

Chenglie Hu

An Introduction to Software Design

Concepts, Principles, Methodologies,
and Techniques

An Introduction to Software Design

Chenglie Hu

An Introduction to Software Design

Concepts, Principles, Methodologies,
and Techniques



Springer

Chenglie Hu
Computer Science
Carroll University
Waukesha, WI
USA

ISBN 978-3-031-28310-9

ISBN 978-3-031-28311-6 (eBook)

<https://doi.org/10.1007/978-3-031-28311-6>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed 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

Preface

According to the Accreditation Board for Engineering and Technology (or ABET as commonly known), software design is one of the core subjects of computer science alongside data structures, computer architecture, algorithms, and programming languages. Yet, unlike the other core areas, there is no widely accepted “table of contents” for software design that can be used for curriculum development. As a result, what software design may be or entail largely depends on the interpretation of an instructor. There are a variety of ways how design is taught in a computer science program, including a learning of the Unified Modeling Language, a learning about design patterns, or simply a chapter-worth coverage in a software engineering course. Though students may have learned pieces of software design in various courses even in elementary programming courses, such fragmented coverages may not always be coherent and consistent. Computer science education research has shown that graduating computer science students may lack necessary software design knowledge and skills due to inadequate design education students have received.

After all, software design is not a well-defined discipline in terms of its foundation (concepts and theories), a defined body of knowledge that allows foundational research to be carried out, its terminologies, its research methods, and, perhaps most importantly, some institutional manifestation in the form of subjects taught at colleges and universities. In fact, we might not even agree on what appears to be a basic question: What is software design? In the extreme, a software design may be fully documented like one we may see in a structural engineering discipline, or it may only exist in code artifacts. But most likely in practice, we often create critical yet often incomprehensive design artifacts for a variety of reasons to communicate about our design thoughts and facilitate the code construction. On the one hand, a design must work logically; thus, doing design might sound like doing some sort of mathematics (though whether both share similar ways of thinking remains debatable). On the other hand, there are many ways we can design a program, and we eventually must choose one. Thus, doing design might be considered a form of art with an “aesthetic” value in the viewpoint of a designer. But at all events, design is unlikely an engineering act or thing because we may start constructing the software when we do not

even have an architecture of the software ready. Regardless of what software design may be, its importance in software's long-term success is undeniable. Despite these conceptual and philosophical challenges, it appears clear that we need a consistent design curriculum in computer science programs to better prepare students for real-world challenges in software development.

Most design books are professionally oriented, focusing on a design process such as continuous design, emergent design, domain-oriented design, agile modeling, etc. They were not intended to be used as college textbooks. Meanwhile, there have been some design textbooks but most with also significant coverage on programming. They vary in style, depth, and topical focus. This diversity seen in existing textbooks may have created difficulties for instructors to choose an appropriate textbook as the author experienced when he started teaching the subject more than a decade ago. The author's exploration in teaching of design started with this fundamental question: What is an appropriate body of knowledge about software design that every computer science student must possess? The Guide to the Software Engineering Body of Knowledge (published by IEEE Computer Society in 2004) specifies what constitutes body of knowledge for software design. The backbone of this knowledge seems to be the design-enabling techniques, which, according to the Guide, include design principles, guidelines, and key notions and concepts in regard to design abstraction, decomposition, modularization, and module cohesion and coupling. Also based on the Guide, some essential knowledge about software architecture should also be included. Therefore, the design-enabling techniques, software architecture, and design of larger software elements (necessary for understanding software architecture) may constitute, in the author's viewpoint, the core of the essential knowledge about software design. This understanding has guided the author to design and improve his teaching every year, leading to the formation of the book.

Chapter 1 is dedicated to an exploration of what software design may be and entail. Design is a way of thinking; therefore, “design thinking”—a widely publicized mode of critical thinking in the recent years—is prompted in the book. Readers are expected to have appropriate knowledge in data structures. However, it is not assumed that readers have also gained good understanding about the paradigm of object orientation. Thus, Chaps. 2 and 3 are designed to help readers better understand object orientation and the essentials of object-oriented design. Methods, standalone or being part of larger units, are essential program units, and design of methods may significantly impact the design attributes of enclosing modules, subsystems, or even the entire system. Chapter 4 is dedicated to design of methods. Object-oriented design is still the primary design paradigm in the real world; therefore, a good portion of the book also includes topics about design of objects, modeling with the Unified Modeling Language, and use of design patterns (Chaps. 5, 6, and 8, respectively), though the coverage of these topics has a focus on intrigued ideas behind the use of the concepts, techniques, tools, and patterns. Despite object orientation being still the center of attention, there is a strong promotion throughout the book that software design should consider all appropriate design paradigms and methodologies. Larger software elements are often directly responsible for the formation of software architecture.

Chapter 7 covers four kinds of larger software elements—libraries, components, frameworks, and microservices—and their architectural implications. Chapter 9 finally brings the presentation of software design to an end with a coverage on software architecture. However, after a well-proportioned discussion on what software architecture may be, the focus of the chapter is on software architectural views. Case studies are important in learning design. But as the author observed, small design examples may be more effective for students to see the applications of design-enabling techniques than larger case studies where we tend to integrate as many such techniques as possible with possibly insubstantial scenarios and missed learning opportunities in exploring diverse design choices and trade-offs. Chapter 10 then gathers some relatively small design case studies that can be used in earlier chapters as appropriate. These case studies can be extended in different ways to provide additional design opportunities. Besides, Chaps. 5 and 10 also include, as exercises, a variety of larger design questions that can be used partially or entirely when covering the respective contents. This book is an attempt by the author to explore a possible software design curriculum that can be used across computer science programs.

The book has adequate materials to be used in a four-credit course on software design or in a three-credit course with certain contents in Chaps. 7–9 left out. Other academic programs such as software engineering, computer engineering, or professional software development may also use the book as either a primary textbook or as a reference. The depth of the book makes it also appropriate for a design textbook at beginning graduate level. It is also appropriate to use the book as a primary reference in a software engineering course. Professionals may also find the book useful in their professional development.

For teaching support materials, please contact the author at chenglie.hu@gmail.com.

Finally, the author is grateful to his wife and other family members for their continued understanding and support during the course of writing. He is also indebted to his students for their efforts in learning design and for their feedback over the years. The invaluable learning outcomes they have demonstrated over the years have not only helped the author improve his own understanding about software design but also become ingredients of the book in many ways.

Lastly, but not the least, the author would like to thank the Springer staffers for their encouragement and technical support throughout the course of this project, and particularly for the excellent editing work.

Waukesha, WI, USA

Chenglie Hu

Contents

1	What Is Software Design?	1
1.1	Overview	1
1.2	The Nature of Software Design	1
1.3	Software Design in the Context of Software Lifecycles	5
1.4	Software Design in the Context of Analytical Thinking	6
1.5	Software Design in the Context of Communication	8
1.6	Software Design in the Context of Design Formalism	12
1.7	Summary	14
	References	17
2	The Paradigm of Object Orientation and Beyond	19
2.1	Overview	19
2.2	What Is Object Orientation?	19
2.2.1	Data Abstraction	20
2.2.2	Object Types	23
2.2.3	Inheritance	24
2.3	The Paradigm of Object-Oriented Design	26
2.4	Embracing Multi-paradigm Design	29
2.5	Summary	32
	References	35
3	Essentials of Object-Oriented Design	37
3.1	Overview	37
3.2	Data Type, Data Abstraction, and Type-Safe Practices	37
3.3	More About Interfaces	41
3.3.1	Software Sustainability	41
3.3.2	The Role of Interfaces Through the Lens of Data Structures . . .	42
3.3.3	Programming to an Interface, Not to an Implementation	45
3.3.4	Interface Segregation	48
3.3.5	Section Summary	50

3.4	Abstract Classes and Design of Type Hierarchies	50
3.4.1	Use of Abstract Classes	50
3.4.2	A Case Study	53
3.4.3	Section Summary	55
3.5	When to Avoid Inheritance	56
3.6	Subtyping with Consistent Object Behavior	59
3.6.1	Representation Invariants	59
3.6.2	Liskov Substitution Principle	60
3.6.3	Design by Contract	62
3.7	Lazy Object Creation Allowing Delayed Decision-Making	64
3.8	Object-Oriented Design in the Large: Design Principles	66
3.9	Summary	69
	References	73
4	Design of Methods	75
4.1	Overview	75
4.2	Essential Characteristics of a Method	76
4.2.1	Procedural Abstraction and Modularity	76
4.2.2	Design Attributes of a Method	78
4.3	Cohesion of a Method	79
4.4	Method Coupling	82
4.4.1	The Phenomenon of Coupling	82
4.4.2	Effects of Coupling	85
4.4.3	Categorization of Coupling	87
4.5	Module Redesign and Code Refactoring	88
4.6	Method Specification	90
4.6.1	The Nature of a Module Specification	91
4.6.2	Method Specification with Some Formalism	93
4.7	A Case Study: Overriding “Equals”	96
4.8	Summary	98
	References	102
5	Design of Objects	105
5.1	Overview	105
5.2	The Context and Process	105
5.3	Essentials of Object Design	107
5.3.1	Object to Model One Thing	107
5.3.2	Diverse Object Design Possibilities	108
5.3.3	Prototyping Object Interaction	110
5.3.4	Designing Objects Around a Structural Style	110
5.3.5	More About Object Discovery	113
5.3.6	Design to Ensure Objects’ Behavioral Correctness	115

5.4	Design of Control Objects	116
5.4.1	Highly Centralized vs. Coordinated Controls	117
5.4.2	Process Control with a Framework	118
5.4.3	Controls for Event-Driven Systems	118
5.4.4	Different Control Roles	119
5.4.5	Objects Are Designed to Control	120
5.5	Object Cohesion and Coupling	121
5.5.1	What Are the Issues?	121
5.5.2	Law of Demeter	124
5.5.3	Objects with No Overlapping Behavior	125
5.6	Iterative Design of Objects	125
5.6.1	Initial Design of Domain Abstractions	126
5.6.2	Subsequent Design Validation and Refactoring	127
5.7	Summary	129
	Further Reading	136
6	Software Modeling Languages and Tools	137
6.1	Overview	137
6.2	Software Analysis and Modeling	138
6.3	Developing Effective Use Cases	141
6.3.1	Use-Case Scenarios	141
6.3.2	Use of Use Cases	144
6.3.3	Development of Use Cases	146
6.3.4	Use-Case Diagrams	149
6.4	Other UML Diagrams	151
6.4.1	Class Diagrams	151
6.4.2	Sequence Diagrams	154
6.4.3	State (Machine) Diagrams	159
6.4.4	Activity Diagrams	168
6.5	Use of UML Diagrams	171
6.6	Dataflow Diagrams	174
6.7	Modeling with Customized Diagrams	179
6.8	Summary	181
	References	186
	Further Reading	186
7	Design of Larger Software Elements	187
7.1	Overview	187
7.2	Software Interfaces and APIs	188
7.3	Characterization of Larger Software Elements	189

7.4	Design of a Library	192
7.4.1	Software Libraries of Different Kinds	192
7.4.2	Characteristics, Benefits, and Risks of a Library	194
7.4.3	Initial Design of a Library	196
7.4.4	Design of a Library's API	197
7.5	Design of Components	198
7.5.1	Component Concepts, Structures, and Models	199
7.5.2	Component Design	205
7.5.3	Component Discovery	208
7.5.4	Component Composition	210
7.5.5	Component-Based Architecture	213
7.6	Design of Application Frameworks	216
7.6.1	Characteristics of an Application Framework	216
7.6.2	Framework Design	218
7.6.3	A Case Study	219
7.7	Microservices	221
7.7.1	Some Background Information	222
7.7.2	The Promises of Microservices	224
7.7.3	Internal Operations of a Microservice	225
7.8	Summary	226
	Further Reading	230
8	Software Design Patterns	231
8.1	Overview	231
8.2	Creational Design Patterns	232
8.2.1	Singleton	232
8.2.2	Abstract Factory	233
8.2.3	Builder	234
8.2.4	Prototype	236
8.2.5	Section Summary	237
8.3	Structural Design Pattern	238
8.3.1	Adapter	238
8.3.2	Proxy	239
8.3.3	Façade	241
8.3.4	Decorator	242
8.3.5	Composite	244
8.3.6	Bridge	245
8.3.7	Flyweight	246
8.3.8	Twin	247
8.3.9	Section Summary	248

8.4	Behavioral Design Patterns	250
8.4.1	Strategy, Servant	250
8.4.2	Command	252
8.4.3	State	253
8.4.4	Visitor	257
8.4.5	Iterator	259
8.4.6	Chain of Responsibility	260
8.4.7	Observer	262
8.4.8	Mediator	264
8.4.9	Section Summary	265
8.5	More About Object Communication	266
8.6	Summary	269
	References	275
9	Software Architecture and Architectural Views	277
9.1	Overview	277
9.2	What Is Software Architecture?	278
9.2.1	A Characterization of Software Architecture	279
9.2.2	Architectural Styles	280
9.2.3	Structural Layers of Software Elements	282
9.2.4	The Impact of Software Architecture	284
9.2.5	Architectural Views	286
9.3	The 4+1 Architectural View Model	289
9.3.1	An Overview of the View Model	289
9.3.2	The Logic View	291
9.3.3	The Process View	292
9.3.4	The Development View	294
9.3.5	The Physical View	295
9.3.6	The “+1” View	297
9.4	The C4 View Model	298
9.4.1	The Layers of the C4 View Model	298
9.4.2	An Example	300
9.5	Effective Development of Architectural Views	303
9.6	Summary	307
	References	309
	Further Reading	310
10	Design Case Studies	311
10.1	Overview	311
10.2	Two Simple Case Studies	312
10.2.1	A Simple Service Application	312
10.2.2	A Generic Data Sorting Framework	315

10.3	A Card Game	317
10.3.1	Analysis	318
10.3.2	Two Abstract Classes	320
10.4	A Framework for Animation	323
10.4.1	Analysis	323
10.4.2	Two Examples of Using the Framework	326
10.5	An Analyzer for Frequencies of Words in a Text	328
10.5.1	Design Story and Use Cases	328
10.5.2	Analysis	329
10.5.3	More Details About Implementation	330
10.5.4	Details of the Application Control	332
10.6	Getting People to Sleep	333
10.6.1	Application Requirements	334
10.6.2	Use Cases	334
10.6.3	Logic View of the Design	335
10.6.4	Process View	338
10.7	Wheel of Fortune Game	342
10.7.1	The Requirements	343
10.7.2	A Use Case	344
10.7.3	First Three Views in the C4 View Model	344
10.7.4	The Code View	348
10.7.5	Console Output	350
	References	359



What Is Software Design?

1

1.1 Overview

Software design is elusive. It can more generally refer to code design, program design, or system design. It can also specifically refer to design at a functional level, at a communication level, or at a level to coordinate software operations. Having learned programming fundamentals, novices may be given guidelines of code design and start learning what might constitute a “good program.” When a program is written, there is a program layout of its elements. This “layout” is what the program has been “designed” whether the program author realized it. If program “design” is what we “naturally” do when we program just like we go about our daily lives according to our individual “design,” it can hardly be given a definition without controversy. When a program involves interactions with other programs and must evolve when the requirements of what a program should do are evolving, program design must be a conscious (mental) activity just like we must plan our day carefully when it is filled with events, travel, and meeting other people. While we might not be able to define software design in more accurate terms, we will, in this chapter, look at what software design means in several contexts.

