as applied to relational database models, is now adding redundant data! While technically this is correct, relational databases have ways to automatically avoid issues with redundant data. Specifically, foreign key constraints within a relational database product (e.g., MySQL, SQL Server, Oracle) will automatically enforce update and delete scenarios. For example, if I changed an id value in a Student instance, this change would automatically be applied to all studentID foreign keys in the TermGPA table that match the original value. Similarly, if I decide to delete a Student instance from the table, all instances in the TermGPA table which match the id value being deleted are automatically deleted.

2. How does a many-to-many relationship in a logical data model get translated into a relational database model?
   The LDM in Fig. 31.5 has a many-to-many relationship: A Student takes many Courses; A Course is taken by many Students. Most of the time, translating this type of relationship in a relational database model results in replacing the many-to-many relationship with a table, which produces two one-to-many relationships in the relational database model. Figure 31.6 shows table StudentCourse, which does not exist as an entity in Fig. 31.5. Note the following regarding this translation of a logical many-to-many relationship to a relational database model.

   - The Student and Course entities are translated into tables.
   - The "takes" relationship is translated into a table called StudentCourse. It is fairly common practice to name this table by combining the entity names from the LDM.
   - The many-to-many logical relationship is now two distinct one-to-many physical relationships: a Student has many StudentCourse instances and a Course has many StudentCourse instances.
   - The StudentCourse table has as its primary key the primary key fields from the two tables it is related to. Thus, the primary key fields in the StudentCourse table are also foreign key fields.
   - The StudentCourse table contains a grade field which corresponds to the LDM showing grade as an attribute of the many-to-many "takes" relationship.

   The result of this translation is the following: (1) A Student can take a Course multiple times by taking the same course in different terms. Each time the Student shall earn a distinct grade; and (2) A Course offered in a term may be taken by multiple Students. Each Student shall earn a distinct grade.

31.3.3 Structured Query Language (SQL)

The structured query language (SQL—pronounced sequel) is a declarative programming language[2] supported by all relational database products.[3] Learning SQL allows you to do two things: create and modify relational database structures; and create, read, update, and delete data in a relational database. An excellent tutorial on SQL is found on the www.w3schools.com web site [3].

[2] A declarative programming language contains statements to indicate what needs to be done. However, the language does not allow the programmer to express how to achieve the result. In the case of SQL, the database software, often called the database engine, determines how to accomplish what needs to be done.

[3] SQL is both an ANSI and ISO standard.

31.3.3.1 Data Definition Language (DDL)

The data definition language (DDL) is the part of SQL used to create and modify relational database structures. DDL capabilities include the following list, which will be described below. Other DDL statements exist to create and modify other aspects of a relational database. These additional DDL features are not discussed in this chapter.

- Create and modify a database.
- Create and modify a table in a database.
- Create and modify an index in a database.

The script shown in Listing 31.15 illustrates some of the common DDL statements. This script is written for MySQL version 5.7 and would create the Digital Library relational database shown in Fig. 31.4. In The MySQL 5.7 Reference Manual [4], see *Chap.* 11 *Data Types* and *Chap.* 13 *SQL Statement Syntax* for details on these DDL statements.

Listing 31.15 DDL Digital Library

```
drop database if exists DigitalLibrary;
create database DigitalLibrary;
use DigitalLibrary;

create table Book
    (isbn          CHAR(13) PRIMARY KEY,
    title          VARCHAR(255),
    publisher      VARCHAR(255),
    yearPublished  CHAR(4));

create table Author
    (id      INTEGER AUTO_INCREMENT PRIMARY KEY,
    name     VARCHAR(255));

create table BookAuthor
    (isbn        CHAR(13),
    authorID     INTEGER,
    PRIMARY KEY (isbn, authorID),
    FOREIGN KEY (isbn)
        REFERENCES Book(isbn)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (authorID)
        REFERENCES Author(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE);

create table Movie
    (title         VARCHAR(255),
    yearPublished  CHAR(4),
    PRIMARY KEY (title, yearPublished));
```

```
create table Director
    (id       INTEGER AUTO_INCREMENT PRIMARY KEY,
    name      VARCHAR(255));

create table MovieDirector
    (title          VARCHAR(255),
    yearPublished    CHAR(4),
    directorID       INTEGER,
    PRIMARY KEY (title, yearPublished, directorID),
    FOREIGN KEY (title, yearPublished)
        REFERENCES Movie(title, yearPublished)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (directorID)
        REFERENCES Director(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE);

/* A few sample indexes */
create index BookIndex_1
    ON Book(title);
create index BookIndex_2
    ON Book(publisher);
create index BookIndex_3
    ON Book(yearPublished);
```

Note the yearPublished fields are defined as CHAR(4) instead of INTEGER. While a year value is an integer, it is unlikely we would need to do arithmetic on a yearPublished value. Thus, we use a string data type for this field.

A few comments regarding the SQL script in Listing 31.15.

- The first two statements delete and then create a relational database named DigitalLibrary.
- The third statement tells the database server that the remaining SQL statements in the script will be using the DigitalLibrary database.
- Most relational databases have both a CHAR and VARCHAR data type. A CHAR data type is a fixed length string. When the data value is less than the fixed length, spaces are added to reach the defined length of the field. In contrast, a VARCHAR data type will not pad the value with spaces.
- The BookAuthor and MovieDirector CREATE TABLE statements show how a foreign key is defined. The constraints ON DELETE and ON UPDATE indicate what should happen when a primary key value is deleted or modified in the referenced table.
- Each PRIMARY KEY automatically generates a database index, allowing good performance when accessing the data using the primary key.

- The sample CREATE INDEX statements show how to create additional indexes for a table. Additional indexes are created for a table when its data is accessed using something other than the primary key fields.

31.3.3.2 Data Manipulation Language (DML)

The data manipulation language (DML) portion of SQL is used to implement CRUD transactions that operate on the data stored in the database. CRUD stands for:

- Create: use SQL INSERT statements to add data to a database.
- Read: use SQL SELECT statements to query a database.
- Update: use SQL UPDATE statements to modify data already stored in a database.
- Delete: use SQL DELETE statements to remove data from a database.

The scripts in this section illustrate the insert, select, update, and delete DML statements. These scripts are written for MySQL version 5.7 and use the Digital Library relational database shown in Fig. 31.4. In the MySQL 5.7 Reference Manual [4], see *Chap.* 13 *SQL Statement Syntax* for details on these DML statements. For readability purposes only, the SQL keywords are in all uppercase letters. With the exception of data values, SQL statements may be written in all uppercase, all lowercase, or mixed case; data values are typically case-sensitive.

Listing 31.16 shows insert statements to add data to the database tables. Assuming the DDL statements in Listing 31.15 are executed just before using these insert statements, the id values inserted into the Author table are 1, 2, 3, and 4. If we were to run the insert statements a second time, the id values would be 5 through 8. When you want to reset the AUTO_INCREMENT so the first insert gets an id value of 1, the best approach is to use DROP TABLE followed by CREATE TABLE. In the case of the Digital Library tables, if you want to reset the Author table's AUTO_INCREMENT, you would need to drop both the Author and BookAuthor tables, since the BookAuthor table has a FOREIGN KEY CONSTRAINT on the Author id field.

When large amounts of data need to be added to a database, using a feature that takes data from a file and loads it into a database table is a more efficient method. In the case of MySQL, you may use the *mysqlimport* client program[4] or the *LOAD DATA* SQL statement.[5]

Listing 31.16 DML Example 1: Create Data using INSERT

```
/* Insert a book with one author */
INSERT INTO Book (isbn, title, publisher, yearPublished)
VALUES ("0441172717","Dune","Chilton Books","1965");

INSERT INTO Author (name) VALUES ("Frank Herbert");
```

[4] See Sect. 4.5 MySQL Client Programs in [4].
[5] See Sect. 13.2 Data Manipulation Statements in [4].

```
INSERT INTO BookAuthor (isbn, authorID) VALUES ("0441172717",1);

/* Insert a book with same author */
INSERT INTO Book (isbn, title, publisher, yearPublished)
VALUES ("0441172695","Dune Messiah","Putnam Publishing","1969");

INSERT INTO BookAuthor (isbn, authorID) VALUES ("0441172695",1);

/* Insert a book with many authors */
INSERT INTO Book (isbn, title, publisher, yearPublished)
VALUES ("0137135599","Blown to Bits","Addison Wesley","2008");

INSERT INTO Author (name) VALUES ("Hal Abelson");
INSERT INTO Author (name) VALUES ("Ken Ledeen");
INSERT INTO Author (name) VALUES ("Harry Lewis");

INSERT INTO BookAuthor (isbn, authorID) VALUES ("0137135599",2);
INSERT INTO BookAuthor (isbn, authorID) VALUES ("0137135599",3);
INSERT INTO BookAuthor (isbn, authorID) VALUES ("0137135599",4);

/* Insert a movie */
INSERT INTO Movie (title, yearPublished)
VALUES ("The Shawshank Redemption","1994");

INSERT INTO Director (name) VALUES ("Frank Darabont");

INSERT INTO MovieDirector (title, yearPublished, directorID)
VALUES ("The Shawshank Redemption","1994",1);
```

Based on the INSERT statements in Listing 31.16, the *SELECT* statement in Listing 31.17 returns the data shown in Table 31.3. The asterisk ("*") in the SELECT statement requests data for all columns in the table.

Listing 31.17 DML Example 2: Read Data using SELECT Example 1

```
/* Get all of the books in the database */
SELECT * FROM Book;
```

Based on the INSERT statements in Listing 31.16, the *SELECT* statement in Listing 31.18 returns the data shown in Table 31.4. The WHERE clause returns only those Author's whose name contains an uppercase "H". The wildcard character percent sign ("%") matches zero or more of any character. Using this as part of a LIKE operator returns any Author name that contains an uppercase "H".

Table 31.3 Rdb SELECT Example 1

isbn	title	yearPublished	publisher
"0441172717"	"Dune"	"Chilton Books"	"1965"
"0441172695"	"Dune Messiah"	"Putnam Publishing"	"1969"
"0137135599"	"Blown to Bits"	"Addison Wesley"	"2008"

Table 31.4 Rdb SELECT Example 2

Id	Name
1	"Frank Herbert"
2	"Hal Abelson"
4	"Harry Lewis"

Table 31.5 Rdb SELECT Example 3

title	yearPublished
"Dune"	"1965"
"Dune Messiah"	"1969"
"Blown to Bits"	"2008"
"Blown to Bits"	"2008"

Listing 31.18 DML Example 3: Read Data using SELECT Example 2

```
/* Get all of the authors that have "H" in their name */
SELECT id, name FROM Author
WHERE name LIKE "%H%";
```

Based on the INSERT statements in Listing 31.16, the *SELECT* statement in Listing 31.19 returns the data shown in Table 31.5. The WHERE clause returns title and yearPublished data only for those books with an Author name containing an uppercase "H". Specifically, the first clause (i.e., WHERE name LIKE "%H%") returns Author row instances where the name contains at least one uppercase "H". The second clause (i.e., AND Author.id = BookAuthor.authorID) returns BookAuthor row instances where the author id values match. Finally, the third clause (i.e., AND Book.isbn = BookAuthor.isbn) returns Book row instances where the isbn values match.

Listing 31.19 DML Example 4: Read Data using SELECT Example 3

```
/* Get all books whose author has "H" in their name */
SELECT title, yearPublished
FROM Book, BookAuthor, Author
WHERE name LIKE "%H%"
  AND Author.id = BookAuthor.authorID
  AND Book.isbn = BookAuthor.isbn;
```

Based on the INSERT statements in Listing 31.16, the *UPDATE* statement in Listing 31.20 changes the name of Author "Hal Abelson" to "Harold Abelson".

Listing 31.20 DML Example 5: Update Data using UPDATE

```
/* Change first name of author Abelson */
UPDATE Author SET name = "Harold Abelson" WHERE id = 2;
```

Based on the INSERT statements in Listing 31.16, the *DELETE* statement in Listing 31.21 removes the row in the Author table containing the name "Harry Lewis". Since a FOREIGN KEY constraint has been defined between Author.id and BookAuthor.authorID, this delete statement will also remove any rows in the

BookAuthor table containing Harry Lewis's author.id value. One important note: without a WHERE clause, a DELETE statement will remove all rows from a table.

Listing 31.21 DML Example 6: Delete Data using DELETE

```
/* Delete author Harry Lewis */
DELETE FROM Author WHERE id = 4;
```

31.3.4 Embedded SQL

SQL statements may be given to a database server for execution in two fundamental ways.

Using client programs: These client programs may be provided by the database server or by a third party. In either case, SQL statements are entered via a command-line or graphical user interface. The MySQL [4] database provides a number of client programs, some allowing DML statements and database configuration commands to be entered (e.g., mysqladmin) and some allowing DDL and DML statements to be entered (e.g., mysql).

Embedding SQL in code: Software developers can embed SQL statements into their code. The details of how this is done are dependent on the specific programming language and the type of database server being used.

Embedding SQL into code is conceptually similar to doing file input/output. The details associated with embedding SQL into Java or Python code are described in the next two sections.

31.3.5 Java Examples

This section will introduce the use of Java to connect to and use a relational database. We'll be using a MySQL database server in these examples.

31.3.5.1 Connect to a Rdb

In order to use a relational database in your code, first, you need to connect to a database server and a database on that server. For Java, we are using the MySQL Connector/J described in *Chap.* 23 *Connectors and API* in [4]. Once the Connector/J is installed, you need to add the jar file (mysql-connector-java-*version*.jar) to the CLASSPATH environment variable.

Listing 31.22 shows the Java code to connect to a MySQL database server on the local host and to the DigitalLibrary database on this local server. The user and password values were created specifically for this example. We give the DriverManager the name of the MySQL driver to use and then we get a connection to the database

on the local machine. The dbConnection variable in this code listing is an instance variable in the class containing this code.

Listing 31.22 Java Example: Connect to MySQL Database

```
try
{
    //Register driver name with the DriverManager.
    //Class.forName("com.mysql.jdbc.Driver").newInstance(); //old
    Class.forName("com.mysql.cj.jdbc.Driver").newInstance();//new
    //Connect to a database
    dbConnection = DriverManager.getConnection(
            "jdbc:mysql://localhost/DigitalLibrary?" +
            "user=dave&password=dave");
    userInput = new Scanner(System.in);
}
catch (SQLException ex)
{
    dbConnection = null;
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError: " + ex.getErrorCode());
}
catch (Exception ex)
{
    dbConnection = null;
    System.out.println(ex);
}
```

31.3.5.2 Query a Rdb

The example in Listings 31.23, 31.24 and 31.25 display Book instances that contain a user-supplied search string. Listing 31.23 shows two different SELECT statements. The first SELECT statement looks for Book instances where the publisher's name contains the user-supplied string value, while the second statement looks for Book instances where the title contains the user-supplied string value. The LIKE BINARY phrase ensures Book instances matching the case of the user-supplied string value are returned, while the question mark ("?") is a placeholder representing where the user-supplied data will be put within the SELECT statement.

Listing 31.23 Java Example: Query a MySQL Database Part 1

```
private final String SELECT_BOOKS_PUB =
     "SELECT isbn, title, publisher, yearPublished FROM Book " +
     "WHERE publisher LIKE BINARY ?;";
private final String SELECT_BOOKS_TITLE =
     "SELECT isbn, title, publisher, yearPublished FROM Book " +
     "WHERE title LIKE BINARY ?;";
```

Listing 31.24 shows the logic associated with the user indicating whether they want to list books matching a publisher search string or a title search string. The userData value represents the search value entered by the user.

Listing 31.24 Java Example: Query a MySQL Database Part 2

```
if (menuChoice.equals(MENU_PUB))
{
  System.out.println("The following books contain<" + userData +
      "> in publisher field:");
  displayBooks(SELECT_BOOKS_PUB, userData);
}
else if (menuChoice.equals(MENU_TITLE))
{
  System.out.println("The following books contain<" + userData +
      "> in title field:");
  displayBooks(SELECT_BOOKS_TITLE, userData);
}
```

Listing 31.25 shows the displayBooks method. A PreparedStatement object is created using the SELECT statement represented by the first formal parameter. The PreparedStatement object is then used to set the value of the placeholder (i.e., question mark) found in the SELECT statement. The statement is then executed, which produces a ResultSet. A ResultSet object contains all of the Book instances returned based on the just executed SELECT statement. The do-while will iterate as long as there are more Book instances in the ResultSet. The method ends by closing the ResultSet and PreparedStatement.

Listing 31.25 Java Example: Query a MySQL Database Part 3

```
private void displayBooks(String selectStmt, String userData)
{
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try
    {
        stmt = dbConnection.prepareStatement(selectStmt);
        stmt.setString(1, PERCENT + userData + PERCENT);
        rs = stmt.executeQuery();
        if (rs.next())
            do
            {
                System.out.println("  isbn:" + rs.getString(1) +
                        "\ttitle:" + rs.getString(2) +
                        "\tpublisher:" + rs.getString(3) +
                        "\tyear published:" + rs.getString(4));
            } while (rs.next());
        else
          System.out.println("  No books satisfy search criteria.");
    }
    catch (SQLException ex)
    {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
```

```
    }
    closeResultSet(rs);
    closeStatement(stmt);
}
```

31.3.5.3 Update a Rdb

The example in Listings 31.26 and 31.27 adds a new author to the first book instance in the database. The insertAuthor method adds the user-supplied name to the Author table. Similar to the SELECT statement example, a placeholder (i.e., question mark) is used to provide the user-supplied author name to the insert statement. The result from executing the insert statement should be an integer 1, indicating that 1 row (aka data instance) was added to the Author table. The insertAuthor method returns the Author id value representing the author's name just added to the table.

Listing 31.26 Java Example: Insert into a MySQL Database Part 1

```
private int insertAuthor(String authorName)
{
    final String INSERT_AUTHOR =
            "INSERT INTO Author (name) VALUES (?)";
    PreparedStatement stmt = null;
    int authorID = 0;
    try
    {
        stmt = dbConnection.prepareStatement(INSERT_AUTHOR);
        stmt.setString(1, authorName);
        int count = stmt.executeUpdate();
        if (count == 1)
        {
            authorID = getMaxAuthorID();
            System.out.println("Added author<" + authorName +
                    "> with id<" +
                    Integer.toString(authorID) + ">.");
        }
        else
          System.out.println("Logic error occurred while inserting" +
                    " new author");
    }
    catch (SQLException ex)
    {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
    closeStatement(stmt);
    return authorID;
}
```

The insertAuthorForBook method in Listing 31.27 uses the Author id value returned by the insertAuthor method along with the isbn value for the first book in the database. These two data values are used to add a row to the BookAuthor

table. Note the use of two placeholders in this INSERT statement. Again, the insert should result in one row being added to the BookAuthor table.

Listing 31.27 Java Example: Insert into a MySQL Database Part 2

```
private void insertAuthorForBook(int authorID, String isbn)
{
    final String INSERT_AUTHOR =
            "INSERT INTO BookAuthor (isbn, authorID) VALUES (?, ?)";
    PreparedStatement stmt = null;
    try
    {
        stmt = dbConnection.prepareStatement(INSERT_AUTHOR);
        stmt.setString(1, isbn);
        stmt.setInt(2, authorID);
        int count = stmt.executeUpdate();
        if (count == 1)
            System.out.println("Added author<" + authorID +
                    "> to ISBN<" + isbn + ">.");
        else
            System.out.println("Logic error occurred while inserting" +
                    " new author");
    }
    catch (SQLException ex)
    {
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
    closeStatement(stmt);
}
```

31.3.6 Python Examples

This section will introduce the use of Python to connect to and use a relational database. We'll be using a MySQL database server in these examples.

31.3.6.1 Connect to a Rdb

In order to use a relational database in your code, first, you need to connect to a database server and a database on that server. For Python, we are using the MySQL Connector/Python described in *Chap.* 23 *Connectors and API* in [4]. The Python connector code is put into the Lib\site-packages folder found within a folder whose name is associated with the Python version you are using (e.g., \Python33).

Listing 31.28 shows the Python code to connect to a MySQL database server on the local host and to the DigitalLibrary database on this local server. The user and password values were created specifically for this example.

Listing 31.28 Python Example: Connect to MySQL Database

```
import mysql.connector
from mysql.connector import Error

def createMySQLconnection():
    global dbConnection
    dbConnection = None
    try:
        dbConnection = mysql.connector.connect(host='localhost',
            database='DigitalLibrary',
            user='dave',
            password='dave',
            raise_on_warnings=True)
        if dbConnection.is_connected():
            print("Connected to DigitalLibrary database")
    except Error as err:
        print(err)
```

31.3.6.2 Query a Rdb

The example in Listings 31.29, 31.30, and 31.31display Book instances that contain a user-supplied search string. Listing 31.29 shows two different SELECT statements. The first SELECT statement looks for Book instances where the publisher's name contains the user-supplied string value, while the second statement looks for Book instances where the title contains the user-supplied string value. The LIKE BINARY phrase ensures Book instances matching the case of the user-supplied string value are returned, while the "%s" is a placeholder representing where the user-supplied data will be put within the SELECT statement.

Listing 31.29 Python Example: Query a MySQL Database Part 1

```
SELECT_BOOKS_PUB = "SELECT isbn, title, publisher, " +
        "yearPublished FROM Book WHERE publisher LIKE BINARY %s"
SELECT_BOOKS_TITLE = "SELECT isbn, title, publisher, " +
        "yearPublished FROM Book WHERE title LIKE BINARY %s"
```

Listing 31.30 shows the logic associated with the user indicating whether they want to list books matching a publisher search string or a title search string. The userData value represents the search value entered by the user.

Listing 31.30 Python Example: Query a MySQL Database Part 2

```
if menuChoice == MENU_PUB:
    print("The following books contain <", userData,
            "> in publisher field:", sep="")
    displaySomeBooks(SELECT_BOOKS_PUB, userData)
```

```
elif menuChoice == MENU_TITLE:
    print("The following books contain <", userData,
          "> in title field:", sep="")
    displaySomeBooks(SELECT_BOOKS_TITLE, userData)
```

Listing 31.31 shows the displaySomeBooks function. A cursor object is created and then executed. The cursor's execute method is given the SELECT statement provided via the first parameter and the value of the placeholder (i.e., "%s") found in the SELECT statement. The placeholder values must be given to the execute method in a tuple. Executing the SELECT statement produces an iterator used to move through the tuples returned from the query. The function ends by closing the cursor.

Listing 31.31 Python Example: Query a MySQL Database Part 3

```
def displaySomeBooks(selectStmt, userData):
    global dbConnection
    try:
      bookCursor = dbConnection.cursor()
      bookCursor.execute(selectStmt,
           (PERCENT + userData + PERCENT,))
      for (isbn, title, publisher, yearPublished) in bookCursor:
         print('  isbn:', isbn, '\ttitle:', title, '\tpublisher:',
             publisher, '\tyearPublished:', yearPublished)
      bookCursor.close()
    except Error as err:
      print(err)
```

31.3.6.3 Update a Rdb

The example in Listings 31.32 and 31.33 adds a new author to the first book instance in the database. The insertAuthor function adds the user-supplied name to the Author table. Similar to the SELECT statement example, a placeholder (i.e., "%s") is used to provide the user-supplied author name to the insert statement. The result from executing the insert statement should be a row count of 1, indicating that 1 row (aka data instance) was added to the Author table. The insertAuthor method returns the Author id value representing the author's name just added to the table.

Listing 31.32 Python Example: Insert into a MySQL Database Part 1

```
def insertAuthor(authorName):
  global dbConnection
  INSERT_AUTHOR = 'INSERT INTO Author (name) VALUES (%s)'
    authorID = None
    try:
      authorCursor = dbConnection.cursor()
      authorCursor.execute(INSERT_AUTHOR, (authorName,))
      if authorCursor.rowcount == 1:
        authorID = authorCursor.lastrowid
        print("Added author <", authorName, "> with id <", authorID,
              ">.", sep="")
      else:
```

```
        print("Logic error occurred while inserting new author")
      authorCursor.close()
    except Error as err:
      print(err)
    return authorID
```

The insertAuthorForBook function in Listing 31.33 uses the Author id value returned by the insertAuthor function along with the isbn value for the first book in the database. These two data values are used to add a row to the BookAuthor table. Note the use of two placeholders in this INSERT statement. Again, the insert should result in one row being added to the BookAuthor table.

Listing 31.33 Python Example: Insert into a MySQL Database Part 2

```
def insertAuthorForBook(authorID, isbn):
    global dbConnection
    INSERT_AUTHOR = 'INSERT INTO BookAuthor (isbn, authorID) ' +
            'VALUES (%s, %s)'
    try:
      authorCursor = dbConnection.cursor()
      authorCursor.execute(INSERT_AUTHOR, (isbn, authorID))
      if authorCursor.rowcount == 1:
        print("Added author <", authorID, "> to ISBN <",
            isbn, ">.", sep="")
      else:
        print("Logic error occurred while inserting new author")
      authorCursor.close()
    except Error as err:
        print(err)
```

31.4 Post-conditions

The following should have been learned when completing this chapter.

- You understand the following regarding eXtensible Markup Language (XML):
  - XML files represent data using a tree structure of tags. An XML tag will contain a data value and/or other tags, is given a name delimited with < and > (e..g, <lastname>), and typically has a matching end tag (e.g., </lastname>).
  - A DOM (document object model) represents an XML file as a tree structure of nodes. You navigate to any node in a DOM by starting at the root of the tree.
  - Both Java and Python provide support for using XML files and the DOM. First, an XML file is parsed to produce a DOM. The DOM contains Element nodes (one per tag) and Text nodes (one per data value).
- You understand the following regarding relational databases (Rdb):

- A Rdb is a collection of one or more tables used to store data. Each table has rows representing data instances (e.g., a student) and columns representing data values (e.g., student has name "Ahmad" and an id of 123456). Relational databases are based on predicate logic and set theory.
- SQL (structured query language) provides capabilities to create and manipulate databases. DDL (data definition language) statements are used to create a database, tables, indexes, and other types of database objects. DML (data manipulation language) statements are used to create, read, update, and delete data in tables.
- Both Java and Python provide support for using Rdb. First, you connect to a relational database server and a specific database. You then execute SQL statements to manipulate the data in the database.

Exercises

Discussion Questions

1. Are certain types of data relationships (i.e., one-to-one, one-to-many, many-to-many) easier to represent using XML? using Rdb?
2. When comparing XML and Rdb, what are the advantages of using XML to persistently store your data?
3. When comparing XML and Rdb, what are the advantages of using Rdb to persistently store your data?