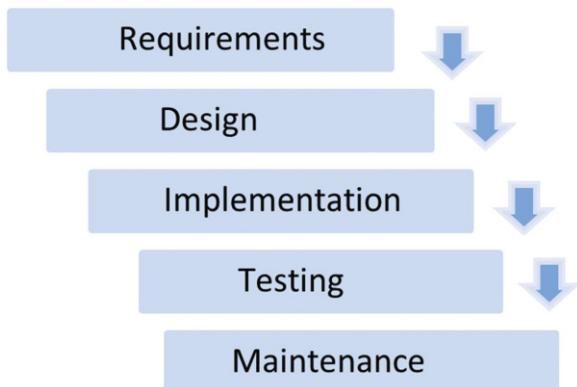
1.2 The Nature of Software Design

It is a widely held view that the most critical ingredient in ensuring a software system’s quality and long-term success is its design. While other concepts, concerns, activities, artifacts, and processes are important to the success of a software engineering project, it is the quality of a system’s design that provides the critical ingredient (Abstract of ICSE ’11). However, software design is perhaps one of the most debated computing concepts in terms of what it is and how it is done. Professionals often have different views about software

design even on how much design they need to perform in a software development process. One thing about software design that few have addressed in the past is how a design commences and evolves. In other words, is there a defined design process that professionals follow or does design take off in one's mind and have its concrete, precise form only in code? When one is uncertain about a process in terms of its nature, things to produce, and, more importantly, its activities, the process would likely be complex. Thus, it is fair to say that designing software is complex, which may then partially explain why professionals have different views on software design, and few would even venture giving a definition. In principle however, professionals all agree that a good design should satisfy software requirements and provide structural flexibility, mobility, and extensibility. But how?

Designing software is unlikely a typical engineering process of any kind. First, software design has no scientific laws and principles to obey. Designers should, yet not be obligated to, follow software design principles and guidelines. Second, an engineering design must be completed before construction takes place. By contrast, a software design is most likely an incremental product due, primarily, to evolving software requirements. Third, in contrast to an architectural design, which is often vital for an engineering product, software professionals have different views about whether there is even a need for a sustainable software architecture in the first place when requirement change is expected in software development. Like an engineering product, however, a software product would go through a cycle of requirements gathering, analysis and design, software construction, testing, delivery, and maintenance before the software is updated, modified, or discontinued in its service—this is known as the software lifecycle. Figure 1.1 depicts a software development process in this software lifecycle where software design is a phase of the process. The process is traditionally known as a waterfall process, where software requirements are analyzed, specified, embedded in design, and realized in software prototypes and in final product in a “waterfall” fashion. A traditional “waterfall” assumes that software requirements would be correct and complete before design begins. Thus, the design as well as the construction and testing could be proceeded in an engineering way. However, it is widely recognized today that this assumption is unrealistic and, in fact, false, and a

Fig. 1.1 The waterfall process of software development



requirement engineering process must be applied iteratively to the entire software lifecycle to make the software usable and sustainable. Thus, software design is a development activity that we must engage in throughout the software lifecycle and is interleaved with other development activities to form a coherent activity cohort. For the same reason, development phases in any process model overlap with no clear boundaries, paving the way for collaborative development activities. Therefore, the nature of software design must be understood in each development context, from different perspectives, and with meanings that are consistent with the needs for a design in any development process.

In an article published almost 30 years ago (Buchanan, 1992), Richard Buchanan termed design to be a wicked problem—a class of social system problems that are ill-formulated where the information is confusing, where there are many clients and decision-makers with conflicting values, and where the ramifications in the whole system are thoroughly incomprehensible in such way that a solution for one aspect reveals an even more complex problem beneath. The notion of wicked problems first arose much earlier in the design involved in social planning problems such as urban renewal and demolition of communities, replaced by high-rise flats. More specifically, wicked problems are characterized by:

1. There are no definitive formulations, only ideas on how to solve the problems.
2. There are no indications when a solution has been reached, much fewer best solutions.
3. There are no “right” or “wrong” solutions, only “good,” “not so good,” or “bad” solutions.
4. Resolving one issue in a design may pose new ones.

These characteristics reveal the complex relationship between determinacy and indeterminacy in design thinking—a thinking process that tackles ill-defined or unknown problems with varied interpretation of the thinking elements involved. The linear model of design thinking is based on determinate problems that have definitive conditions. The designer’s task is to identify those conditions precisely and then calculate a solution. In contrast, the wicked-problems approach suggests that there is a fundamental indeterminacy in all but the most trivial design problems. Indeterminacy is very different from undetermined in that indeterminacy implies that there are no definitive conditions or limits to design problems, whereas undetermined or under-determined requires further investigation to make them more determinate. Design problems are “indeterminate” (or “wicked”) fundamentally because design has no special subject matter (potentially universal in scope) of its own apart from what a designer conceives it to be. Designers may discover or even create a subject out of the problems or issues of specific circumstances. This sharply contrasts with the disciplines of science where things are what they are governed by the scientific principles, laws, and rules, not how they are conceived to be. The history of design is more likely a record of the design historians’ views regarding what they conceived to be the subject matter of design.

Like Donald Knuth who thought computer programming is an art (Knuth, 1974), if we think of design from the perspective of art, then design may be viewed as a discipline of systematic thinking with its own technological methods in communication, construction, strategic planning, and systemic integration. It may then also be plausible to consider design, regardless of a context, as a process of planning and well-intentioned execution of a working hypothesis—a significant factor in shaping humane experience. As design thinking continues to expand its meanings and connections in contemporary culture and in our social and economic lives, perceiving the existence of such an art only opens the door to further inquiry to explain what that the art is, how it operates, and why it succeeds or fails in certain situations. The challenge is to gain a deeper understanding of design thinking so that more cooperation and mutual benefit is possible between subject matters and those who apply design thinking to remarkably different problems. This perspective will help to make the practical exploration of design more intelligent and meaningful.

Even though designing software is unlikely an engineering process in a traditional sense, it shares some commonalities with other professional design processes. These are:

- Visual communication tools to effectively convey design ideas
- Commonly recognized forms of design for professional documentation
- A managed design process, no matter how elusive it might be

Like any other professional design, a software design process starts with requirement analysis where we explore design possibilities by applying thinking that can lead to divergent outcomes, which might be termed “divergent thinking” by professionals. Design analysis is to define design problem, in which the design elements are determined against design requirements and goals. In contrast, software architecture is a product of requirement synthesis, where we narrow down more probable design choices to lead to design decisions, which might be termed “convergent thinking” in contrast to “divergent thinking.” Design synthesis is a process in which the various possible pieces of design are combined and balanced against each other, yielding a final plan to be carried into construction. Design analysis and synthesis are two processes that are proceeded in an interleaved fashion as the designer deems helpful when focusing on a particular aspect of design or in resolving certain design issues. Software may be constrained by environmental conditions, memory usage, or scalability expectations. Therefore, in a divergent-convergent design thinking process, frequent weighing on design tradeoffs is also unique to software design, as a tradeoff is often a conscious decision based on the evaluation of the unity of a functioning and balanced whole.

Given the discussion above, while the true nature of software design might remain elusive, design appears to be a process that requires analytical and logical thinking, artistic intuition, endurance on uncertainty, and being accustomed to resolving local design issues while not losing sight on the overall architecture. We will further elaborate below several contexts in which we look at software design as an activity, as a way of thinking, as an expression of artifacts, and as an underlying representation of computing.

1.3 Software Design in the Context of Software Lifecycles

Software lifecycle starts with software requirement elicitation and waterfalls through the phases depicted in Fig. 1.1. The process of a lifecycle is a way of managing the execution of the (waterfall) phases with a common goal to deliver the software product with pre-specified quality attributes. A lifecycle process model is featured by its underlying philosophy, the phase-wise activities, and an organization of the development artifacts. Each process model has its distinctive benefits and inevitable limitations or even drawbacks. While all process models characterize how a development would proceed through the “waterfall” phases, one traditional process model is literally labeled *Waterfall Process Model*—a documentation-driven process with defined goals, milestones, and deliverables for each development phase, mimicking a traditional engineering process. In the Waterfall Model, design is a phase that has defined time span with little overlap with a timeframe for software construction. Design would be formally documented with a widely recognized standard such as IEEE Software Design Descriptions (IEEE 1016, 2009). The recommended descriptions are organized into one or more design views (which we will discuss in Chap. 9). However, this documentation-driven process has long been questioned for its impracticality with, as many argued, a flawed underlying assumption that elicitation of complete and correct requirements is achievable. Meanwhile, several separate developments of an object-oriented approach to software development led to a unification of process models known as *Unified Process* (UP), which recognizes the vulnerability of software requirements with its own phases. The UP Model also features software modeling with modeling tool *Unified Modeling Language* (UML). For a brief account, the UP Model has four phases: *Inception*, *Elaboration*, *Construction*, and *Transition*. The “waterfall” workflows go across all phases over iteration cycle (Fig. 1.2). The

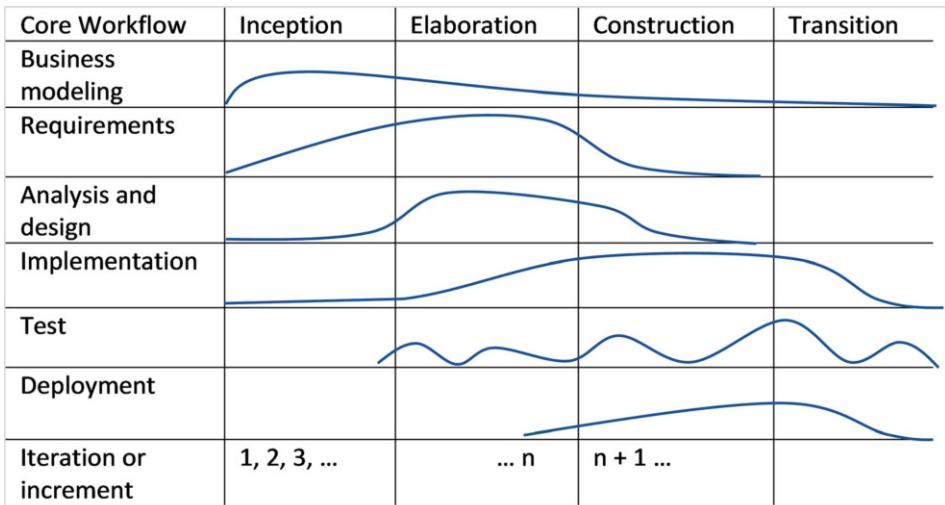


Fig. 1.2 Illustration of unified process

UP is an iterative and incremental development process with four phases divided into a series of time-constrained iterations. Each iteration results in an incremental release of the software containing added or improved functionality. Figure 1.2 also describes the amount of each workflow needed in different phases. A simple fact of the UP process is that though most iterations need all workflows, the relative effort and emphasis will change over the course of the project. The UP Model is said to be an architecture-centric process with software architecture being at the heart of the development to shape the system. Executable architecture baseline, once created during the Elaboration phase, is one of the most important deliverables of a UP process. Incremental partial delivery of the system serves as validation cycles for the architecture. Software design with the UP Model takes place as early as during Inception phase when software feasibility is thoroughly studied. Design activities proliferate during Elaboration phase and continue in the Construction phase and even after Transition phases starts.

In contrast, agile software practices promote a lightweight process such as the *Scrum Model*, featuring practices of continuous or evolutionary design without necessarily a defined architecture. The underlying assumption of “agile design” is that software requirements will change, and the architecture will self-adjust to accommodate incremental design pieces through design and code refactoring—making informed modifications to the design or code as needed. This fundamental departure from traditional, more engineering-inclined design practices has made some wonder whether “Design Is Dead” (Fowler, 2000) or to what extent the traditional design process and practices are still relevant.

Despite the controversy of design in relation to its role in a software lifecycle, actual design practices may likely vary in organizations depending on a local software development culture. What seems more agreeable is that design uncertainties are driven by instability of software requirements; thus, design architecture needs to capture such uncertainties as much as practically possible in a requirement engineering process. Ideally, an architecture should be stable enough to allow meaningful development of components to begin, but fluid enough to allow architectural extension and calculated structural mobility. Arguably, letting an architecture drift as it needs to can be risky for large projects. The wickedness of design cannot be cured, but can only be tamed with requirements engineering, principled design architecting, and software prototyping to test, verify, and ensure structural sustainability. A lifecycle model can impact approaches to software design. But any approach we use to design software is likely interleaved with other lifecycle activities we engage in with technical, economical, and social consequence. Design outcomes bear such consequences too.

1.4 Software Design in the Context of Analytical Thinking

Professionals have long suspected that designing software might well be a cognitive process that happens inside one’s brain at a speed faster than lightening, and the essence of design, then, is a rapid modeling and simulation process that is proposing solutions and allowing them to fail (Glass, 2006). People have also studied how design really works in

one's brain. They found that thinking about a design is often fixated at a pair of design aspects, each taking turn to be focused on as the designer weighs pros and cons of design tradeoffs before deciding (Baker & Van der Hoek, 2010). Other studies suggested that good designers can maintain sight of the big picture by engaging in systems thinking. They can tolerate ambiguity, navigate through uncertainty, think, and communicate in the several design languages (Dym et al., 2005). Furthermore, designers ask generative questions allowing design options to diverge from facts to the possibilities and deep reasoning questions attempting to converge on the facts again with design decisions. This iterative divergent-convergent thinking process might constitute the core of design thinking, applicable to many engineering design undertakings and even social and economic decision-making processes. Based on these findings, we might characterize software design, with a more abstract perspective, as *a systematic, intelligent process in which designers generate, evaluate, and specify software concepts and elements with their forms and functions satisfying the software requirements.*

Learning to effectively use design paradigms, principles, and proven technical practices is essentially about learning design thinking. Software analysis is where design thinking may start on a potential software model based on the software requirements. This model, with several baseline structural options, is a mental product of high-level design decisions. However, to create sustainable architectures, divergent thinking may also inevitably dive deep into programming to provide mental feedback on perspectives of the tradeoffs. In other words, design thinking may help us assure a conformance between the architecture and its code realization.

Researchers have recognized the importance of reducing the cognitive distance between the concepts that may exist in our minds and those that are realizable in the implementation medium of computing such as objects. On the one hand, divergent thinking may allow us to evaluate the impact of applying design patterns and other design methods on potential components and their interactions that are more easily and predictably realizable in our implementation mediums. On the other hand, convergent thinking may help us decide by applying appropriate programming paradigms to weigh the possibilities and bring what's implementable to the level of components we conceive mentally. This process of design thinking is not linear, and the cognitive distance always exists and must be traversed during a divergent-convergent thinking process. Figure 1.3 summarizes and further illustrates the process of design thinking.

“Design thinking” has been a popular concept in the recent years, though in a much more general context. It refers to a set of cognitive, strategic, and practical procedures and the body of knowledge for reasoning when engaging with design problems. As an iterative, non-linear process, design thinking includes activities such as context analysis, problem framing, ideation and solution construction, prototyping, user testing, and evaluating. In many business and social contexts, problems that need design of solutions are also ill-defined or “wicked.” For such problems, innovation and creativity are often required to create a natural flow from research to product rollout. Some characteristics of design thinking include brainstorming, immersion in the customer experience, transforming data into insights, and preparing prototypes for real-world experiments.