References

1. Refsnes data: XML tutorial. Available via https://www.w3schools.com/xml/default.asp. Cited 17 Jul 2019
2. Refsnes data: XML DOM tutorial. Available via https://www.w3schools.com/xml/dom_intro.asp. Cited 17 Jul 2019
3. Refsnes data: SQL tutorial. Available via https://www.w3schools.com/sql/default.asp. Cited 19 Jul 2019
4. Oracle corporation: MySQL 5.7 reference manual. Available via https://dev.mysql.com/doc/refman/5.7/en/. Cited 19 Jul 2019



OOD Case Study: Persistent Storage

32

The objective of this chapter is to illustrate the use of two persistent storage technologies—eXtensible Markup Language (XML) and Relational Database (Rdb)—by updating the case study designs.

32.1 OOD: Preconditions

The following should be true prior to starting this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating a LDM to a physical data model needs to consider the limitations of the physical storage format.
- You understand the process of normalization and have applied normalization forms to a data design.
- You understand the following regarding eXtensible Markup Language (XML):
  - XML files represent data using a tree structure of tags. An XML tag will contain a data value and/or other tags, is given a name delimited with < and > (e..g, <lastname>), and typically has a matching end tag (e.g., </lastname>).
  - A DOM (document object model) represents an XML file as a tree structure of nodes. You navigate to any node in a DOM by starting at the root of the tree.
  - Both Java and Python provide support for using XML files and the DOM. First, an XML file is parsed to produce a DOM. The DOM contains Element nodes (one per tag) and Text nodes (one per data value).

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_32

- You understand the following regarding relational databases (Rdb):
  - A Rdb is a collection of one or more tables used to store data. Each table has rows representing data instances (e.g., a student) and columns representing data values (e.g., student has name Ahmad and an id of 123456). Relational databases are based on predicate logic and set theory.
  - SQL (structured query language) provides capabilities to create and manipulate databases. DDL (data definition language) statements are used to create a database, tables, indexes, and other types of database objects. DML (data manipulation language) statements are used to create, read, update, and delete data in tables.
  - Both Java and Python provide support for using Rdb. First, you connect to a relational database server and a specific database. You then execute SQL statements to manipulate the data in the database.

- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

32.2 OOD: ABA Persistent Storage Designs

The list of requirements found below has been modified from what was stated in Chap. 18. One requirement was changed to make the data modeling a little more interesting.

1. Allow for entry and (non-persistent) storage of people's names.
2. Store for each person a single phone number and a single email address. *Include the type of phone number and the type of email address for this person. The type of phone number may be either: home, cell, work, or other. The type of email address may be either: personal, work or other.*
3. Use a simple text-based user interface to obtain the contact data.

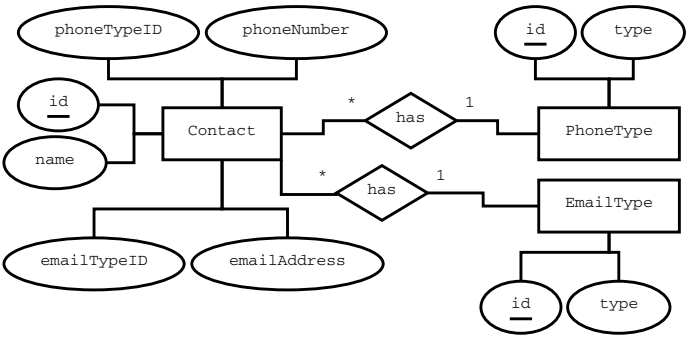


Fig. 32.1 ABA logical data model

4. Ensure that each name contains only upper case letters, lower case letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.

32.2.1 OOD: ABA Data Models

Figure 32.1 shows the logical data model based on the requirements listed above. Since each contact has one phone and one email, the Contact entity has attributes for phone number and email address. This includes identifying the type of phone number (e.g., cell, work) and type of email address (e.g., personal, work). The purpose of the PhoneType and EmailType entities is to keep track of each type of phone and email currently used by the ABA.

32.2.1.1 OOD: ABA XML Physical Data Model

The hierarchy chart in Fig. 32.2 shows the XML tag structure. One <AddressBook> tag will exist in the XML file and is the root node in the DOM tree. The XML file will contain zero or more <Contact> tags, each representing a unique contact name for the ABA. The XML file will also contain a fixed number of <EmailType> and <PhoneType> tags, representing the choices the user has for identifying the type of email address and phone number, respectively. Listing 32.1 shows three email types and four phone types that are part of the ABA. One contact is shown in this XML file. The first line in this XML file is the XML Declaration statement, used by software programs to identify the file as containing XML tags.

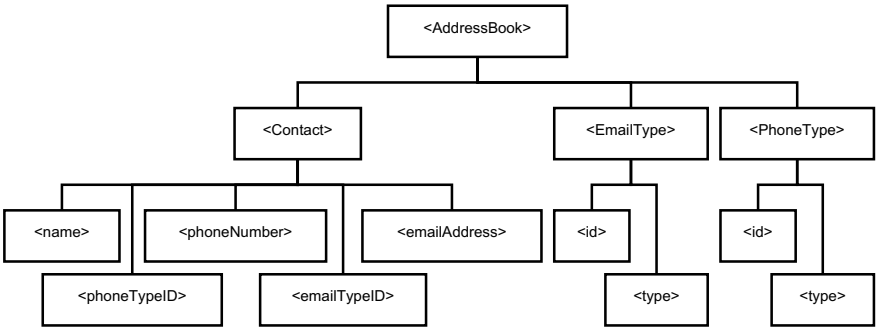


Fig. 32.2 ABA physical data model—XML

Listing 32.1 Sample ABA XML File

```
<?xml version="1.0" encoding="UTF–8" standalone="no"?> <AddressBook>
<EmailType><id>1</id><type>personal</type></EmailType>
<EmailType><id>2</id><type>work</type></EmailType>
<EmailType><id>3</id><type>other</type></EmailType>
<PhoneType><id>1</id><type>cell</type></PhoneType>
<PhoneType><id>2</id><type>home</type></PhoneType>
<PhoneType><id>3</id><type>work</type></PhoneType>
<PhoneType><id>4</id><type>other</type></PhoneType>
<Contact><name>Dave</name><phoneTypeID>1</phoneTypeID>
<phoneNumber>3155551234</phoneNumber><emailTypeID>1</emailTypeID>
<emailAddress>dave@david.com</emailAddress></Contact> </AddressBook>
```

32.2.1.2 OOD: ABA Rdb Physical Data Model

The relational database model in Fig. 32.3 shows the database table structure. The Contact table will contain zero or more rows (aka instances), each representing a unique contact name for the ABA. The database will also contain a fixed number of EmailType and PhoneType rows/instances, representing the choices the user has for identifying the type of email address and phone number, respectively. Listing 32.2 shows the SQL data definition language (DDL) statements to create the AddressBook database. The bottom of this listing includes DML statements to add three email types and four phone types to the respective tables.

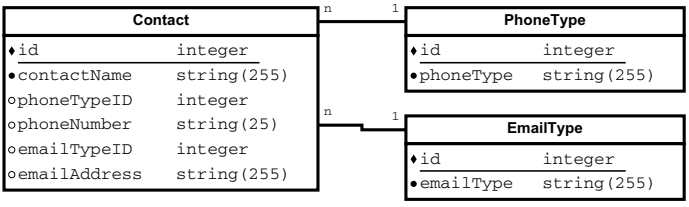


Fig. 32.3 ABA physical data model—Rdb

Listing 32.2 Sample ABA SQL DDL statements

```
drop database if exists AddressBook;
create database AddressBook;
use AddressBook;

create table PhoneType
  (id          INTEGER AUTO_INCREMENT PRIMARY KEY,
  phoneType    VARCHAR(255) NOT NULL);

create table EmailType
  (id          INTEGER AUTO_INCREMENT PRIMARY KEY,
  emailType    VARCHAR(255) NOT NULL);

create table Contact
  (id              INTEGER AUTO_INCREMENT PRIMARY KEY,
  contactName VARCHAR(255) NOT NULL UNIQUE,
  phoneTypeID INTEGER,
  phoneNumber VARCHAR(25),
  emailTypeID INTEGER,
  emailAddress     VARCHAR(255),
  FOREIGN KEY (phoneTypeID)
     REFERENCES PhoneType(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE,
  FOREIGN KEY (emailTypeID)
     REFERENCES EmailType(id)
     ON DELETE CASCADE
     ON UPDATE CASCADE);

/* Indexes to improve performance */
create index Contact_Index_1
   ON Contact(contactName);
create index PhoneType_Index_1
   ON PhoneType(phoneType);
create index EmailType_Index_1
   ON EmailType(emailType);

/* Add phone types */
INSERT INTO PhoneType(phoneType) VALUES ("cell");
INSERT INTO PhoneType(phoneType) VALUES ("home");
INSERT INTO PhoneType(phoneType) VALUES ("work");
INSERT INTO PhoneType(phoneType) VALUES ("other");

/* Add email types */
INSERT INTO EmailType(emailType) VALUES ("personal");
INSERT INTO EmailType(emailType) VALUES ("work");
INSERT INTO EmailType(emailType) VALUES ("other");
```

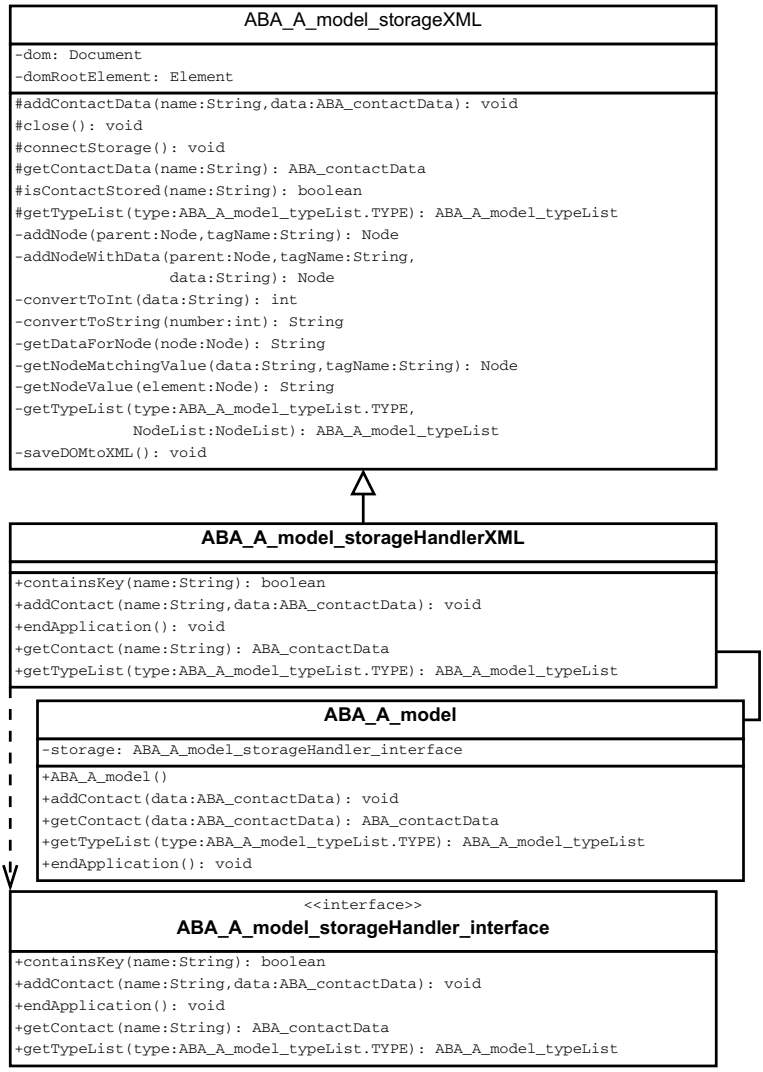


Fig. 32.4 Class diagram—ABA model component for XML

32.2.2 OOD: ABA Design—XML

The class diagram in Fig. 32.4 shows the classes in the model component responsible for the processing of the XML file and DOM. The ABA_A_model_storageXML class is abstract. It contains all of the XML and DOM specific logic for the ABA.XML file. The protected methods (shown with a pound sign "#" before each method name) are called from the ABA_A_model_storageHandlerXML class, which inherits the behavior of the storageXML abstract class and implements the ABA_A_model_storageHandler_interface.

The only attribute shown in the ABA_A_model class is of type ABA_A_model_storageHandler_interface, allowing the model to instantiate a storageHandler for XML or Rdb. Listing 32.3 shows how the model constructor will instantiate one of the storageHandler objects. The storageType parameter shown for the model constructor comes from the ABA_A.main method, which accepts a single parameter to identify whether XML or Rdb is used as the persistent data storage technology.

Listing 32.3 ABA model class constructor

```
private ABA_A_model_storageHandler_interface storage;

public ABA_A_model(String storageType)
{
  if (storageType.equals(ABA_A.ARG_RDB))
    storage = new ABA_A_model_storageHandlerRdb();
  else
    storage = new ABA_A_model_storageHandlerXML();
}
```

Listing 32.4 shows part of the main method, which calls the getFirstArgs method (not shown) to obtain the type of persistent storage to use. The main method then passes this to the constructor, which is responsible for constructing the model object.