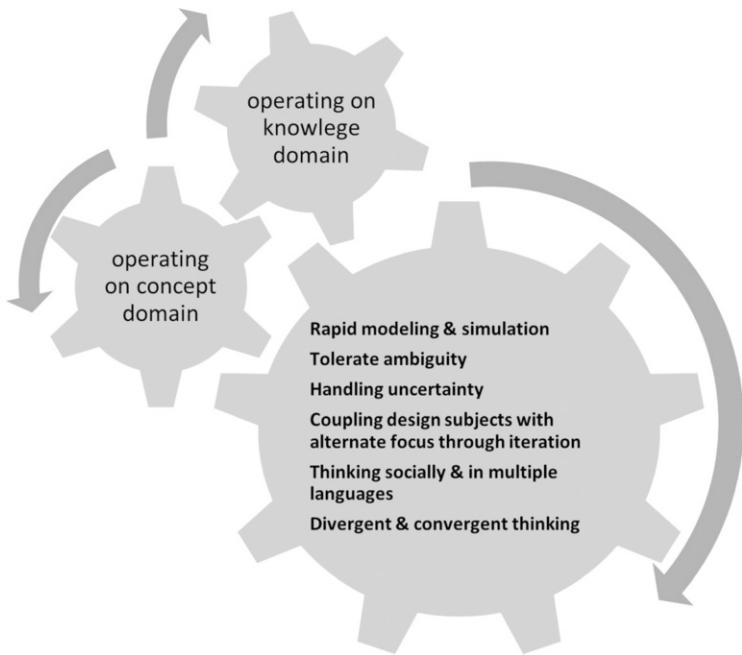


Fig. 1.3 Illustration of design thinking

Exploiting a collective, cognitive strength of human beings who are motivated by varying perspectives and emotions, design thinking emphasizes engagement, dialogue, and learning. By involving various kinds of stakeholders in processes of developing solutions, design thinking embraces change, facilitates collaboration, and overcomes difficulties and constraints. This paradigm of thinking is consistent with design thinking we discussed earlier with perhaps a different manifestation. In a business or social context, design thinking is often a collective effort. While collective design effort is common in software design too, design of a software architecture often requires consistent and coherent thinking expected of individual designers with high cognitive loads.

1.5 Software Design in the Context of Communication

A modeling language such as the UML aids, not replaces, a thinking process, though the way the UML is used in design, particularly in a thinking process, is still unclear as studies have found. As discussed earlier, design is primarily a mental process. Thus, whether we use tools to assist a thinking process may be not essential. Modeling languages, however, help designers communicate the architecture to the stakeholders. According to one account among perhaps hundreds, software architecture is a three-aspect entity: software elements, the forms of their relations and interactions, and rationale of the design. These aspects must be effectively communicated to all stakeholders for them to understand the complexity of

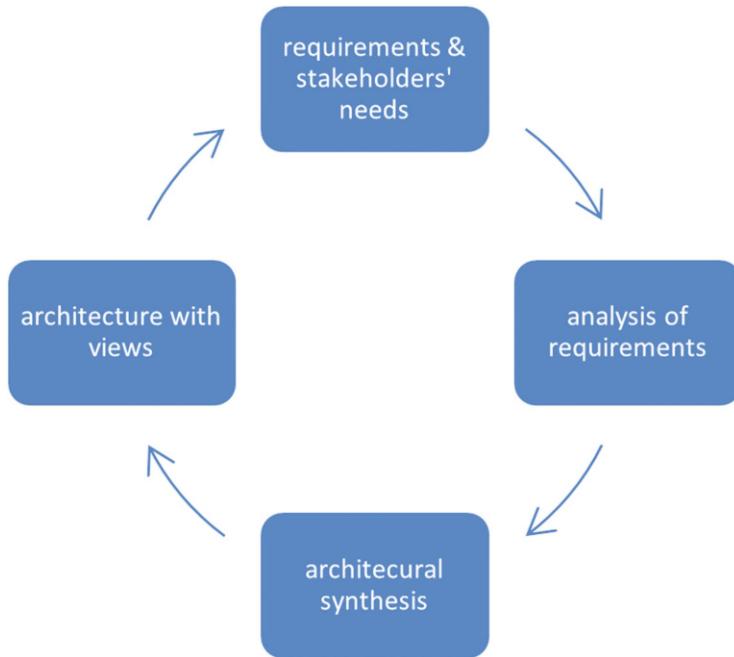


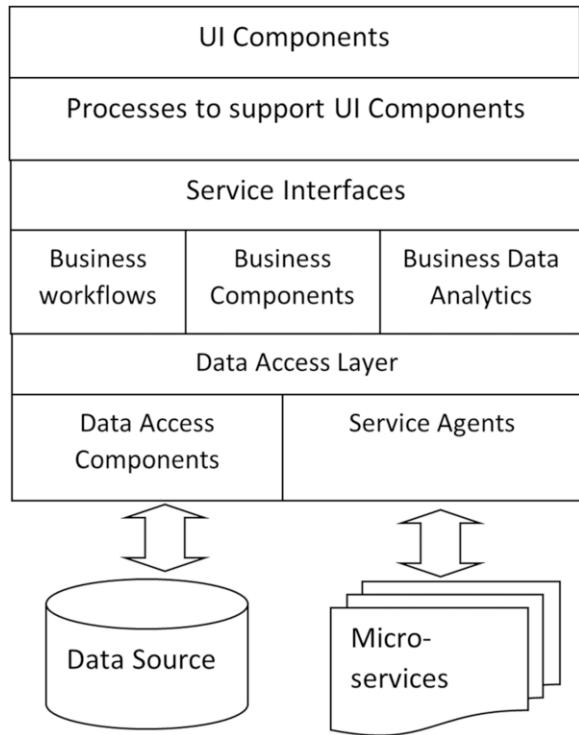
Fig. 1.4 Evolutionary process leading to software architecture and its validation cycles

the software. Architectural views are a primary means of communication with varied perspectives, customized for different stakeholders. These views must be updated in an iterative process (depicted in Fig. 1.4) to allow frequent feedback on the views, view updates, and construction of new views to satisfy the vested interests of all stakeholders.

There are many ways architectural views can be constructed (we will discuss in Chap. 9). For example, Fig. 1.5 is a layered view about system's components, data resources, and software service providers. This view allows visualization of how components communicate with one another or with resources. Despite its simplicity, such a view can be adequate to the management of an organization. In contrast, Fig. 1.6 is a flowchart about a self-checkout process of a physical store. This view describes not only how the software works procedurally but also how it communicates with other devices (such as a scanner) and external processes (such as a credit card provider). This view can be appropriate for a technical person who may play certain role in the development process.

Developers may appreciate architectural views by perspective or by layer of abstraction. Perspective views typically include a logic view about software elements, relations, and interactions to understand the software operation, a process view about software interactions with other software or hardware processes, and an organization view about software artifacts. In a layered approach to views, a view about the operational context of the software system constitutes the top layer. We would then “zoom in” on the system to view its element containment structures and further on compositions of the software

Fig. 1.5 An architectural view of component layers



elements. To effectively construct views, we use more focused views known as view packets to satisfy more specific interests of the stakeholders. For example, a database administrator might appreciate a view that includes all data objects, their containers, and immediate data clients. However, views and view packets alone may not provide a clear and adequate justification of the rationale for the architecture. Often, designers may also provide narratives about the design ideas behind the architecture and other possibilities that have been considered. In addition, software prototypes of various kinds constructed for various uses may also support the rationale for the architecture. In other words, designers must use conceptual, visual, verbal, or physical means to communicate with the stakeholders about the design, particularly the architecture of the software. Communications to the stakeholders are often critical for the validation of software requirements and further requirement elicitation.

In addition to providing communications with architectural views, a design is also capable of communicating directly about ways the software system is built. For example, if the software is essentially built on an enterprise-level software framework (such as those provided by Oracle), a design of the system may directly convey how exactly the development team would customize the framework to provide organization-specific functionalities. If, instead, a development framework is used as a means to construct the software, we must also communicate what design means within the capacity of the development framework.

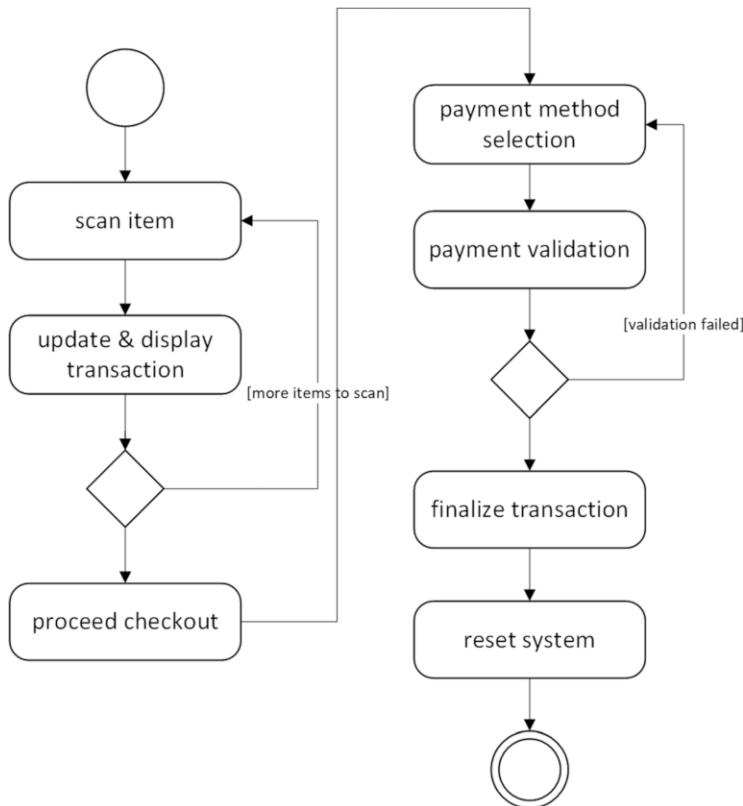


Fig. 1.6 A process flow of a self-checkout process

Figure 1.7 is an architectural view of a popular web application front-end development framework (and library) React, developed by Facebook. React uses a declarative paradigm that makes it easier to reason about application logic and more efficient to update and render components when data changes. React uses JSX (which is simple syntactic sugar) for the easy creation of JavaScript objects when working with the virtual document object model (DOM). These JavaScript objects would be rendered through the recursive creation of DOM nodes and appending the nodes to the actual DOM hierarchy. With this React framework, components design through composition may constitute the majority of the design work (so that resources, including the necessary training, can be appropriately arranged by the management).

Use of software frameworks that focuses on reuse may require a very different software development process. The designer must communicate to the stakeholders about the design scope, methods, and limitation to solicit administrative guidance and support.

Finally, software design can be documented when needed as a written communication for the client and developer to agree on a contract. A documentation can also help the stakeholders understand the scope of the software and its operations at a high level. There

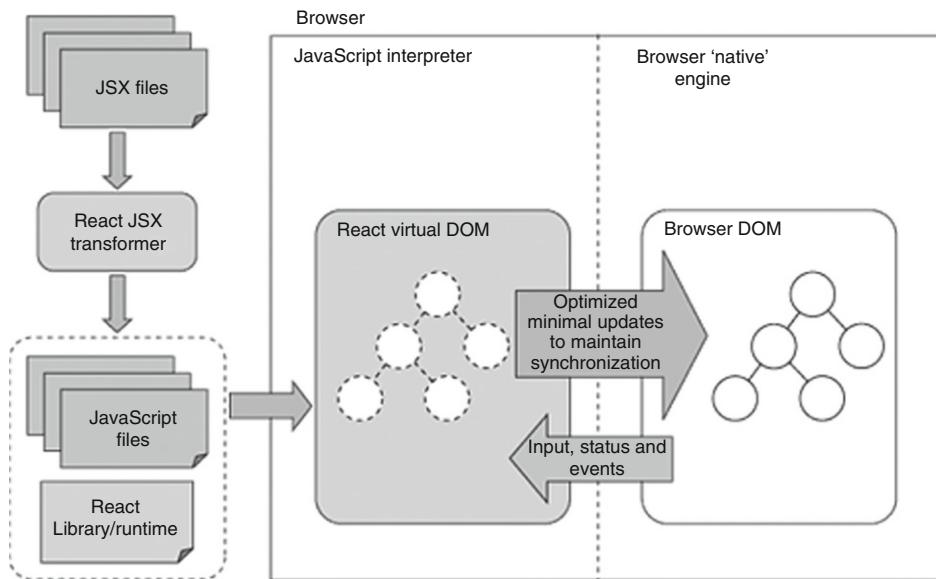


Fig. 1.7 React framework architecture

are professional standards (such as IEEE standard 1016-1998) that specify what to document about a design. A typical design documentation includes a design overview, the architecture, description of components, and a design of user interfaces. However, documenting a design can be controversial because of the potential that requirement changes may invalidate the design partially or even entirely, resulting in a waste of time and effort. Some argued that the only reliable communication mechanism about the design is the code (Reeves, 2005). Nonetheless, design is about communication in many ways, and effective communications about design sustain design efforts, effectiveness, and outcomes. To avoid pitfalls, we document design with effective risk mitigation strategies. For example, we may exclude documenting the design of lower-level modules as such modules may change frequently. We also want to communicate the design early to the stakeholders and frequently at least in the early phases of the development to ensure an effective process of requirement validation and consistent requirement elicitation efforts.

1.6 Software Design in the Context of Design Formalism

Design paradigms promote ways of thinking and influence design methodologies. Structured design, featured with procedural abstraction, was the primary design paradigm in the 1970s and 1980s. Object-oriented paradigm, based on data abstraction, has become dominant since the mid-1990s. The programming languages we choose to use directly affect our adoption of design paradigms. Even a language construct may be consequential in software design such as the well-known “GOTO” construct that led to a debate on its

controversial benefits in program design. The debate has lasted even to this day. Software today is complex for multiple reasons—complex functionality, user experience demands, being heterogeneous, and having to endure frequent updates to stay competitive. Consequently, software development today is often an endeavor using a multi-language environment involving technologies of different kinds. Older object-oriented languages have now embraced structural and functional constructs and libraries in their new releases that language designers failed to recognize in the first place. As a result, software design and construction are increasingly using paradigm-neutral approaches.

Contrary to a classical notion that software may retire someday, software today is built to last. Paradigm-neutral design approaches with a paradigm-rich set of language constructs may offer much more design possibilities than any other paradigm-specific design methodology would. More design possibilities and development technologies have fueled development efforts in building software frameworks, application programming interfaces (APIs), and infrastructures to support the creation of microservices (independently deployable software units). These advancements might allow us to effectively address the lack of codification in software development that we build similar applications repeatedly in different organizations at different times and repeatedly make the same mistakes in design and development. These advancements are also changing the way we build software and have inched us closer to making software design and development a “scientific” process.

Researchers have argued that an important step to make design more scientific is to make design a more formal process much like showing why mathematical theorems work. We use “formal software-building ingredients” such as APIs, frameworks, or general-purpose software services in a design effort much like we use propositions, theorems, equations, and formulas to construct mathematical proofs. More essentially, perhaps, designing software, like design of mathematical proofs, has great informality, where formal parts are embedded in an informal intellectual structure that gives them meaning. *Barbara Liskov*, a well-known computer scientist, was referring to her work on abstract data types in an interview: ‘I was interested in the underlying work. ‘How do you organize software?’ was a really interesting problem. In a design process, you’re faced with figuring out how to implement an application. You need to organize the code by breaking it into pieces. Data abstraction helps with this. It’s a lot like proving a theorem. You can’t prove a theorem in one fell swoop. Instead, you invent some lemmas, and you decompose the problem. In my version of computational thinking, I imagine an abstract machine with just the data types and operations that I want. If this machine existed, then I could write the program I want. But it doesn’t. Instead, I have introduced a bunch of subproblems—the data types and operations—and I need to figure out how to implement them. I do this over and over until I’m working with a real machine or a real programming language. That’s the art of design’ (Van Vleck, 2016).