Listing 32.4 ABA main method

```
public final static String ARG_RDB = "Rdb";
public final static
String ARG_XML = "XML";

public static void main(String[] args)
{
  String storageType = getFirstArg(args);
  ABA_A_controller aba = new ABA_A_controller(storageType);
  aba.go();
}
```

Below are two questions regarding the design of the model component.

1. Why does the design of the model component use an abstract class?
   An abstract class allows a developer to express behavior to be used by other classes, without having to explicitly construct an object instance (since an abstract class cannot have a constructor method). When a concrete class inherits from an abstract class, it allows the behavior expressed in the abstract class to be included through generalization. In this design, we've flipped the use of generalization (aka inheritance). That is, a superclass (i.e., what is being inherited) generally contains more general behavior while the subclass contains more specific behavior. In this design, the abstract class ABA_A_model_storageXML is the superclass. But it contains specific behavior related to processing an XML file and DOM. In contrast, the subclass ABA_A_model_storageHandlerXML contains general behavior as defined by the ABA_A_model_storageHandler_interface.

2. Why does the design of the model component use an interface?
   The use of this interface is critical to how the model constructor method creates an object instance representing the persistent data storage technology being used. As shown in Listing 32.3, the storage instance variable can refer any object instance whose concrete class implements the ABA_A_model_storageHandler_interface. This allows us to easily add another type of persistent storage (e.g., non-relational database) in some future version of the ABA.

32.2.3 OOD: ABA Design—Rdb

The class diagram in Fig. 32.5 shows the classes in the model component responsible for the processing of the relational database. The ABA_A_model_storageRdb class is abstract. It contains all of the SQL DML specific logic for the AddressBook database. The protected methods (shown with a pound sign "#" before each method name) are called from the ABA_A_model_storageHandlerRdb class, which inherits the behavior of the storageXML abstract class and implements the ABA_A_model_storageHandler_interface.

The only attribute shown in the ABA_A_model class is of type ABA_A_model_storageHandler_interface, allowing the model to instantiate a storageHandler for XML or Rdb. Listing 32.3 shows how the model constructor will instantiate one of the storageHandler objects. The storageType parameter shown for the model constructor comes from the ABA_A.main method, which accepts a single parameter to identify whether XML or Rdb is used as the persistent data storage technology.

Listing 32.4 shows part of the main method, which calls the getFirstArgs method (not shown) to obtain the type of persistent storage to use. The main method then passes this to the constructor, which is responsible for constructing the model object.

Refer to the two questions at the end of Sect. 32.2.2 for a discussion on the use of an abstract class and an interface in this design.

32.2.4 OOD: ABA Design—Summary

The above two sections illustrate one of the strengths of the MVC architectural framework. We are able to use different persistent storage technologies without making any changes to the controller and view components.

The two class diagrams in Figs. 32.4 and 32.5 shows a model component that can be easily extended to use a third type of persistent storage. For example, using a text file would result in creating two new classes: ABA_A_model_storageHandler TextFile which will implement the ABA_A_model_storageHandler_interface and ABA_A_model_storageTextFile which will be an abstract class containing behavior for using a text file as the ABA persistent storage technology. The only other change necessary would be to modify the ABA_A_model constructor method and the getFirstArgs method to add "TXT" as a valid parameter used when starting the ABA.

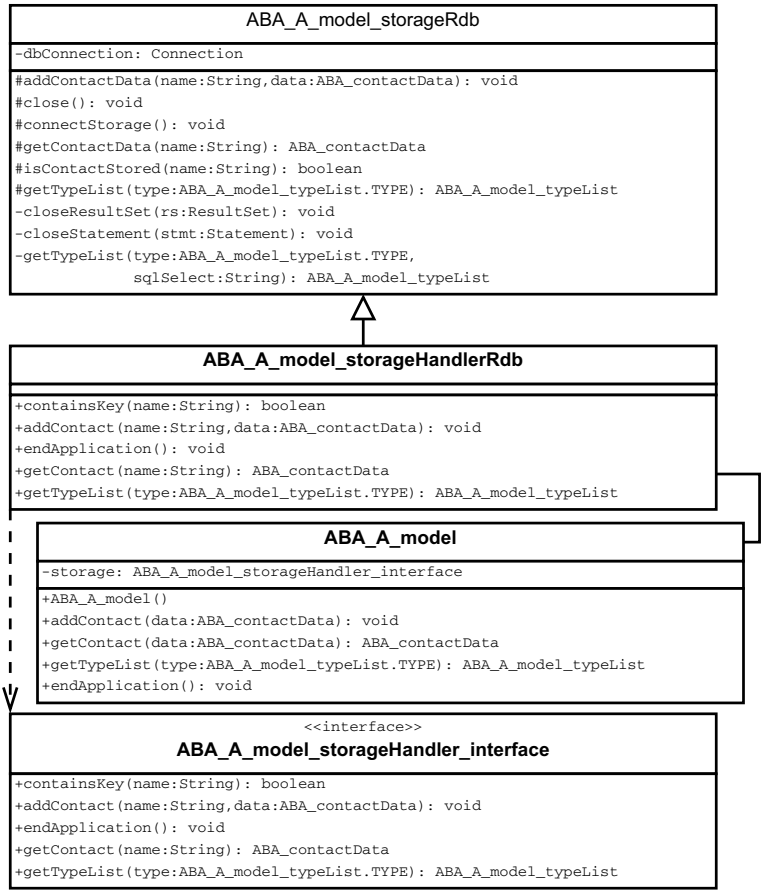


Fig. 32.5 Class diagram—ABA model component for Rdb

32.3 OOD Top-Down Design Perspective

We'll use the personal finances case study to reinforce data design choices as part of a top-down design approach.

32.3.1 OOD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 12, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings

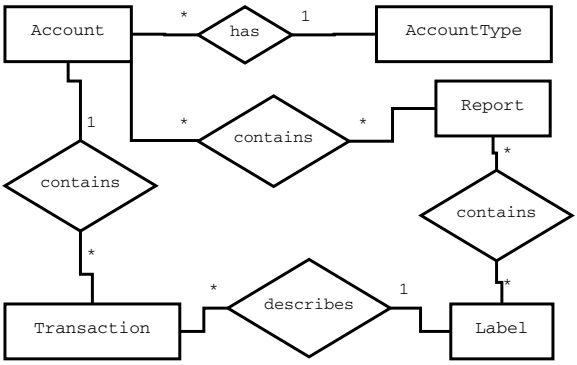


Fig. 32.6 Entity relationship diagram—PF

accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.

- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.
- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

32.3.2 OOD Personal Finances: Data Models

The ERD in Fig. 32.6 shows the entities and relationships, based on the requirements listed above. The relationship rules based on the cardinality are as follows. Note the two many-to-many relationships.

- An Account contains many Transactions; A Transaction is for one Account.
- An Account has one AccountType; An AccountType describes many Accounts.
- An Account is included on many Reports; A Report contains many Accounts.
- A Transaction is described by a Label; A Label describes many Transactions.
- A Label is included on many Reports; A Report contains many Labels.

The LDM in Fig. 32.7 shows the attributes associated with each data entity. Four of the entities are using an id field as the primary key. This would be implemented in a physical data model to ensure each id value is unique. The Account primary key is

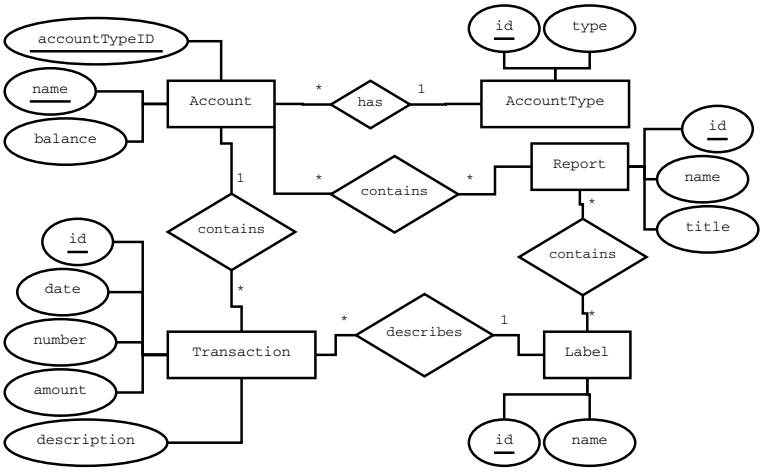


Fig. 32.7 Logical data model—PF

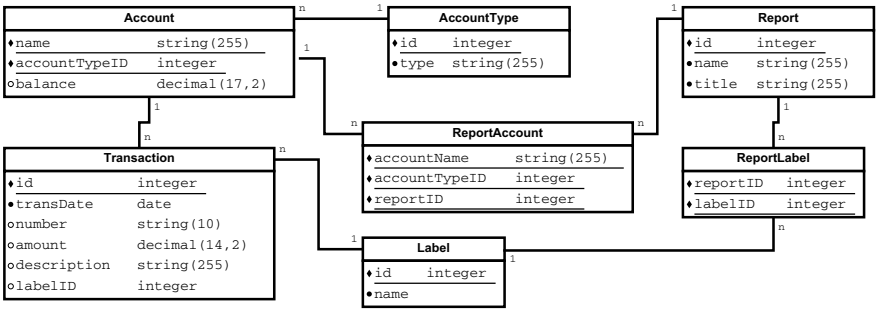


Fig. 32.8 Relational database design—PF

the combination of its name and id value associated with the account type assigned to the account.

The relational database model in Fig. 32.8 shows the translation of the LDM into a relational database. The two many-to-many relationships in the LDM have been replaced with a table, resulting in two additional one-to-many relationships in the physical data model. With a relational database, the id fields would be implemented using the AUTO_INCRMENT feature.

The hierarchy chart in Fig. 32.9 shows the structure of XML tags. With an XML file, the id fields would be implemented to allow the display of the personal finance data in the order they were created. Note the <MaxID> tag, which is used to keep track of the largest id value for each of the four entities having an id field.

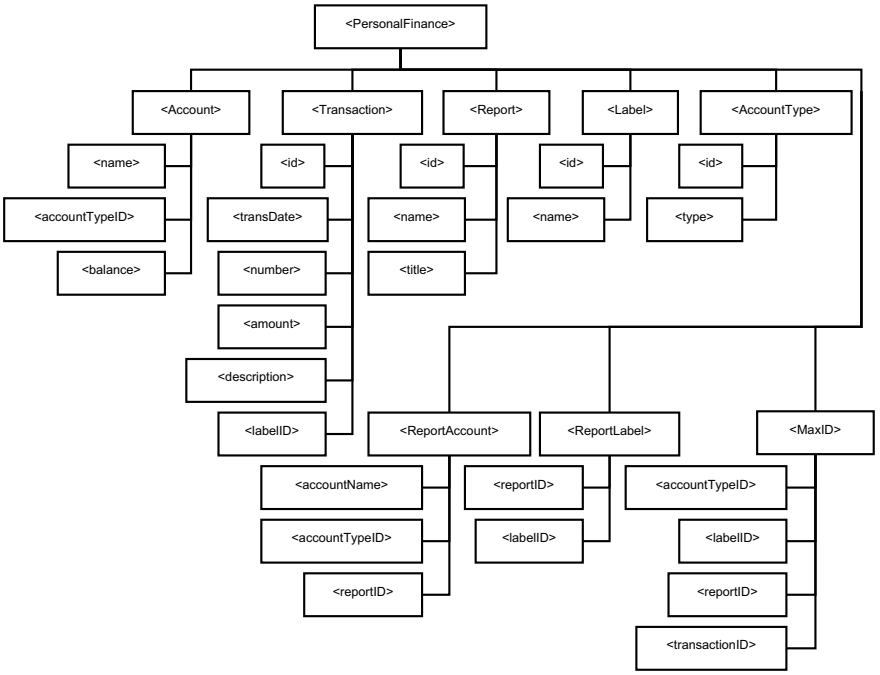


Fig. 32.9 XML design—PF

32.4 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating a LDM to a physical data model needs to consider the limitations of the physical storage format.
- You understand the process of normalization and have applied normalization forms to a data design.
- You understand how to design XML files, and how to use Java to process an XML file.
- You understand how to design a Rdb, and how to use Java to process data stored in an Rdb.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing object-oriented design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a UML class diagram, UML package diagram, IDEF0 function model, and data flow diagram are design models used in object-oriented solutions to illustrate the structure of your software design.
- You understand that an IDEF0 function model, data flow diagram, UML communication diagram, and UML statechart are design models used to illustrate the behavior of your software design.
- You have created and/or modified models that describe an object-oriented software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Use an existing code solution you've developed, develop alternative data models to show ways in which the application data could be structured. Apply the normalization forms on your logical data models. How good or bad is your persistent data storage design?
2. Use your development of an application you started in Chap. 3 for this exercise. Modify your application to use persistent data storage. Evaluate your data models using the normalization forms.
3. Continue hands-on exercise 3 from Chap. 12 by developing a design for a persistent data storage technology. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 12 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)
    - Bus transportation system
    - Course-class enrollment
    - Digital library
    - Inventory and distribution control
    - Online retail shopping cart
    - Personal calendar
    - Travel itinerary



SD Case Study: Persistent Storage

33

The objective of this chapter is to illustrate the use of two persistent storage technologies—eXtensible Markup Language (XML) and relational database (Rdb)—by updating the case study designs.

33.1 SD: Preconditions

The following should be true prior to starting this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating an LDM to a physical data model needs to consider the limitations of the physical storage format.
- You understand the process of normalization and have applied normalization forms to a data design.
- You understand the following regarding eXtensible Markup Language (XML).
  - XML files represent data using a tree structure of tags. An XML tag will contain a data value and/or other tags, is given a name delimited with < and > (e..g, <lastname>), and typically has a matching end tag (e.g., </lastname>).
  - A DOM (document object model) represents an XML file as a tree structure of nodes. You navigate to any node in a DOM by starting at the root of the tree.
  - Both Java and Python provide support for using XML files and the DOM. First, an XML file is parsed to produce a DOM. The DOM contains Element nodes (one per tag) and Text nodes (one per data value).

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_33

- You understand the following regarding relational databases (Rdb).
  - An Rdb is a collection of one or more tables used to store data. Each table has rows representing data instances (e.g., a student) and columns representing data values (e.g., student has name Ahmad and id of 123456). Relational databases are based on predicate logic and set theory.
  - Structured query language (SQL) provides capabilities to create and manipulate databases. Data definition language (DDL) statements are used to create a database, tables, indexes, and other types of database objects. Data manipulation language (DML) statements are used to create, read, update, and delete data in tables.
  - Both Java and Python provide support for using Rdb. First, you connect to a relational database server and a specific database. You then execute SQL statements to manipulate the data in the database.

- You understand Model–View–Controller as a software architecture that separates the user interface (view) from the application data (model). This separation is achieved by putting the domain logic in the controller and enforcing constraints on how these three components communicate with each other.
- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security. You have evaluated program code using these criteria.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.

33.2 SD: ABA Persistent Storage Designs

The list of requirements found below have been modified from what was stated in Chap. 19. One requirement was changed to make the data modeling a little more interesting.

1. Allow for entry and (non-persistent) storage of people's names.
2. Store for each person a single phone number and a single email address. *Include the type of phone number and the type of email address for this person. The type of phone number may be either: home, cell, work, or other. The type of email address may be either: personal, work or other.*
3. Use a simple text-based user interface to obtain the contact data.

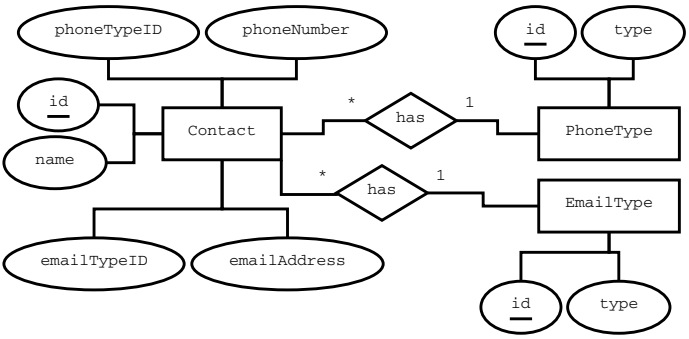


Fig. 33.1 ABA logical data model

4. Ensure that each name contains only uppercase letters, lowercase letters, and spaces. At least one letter must be entered for a name. There is no maximum limit on the size of the name entered.
5. Ensure that each phone number contains only the digits zero through nine. At least one digit must be entered for a phone number. There is no maximum limit on the size of the phone number entered.
6. Prevent a duplicate name from being stored in the address book.
7. Allow for display of contact data in the address book.

33.2.1 SD: ABA Data Models

Figure 33.1 shows the logical data model based on the requirements listed above. Since each contact has one phone and one email, the Contact entity has attributes for phone number and email address. This includes identifying the type of phone number (e.g., cell, work) and type of email address (e.g., personal, work). The purpose of the PhoneType and EmailType entities is to keep track of each type of phone and email currently used by the ABA.

33.2.1.1 SD: ABA XML Physical Data Model

The hierarchy chart in Fig. 33.2 shows the XML tag structure. One <AddressBook> tag will exist in the XML file and is the root node in the DOM tree. The XML file will contain zero or more <Contact> tags, each representing a unique contact name for the ABA. The XML file will also contain a fixed number of <EmailType> and <PhoneType> tags, representing the choices the user has for identifying the type of email address and phone number, respectively. Listing 33.1 shows three email types and four phone types that are part of the ABA. One contact is shown in this XML file. The first line in this XML file is the XML Declaration statement, used by software programs to identify the file as containing XML tags.

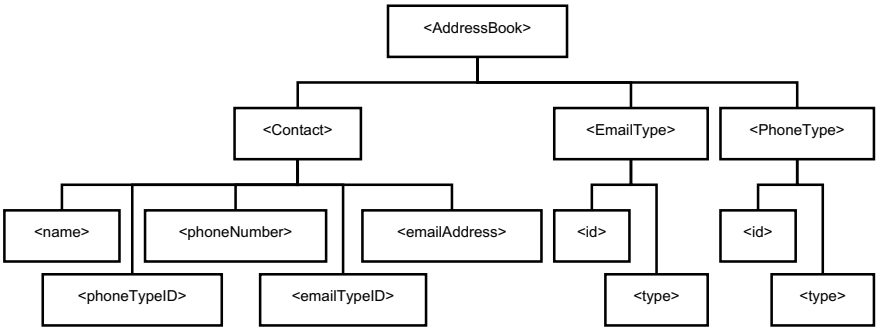


Fig. 33.2 ABA physical data model—XML

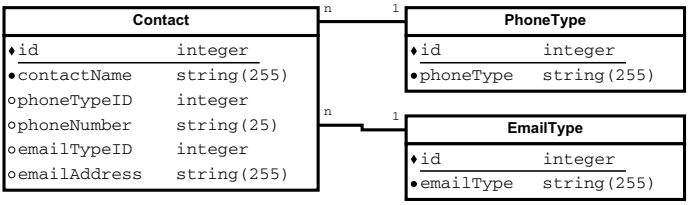


Fig. 33.3 ABA physical data model—Rdb

Listing 33.1 Sample ABA XML File

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<AddressBook>
<EmailType><id>1</id><type>personal</type></EmailType>
<EmailType><id>2</id><type>work</type></EmailType>
<EmailType><id>3</id><type>other</type></EmailType>
<PhoneType><id>1</id><type>cell</type></PhoneType>
<PhoneType><id>2</id><type>home</type></PhoneType>
<PhoneType><id>3</id><type>work</type></PhoneType>
<PhoneType><id>4</id><type>other</type></PhoneType>
<Contact><name>Dave</name><phoneTypeID>1</phoneTypeID>
<phoneNumber>3155551234</phoneNumber><emailTypeID>1</emailTypeID>
<emailAddress>dave@david.com</emailAddress></Contact>
</AddressBook>
```

33.2.1.2 SD: ABA Rdb Physical Data Model

The relational database model in Fig. 33.3 shows the database table structure. The Contact table will contain zero or more rows (aka instances), each representing a unique contact name for the ABA. The database will also contain a fixed number of EmailType and PhoneType rows/instances, representing the choices the user has for identifying the type of email address and phone number, respectively. Listing 33.2 shows the SQL data definition language (DDL) statements to create the AddressBook

database. The bottom of this listing includes DML statements to add three email types and four phone types to the respective tables.

Listing 33.2 Sample ABA SQL DDL statements

```
drop database if exists AddressBook;
create database AddressBook;
use AddressBook;

create table PhoneType
    (id          INTEGER AUTO_INCREMENT PRIMARY KEY,
    phoneType    VARCHAR(255) NOT NULL);

create table EmailType
    (id          INTEGER AUTO_INCREMENT PRIMARY KEY,
    emailType    VARCHAR(255) NOT NULL);

create table Contact
    (id              INTEGER AUTO_INCREMENT PRIMARY KEY,
    contactName VARCHAR(255) NOT NULL UNIQUE,
    phoneTypeID INTEGER,
    phoneNumber VARCHAR(25),
    emailTypeID INTEGER,
    emailAddress     VARCHAR(255),
    FOREIGN KEY (phoneTypeID)
        REFERENCES PhoneType(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    FOREIGN KEY (emailTypeID)
        REFERENCES EmailType(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE);

/* Indexes to improve performance */
create index Contact_Index_1
    ON Contact(contactName);
create index PhoneType_Index_1
    ON PhoneType(phoneType);
create index EmailType_Index_1
    ON EmailType(emailType);

/* Add phone types */
INSERT INTO PhoneType(phoneType) VALUES ("cell");
INSERT INTO PhoneType(phoneType) VALUES ("home");
INSERT INTO PhoneType(phoneType) VALUES ("work");
INSERT INTO PhoneType(phoneType) VALUES ("other");

/* Add email types */
INSERT INTO EmailType(emailType) VALUES ("personal");
INSERT INTO EmailType(emailType) VALUES ("work");
INSERT INTO EmailType(emailType) VALUES ("other");
```

33.2.2 SD: ABA Design

The structure chart in Fig. 33.4 shows the function calls as the ABA is started. The addressBook function has a parameter provided by the caller (i.e., user) to indicate whether to use a relational database ("Rdb") or an XML file ("XML") as the persistent storage technology. When the parameter value is invalid, the ABA will use a relational database. The init function in the ABA_A_controller is given the valid storage type, which is used by the model component to initialize the persistent storage technology. The init function in ABA_A_model_data will call the appropriate connectStorage function based on the storageType parameter. The ABA_A_model_storageRDB.connectStorage function will connect to the Address-

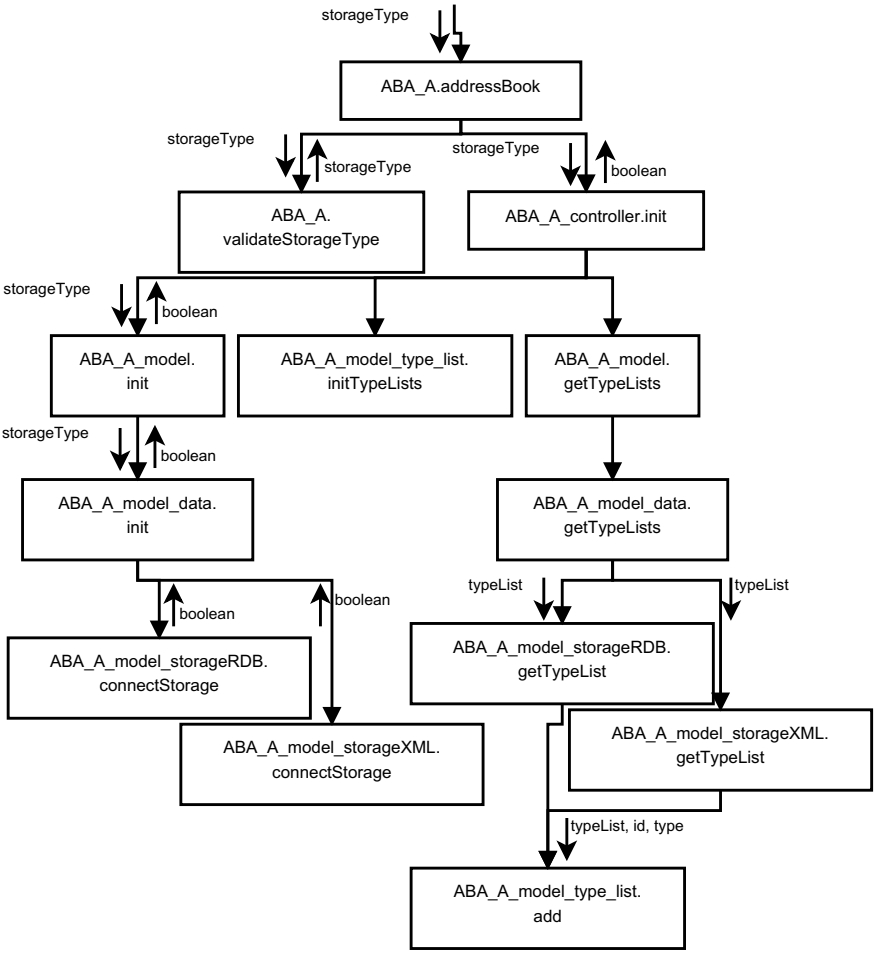


Fig. 33.4 Structure chart—ABA init

Book database while the storageXML.connectStorage function will create a DOM using the ABA.XML file. Listing 33.3 shows the ABA_A_model_data.init function. Note the global variable storageType, which is initialized by the init function then used by all the other functions in this module.

Listing 33.3 ABA_A_model_data module init function

```
def init(persistentStorageType):
    global storageType
    storageType = persistentStorageType
    if storageType == ABA_A.ARG_RDB:
        initResult = ABA_A_model_storageRDB.connectStorage()
    else:
        initResult = ABA_A_model_storageXML.connectStorage()
    return initResult
```

After the connectStorage function completes, the last initialization step is to obtain the list of email and phone types stored in the persistent storage technology. The list of types is kept in the ABA_A_model_type_list module. Listing 33.4 shows how two global variables are initialized and used to store the (id, type) pairs for email and phone types. The idMinMaxList global stores the smallest and largest id value for email and phone types. This list is used to validate the user's entry of a type id value displayed in a menu.

Listing 33.4 ABA_A_model_type_list module initTypeLists function