How might design exploit “formal ingredients” of software in an informal software structural formation? Or is there a way we might better understand the inner workings of software structural formation? Computing is a process of sequence of information

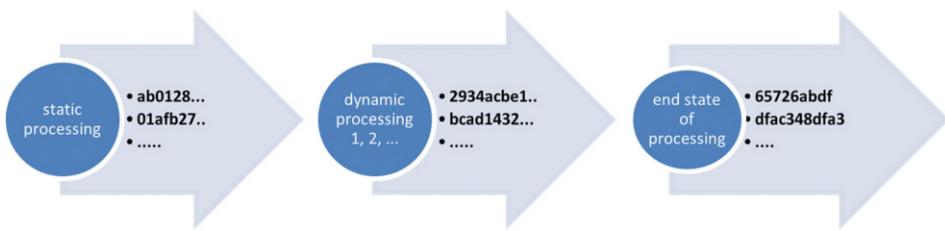


Fig. 1.8 State-wise representations of computing

representations that computer “understands” and may have several structural levels, from bits to bytes to groups of bytes to groups of groups of bytes, etc. Representations are operative and transformative in a way we designed (Fig. 1.8). By extension, design is as much for programming as it is for modeling the world and translating a portion of the world into different groups of bits and bytes at different levels by applying design-specific transformations. A design controls symbol transformation to capture functional operations of the software and information transformation from one layer to another. The truth is that different designs may all work when replaced with one another. In other words, the logic outcomes of different symbol transformations may be the same. It is interesting, therefore, to investigate the relations between a design and the inner workings of symbol representation and transformation to better understand, in a more formal way, how software design might work in abstract terms. When there are multiple ways to put pieces together, certain ways might be judged more “elegant” than others in the same aesthetic sense. Nonetheless, until we discover and develop schematic approaches to put software elements together, software design remains an art in assessing its artifact, in the way we think, and in the design process we manage. As software design continues to evolve in both theory and practice, discussions on making the design of software more of a formal process will likely remain at a philosophical level and stay only in the research community. What remains of practical value is a continued exploration about the ingredients of design thinking applied to software design.

1.7 Summary

We have discussed in this chapter about the nature of software design and what software design may mean in several contexts. Software design is carried out at an individual level in the context of analytic thinking. Learning software design is essentially about learning a way of thinking, though this ability of thinking may likely take a long time to develop. Software design is not an isolated activity and must be carried out and understood about its role and impact in a context of software lifecycle. In this context, design outcomes are affected not only by the designer’s cognitive strength but also by the software development environment and the culture of development teams in which design work is carried out. In a

sense, a design is mostly for current and potential stakeholders to read and understand, only one time for people to implement. Thus, communication about a design is as important as producing the design, if not more so. In this context, effective communication about design is an integral part of a design process. Though such communications may appear to be technical, they often have economic, social, and managerial consequences. Finally, to show something can be done or simulated with a computer program might conceptually look like a demonstration that certain mathematical proposition is true with a proof. But the contexts, in which mathematical proofs are created, are very different from those in which computer programs are developed. We can only show the correctness of a proof by the most general logical means, whereas we show a design works by implementing the design and by testing. It is more likely, however, that the elements of design thinking may apply to both software design and mathematical proofs. As software design becomes more component-oriented, framework-oriented, service-oriented, or API-oriented, we will move closer to achieving a design formalism. Until then, software design remains a work of art, and the work must be carried out with contextual influences we discussed in this chapter.

Exercises

1. Compare and contrast between developing a piece of software and building a physical bridge.
2. What would be similarities and differences between code design of a program and planning for an extended vacation trip?
3. A “waterfall” process refers to a forward-only progression of software development from requirements elicitation, design, implementation, testing, to maintenance. How might you apply the “waterfall” process to your problem-solving no matter how simple the program might be?
4. What would be potential issues you might encounter if you solve a program without planning prior to coding your solution?
5. Would design for solving a scientific problem with a computer program a wicked problem? Why or why not?
6. Suppose you are designing a website for anyone to post stories, categorize them with hashtags, and search for stories using keywords. Would that be a wicked problem? Why or why not?
7. What makes software design different from writing a program or coding?
8. How would we assess the quality of a software design?
9. Suppose you are designing a program for playing paper-rock-scissors game; what might be your design considerations prior to implementing them?
10. As Fig. 1.2 shows, the analysis-and-design workflow of the UP Model generally starts late in the Inception phase, but quickly reaches plateau and remains relatively plateaued in the entire Elaboration phase, and then becomes subsiding and (gradually) diminishing once the Construction phase starts. Is this workflow pattern justified?

11. It is said that to produce an innovation, people ask questions, consider diverse voices on something that doesn't exist, get out of one's comfort zone, fuel imagination with inspiration one can find throughout the day, build prototypes to test ideas, and communicate ideas to stakeholders to gain support. How are these activities applied to software design (which is often innovative)?
12. Design thinking has gained popularity in recent years. There are many online discussions about what design thinking is in terms of its much broader applicability. Essentially, design thinking is said to involve the following activities:
 - Empathize (with users and clients).
 - Define (users' needs, their problem, and designer's insights).
 - **Ideate** (by challenging assumptions and creating ideas for innovative solutions).
 - Prototype (to start creating solutions).
 - **Test** (solutions).
 - (a) How would these activities translate into activities in software design, if applicable?
 - (b) What software design activities might you see that are not contained in this list?
 - (c) How would you reconcile this list with the list in Question 11?
 - (d) Make a similar list for designing software.
13. Choose a recent day of your life that was more eventful, and then, like Fig. 1.5, create a diagram to describe the activities that happened during the day using the following graphic constructs:
 - (a) A circle to indicate the start of the day.
 - (b) A double circle indicates the end of the day.
 - (c) A rectangular box to represent an activity.
 - (d) A long arrow for progression of the day.
 - (e) A diamond where activity flow may branch out due to certain conditions or personal decisions.
14. Empirical studies have shown that design diagrams often do not match the code and may become obsolete quickly as they are not necessarily updated when code changes. Thus, communications to software's stakeholders with views (often consisting of diagrams) may become inaccurate or even misleading when significant disparity exists between the artifacts and the development progress.
 - (a) What might be implications of these problems if these problems can't be avoided in practice?
 - (b) Suggest a few ways to mitigate such problems.
15. A company uses a linear model $r = a + b \times t$ and sales data of the last 12 months to predict future revenue, where a is the average sales of the past 12 months and b is the average monthly revenue growth rate of the last 12 months. Note that a and b must be updated as new sales data becomes available (so they are called *moving average initial sales* and *moving monthly sales rate of change*).

- (a) Use an interface to define an abstract data type we can use to update a and b , and hence implement the revenue prediction model described above (no implementation of the interface is necessary).
 - (b) Write a *main()* method to demonstrate use of this data type in part a) to predict future revenue given time t (though you are unable to run the program without an implementation of the interface).
 - (c) Use a flowchart to describe how input data t transforms into an output revenue prediction.
16. An object-oriented program is essentially about using the data types we design or we can use to implement computing tasks. Is this statement accurate? Why, or why not?
17. We may explicitly define design tasks, for example, as follows:
- Examine software requirements to understand domain model and design appropriate data structures used by data objects.
 - Select an appropriate architectural style based on the initial analysis.
 - Decompose the system function into sub-functions, design software elements enabling such functions, and fit the modules into the architecture.
 - Create design elements, interfaces, and other data types, including those required for using external systems or devices.
 - Assess the needs of each structural layer.
 - Develop a construction plan.
- Is such a to-do list helpful? Why, or why not?
18. If software is a form of art, how would you assess or evaluate its aesthetic value in terms of the capacity of the design to elicit its structural and behavioral satisfaction or dissatisfaction or in terms of a set of criteria you would like to define?

References

- Abstract. (2011). *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. <https://dl.acm.org/doi/proceedings/10.1145/1985793?cftoken=32207296&cfid=992392223>
- Baker, A., & Van der Hoek, A. (2010). Ideas, subjects, and cycles as lenses for understanding the software design process. *Design Studies*, 31(6), 590–613.
- Buchanan, R. (1992). Wicked problems in design thinking *Design Issues*, 8(2) Spring. <https://www.jstor.org/stable/i267284>
- Dym, C., et al. (2005). Engineering design thinking, teaching, and learning. *Journal of Engineering Education*, January, 103–120.
- Fowler, M. (2000). Is design dead? *Keynote Speech at XP 2000 conference*, article. <https://www.martinfowler.com/articles/designDead.html>
- Glass, R. (2006). Software Conflict 2.0: The art and science of software engineering. *developer.* Books*. https://www.developerdotstar.com/books/software_conflict_glass.html
- IEEE 1016-2009. (2009). *IEEE Standard for Information Technology – Systems Design – Software Design Descriptions*. <https://standards.ieee.org/standard/1016-2009.html>

- Knuth, D. (1974, December). Computer programming as an art. *Communications of the ACM*. <http://www.paulgraham.com/knuth.html>
- Reeves, J. (2005). Code as Design: Three Essays by Jack W. Reeves. *developer.* Books*. https://www.developerdotstar.com/mag/articles/reeves_design_main.html
- Van Vleck, T. (2016). *Interview with Barbara Liskov*. ACM Turing Award Recipient, transcripts. <https://amturing.acm.org/pdf/LiskovTuringTranscript.pdf>



The Paradigm of Object Orientation and Beyond

2

2.1 Overview

Before being exposed to design theories, methods, and practices, learners need a good understanding about *object orientation*, which is still by far the most dominant paradigm for programming and software design. This chapter provides a discussion about what object orientation is about. We assume that readers have operational knowledge about object-oriented programming, though their understanding and experience may certainly vary. The discussion provided here, we hope, will improve one's understanding about object orientation and its role in design. We will start with a discussion about what object orientation is. We will decompose the notion into elements and explore their meanings in perspectives. We will then briefly introduce how object orientation is used in software design, and a more thorough coverage will follow in subsequent chapters. Systems today are often developed with multiple languages, operate in heterogeneous environments, and interact with various external systems and devices. Thus, object orientation is not an exclusive paradigm to use in software design today. We will discuss, before we close the chapter, how multi-paradigm approaches may work in software design. These preliminary, mostly conceptual discussions provide a necessary preparation for more in-depth discussions later about paradigm-neutral design practices.

2.2 What Is Object Orientation?

Researchers (Cardelli & Wegner, 1985) defined the notion of object orientation to be:

$$\text{object orientation} = \text{data abstraction} + \text{object type} + \text{inheritance}$$

Data abstraction is a separation between the definition of data representation and its implementation. An object type (OT) is a type that objects are associated with. In this association, objects are forms of data abstraction with an interface of named operations and a hidden state. The term of OT is not widely used, but the notion is closely related to a better-known term *abstract data type* (ADT). According to the National Institute of Standards and Technology, an ADT is a set of data values and associated operations that are precisely specified independent of any particular implementation. OT can be viewed as a weak form of ADT in the sense that named operations may not be precisely defined. Both OT and ADT are implementations of data abstraction, however.

An object-oriented programming (OOP) language provides constructs for using object orientation in programming. Researchers described OOP to be the creation of clusters of abstractions for entities, controls, and interfaces. However, the practice of OOP can vary significantly. For example, a simple line of code:

```
balance = balance * (1.0 + interestRate);
```

could be replaced by an “object-rich” statement with decreased level of clarity:

```
balance.add(balance.get().times(interestRate));
```

Clearly, excessive use of objects can lead to unnecessary software complexity, causing maintenance difficulties as opposed to improving software maintainability that OOP is touted to bring.

2.2.1 Data Abstraction

Barbara Liskov, to whom the notion of ADT was attributed (Liskov & Zilles, 1974), suggested approaching programming not as a technical problem, but as a mathematical problem—something that could be informed and guided by logical principles and aesthetic beauty. While a young professor at the Massachusetts Institute of Technology, she led a team that created programming language CLU (short for “cluster”), which relied on an approach, guided by ADT, to organize code into modules (without a “GOTO” construct). Major programming languages used today, including C++, Java, and C#, are descendants of CLU. This modular programming practice at the time was independent of the practice of OOP, which first appeared in the language *SIMULA* in 1967.

ADT is a type abstraction and essentially a mathematical structure rooted in abstract algebra. As a mathematical entity, the operations of an ADT are defined precisely, though axiomatically. For instance, operations of a stack ADT satisfy the following axioms, if s is an instance of ADT Stack:

1. Upon instantiation, `s.isEmpty()` is true. (Assume instantiation defines an empty stack.)
2. If `s.push(e)`, then `s.isEmpty()` is false. (Thus, operation `isEmpty` is defined, as is push operation.)
3. If `s.push(e)`, then `s.top()` returns `e`. (This defines operation `top`.)
4. If `s.push(e)` and `s.pop()`, then the original stack `s` is intact. (This defines operation `pop` and the property of last in first out.)

When defined precisely, the primary operations of a stack cannot be behaviorally altered regardless of any potential implementation. ADTs, by design, prohibit operational tempering and extension just like all built-in types do with verifiable behavior of its operations (Cook, 2009). ADTs depend upon a static type system to enforce type abstraction. A programming construct known as “class” is used to implement ADTs in most today’s programming languages. Like an ADT, a class can be used to define a data type, and it is often convenient to do so. However, because a class can be extended and behaviorally altered in subclasses, using classes to define data types may cause verification difficulties when we want to perform correctness checks as there isn’t a language-enforced mechanism to ensure consistent behavior of an extended type. Therefore, in an ideal world, we should only use classes (abstract or not) to implement data types and construct objects, but not to represent data types. The *interface* construct, which OOP languages typically provide, is appropriate for providing data abstraction in the forms of OT or ADT. An interface can only contain abstract operations with implementation not permitted (unless it is a default operation). Like an ADT, an interface provides type abstraction. Unlike an ADT, an operation of an interface is open to implementations with its behavior not (yet) semantically enforceable by the language. Thus, when required, verification can still be difficult on interface instances.

The notion of data abstraction applies to any data types including all built-in primitive types. For instance, `int` type in Java represents all integers we can use with operations we take for granted. However, as ordinary users, we have no knowledge about computer’s internal representations of integers, neither do we know, and few would be interested in, how exactly the operations were implemented. That’s the beauty of data abstraction—a separation of what we must know from what we don’t need to. Computer software makers are responsible for deciding on how integers are represented in terms of their bit patterns in computer memory and implementing the representation based on the given computer architecture. Language implementers further decide how the operations are implemented. For instance, `2 + 3.0` may not be a legal operation as one of the operands is not an integer and explicit type casting would be required if the language is strongly typed. Alternatively, a language may impose type conversion by default depending on the type of designation variable. Names of data instances are largely circumstantial. We use `2` to represent an instance of integer type because of the universally accepted convention. The number instance `-2` may also just be a more conventional way of representing `2.negate()`. Therefore, built-in primitive data types are data abstractions too.