```
EMAIL_TYPE = 0
PHONE_TYPE = 1

def initTypeLists():
    global typeLists
    global idMinMaxList
    #A list containing two dictionaries
    #   typeLists[EMAIL_TYPE] is a dictionary.
    #   typeLists[PHONE_TYPE] is a dictionary.
    #       Both contain (id,type) pairs where id is key value.
    typeLists = [{}, {}]
    #A list containing two lists
    #   idMinMaxList[EMAIL_TYPE] is a list.
    #   idMinMaxList[PHONE_TYPE] is a list.
    #       Both contain [min, max] id value for email or phone types.
    idMinMaxList = [[None, None], [None, None]]
```

Listing 33.5 shows the add function in the ABA_A_model_type_list module. This is called by either storageRDB or storageXML each time this logic wants to add another email or phone type to the two global data structures just described. The type_list parameter will be one of the global variables (EMAIL_TYPE or PHONE_TYPE) defined in this module. Note how the idMinMaxList is updated to keep track of the minimum and maximum id value for each type of list being stored.

Listing 33.5 ABA_A_model_data module add function

```
def add(type_list, id, type):
    global typeLists
    global idMinMaxList
    if type_list == EMAIL_TYPE or type_list == PHONE_TYPE:
        #Get dictionary for email or phone type lists
        dict = typeLists[type_list]
        dict[id] = type
        #Get list for email or phone min/max id values
        minMax = idMinMaxList[type_list]
        #Update minimum ID value
        if minMax[MIN_IDX] is None:
            minMax[MIN_IDX] = id
        elif id < minMax[MIN_IDX]:
            minMax[MIN_IDX] = id
        #Update maximum ID value
        if minMax[MAX_IDX] is None:
            minMax[MAX_IDX] = id
        elif id > minMax[MAX_IDX]:
            minMax[MAX_IDX] = id
```

The structure chart in Fig. 33.5 shows the function calls after the user has entered data to create a new contact. The isCreateContact function would return true, causing the user-supplied contact data to be validated. Assuming the validateNewContact function returns true, the ABA_A_model.storeContactData is called. The addEntry function in the ABA_A_model_data module determines which type of persistent storage is being used, then calls the appropriate isContactStored and addContactData functions to ensure the name is not already in the address book and to store the contact data, respectively.

Listing 33.6 shows the addEntry function. Note the use of the storageType global variable to determine whether to use the storageRdb OR storageXML module.

Listing 33.6 ABA_A_model_data module addEntry function