However, there are operational differences between using data types a library provides (or we designed) and those built-in primitive types. For example, consider Java library interface *List*, which is a data abstraction for operations of a list. When using it, we define a concrete list instance (and is conventionally called an object) like:

```
List<Integer> lst = new ArrayList<Integer>();
```

The object *lst* must be constructed with an explicit process and can only access (without a type cast) operations the type *List* defines, whereas an object of a built-in type is not explicitly constructed by user. An object must be constructed before its use according to an implementation of *List<Type>*, i.e., *ArrayList<Type>* in this case. Objects are a “double-edged sword” with the following features (Cook, 1990, 2009):

- Objects use procedural abstraction with its interface allowing objects’ behavioral alteration.
- Objects work like first-class modules (meaning they can be passed as parameters and returned from a method).
- Objects cause difficulties in program verification with their imperative states and first-class status (thus, an object’s state, defined by its instance data, can be altered unpredictably).
- Objects are dynamic entities, thus cannot depend upon a static type system to enforce type abstraction (i.e., they can be dynamically cast to another type for accessing its operations); thus, what appears to be essential for objects is some form of first-class functions or processes.

An object has encapsulated instance data that has an implied scope of procedural modifiability, which determines the validity of an object. In this sense, researchers have used the term *procedural data* to describe an object. When we use an object such as:

```
lst.aDefinedListOperation( ... );
```

there are two implications:

1. The operation is defined in a data abstraction (*List* in this case).
2. The operation is implemented in a concrete class (*ArrayList* in this case).

If *lst* was constructed to be an instance of *LinkedList* (another concrete class that implements *List*), the above method invocation would remain valid because validity only checks where this operation is defined. A different implementation of an operation defined in a data abstraction may offer different non-functional benefits such as memory usage or algorithm selection but is expected to follow the same pre- and postconditions of the operation.

A concrete class can implement multiple interfaces. For example, class *LinkedList* also implements *Queue* interface. In other words, we may choose to implement multiple data abstractions with a single concrete class when convenient and beneficial. If we use a concrete class to define a data type such as:

```
LinkedList<Integer> lstObj = new LinkedList<Integer>();
```

the effect of this declaration is that code that uses *lstObj* is tied to a concrete implementation with potentially “messy” details, and the object is free to use any operation supported by *LinkedList* even when such a use is not justified. Besides, this object would be unable to be assigned to an instance of *List*, making the code more rigid and less flexible.

To manage software complexity effectively and provide software extensibility, we need a clear separation between what a computation is and does and how the operations are implemented. Data abstraction supports that separation. With the separation,

- We design data abstractions based on software requirements to ensure that high-level “business logic” can be implemented using the data abstractions only with an understanding that actual instances will be provided at runtime.
- Concrete classes that implement data abstractions can be constructed later to allow more stable requirements that implementation details reply on.
- We use object “factories” to create and dispatch “just-in-time” instances of data abstractions at runtime.
- We make separate design decisions on implementations of the data abstractions, whether we want to use multiple classes or a single class to implement all data abstractions as appropriate. Regardless of what we do to implement the details, it is all hidden behind the “walls” of the “factories.”

2.2.2 Object Types

An object type (OT) may refer to any data type that can be used for declaring objects. An abstract data type (ADT) is an OT (though the converse is untrue). To represent OT, we use interfaces, abstract classes, and, to a much lesser extent, concrete classes. Ideally, we use a concrete class to define an OT only if implementations of the operations are stable such as those of a software tool. When we want to postpone the decisions on objects’ representation details, we use interfaces to represent OTs. However, when certain operations can be stably implemented, or a control process in certain situations can be safely implemented in principle, we may use an abstract class to represent an OT.

To make programs flexible and extendible, we want objects to be replaceable at runtime when alternative details are desired. This would give us advantages of delaying decision-making on details as much as practically necessary to allow details to mature. We need OTs

to give us such abilities. This is the reason why we always use interfaces and abstract classes to represent OTs to model things or processes that can potentially change in operational details.

Objects can be cast to different OTs at runtime like below:

```
If(IstObj instanceof Queue){
    ((Queue) IstObj).enqueue("hello");
}
```

Such code makes program verification more difficult and weakens the meaning of an OT. Thus, object being cast to a different OT at runtime should be a deliberate action with full awareness about its consequences.

When a class contains only static members, it does not represent an OT as statics operations should not be invoked using objects. Such a class serves as a module in a traditional sense. Textbooks often describe a class as a “blueprint” of objects. In fact, it is anything but a “blueprint”:

- A class is primarily to provide an implement to some “blueprints” (i.e., interfaces or abstract classes).
- A class is a syntactic representation of objects, and its constructor creates its memory representations.
- A class, as a subclass, can “modify” an implementation of some “blueprints.”
- A class, when used to group static variables and operations, is a classic module.

An interface specifies what operations are and do; it is a “blueprint” in that sense (though an ADT assumes that role more accurately). An abstract class then is only partially a “blueprint.” Though OT is mostly to mean a “blueprint” in the book based on the original intent of the term, we use the term more loosely.

2.2.3 Inheritance

The third element of object orientation is *inheritance*. Inheritance is the mechanism of extending classes or implementing interfaces. Inheritance allows new (sub)classes to be constructed based on existing classes to model more elaborated objects with additional properties and behavior. Through inheritance, operations implemented in superclasses can also be reimplemented in subclasses. For example, *stream* may be a useful software notion with concrete streams like information stream, file stream, network stream, etc. Streams share similar operations regardless of the kind of stream it is. Thus, it is justified to create a base class to implement, in principle, some common operations such as *map*, *sort*, and *filter* to deal with things that are “iterable.” Subclasses would handle the individual storage appropriate for the stream content. An apparent benefit of inheritance is code reuse as a

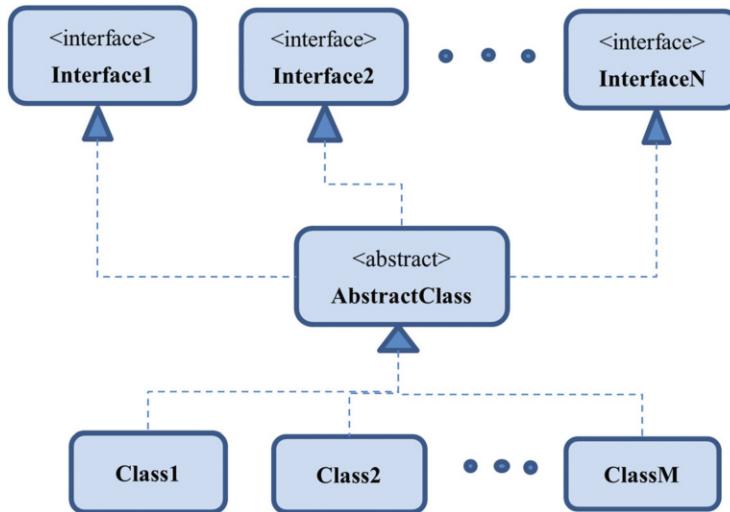


Fig. 2.1 A plausible scenario of class inheritance

subclass can inherit (visible) code from a superclass. Empirical studies have shown, however, that inheritance generally doesn't compete more favorably on code reuse with traditional approaches based on modularity.

Inheritance adds complexity to a program. There is added code rigidity when concrete (non-leaf) classes are used in an inheritance hierarchy because of a potential deep hierarchy that any changes close to the base may affect all branches of the class where changes occur. The drawbacks, when inheritance is used inappropriately, may well overweight the benefits.

Figure 2.1 describes one plausible scenario of inheritance, where an abstract class implements multiple interfaces to provide code base and common services needed by eventual implementing classes. In this scenario, interfaces should generally be small and focused to allow versatile conceptual combinations. The sandwiched abstract classes are optional, however, and they are only needed when concrete subclasses do share significant amount of implementation details such as hierarchies of data structures. An inheritance with interfaces at the root is often termed “interface inheritance” with a primary focus on polymorphic implementations of interface operations. Such hierarchies are generally shallow with two or three layers as they are primarily for creating replaceable objects with polymorphic behavior. In contrast, “implementation inheritance” or “code inheritance” refers to creating subclasses most likely for implementation or code benefit. In particular, when a hierarchy rooted at a concrete class, ramifications on code rigidity and program verifiability can be unpredictable. Thus, the potential harm of “code inheritance” may well overweight the benefits.

When a program is divided into clusters of objects, polymorphism provides different paths that data and information can transform and hence ways a program can be extended.

By extension, inheritance (with mostly “interface inheritance”) is a mechanism to support structural and behavioral extensibility of a system. Object orientation is, therefore, to orient the objects we design toward achieving a system’s structural and behavioral extendibility.

2.3 The Paradigm of Object-Oriented Design

A paradigm may be a pattern, a model, or more broadly a philosophical and theoretical framework of some kind. A paradigm of study in science refers to a shared collection of the underpinning conceptions, assumptions, and scientific evidence and a set of exemplary experiments that can be emulated. These characteristics of a paradigm apply to software design too. The constructs of object-oriented programming enable a paradigm of software design. Good design practices using object orientation can be considered as exemplary design experiments, and object-oriented design principles are a shared collection of guiding forces in the design of objects.

Object-oriented design (OOD) is to use the notion of object orientation, as discussed in the previous section, in design practices. Arguably, OOD is an extension of the traditional structured design. To understand this paradigm transition, consider how the notion of data has evolved since the introduction of object types. Objects are based on composite data known as (static) record types. For example, the following class defines a record type containing a composition of different data types.

```
class StringLog{  
    String[ ] log;  
    int size;  
}
```

It was a significant advancement in programming when languages supported composite data types as first-class entities that can be passed as parameters and returned from functions. Programs using composite types may improve effectiveness of data use and code maintainability. However, use of composite types is not a paradigm shift in program design because richness of data does not change the way data is used. But when methods are integrated into a record type like the following code, it changes the way modules work in a program.

```
class StringLog  
{  
    String[ ] log;  
    int size;  
    void insert(String s){ ... }  
    String getLog(int pos){ ... }  
}
```

The dynamic nature of objects opens up an array of design possibilities beyond the philosophical and technical framework of the structured design. An object is like a self-sufficient computing machine with its internal data state. When such machines can be composed with other machines, i.e., they can be defined in, passed into, or returned from a module, we must develop methodologies and guiding principles in terms of how these machines may best work collaboratively to solve problems. This is a paradigm shift because the way we solve problems has changed by using objects:

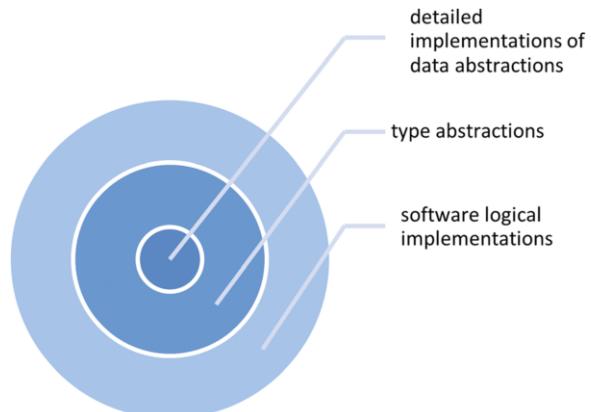
1. It is a unity between data and operations because of the ways objects are used.
2. Each object has its designed share of intelligence, contributing to an overall task completion.
3. Program complexity is distributed among designed objects.

Data abstraction remains central to object-oriented design with a set of OTs that provide an infrastructure for an extendable system. Data abstractions are like conceptual “virtual machines” to be used when software operations are implemented. The implementation details of the data abstraction are like a “processing kernel” where actual computation takes place, as illustrated in Fig. 2.2. This object-oriented approach to software design achieved these benefits:

1. Implementations that depend only on abstractions are not affected by changes to implementation details.
2. This separation, in turn, allows decisions on implementation details to be delayed so that we can proceed with higher-level design and implementation early while allowing details to evolve and mature.

Functional decomposition of a system still applies to object-oriented design, however. But the decomposed tasks are responsibilities of the OTs we design and accomplished

Fig. 2.2 A layered view of software operation



through collaborations among objects. High-level functional decomposition also reveals software entities that play roles of coordination or control over task delegation and integration. Decompositions also discover the needs for connectors to resources and structurers that facilitate resource sharing and bridge task completion. Eventually, functional decompositions identify “kernel operators”; and these are implementers of the OTs, data structures, object factories, and special-purpose entities. In an object-oriented design, functional decompositions of a system introduce clusters of OTs. Meanwhile, decompositions apply to system behavior and domain concepts too. The outcomes of decompositions are inputs to the design of objects, forming themed object clusters with distinctive responsibilities.

Object-oriented design is complex because we are facing a wide range of design possibilities, constraints, and tradeoff considerations to seek a “balanced” design that has the attributes consistent with our design goals. The “high-level” separation discussed earlier between the design of OTs and design of implementation mechanisms hidden in object “factories” is only a high-level “divide-and-conquer” strategy that must be supported by more concrete object-oriented design practices that we will elaborate in subsequent chapters.

To see how the structure of a program can be sustained with OOD, consider the following multi-conditional statement, common in a business processing layer.

```
if( cmd == 'choiceA' ) doThis();
else if( cmd == 'choiceB' ) doThat();
else doSomethingElse();
```

This code is vulnerable to changes if more conditional cases are to be added, subtracted, or modified. We can use polymorphism to defer the decision on “what to do” until a specific condition occurs (at runtime). This can be achieved with an OT structure shown in Fig. 2.3:

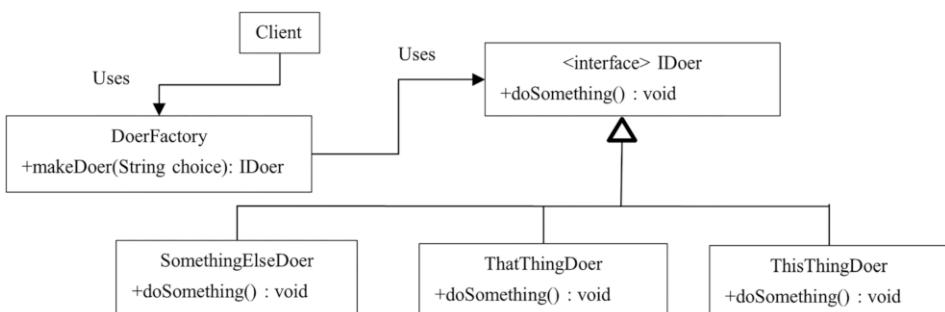


Fig. 2.3 An object structure to replace conditional logic

Interface *IDoer* captures what needs to be polymorphic. *DoerFactory* is a service class responsible for creating a relevant object as needed. The code above is now equivalent to the following more sustainable condition-less statement.

```
IDoer doer = DoerFactory.makeDoer( cond ); doer.doSomething();
```

The conditionals are not eliminated, however. Rather, they are moved into the service method *makeDoer* to control object creation. When a new process was needed with an additional command, we would create another implementing class of *IDoer* and modify the factory to “schedule” a dispatch of an object. This modification to a factory is generally safer as object factories are likely static entities working with well-intended OT hierarchies. This is a tradeoff between a vulnerable conditional statement (potentially in a high-level business processing module) and a more sustainable code statement with an OT hierarchy and a factory of delayed object creation. We trade a more complex code structure (which is also less efficient) for code sustainability. This is often a worthwhile tradeoff when high-level code stability is at priority.

A design must be mapped in two directions—software requirements in one direction and implementation in the other. It might be expected that an OOD would allow us to model the real-world entities as we observe in terms of what the attributes are and how they behave. This is most likely an incorrect expectation. We model real-world entities only when they are helpful, keep their attributes only when they are necessary, and introduce behavior only if it is part of a solution process. Most importantly however, a design must ensure satisfying the software requirements with the designed software entities and their functionalities. The mapping in the other direction to an implementation is based on certain characteristics of the design we want to achieve such as software sustainability, scalability, and robustness, among others. The forces that keep the mappings in the two directions “synchronized” are a good understanding about requirements vulnerability and changeability, which may determine the OTs we design to encapsulate the things and processes that are likely to change. This “principle of encapsulation” is consistent with what was proposed more than half a century ago for modular design (Parnas, 1972), though object orientation offers a more powerful encapsulation mechanism.

2.4 Embracing Multi-paradigm Design

Programming languages are increasingly moving away from being pure object-oriented languages, as are software design methodologies. With an increased use of application programming interfaces developed with mixed paradigms, new design opportunities are emerging that drive the advancement of the design methodologies. Today, we probably want to start a design with a paradigm-neutral approach even if object orientation may still be a design focus. This can be done by analyzing the application domain to understand the

dimensions of commonality and variability—in what aspects and to what extent things are common or vary—to design appropriate modeling elements (Coplien, 1999). This analysis is based solely on the principles of abstraction and system decomposition that underlie all design paradigms. Software analysis may uncover groupings of abstract concepts and artifacts that are tied together by their commonalities and perhaps by the similar nature of their variabilities. These groupings will constitute the basis for designing procedural and data abstractions within the software constraints. The conceptual families of software elements may also form groupings or subsets of groupings for further analysis. When responsibilities of the groupings are fully understood, decisions can then be made on design paradigms that can best handle the groupings or intersections of the groupings. These ideas are rooted in classical design principles to use modules to capture and hide from other modules difficult design decisions or implementations that are likely to change (Parnas, 1972).

Prescribing an exclusive design paradigm can be inadequate or unhelpful for producing sustainable systems when certain design issues become more prominent. Traditional structural or modular design focuses on static behavior of a system, whereas object-oriented design captures how a system may behave dynamically. Yet, we have seen functional languages that seem to capture interactions between static and dynamic aspects of a system. For example, functional languages have long supported “higher-order” functions to accept functions as parameters and return functions as if they were values. In the same sense, object-oriented programming can be thought of as “higher-order” procedural programming with enhancement on procedural aggregation and override, though in the name of object. Making design or code more flexible and extendable may have a bit different flavor in higher-order procedures, where code uses explicit higher-order parameters to define “hook points” to allow “polymorphic” procedures. The following (JavaScript) code example demonstrates such a possibility:

```
const saveRecord = (record, save, beforeSave, afterSave) => {
    const defaultSave = (record) => {
        // default save functionality
    }
    if (beforeSave) beforeSave(record);
    if (save) { save(record); }
    else { defaultSave(record); }
    if (afterSave) afterSave(record);
}
```

To call the module,

```
const customSave = (record) => { ... };
saveRecord(myRecord, customSave);
```

newer programming languages and updates of old languages increasingly support constructs that blur the differences between procedures and objects. For example, lambda expressions extend the traditional realm of lambda functions to associate data with dynamic behavior of a system. Extension methods of C# enable adding (static) methods to existing types without inheritance to offer a logical association between different paradigms. Code reliability has motivated professionals to rethink the value of static typing. The language GO (or Golang) has been popular because of its static typing nature with a much faster runtime and its multi-paradigm support (in addition to its features and libraries close to what C and C++ offer). The popularity of GO might be a sign that inheritance may not be as important in modern software design as it once was. These developments have diluted the boundary between static and dynamic behavior of a software system and opened the door for more design options.

Meanwhile, software applications have entered a new era of software application as a service with integrated capabilities in Internet of Things, machine learning, and data intelligence. Dynamically typed languages such as Python, Ruby, and JavaScript have replaced traditional language frontrunners for software projects of any nature. Because of the versatilities of these languages and their powerful language features, many libraries have been developed to support software development in many application areas. We have seen a new phenomenon that a popular language is supported by an ecosystem, development frameworks, and a community of open-source enthusiasts. Paradigm neutral or paradigm open is becoming a new norm of software development.

Encyclopedia Britannica describes problem-solving as seeking “a kind of thinking that facilitates question answering.” Paradigms are evolutionary methodologies that may change, as technologies advance, the way we think and facilitate answering design questions. Older paradigms are not replaced by new ones. Rather, they merge as one collective paradigm with concepts and techniques from a collective repertoire. Thus, thinking in objects is a natural progression of thinking in modules. Thinking in functional expressions might require a mental evaluation on effects of interactions between static and dynamic processes. This multi-paradigm thinking also applies to design artifacts we produce. For example, structured diagrams from the 1970s were based on functional decomposition, task composition, and task delegation. This way of thinking, as professionals found, became also useful in seeking object-based design solutions of handling concurrent event execution, where events are delegated to the most suitable business objects to handle the business behavior, leading to a design of business objects (Gillibrand, 2000). Classical dataflow diagrams are still popular. When the processes in a dataflow diagram are associated with objects, we can introduce “object channels,” as shown in Fig. 2.4, along with static processes if appropriate and meaningful.

Today’s software applications are frequently complex systems made up of many subcomponents that require a mixture of technologies. For instance, people may still rely on mainframe computers for handling raw financial data. But they also build networked data objects of various forms to filter raw data for web-based client applications. At the same time, these client applications may also use many cloud-based financial services from

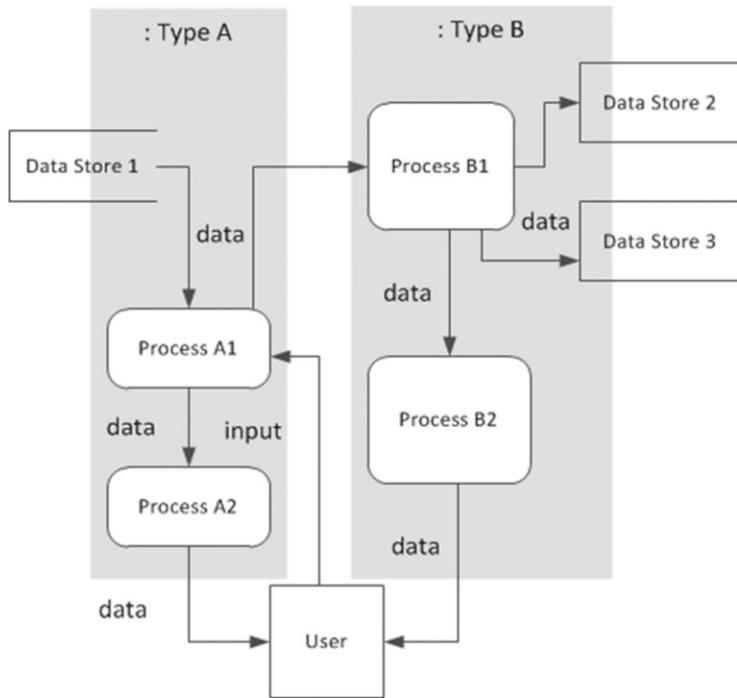


Fig. 2.4 A dataflow diagram with object channels

different vendors. This level of system integration requires, in turn, software design to use a mixture of paradigms to effectively tame the complexity associated with interoperability of the systems. We need less layers for application code, fewer code dependencies, but more service-focused frameworks, structural connectors, and better practices to keep domain and validation logic close to the sources. Paradigm mixing allows us to achieve these goals.

2.5 Summary

Despite a clear need for multi-paradigm effort in software design, object orientation is and will remain the primary paradigm in software design for years to come. This chapter has elaborated the notion of object orientation in the perspectives of data abstraction, object type, and inheritance. The key to object orientation is data abstraction to support objects' behavioral alteration or extension. The chapter also provided an analysis on what object orientation may mean to object-oriented software design. An example was given to show how we could avoid vulnerable processing logic by creating appropriate data abstractions to allow delayed design decisions when software requirements are still evolving or implantation details are yet to mature. These design ideas will be further developed in upcoming chapters.

Software today is frequently a system of systems, and interoperability is essential. Because of the need for integration of systems developed at different times, using different technologies, and under different social-technical circumstances, multi-paradigm design is becoming a new norm in software development. Multi-paradigm design is not just about using an appropriate design paradigm when circumstances warrant; it is also about paradigm integration with design ideas intertwined in the design of software elements, their relations, and architectural structures that facilitate system interoperability.

Exercises

1. Describe connections and differences between data abstraction and abstract data type.
2. A class in an object-oriented language defines a data type,
 - (a) Is this data type object type, and why?
 - (b) Is this data type abstract data type, and why?
3. The notion of abstract data type (ADT) predates the first widely used object-oriented programming language C++ by more than a decade. In other words, ADT was originally defined imperatively without using interfaces or classes. Read Wikipedia at https://en.wikipedia.org/wiki/Abstract_data_type for more information, and then describe how stack as an ADT was defined in an imperative fashion.
4. We are used to console input and output mechanisms of a language generally without knowing (or even being curious about) the details. That's the beauty of data abstraction. Explore and then describe what each element is in `System.out.println(. . .)` using publicly available Java documentation, and you may start here: <https://docs.oracle.com/javase/7/docs/api/java/lang/System.html>.
5. If you are asked to write a data type, say, *Student*, most likely you would use a class to encapsulate instance data fields, such as student name, total grade points, total credit hours completed, etc., and instance methods to retrieve student information, update data fields, check whether student is in a good academic standing, etc. Then, you would declare an object like:

```
Student student = new Student("John", 21, 10);
```

- (a) Explain how this approach might lead to rigid and inflexible code.
- (b) Describe how you might design this data type differently.
6. It is possible to use a class only for data. For example,

```
class Student { public String studentName; public int totalGradePoints; public int totalCreditHours; }
```

We then define a collection of (static) service methods also using a class, such as:

```
class Services {  
    public static double getGPA(Student stu){ return stu.totalGradePoints / stu.totalCreditHours; }  
    .... }
```

Here is how we would use this approach in code:

```
Student stu = new Student();
stu.studentName = "John"; stu.totalGradePoints = 21; stu.totalCreditHours = 10;
double gpa = Services.getGPA(stu);
```

Discuss pros and cons of this approach in terms of supporting data abstraction, code flexibility, and code reliability (compared with your solution to Question 5).

7. Someone might argue that even though class *Student* in Question 5 is concrete, we can still treat it as a data abstraction and write subclasses and override behaviors when necessary. For example, we may use the following class to model a graduate student:

```
class GraduateStudent extends Student {
    public GraduateStudent(String name, int totalGradePts, int totalCredits){
        super(name, totalGradePts, totalCredits);
    }
    public boolean isStudentInGoodStanding(){
        double gpa = (double) totalGradePoints / totalCreditHours;
        return gpa >= 3.0? true : false;
    }
}
```

An object can be created like:

```
Student gradStu = new GraduateStudent("John", 21, 10);
```

Describe the pros and cons of the approach.

8. What are the pros and cons of using an abstract class to represent a data abstraction?
9. Airlines assign seats at gates for those who need them. There are three types of customers: regular customer, regular customer with elite frequent flyer status, and customer who is airline's employee. Seats are assigned based on the policy for each kind of customers. For example, a customer with elite frequent flyer status can be upgraded to a business seat upon availability. A customer who is airline's employee should be assigned a business seat, then an economy seat all upon availability, and should be bumped first if seats are tight. Suppose the software used to assign seats has a type *SeatAssignment*, which has a method to return a seat string when seat assignment is successful based on customer's information and airline's policies or empty string if seat assignment is unsuccessful.

```
public String assignSeat(Customer c){ ... }
```

Design data abstractions that can be used in a possible implementation of the method.

10. We apply paradigms in our lives too, though we may not be consciously aware most of time. For example, to visit a foreign country, we can take a do-it-yourself approach or

contact a travel agent to plan for you. These can be considered two different travel paradigms. A multi-paradigm planning for a trip might be a risk-driven approach to identify portions of the trip that seem uncertain, least known, or most risky. We might consider whether a reliable local travel agent would be available. We might also use social networking to ask people who might have had comparable travel experiences for suggestions. But perhaps more importantly, we want to look closely into the uncertainties with multiple alternative plans before we settle on any concrete, feasible plan.

Think about a difficult or extensive task that requires planning (such as running for office, going through a challenging semester, starting a new business, surviving a wedding ceremony, etc.), and then describe how you might go about planning using a “multi-paradigm” approach.

References

- Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *Computing Survey*, 17(4), 471–522.
- Cook, W. (1990). Object-Oriented Programming versus Abstract Data types. In *Proceedings of the REX Workshop/School on the Foundation of Object-Oriented Languages* (LNCS 173) (pp. 157–117). Springer.
- Cook, W. (2009). *On understanding data abstraction, revisited*. Proceedings of OOPSLA 2009, Orlando, FL.
- Coplien, J. (1999). *Multi-paradigm design for C++*. Addison-Wesley.
- Gillibrand, D. (2000). Essential business object design. *Communications of the ACM*, 43(2), 117–119.
- Liskov, B., & Zilles, S. (1974). Programming with abstract data types. *Proceedings of the ACM SIGPLAN symposium on Very high-level languages*, 50–59.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058.



Essentials of Object-Oriented Design

3

3.1 Overview

A computer program that solves a problem has two essential ingredients: algorithms describing the solution logic and data types designed to facilitate a solution process. At a more abstract level, problem-solving relies on data types and their structures for accessing controls of software operations and functions of domain elements. For an object-oriented solution, design of inheritance hierarchies can be at the core of a solution design. This chapter provides an in-depth discussion about the practices of using object orientation in program design, including appropriate use and misuse of the related concepts. The chapter is an overview about the essential concepts, principles, and practices about object-oriented design. It also prepares readers for in-depth discussions in the subsequent chapters. We will start with more elaboration on data abstraction about its primary advantage in providing behavioral polymorphism. We will then take a closer look at three important aspects of object-oriented design—design of interfaces, appropriate use of abstract classes, and ensuring behavioral compatibility when extending an inheritance hierarchy. We will also provide a more elaborated discussion on use of inheritance in design. We will conclude the chapter with an introduction to major object-oriented design principles. When helpful, code fragments will be given in Java syntax.

3.2 Data Type, Data Abstraction, and Type-Safe Practices

Data type (DT) is a set of “values” (they can be objects) with a set of operations defined for the “values.” DT is perhaps among the first programming concepts novices would learn. It might be adequate to use data values to explain to novices what a data type is. For example, we might say data type *int* consists of all values of integers we practically use. A primitive

data type is a set of all possible values with certain (computer) representation and a set of defined operations applied to these values. Language built-in primitive data type *int* represents a set of integers that can be represented in a computer. Different sizes (8, 32, or 64 binary digits) used for an integer representation in computer memory determine a range of integers we can use. The operations applicable to the members of a data type are part of the data type. Thus, operations $+$, $-$, $*$, $/$, $\%$, etc. are part of the integer type *int*. Even though the same operators apply to values of data type *float*, the underlying implementations of the operations are different because the values of *float* are represented differently in memory. Besides, a language also imposes rules of how operations should be carried out. For operation like $2 + 3.4$, a language may convert integer 2 to a float number 2.0 and then apply “ $+$ ” defined for the type. However, a language with more stringent type rules could simply label the expression “illegal” and force a user to write $2.0 + 3.4$ instead.