```
def addEntry():
    global storageType
    name = ABA_A_contact_data.getName()
    if storageType == ABA_A.ARG_RDB:
        if not ABA_A_model_storageRDB.isContactStored(name):
            ABA_A_model_storageRDB.addContactData()
    else:
        if not ABA_A_model_storageXML.isContactStored(name):
            ABA_A_model_storageXML.addContactData()
```

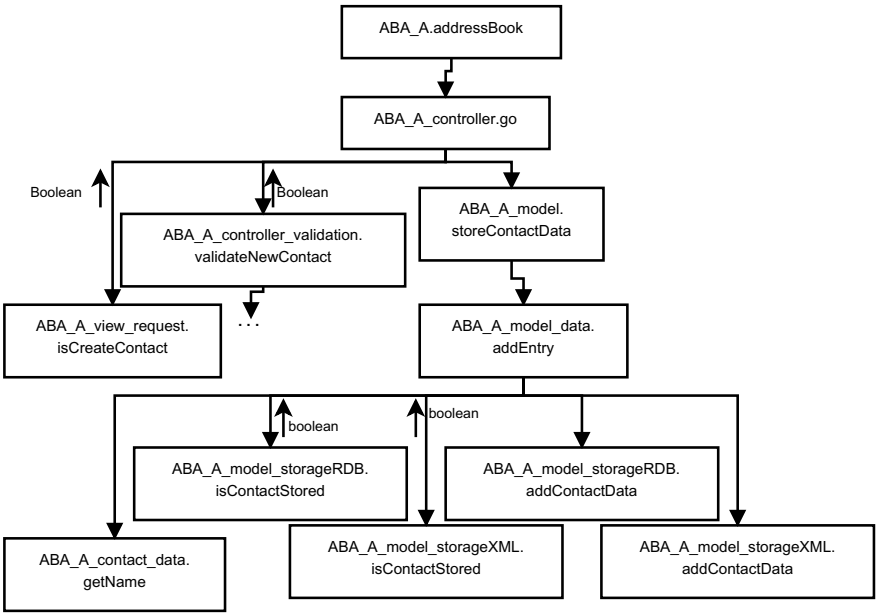


Fig. 33.5 Structure chart—ABA create a contact

33.3 SD Top-Down Design Perspective

We'll use the personal finances case study to reinforce data design choices as part of a top-down design approach.

33.3.1 SD Personal Finances: A Second Case Study

The requirements for personal finances, as stated in Chap. 13, are listed below.

- Allow a user to create as many accounts as they would like. Each account represents a single financial asset or liability that is provided as a financial service to an individual. Accounts are typically created to represent checking and savings accounts, credit card accounts, and loans for an automobile, school, or home. Each account has a name and a balance.
- Allow a user to enter as many transactions as they would like in each account. Each transaction represents a credit or debit within the account. Each transaction has a date, description, and amount (which can be a credit or debit). A transaction may optionally have a number (e.g., a check number for a checking account transaction) and zero or more labels.

- Allow a user to create labels, which are used to categorize a transaction. For example, labels may be created to indicate whether a transaction is associated with a charity, groceries, or home improvement.
- Allow a user to create as many reports as they would like. Each report includes one or more accounts and zero or more labels.
- Allow a user to modify or delete any account, transaction, label, or report.

33.3.2 SD Personal Finances: Data Models

The ERD in Fig. 33.6 shows the entities and relationships, based on the requirements listed above. The relationship rules based on the cardinality are as follows. Note the two many-to-many relationships.

- An Account contains many Transactions; A Transaction is for one Account.
- An Account has one AccountType; An AccountType describes many Accounts.
- An Account is included on many Reports; A Report contains many Accounts.
- A Transaction is described by a Label; A Label describes many Transactions.
- A Label is included on many Reports; A Report contains many Labels.

The LDM in Fig. 33.7 shows the attributes associated with each data entity. Four of the entities are using an id field as the primary key. This would be implemented in a physical data model to ensure each id value is unique. The Account primary key is the combination of its name and id value associated with the account type assigned to the account.

The relational database model in Fig. 33.8 shows the translation of the LDM into a relational database. The two many-to-many relationships in the LDM have been replaced with a table, resulting in two additional one-to-many relationships in the physical data model. With a relational database, the id fields would be implemented using the AUTO_INCREMENT feature.

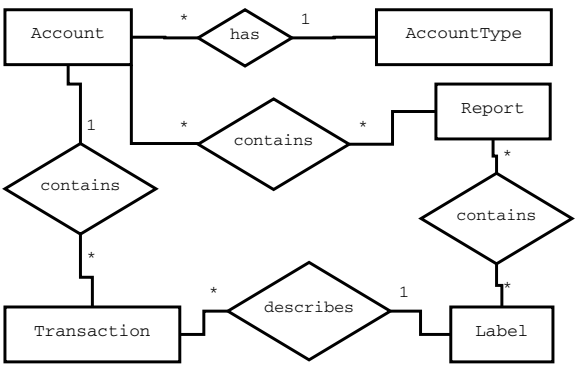


Fig. 33.6 Entity relationship diagram—PF

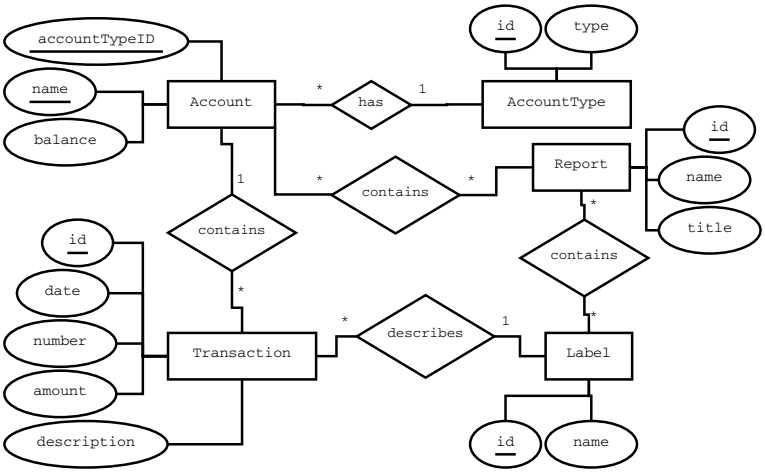


Fig. 33.7 Logical data model—PF

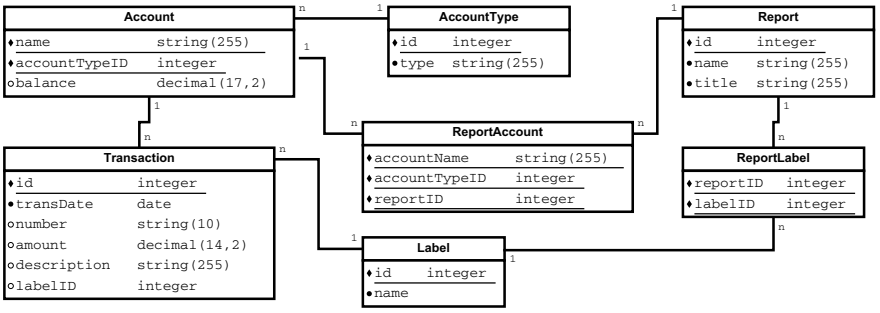


Fig. 33.8 Relational database design—PF

The hierarchy chart in Fig. 33.9 shows the structure of XML tags. With an XML file, the id fields would be implemented to allow the display of the personal finance data in the order they were created. Note the <MaxID> tag, which is used to keep track of the largest id value for each of the four entities having an id field.

33.4 Post-conditions

The following should have been learned when completing this chapter.

- You understand how to develop data models, including entity relationship diagrams and fully-attributed logical data models. You also understand that translating an

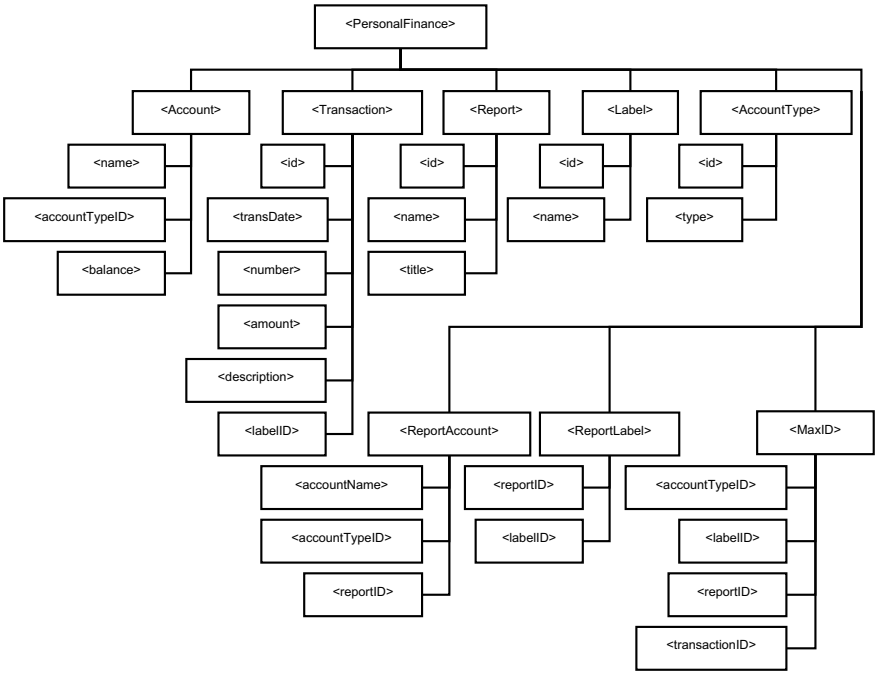


Fig. 33.9 XML design—PF

LDM to a physical data model needs to consider the limitations of the physical storage format.

- You understand the process of normalization and have applied normalization forms to a data design.
- You understand how to design XML files, and how to use Python to process an XML file.
- You understand how to design an Rdb and how to use Python to process data stored in an Rdb.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing structured design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security. You have evaluated a software design using these criteria.
- You understand that a hierarchy chart, IDEF0 function model, structure chart, structure diagram, and data-flow diagram are design models that may be used to illustrate the structure of your software design.

- You understand that an IDEF0 function model, data-flow diagram, structure chart, structure diagram, and UML state machine are design models that may be used to illustrate the behavior of your software design.
- You have created and/or modified models that describe a structured software design. This includes thinking about design from the bottom-up and from the top-down.

Exercises

Hands-on Exercises

1. Use an existing code solution you've developed, develop alternative data models to show ways in which the application data could be structured. Apply the normalization forms on your logical data models. How good or bad is your persistent data storage design?
2. Use your development of an application you started in Chap. 4 for this exercise. Modify your application to use persistent data storage. Evaluate your data models using the normalization forms.
3. Continue hands-on exercise 3 from Chap. 13 by developing a design for a persistent data storage technology. Be sure to develop design models to illustrate the structure and behavior of your design. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chap. 13 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)
    - Bus transportation system
    - Course-class enrollment
    - Digital library
    - Inventory and distribution control
    - Online retail shopping cart
    - Personal calendar
    - Travel itinerary

Part IV
Wrap-Up

This fourth part introduces the notion of a software design document and presents some ideas for what you should learn next.



Software Design Document

34

The objective of this chapter is to present a template for a software design document that may be used to consolidate design knowledge into a single document.

34.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_34

34.2 Concepts and Context

Documents are produced in software development projects to encapsulate the relevant information into a well-structured and thorough presentation. As described in this book, a software design contains knowledge represented using many types of models and textual descriptions. Including all of this into a single document allows for an easy way to share your design information. The question addressed by this chapter is how should we organize our design knowledge?

34.2.1 What Would a Software Design Artifact Describe?

Pressman describes four types of design information [1].

Architecture: A high-level description of the observable elements making up the system.
Data: Includes logical data models, physical data models, and a data dictionary.
Interfaces: Describes human–computer interactions, external interfaces, and internal interfaces between components.
Components: Includes design descriptions of each component making up the system. Includes both observable components (shown in architecture) and non-observable components.

34.2.2 What Would a Software Design Artifact Look Like?

The structure of a design document is based on these four types of design information. Table 34.1 shows an outline for a software design document.

Each section of the design artifact is briefly described below.

34.2.2.1 Introduction

This section describes the purpose and scope of the design document.

34.2.2.2 Architecture

Create and insert one or more diagrams to show the high-level design of the application. The types of diagrams you use to show your application architecture can include one or more of the following. Please note these diagram types are not listed in order of preference.

- A data-flow diagram (DFD) to show the major components of the design and how these components are related to each other via data flows and data stores.
- An IDEF0 function model to show the major components of the design and how these components are related to each other via input, control, output, and mechanism data flows.

Table 34.1 Design document Structure

1. Introduction
Describe the purpose and scope of the design document.
2. Architecture
Describe the high-level design of the application by inserting one or more software architecture diagrams.
3. Data
This section contains a description of the logical and physical view of the data used by the application.
3.1 Logical Data Model
Insert a logical data model here, and then list the data rules based on the cardinalities shown in the logical data model.
3.2 Physical Data Model
Insert a description of the persistent data store. This may include a physical data model (e.g., relational database design or hierarchy of XML tags).
3.3 Internal Data Structures
Describe the memory-based data structures constructed and used during execution of the application.
3.4 Data Dictionary
Include a table to describe the characteristics of each attribute shown in the data models.
4. Interfaces
This section contains a description of interfaces that must be supported by the application.
4.1 Human-computer Interaction
Describes the user interactions of the application/system.
4.2 Communication with External Entities
Describes the ways in which the application will communicate with different types of external entities.
4.3 Communication between Internal Components
Describes the ways in which the components of this application/system will communicate with each other.
5. Components
This section contains a more detailed description of each component described in the Architecture section.
5.1 Component X
Include text and design models that describe the component. Each component section should describe the behavior and structure.

- For an object-oriented design, a UML package and/or class diagram to show the major components of the design and how the classes in each package are related to classes in other packages.

34.2.2.3 Data

The Data section will contain descriptions for logical data models, physical data models, internal data structures, and a data dictionary.

In section *3.1 Logical Data Model*, create and insert a logical data model (LDM) for the application. This should show the strong entities, weak entities, and the relationships between each data entity. The model should show the attributes associated with each data entity and which attribute(s) represent the primary key.

After the LDM, list the rules shown in the logical data model. Each rule is a single sentence describing a relationship between two logical entities. A few example rules are listed below, based on an LDM that describes contact information. (The text in parentheses is an explanation of what the LDM would look like in order to generate this rule. This text is part of these instructions and not intended to be included in your design artifact.)

- A contact has a name that is not required to be unique. (The LDM shows a contact entity with an id attribute as the primary key and a name as a non-key attribute.)

Table 34.2 Data dictionary information—text file

Attribute	Type	Size	Format/range	Structure

- A contact has zero or more phone numbers. (The LDM shows a relationship between a contact entity and a phone number entity. The cardinality shows that a phone number instance is not required for each contact instance.)
- A contact helps to identify their phone numbers. (The LDM shows a relationship between a strong contact entity and a weak phone number entity.)

In section *3.2 Physical Data Model*, when a persistent data store is being used by the application, create and insert a physical data model to show the physical representation of the data as it will exist in the persistent data store. When a persistent data store is not being used, this section may be removed from your design artifact.

The type of physical data model used is based on the type of persistent data storage to be used by the application. The following is not intended to be an exhaustive list of persistent data storage types.

- For a text file, this section needs to describe the contents and format of the lines of text. When there are different types of text lines in the text file, each type of text line needs to be described.
- For a text-based XML file, this section needs to describe the hierarchy of XML tags that describe the structure of the data.
- For a relational database, this section needs to describe the tables, primary keys, foreign keys, non-key fields, and indexes based on the logical data model.

In section *3.3 Internal Data Structures*, when non-persistent data structures (e.g., linked list, array, tree, graph, document object model) will be used by the application, provide a description for each data structure. Otherwise, this section is not applicable and may be removed from your design artifact.

In section *3.4 Data Dictionary*, list the attributes shown in the logical data model and describe their physical characteristics. Use one of the following tables based on the type of physical data store being used. Tables 34.2, 34.3, and 34.4 show three different formats for a data dictionary, each containing the following columns.

Attribute: the name of the attribute on the logical data model.
Type: the type of data stored in the attribute.
Size: the size of the attribute, based on its Type.
Format/Range: any special format rules or range of values the attribute must adhere to.

Table 34.2 shows what a data dictionary might look like for a text file. The *structure* column identifies the internal data structure(s) used to store values associated with this attribute.

Table 34.3 Data dictionary information—XML file

Attribute	Type	Size	Format/range	XML tag

Table 34.4 Data dictionary information—relational database

Attribute	Type	Size	Format/range	Table

Table 34.3 shows what a data dictionary might look like for an XML file. The *XML Tag* column identifies the XML structure(s) used to store values associated with this attribute.

Table 34.4 shows what a data dictionary might look like for a relational database. The *Table* column identifies the table(s) containing the attribute.

34.2.2.4 Interfaces

This section contains a description for three types of interfaces: the human–computer interaction; the ways the application will communicate with external entities as described in the Architecture section; and how components of this application communicate with each other.

When one of these interface types does not exist for your application you may remove that section from your design artifact.

In section *4.1 Human–computer Interaction*, when the application has an interface with a human user, create and insert a state machine diagram (aka statechart) to explain how the user would navigate through the application. When the HCI design is large or complex, you may want to have more than one statechart or you may want to include descriptive text to help explain the interactions between the application and a user. In addition, a large or complex HCI should also be described using pictures to illustrate its appearance. When doing this, adding text to connect the state machine diagram to the picture(s) would help in understanding the HCI design.

In section *4.2 Communication with External Entities*, when the application must communicate with one or more external entities (as shown in the Architecture section), create and insert a structure or behavior model to show the way(s) in which the application will communicate with each type of external entity. These diagrams would show the application programming interface (API) to be used between your application code and the external entity.

In section *4.3 Communication between Components*, when your architecture/design contains two or more components, this section should describe how these components communicate with each other. This description would likely include both text and diagrams, using one or more behavior and structure diagrams.

34.2.2.5 Components

This section contains a more detailed description of each component described in the Architecture section. Create a separate [Component X] section for each component and rename each section to one of the component names.

In section *5.1 [Component X]*, each component in your architecture/design will have at least two diagrams—one that shows the structure and another that shows the behavior of the component.

34.3 Post-conditions

The following should have been learned when completing this chapter.

- You understand one approach for consolidating design knowledge into a single document/artifact. This approach describes a design at a high-level of abstraction via the Architecture section; data models representing both memory-based and persistent data storage; interface descriptions for the user interface, to external entities, and between components within the design; and details on each component described in the Architecture section.

Exercises

Hands-on Exercises

1. Using an existing design solution, create a design document containing all of the design knowledge you've developed for your solution.
2. Use your development of an application you started in Chaps. 3 or 4 for this exercise. Create a design document containing all of the design knowledge you've developed for this application.
3. Continue hands-on exercise 3 from Chaps. 12 or 13 by developing a design document containing all of the design knowledge you've developed for this problem domain. The list below serves as a reminder of the application domain you may have chosen for this exercise. Refer to the Hands-on Exercises in Chaps. 12 or 13 for details on each domain.

    - Airline reservation and seat assignment
    - Automated teller machine (ATM)
    - Bus transportation system
    - Course-class enrollment
    - Digital library
    - Inventory and distribution control
    - Online retail shopping cart
    - Personal calendar
    - Travel itinerary