As users of primitive data types, we must know the correct use of the operations defined for a data without necessarily knowing how the operations are implemented (different languages may implement *int* type differently). Thus, primitive data types are in fact ADTs, and their constructions follow the same principles of data abstraction. Data type is more of a programming term as it often refers to a primitive data type that we, as users, do not design. In comparison, object type is more of a design term because we design them. Nonetheless, we may use DT, OT, and ADT interchangeably as appropriate. All three notions about data are based on the notion of data abstraction—the principle that underpins the paradigm of object orientation. To revisit, data abstraction provides a separation between use of the operations defined by a type and an implementation of the operations. This separation has the following primary advantages:

- It allows high-level code modules to be constructed based only on the operations the data types define.
- It allows implementations of the data types to be delayed when details are uncertain or emerging or implementations are simply not a current priority.

The primary benefit of data abstraction is to allow high-level processing modules to be implemented using data abstractions only based on their logical interfaces without concerning lower-level details. Such constructions can take place potentially much earlier than the time when these abstractions must be implemented. In other words, data abstraction gives us an ability to focus on high-level design and implementation activities with good sustainability. Abstract methods play a role of enforcing the separation. Consider the following object type that offers three operations to define a data service:

```
interface IDataService{  
    List<String> retrieveData(String connStr);  
    void saveData(List<String> data, String connStr);  
    String updateData(String str);  
}
```

We can use the type to construct the high-level module independent of any potential implementation of the interface:

```
void processsData(IDataService s, String srcConnStr){  
    List<String> lst = s.retrieveData(srcConnStr);  
    for (int i = 0; i < lst.size(); i++) {  
        String current = lst.get(i);  
        String str = s.updateData(current)  
        lst.replace(i, str)  
    }  
    s.saveData(lst, srcConnStr);  
}
```

In this example, the following benefits of using data abstraction are also clearly observable:

- The code sustains when implementation details (of the interface) change.
- Instances of *IDataService* are replaceable when needed (such as changes in a data source or a connection method, which would require a new implementation of the interface).

If we need data services for two different data sources, we implement the interface, respectively, as follows:

```
class DatabaseService implements IDataservice { .... }  
class DatafileService implements IDataservice { .... }
```

The method can be invoked as follows (likely at different times):

```
processsData(new DatabaseServer(), 'my_connection_string');  
processsData(new DatafileServer(), 'my_file_path');
```

However, if either *DatabaseService* or *DatafileService* was used as the type of the parameter *s*, the method would be confined to a concrete datatype implementation, and parameter *s* would be only replaceable by an instance of a subtype. Consequently, the code would be more rigid and less sustainable.

There are type hazards in our design or in use of data types we need to be aware. Abstract classes are often used as intermediate data types to provide common code base to be inherited in subclasses. This is more prominent in implementations of data structure hierarchies where code reuse is common. When an abstract class implements multiple interfaces and also serves as a data type, it has a side effect that an instance would have access to all operations that the abstract class implements. This can cause program verification difficulties and unintentional misuses. An abstract class may serve as a data type

without much side effect if it implements a single interface or is standalone while still providing data abstraction to certain extent.

Adding more methods to an existing interface later may cause more issues than it resolves. A simple reason is that it would then require all implementing classes to implement the added operations even if some clients may not need the operations or have no appropriate ways to implement them. Instead, we typically have the following two options:

- Add methods to an implementing class where needs are. For example, suppose we add the following method to the class *DatabaseService*:

```
void displayData(){ .... },
```

and create an instance:

```
IDataService service = new DatabaseService().
```

The following code demonstrates a safe way to use the method:

```
if ( service instanceof DatabaseService )
    ((DatabaseService) service).displayData();
```

However, explicit verification of an object's type also introduces code rigidity, making code more vulnerable to changes.

- A better option (but a bit more complexity, thus a tradeoff) is to include a new interface to include the desired methods, forming a new data abstraction. As appropriate, we could either modify the existing class *DatabaseService* or extend it with another class; in either case, the class also implements the new interface as follows:

```
interface IDataDisplay{
    void displayData();
}
class NewDatabaseService extends DatabaseService implements IDataDisplay { .... }
IDataDisplay d = new NewDatabaseService(...);
d.displayData();
```

Thus, an object can be typed differently with limited scope of operation access—a safer practice. This example may also suggest use of small interfaces to allow flexibility of interface inheritance, which we will elaborate later.

3.3 More About Interfaces

In this section, we continue the discussion about use of interfaces to support data abstraction. We will further elaborate the unique benefit of using interfaces in software design.

3.3.1 Software Sustainability

Interface inheritance (as opposed to code inheritance as discussed earlier) is one of the ways in design to sustain software implementations because of multiple implementations of the same interface that allows behavioral polymorphism of the operations, which makes an object's behavior extendable. In type theory, polymorphism is a notion to mean the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types. Programming languages typically support three kinds of polymorphism; all three provide ways to sustain software design and implementation.

- *Ad hoc polymorphism* refers to polymorphic functions that can be applied to arguments of different types but behave differently depending on the type of the argument to which they are applied. The term is better known as operation overloading. The term *ad hoc* in this context is not intended to have a negative connotation; it simply refers to the fact that this type of polymorphism is not a fundamental feature of the type system.
- *Parametric polymorphism* allows a function or a data type to be written generically so that it can handle values uniformly without depending on their type. Parametric polymorphism is a way to make a language more expressive while still maintaining full static type safety. The notion is better known as *generic type* in most object-oriented languages, for example, a list with elements of a generic type when list is defined and implemented but replaced with any actual type when a concrete list is constructed.
- *Subtype polymorphism* (or inclusion polymorphism) refers to the name of a type applying to instances of many different classes related by the common supertype. For instance, if S_1 and S_2 are subtypes of T (name of an interface or a class), then an object of T can be assigned with either an instance of S_1 or S_2 , i.e., $T t = new S_1()$; $T t = new S_2()$; are both valid (i.e., instances are swappable). For the abstract methods in T , S_1 and S_2 would generally implement them differently. Subtype polymorphism is a runtime program property that invocation of a method would be bound to a correct implementation only at runtime. As discussed in the previous chapter, subtype polymorphism constitutes the core technique in object-oriented design. To maintain object's behavioral compatibility at runtime, any implementation of an overridden method must satisfy

contractual pre- and postconditions of the method. This requirement is generally known as *Liskov Substitution Principle* to ensure an object's substitutability without causing incompatible behavior at runtime. We will provide a focused discussion later in this chapter.

Vulnerable details can make code less sustainable. We use interfaces to provide some leverage on code vulnerability. Consider the following abbreviated method to process a collection of orders with a parameter of *Collection*, a Java library interface:

```
void processOrders(Collection<Order> orders){ .... }
```

Often, it can be too early to think about the kind of collection that would be appropriate for this operation. For example, a queue would be more appropriate if orders must be processed on a first-come-first-serve basis. A set could be used if orders must not be duplicated, etc. The data abstraction *Collection* supports virtually all kinds of data structures that are not maps. Thus, implementation of the method can be proceeded using the operations defined in *Collection*, while a decision on a concrete collection to use can be delayed.

Implementation details often involve business rules, computational algorithm selections, user inputs, or simply software requirements that are yet to mature. Use of interfaces allows high-level structural or operational decisions to be made early so that high-level modules can be constructed early to test the feasibility of the software architecture. We can postpone decisions on implementation details until the details become important in integration testing of the software.

In an application, details may not be known until runtime such as system responses to user inputs or a data stream. Interfaces allow “lazy” objects to be created at runtime with polymorphic details. Factories are typically used to dispatch appropriate just-in-time objects at runtime. Factories and concrete classes that implement interfaces are, or close to, “leaf modules” of a structural hierarchy of the software, and their modifications, if needed, tend to be less consequential to the software structure. Appropriate use of interfaces protects high-level code modules from changes. In many cases, software extension may simply be a matter of adding new implementations of the existing interfaces.

3.3.2 The Role of Interfaces Through the Lens of Data Structures

Java’s collection library is a good example of using interfaces. Many implementing classes have been added since Java Development Kit (JDK) 1.0. However, the essential elements in the inheritance hierarchies of the library remain stable over the years. With each new release of Java, existing application codes with appropriate design are not affected. Despite newer equivalent, better-performing classes, legacy classes are continually supported. For

example, applications using legacy class *Vector* (despite the availability of better-performing *ArrayList*) continue to work. Data structure “deque” stands for double-ended queue and is a linear data structure where the insertion and deletion operations are performed from both ends. Deque is hardly a needed data structure given the existing library capability. However, by introducing this interface, the library has achieved the following effects:

- Directly support applications where deque structures are more appropriate.
- By creating the implementation class *ArrayDeque*, stack operations perform better than the legacy class *Stack* does (which is based on *Vector*).
- By implementing interface *Queue*, *ArrayDeque* offers better-performing queue operations than *LinkedList* does (which also implements the *Queue* interface).

The interface *Collection* defines rudimentary operations for non-map data structures such as list, set, or queue (with more operations defined in sub-interfaces *List*, *Set*, and *Queue*). Figure 3.1 (partially) illustrates the structural relations of the hierarchy (where a dotted arrow means interface implementation and a solid arrow represents a type extension). To support code reuse, abstract classes are used to provide baseline implementation of the interfaces. By extending the abstract classes, concrete classes provide final implementations of various data structures. It is interesting to observe that concrete classes often implement multiple interfaces. For example, *LinkedList* implements not only expected interfaces like *List* but also “unexpected” interfaces like *Deque* and *Queue*. In other words, a class may implement those interfaces when implementation details can be shared and more efficiently managed, regardless of the conceptual implications of the class name. Thus, an instance of *LinkedList* can be typed as a *Queue* or *Deque* yet has access to operations that are incompatible with those of a queue or deque. Similarly, an instance of type *ArrayDeque* can be typed as *Deque*, *Stack*, or *Queue* but used to access operations that would potentially invalidate the intended data structure. Compare the following two declarations:

```
Deque obj1 = new ArrayDeque();
ArrayDeque obj2 = new ArrayDeque();
```

The instance *obj1* can access methods defined in *Deque* and in *Queue*, but not other methods that are also defined in *ArrayDeque* without an explicit type cast. In contrast, *obj2* can access all operations implemented in *ArrayDeque*. With “uncontrolled” access to the operations with *obj2*, misuses or undisciplined practices may happen, which may make program verification difficult or even impossible. This example reinforces an earlier discussion that concrete classes are for implementation details and generally unsuitable for use as data types. Operations a concrete class implements may often be conceptually incoherent, and these operations must be decoupled from high-level code modules using appropriate interfaces.

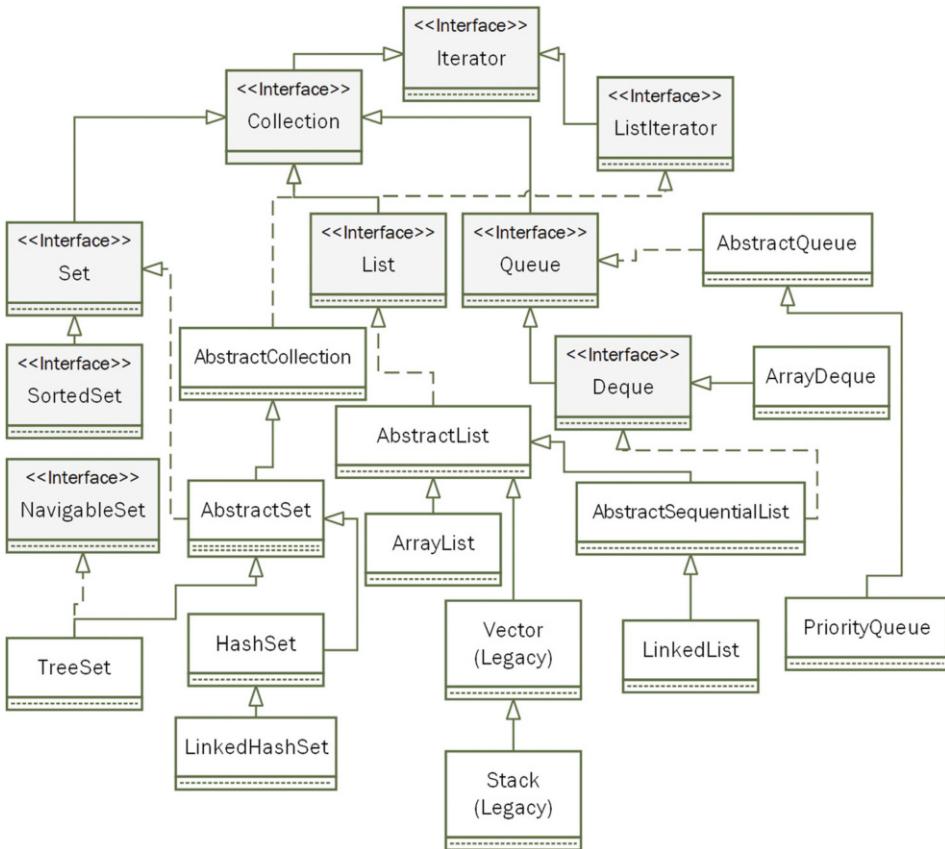


Fig. 3.1 Java collections framework (partial)

There are other interfaces the collection library also uses such as `Iterator`, `Cloneable`, `Serializable`, or `Comparator` (though most of which are not included in Fig. 3.1). These interfaces are for enabling data structures with additional properties. For example, the interface `Iterator` allows client to use the elements of a data structure without knowledge about the nature of the data structure and much less its implementation. These interfaces are small, each focusing on an area of concern. In other words, an interface can be large or small depending on a concept we want to model. Data structures are service entities; thus, their abstractions are generally large reflecting a wide range of service operations users may potentially need. In contrast, we decompose a concept into sub-concepts with small interfaces capturing the essential operations of each sub-concept to provide conceptual flexibility to model something larger, if we need to, by combining appropriate interfaces.

3.3.3 Programming to an Interface, Not to an Implementation

Functional languages support higher-order functions. A higher-order function is a function of functions (as parameters). Object-oriented languages like Java may introduce higher-order functions in their updates, but the underlying implementations of the functional features may well be based on interfaces. Consider computation of a sum $\sum_{i=1}^n f(i)$ for any given function f and integer n . Since objects are higher-order modules, we can achieve a similar effect of a higher-order function with the following interface:

```
interface IFunction{ double evaluate(float x); }
```

The following method then emulates a higher-order function to compute the sum:

```
static float functionalSum(IFunction func, int n){
    float sum = 0.0;
    for(i = 1; i <= n; i++){
        sum += func.evaluate(i);
    }
    return sum;
}
```

The following code computes sum $\sum_{i=1}^{100} \cos(i)$ (with an instance created by an anonymous class):

```
float result = functionalSum(new IFunction(){
    public double evaluate(float x){
        return Math.cos(x); }, 100); }
```

“Program to an interface, not to an implementation” is one of the most useful object-oriented design practices. For the example above, the alternative would be to create a method for each of the functions we want to use to compute the sum or modify a single method repeated when we need a different function. In either case, we would experience either code repetition or code modification—consequences of programming to an implementation. In contrast, at the time to implement *functionalSum*, we do not need to know what function we might use; that’s the beauty of programming to an interface. The method’s implementation is stable. For a different function, we write new code to implement the interface as opposed to modifying existing code.