Reference

1. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. McGraw-Hill, New York



What's Next?

35

The objective of this chapter is to discuss what you should do next to further your understanding and experience in developing software.

35.1 Preconditions

The following should be true prior to starting this chapter.

- You have read relevant portions of this book based on your learning needs.

35.2 What Should You Do Now That You've Been Introduced to Software Design?

There are lots of ways the author can address this question. Below are a few ideas for you to ponder, and perhaps pursue.

35.2.1 Dive Deeper into Book Topics

You should continue to think about and apply concepts covered in this book. From a programming perspective, the three program design criteria and six software design criteria, listed below, should be strongly considered whenever you are developing a software solution. If you get nothing else from this book, hopefully these nine criteria will be an integral part of your software development practice.

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_35

- Program design criteria
  - Separation of concerns
  - Design for reuse
  - Design only what is needed
- Software design criteria
  - Simplicity
  - Coupling
  - Cohesion
  - Information hiding
  - Performance
  - Security

Thinking more abstractly when developing software is critical for developers whom would like to participate in the early stages of a software development project. When the need for a large software system is being identified, early efforts focus on describing the scope of the project. A project scope often includes a description of the system and its subsystems. This top-down description may include design models to show both structure and behavior of the system. As discussed in this book, both the data-flow diagram and IDEF0 function model allow you to abstract away many details while describing the behavior and interactions of subsystems. In addition, when you know that the design and implementation will use the object-oriented paradigm, a UML package diagram may also be a useful design model to describe scope.

Using the design models discussed in this book will help you become a better software developer, regardless of the level of design detail you are describing. The list below identifies all of the design models discussed in this book. The first three design models in this list provide lots of flexibility in how you choose to use abstraction and in decomposing a large system into subsystems, components, and subcomponents.

- Data-flow diagram
- IDEF0 function model
- UML statechart
- Data modeling: entity–relationship diagram, logical data model, physical data model
- Object-oriented design models: UML class diagram, package diagram
- Structured design models: Hierarchy chart, structure chart, structure diagram

The five perspectives on software design introduced in Part III are each worthy of further exploration. While this book only has one chapter on quality assurance, this topic is critical in improving software regardless of the design perspective, application domain, or scope of software. Using the techniques described in Chap. 23 will help you create software with fewer defects and vulnerabilities. Go back and read this

chapter again, start using these techniques, and convince your peers, colleagues, and project team members to use them!

All developers can improve their understanding and development of secure software. We should strive to develop more robust software; software capable of reacting in a secure way to expected and malicious use, and to accidental misuse. The 13 security design principles introduced in Chap. 24 represent a great start to creating more robust software. While applying the security design principles during design and implementation is critical, it is just as important to think about security while planning, gathering requirements, and testing.

Human–computer interaction design is a challenging and creative process often overlooked by developers focused on the more technical aspects of software development. Creating a great software product starts with the user interface. Developing a user-interface design that is useful, easy to learn, and allows the user to be productive should result in a software product people will want to use. Continuing to study and apply HCI topics allows one to blend the arts and sciences.

Persistently storing data is often a critical feature of software. The introduction to this design topic, including an introduction to XML and relational databases, is an area with lots of potential for further exploration. With XML, you could continue to learn about text-based XML files and start to learn about binary-based XML files. With relational databases, you could continue to learn about DDL and DML, logical and physical data modeling, or learn more about database administration.

The introduction to design patterns included in this book is just the beginning. If you are serious about becoming a better software designer, particularly using an object-oriented paradigm, you should consider diving much deeper into this topic.

35.2.2 Explore Other Design Topics

You should begin to learn and apply other design topics. These include

- Metrics
  Can we measure the quality of a software design? For example, can we describe the amount of coupling between design components using an easy to understand metric? When these measures exist, do they look only at program code or do they evaluate design models? Is a quantitative or qualitative measure more useful to us in describing the quality of a software design? Can any of the existing measures be automatically *computed* in a software development tool, regardless of whether the measure uses code or models?
- Layers of design abstraction
  What does it mean to be a software architect? How much abstraction is too much when creating a very high-level description of a software system? When creating a conceptual design, how much design is needed to convey intentions and to prevent misinterpretations? When creating a detailed design, at what point is the design too detailed? That is, when should someone stop creating a detailed design?

- Design within a software development processes (SDP)
  How does an organization maintain design artifacts for software that evolves over many decades? What are the benefits and limitations associated with maintaining design artifacts?
- Design within an agile software development processes (SDP)
  How is design done in an agile SDP? How much design should be documented while using an agile SDP while staying true to the agile manifesto? [1]

35.3 Post-conditions

The following should have been learned when completing this chapter.

- You have a deeper appreciation for software design and realize there are many ways to further your understanding of this complex subject area.

Reference

1. Beck K et al (2001) Manifesto for agile software development. https://agilemanifesto.org/. Accessed 30 Jul 2019

Index

A
Abstraction, 3, 4, 10, 132
  software design models, 133
  software design patterns, 134

C
Case study
  approach, 20, 27, 41, 147, 167
  requirements, 28, 35, 42, 48, 61, 66, 70, 77, 82, 86, 104, 115, 182, 220, 268, 286, 306, 320, 362, 376, 404, 407, 416, 421, 476, 490
Class diagram, 21, 30, 34, 64, 69, 73, 108, 110, 138, 148, 190, 197, 209, 268, 275, 308, 313, 366, 367
  Fernandez Symmetric Encryption, 407
  GoF Command, 405
  GoF Facade, 404
  Rdb data store, 482
  XML data store, 480
Cryptography, 354
  asymmetric, 355
  digital certificates, 355
  digital signature, 355
  hash function, 355
  symmetric, 355
Customer survey, 342

D
Data-flow diagram, 149, 168, 209, 243
Data modeling
  1st normal form, 436, 439
  2nd normal form, 436, 439
  3rd normal form, 436, 439
  4th normal form, 438, 440
  attribute, 429
  Boyce-Codd normal form, 437, 440
  cardinality, 429, 432
  entity, 428
  ERD, 428, 430–432
  LDM, 429–431, 433, 477, 491
  logical, 428
  normalization, 435
  physical, 434
  primary key, 431
  relationship, 428
  weak entity, 429
Denial-of-service attack, 92
Design pattern, 390
  Fernandez, 396
  Fernandez Asymmetric Encryption, 399
  Fernandez Authenticator, 407, 421
  Fernandez Digital Signature, 400
  Fernandez Secure Model–View–Controller, 401, 408, 422
  Fernandez Symmetric Encryption, 398, 407, 421
  Fernandez Symmetric Encryption class diagram, 407
  Fernandez Symmetric Encryption structure chart, 421
  GoF, 390
  GoF Command, 404, 418
  GoF Command class diagram, 405
  GoF Command python, 418
  GoF Command structure chart, 418
  GoF Facade, 393, 404, 417
  GoF Facade class diagram, 404
  GoF Facade python, 417
  GoF Facade structure chart, 417
  GoF Iterator, 393
  GoF Singleton, 392, 405, 420
  GoF Singleton java, 406
  Larman, 394

© Springer Nature Switzerland AG 2020

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2

Larman Creator, 395
Larman High Cohesion, 131, 406, 420
Larman Low Coupling, 130, 406, 420
OOD versus SD, 416
Document object model, *see* DOM
DOM, 445
example, 449, 452, 453
structure, 446

E
Entity relationship diagram, *see* ERD
ERD, *see* data modeling
notation, 428
Extensible markup language, *see* XML

F
Formal review, 335
case study example, 346
process, 336
roles, 336
tester, 338

G
GUI
ABA design evaluation, 312, 326
check box, 302
class diagram, 308, 313
concepts, 300
java, 309
label, 302
list box, 302
objects and listeners, 301
push button, 302
python, 324
radio button, 303
security, 362, 378
statechart, 306, 313, 320, 327, 362, 376
structure chart, 321, 323
text fields, 303
UML communication diagram, 308, 311
window example, 306, 313, 320, 327

H
HCI, 252
HCI design goals, 255
be consistent, 257
know the user, 255
optimize user abilities, 256
prevent user errors, 256
HCI design steps, 257
HCI evaluation criteria, 252
efficiency, 253
learnability, 253
user satisfaction, 254
utility, 254
Hierarchy chart, 21, 43, 47, 50, 80, 85, 88, 117, 119, 161, 168, 243
Human–computer interaction, *see* HCI

I
IDEF0 function model, 138, 158, 190, 225, 362, 376
Information assurance, *see* security, information assurance
Information security, *see* security, information security

L
LDM, *see* data modeling
notation, 429
Logical data model, *see* LDM

M
Model–View–Controller, *see* MVC
MVC, 176
2-tier generic data-flow diagram, 178
class diagram, 190, 197
cohesion, 194, 229, 240
coupling, 191, 226, 236
generic data-flow diagram, 176
generic IDEF0 function model, 176
GUI design, 298
IDEF0 function model, 190, 225
information hiding, 195, 230, 241
java, 183–186, 188, 199–201
map requirements to components, 182, 220, 231, 321, 379
package diagram, 183, 196
performance, 196, 231, 242
python, 221–224, 230, 233, 235
security, 196, 231, 242
simplicity, 190, 226, 236
statechart, 190, 197, 225, 232
structure chart, 224, 226–228, 234, 235, 237
structure diagram, 222, 235
TUI design, 298
UML communication diagram, 185, 186, 188, 198, 199

N
Nassi–Shneiderman diagram, 22, 30, 37, 43, 50, 65, 80, 108, 118

O
Object-oriented language features, 212
  abstract class, 212
  class diagram, 214, 215
  code examples, 213–215
  interface, 212
OOD case study
  class diagram, 138, 190, 197, 268, 308, 366, 367
  cohesion, 143, 194, 203
  coupling, 142, 191, 202
  GUI design evaluation, 312
  IDEF0 function model, 138, 190, 362
  information hiding, 145, 195, 204
  package diagram, 138, 183, 196, 364
  performance, 145, 196, 204
  security, 146, 196, 204
  simplicity, 141, 190, 202
  statechart, 140, 142, 190, 197, 268, 306, 362, 364
  TUI design, 262, 266
  TUI design evaluation, 265, 267
  UML communication diagram, 139, 141, 185, 186, 188, 198, 199, 269, 308, 311
OOD top-down, 147, 204, 270, 312, 370, 408, 483
  class diagram, 148, 209, 275, 313
  cohesion, 152, 212
  coupling, 152, 212
  data-flow diagram, 149, 209
  ERD, 484
  Fernandez patterns, 411
  GoF patterns, 409, 410
  GUI design, 313
  GUI design evaluation, 314
  information hiding, 152, 212
  Larman patterns, 409, 410
  LDM, 484
  package diagram, 147, 205
  performance, 152, 212
  Rdb, 485
  requirements, 147
  security, 152, 212, 371
  security design principles, 371, 372
  simplicity, 152, 211
  statechart, 150, 274, 313
  TUI design, 271
  TUI design evaluation, 273
  UML communication diagram, 150, 209, 210
  XML, 485
OOP case study
  class diagram, 30, 34, 64, 69, 73, 108, 110
  data input validation, 104
  data output validation, 111
  design for reuse, 31, 35, 37, 70, 74
  design only what is needed, 32, 35, 38, 70, 74
  exception handling, 112
  fail-safe defaults, 113
  java, 29, 32, 36, 63, 67, 71, 106
  memory performance, 66, 70, 75
  Nassi–Shneiderman diagram, 30, 37, 65, 108
  python, 28, 32, 35, 62, 66, 71, 104
  separation of concerns, 30, 34, 37, 70, 73
  statechart, 30, 37, 65, 108
  time performance, 66, 70, 74

P
Package diagram, 138, 147, 183, 196, 205, 364
Program design criteria, 20
  design for reuse, 21, 31, 35, 37, 45, 47, 51, 70, 74, 85, 89
  design only what is needed, 21, 32, 35, 38, 45, 47, 51, 70, 74, 85, 89
  memory performance, 58, 66, 70, 75, 81, 86, 89
  separation of concerns, 20, 30, 34, 37, 44, 47, 51, 70, 73, 85, 89
  time performance, 56, 66, 70, 74, 81, 86, 89
Program state, 4, 10

R
Rdb, 455, 478, 492
  PDM example, 457, 458
  predicate logic, 455
  set theory, 456
  table, 455
  translate LDM to PDM, 456
Relational database, *see* Rdb

S
SD case study
  cohesion, 164, 229, 240
  coupling, 163, 226, 236
  GUI design evaluation, 326
  hierarchy chart, 158, 161
  IDEF0 function model, 158, 225, 376
  information hiding, 165, 230, 241
  performance, 166, 231, 242
  security, 166, 231, 242
  simplicity, 161, 226, 236

statechart, 160, 162, 225, 232, 286, 320, 376, 378
structure chart, 159, 161, 224, 226–228, 234, 235, 237, 287, 321, 323, 379, 381, 382
structure diagram, 159, 162, 222, 235
TUI design, 280, 285
TUI design evaluation, 283, 286
SD top-down, 167, 242, 287, 326, 384, 422, 497
cohesion, 171, 246
coupling, 171, 246
data-flow diagram, 168, 243
ERD, 498
Fernandez patterns, 424
GoF patterns, 423, 424
GUI design, 327
GUI design evaluation, 328
hierarchy chart, 168, 243
information hiding, 171, 246
LDM, 498
performance, 171, 246
Rdb, 498
requirements, 167
security, 171, 246, 384
security design principles, 384, 385
simplicity, 171, 245
statechart, 169, 291, 327
structure chart, 169, 244, 291, 292, 327
TUI design, 289
TUI design evaluation, 291
XML, 499
Security
additional concepts, 352
cryptography, 354
data input validation, 93
data output validation, 93
design principles, 352
exception handling, 93
fail-safe defaults, 93
goals, 352
information assurance, 92
information security, 92, 93
java, 366–370
NIST Cybersecurity Framework, 92
python, 380, 382, 383
type-safe languages, *see* type-safe languages
Security design principles, 352
be reluctant to trust, 354, 356, 357, 372, 385
complete mediation, 353, 356–358, 372, 385
defend in depth, 354, 356–358, 371
economy of mechanism, 353, 356–358
fail-safe defaults, 353, 356–358, 371, 384
least common mechanism, 353, 358, 372, 385
least privilege, 353, 357, 358, 372, 385
open design, 353, 356–358
promote privacy, 354, 356, 357, 371, 385
psychological acceptability, 353, 358, 371, 384
secure the weakest link, 354, 358, 372, 385
separation of privilege, 353, 356–358, 371, 384
use your resources, 354, 359
Software assurance, 334
Software design approach, 6
bottom-up, 7
data-driven, 8
hybrid, 9
object-oriented, 8
process-oriented, 8
structured, 9
top-down, 6, 147, 167
Software design artifact, 4, 506
architecture, 506
components, 509, 510
data, 507
four design categories, 4
interfaces, 509
outline, 506
Software design characteristics, 127
cohesion, 131, 143, 164
coupling, 129, 142, 163
information hiding, 132, 145, 165
performance, 132, 145, 166
security, 132, 146, 166
simplicity, 129, 141, 161
Software design document, *see* software design artifact
Software design models, 2, 133
behavior, 3, 22
class diagram, 21
data-flow diagram, 149, 168
hierarchy chart, 21
IDEF0 function model, 138, 158
Nassi–Shneiderman diagram, 22
package diagram, 138
statechart, 23
structure, 3, 21
structure chart, 159
structure diagram, 159
UML communication diagram, 139
Software design patterns, *see* design pattern

Software development process, 2, 11
agile, 13
common steps, 11
incremental, 13
phases, 2
waterfall, 12
Software quality, 334
Software quality assurance, 334, 335
customer survey, 342
formal review, 335
goals, 345
informal review, 341
software testing, 342
software validation, 335
software verification, 335
summary, 344
walkthrough, 341
Software testing, 342
acceptance, 344
integration, 343
summary, 345
system, 344
unit, 343
SP case study
C++, 42, 46, 48, 79, 83, 87
data input validation, 116
data output validation, 120
design for reuse, 45, 47, 51, 85, 89
design only what is needed, 45, 47, 51, 85, 89
exception handling, 121
fail-safe defaults, 122
hierarchy chart, 43, 47, 50, 80, 85, 88, 117, 119
memory performance, 81, 86, 89
Nassi–Shneiderman diagram, 43, 50, 80, 118
python, 42, 45, 48, 78, 82, 86, 116
separation of concerns, 44, 47, 51, 85, 89
statechart, 43, 50, 80, 119
time performance, 81, 86, 89
SQL, 460
data definition language, *see* DDL
data manipulation language, *see* DML
DDL, 461
DDL example, 461, 478, 492
DML, 463
DML example, 463–466
embedded, 466
java, 466–470
python, 470–473
Statechart, 23, 30, 37, 43, 50, 65, 80, 108, 119, 140, 142, 150, 162, 169, 190, 197, 225, 232, 268, 274, 286, 291, 306, 313, 320, 327, 362, 364, 376, 378
Structure chart, 159, 161, 169, 224, 226–228, 234, 235, 237, 244, 287, 291, 292, 321, 323, 327, 379, 381, 382
Fernandez Symmetric Encryption, 421
GoF Command, 418
GoF Facade, 417
Rdb data store, 494, 496
XML data store, 494, 496
Structure diagram, 159, 162, 222, 235
Structured query language, *see* SQL

T
Text-based user interface, *see* TUI
Time complexity, *see* program design criteria, time performance
TUI
ABA design, 262, 266, 280, 285
ABA design evaluation, 265, 267, 283, 286
class diagram, 268, 275
design alternatives, 262, 280
statechart, 268, 274, 286, 291, 362, 364, 376, 378
structure chart, 287
UML communication diagram, 269
Type-safe languages
Java, 94
not C++, 97
Python, 96

U
UML communication diagram, 139, 141, 150, 185, 186, 188, 198, 199, 209, 210, 269, 308, 311
UML state machine, *see* statechart

W
Walkthrough, 341
case study example, 347

X
XML, 444
document object model, *see* DOM
example, 444, 445, 447
hierarchy chart, 477, 491
java, 447, 448, 450
python, 451, 452, 454
tags, 444, 477, 492