Because of the complexity associated with inheriting multiple classes, most object-oriented languages support only single inheritance. However, a class can implement

multiple interfaces. As discussed earlier, interfaces enable objects to possess additional properties and behaviors. Consider the following abbreviated subclass of *List*. By implementing interfaces *Copyable* and *Iterable*, *MyList* is enabled with additional properties. The small interfaces provide just enough operations for the needs of the operation *processOrders*. This operation is more likely sustainable as it is also prevented from coupling with other implementation details of an order list and its underlying structures.

```
class MyList<T> extends AbstractList<T> implements Iterable<T>, Copyable<T>{ .... }
interface Copyable<T>{
    MyList<T> copy();
}
interface Iterable<T>{
    Iterator<T> iterator();
}
interface Iterator<T>{
    boolean hasNext();
    T next();
}
void processOrders(Copyable<Order> orders){
    Iterable<Order> tempOrders = orders.copy();
    Iterator<Order> it = tempOrders.iterator();
    while(it.hasNext()){
        action.use(it.next());
        ....
    }
}
```

To elaborate further the idea of programming to an interface, consider the software components illustrated in Fig. 3.2 (which is a component diagram we will introduce later). It is a scenario of an order component interacting with two other components. The component *Customer Repository* relies on an interface to update its customer information. Such an interface might look like:

```
interface CustomerLookUp{
    String updateCustomerEmail(String custId);
    String updateCustomerPhoneNum(String custId);
    ....
}
```

The component *Order System* is where customer information is likely to be updated and hence is responsible for implementing the above interface (a ball notation indicating the implementation of a provided interface). Component *Customer Repository* requires this interface (with a socket notation) to implement its own operations without knowledge of how the interface methods would be implemented (i.e., programming to this interface). Meanwhile, the *Order System* might be just one of the systems that could implement the

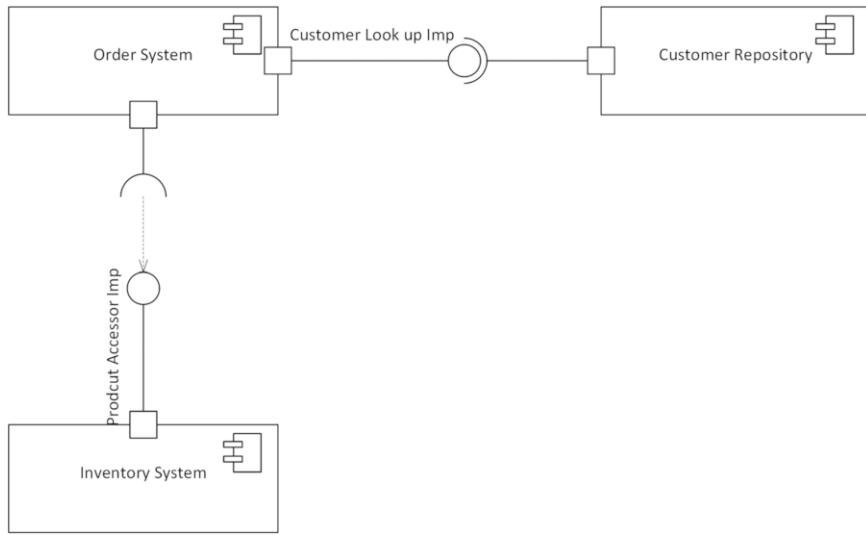


Fig. 3.2 Component communication with ball/socket interfaces

interface. An in-person store system, for example, could also provide an implementation of the interface. A plausible scenario is that the component *Customer Repository* depends on a runtime instance of the interface whether it is an implementation of an in-person store or an online store operation. Similarly, *Order System* needs an access to product information for its implementation to work. For example, the component might be implemented using the following interface:

```

interface ProductAccessor{
    ProdDetail getProductDetail(String prodId);
    boolean checkProductAvailability(String prodId);
    ....
}

```

Component *Inventory System* has detailed information for implementing the above interface. Again, *Inventory System* might not be the only component to be able to implement the interface; a third-party inventory system could do so also (as a reliable order system depends on multiple product sources). To be more specific, a class in the component *Order System* may provide an implementation of the interface *CustomerLoopUp*. Figure 3.3 illustrates a possible scenario where implementation of the interface is provided in class *CustomerManager*.

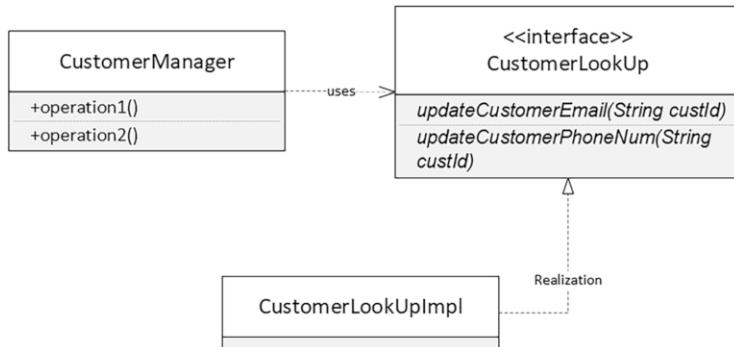


Fig. 3.3 What might happen inside component Order System

3.3.4 Interface Segregation

When a class implements multiple interfaces, it provides flexibility for users to choose the right property or behavior to use. When an object is typed with an appropriate interface, it is limited to access only the operations defined in the interface (without casting), and this is a safe practice of “programming to interfaces” while providing good verifiability of the program. It is thus beneficial to design small interfaces based on a conceptual decomposition to provide diverse needs of an object’s behavior. When a need arises, we can write a sub-interface by extending one or multiple small interfaces. This strategy of using small interfaces to provide flexibility of software modeling is commonly known as *Interface Segregation Principle* (though the original form of the principle was “no code should be forced to depend on methods it does not use”). Figure 3.4 gives a scenario that the concept of shipment can be conceptually decomposed into sub-concepts about goods type, goods dimension, and transportation means (among other possibilities) with separate interfaces. These interfaces can be used independently or in groups to satisfy the needs of different application scenarios. For example, a type `GoodsDescription` could be formed by extending interfaces `Dimensions` and `GoodsType`.

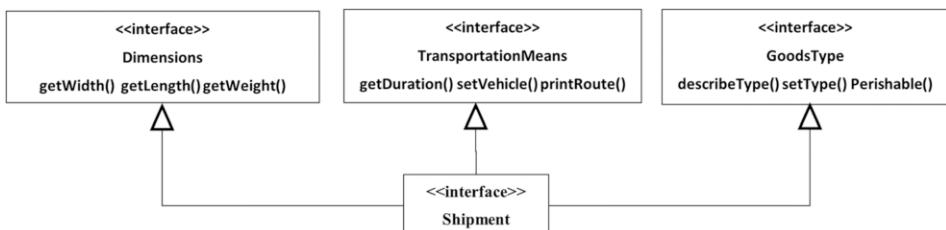


Fig. 3.4 Example of using small interfaces

Segregated interfaces may also facilitate capturing similarities or common behavior among unrelated entities. For example, *Student* and *CompanyCEO*, as object types, model two very different entities. However, a student and a CEO may both serve on a school's advisory board. Thus, both types can implement an interface such as one below for a member to respond to a board event, as in the following method:

```
interface IBoardMember{
    void act(BoardEvent e);
}

void processBoardEvent(List<IBoardMember> members){
    BoardEvent e = getEvent(View.getInput())
    for(IBoardMember m : members){
        m.act(e);
    }
    ...
}
```

By using small interfaces, we can appropriately introduce desired behavior among very different objects without artificially forcing stricter, more complex inheritance relationships.

Small interfaces can also be used to introduce desired properties among objects of the same type. For example, either one of the following interfaces can be used to introduce an order between objects. The first interface would more likely be implemented if we know in advance that objects of the intended type must be comparable. Thus, being able to compare to another object is part of an object's behavior: *obj1.compareTo(obj2)* (a positive integer means *obj1* is “greater than” *obj2*).

```
interface Comparable<T>{
    int compareTo(T o);
}

interface Comparator<T>{
    int compare(T o1, T o2);
}
```

If objects are not designed to be comparable, but we need to introduce an order at runtime, we can use the other interface *Comparator* to compare two objects: *comparator.compare(obj1, obj2)*. This can be convenient. For example, the following code demonstrates use of *Comparator* to sort an array of integers in a descending order using a static method from Java library class *Arrays*:

```
Arrays.sort(array, new Comparator<Integer>(){
    public boolean compare(Integer o1, Integer o2){
        return o2.intValue() - o1.intValue();
    }
});
```

3.3.5 Section Summary

Interfaces are the primary tool we use to design object types. They provide separation between use of objects' operations and their implementations. This separation is critical for code sustainability and extension. We have discussed how interfaces are used to address various design needs. To summarize, we use interfaces to:

- Realize data abstraction and enable polymorphic behavior of the operations
- Improve module reusability, flexibility, and sustainability by applying “programming to an interface”
- Introduce or extend objects’ properties and behaviors
- Capture behavioral commonality of heterogeneous objects
- Support conceptual decomposition and reorganization to support flexibility of object modeling

However, use of interfaces is not without pitfalls. For example, ad hoc use of interfaces to extend object’s behavior with multiple subclasses can lead to a deep hierarchy with increased complexity and vulnerable behavioral relations. Less thoughtful interfaces with missing operations may lead to bloated interface hierarchies with weak conceptual connections. Use of interfaces doesn’t necessarily guarantee the benefits it may bring and should always be proceeded with abundance of caution and tradeoff consideration.

3.4 Abstract Classes and Design of Type Hierarchies

Abstract classes play a special role in the construction of object types. An abstract class provides a partial implementation of an object type while containing some abstract methods. Like a concrete class, an abstract class may contain constructors, which can only be invoked by a subtype’s constructor to initialize instance variables visible in its scope. Classes that extend an abstract class can be either concrete if they provide implementations of all abstract methods of the supertype or abstract themselves if they also contain abstract methods (inherited or newly introduced). A subclass of a concrete class can be abstract; this can happen if certain behavior becomes uncertain or must be altered. By design, implemented parts of an abstract class are reusable in subclasses.

3.4.1 Use of Abstract Classes

An abstract class can support data abstraction if the implemented portion is relatively stable. A widely used example of an abstract class is *Employee* class to model different kinds of employees in an organization. Employees of different categories do share many stable commonalities such as employee identification, basic company benefit, and being

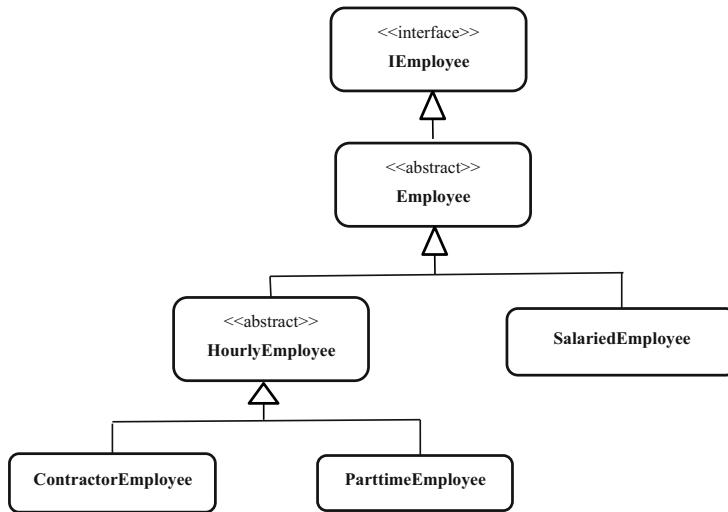


Fig. 3.5 An inheritance hierarchy to model employees

obligated to comply with organizational policies. However, employees are also different in their pay rates, roles, and responsibilities, among others. Therefore, certain methods such as *pay()*, *playRole()*, *assumeResposibility()*, etc. are abstract, and their implementations are left to subclasses like *HourlyEmployee*, *SalariedEmployee*, *Intern*, etc. These subclasses provide implementations based on appropriate employee categories they model. It is possible that some subclasses are abstract themselves. For instance, hourly employees might be part-time workers or contractors; as such, their pay rates and benefit packages can be very different. Figure 3.5 is a possible inheritance hierarchy for modeling employees based on the discussion above.

In large inheritance hierarchies such as those of data structures, abstract classes are often used to support code reuse. For example, several abstract classes (shown in Fig. 3.1) are used to implement *Collection*, *List*, or *Set* interfaces to provide essential code base and data representation shared in concrete classes. Since such intermediate use of abstract classes can be less strategic, implementation details can be complex, non-conventional, or even ad hoc (when it is used to provide remedial solutions to emerging issues). When abstract classes play such intermediate roles, they are not suitable for use as data types.

Perhaps the most important use of abstract classes is to provide application frameworks, which we will discuss in detail later. Briefly, an application framework provides a fundamental structure to support the development of applications. Such a framework often uses abstract classes to provide generic controls of an application, which can then be customized by implementing the abstract methods and required interfaces. For example, we may design an abstract class to control a login process by implementing how user inputs are collected, how a user can be registered, how in principle the input credentials are verified against

stored credentials, and the process of credential recovery and change. Certain subordinate processes are left as abstract methods such as the exact amount of user information to collect when they register, the exact information required for credential recovery, or the actual steps in a registration process. Some others can be abstracted as interfaces such as a credential encryption process. This kind of use of abstract classes has several benefits:

- It helps streamline control processes and applies to a variety of business scenarios.
- It simplifies the process of application development with consistent quality of the products.
- It allows rapid application development with templated control operations.

Design of an inheritance hierarchy can be either top-down or bottom-up. For hierarchies of data structures, a top-down design might be common as the relations between data structures are well understood. We may start with interfaces containing operations appropriate for any potential subtypes. For example, keyless data structures are fundamentally different from key-value data structures. Thus, we design an interface for each kind of data structures to define the essential operations. Since concrete “leaf” classes are well understood, abstract classes can be more effectively designed to maximize code reuse. A drawback of the top-down approach is potential exclusion or overlook of some properties and operations that may be uncommon or hard to envision initially but become apparent omissions later. For example, to support data processing in functional styles, Java’s Collections library added operation *Stream<E> stream()* to its *Collection* interface (as a default method) in JDK 1.8. But how the *Stream* interface can be implemented for every collection subclass without significant code duplication might have been a challenging design question.

A bottom-up development of an inheritance hierarchy may start with an understanding of potential concrete object types (or domain object models) that share the same “theme.” The better we understand the commonalities and variabilities among potential object types, the more reliable the abstractions are we can design to support operational needs of potential implementing classes (and the better we can design abstract classes to maximize the reuse). A top-level interface should contain the union of the operations in potential concrete classes. Sub-interfaces can be designed by decomposing the “theme” into “sub-themes.” A bottom-up design of an inheritance hierarchy appears to allow better opportunities for the design of polymorphic behavior.

Regardless of a design approach, application hierarchies (as opposed to hierarchies of data structures) should be generally shallow. Horizontal extensions of hierarchies are generally expected as programs are designed to extend. Vertical extensions of hierarchies should always be proceeded with caution even when they are justified.

3.4.2 A Case Study

We look at the design of the hierarchy of two-dimensional geometric shapes in Java package `java.awt.geom`. At the top, *Shape* is an interface that has operations that check relations between the “shape” and its bounding rectangle, identify the shape boundary, and retrieve an iterator over the pieces of a shape’s edge. Expectedly, there are abstract classes for shapes of different kinds. One of them is *RectangularShape* with possible use as a bounding rectangle to facilitate drawing shapes like arcs, ellipses, and, of course, rectangles. This class implements some of the abstract methods in *Shape* yet introduces new abstract methods whose implementations may depend on how value precisions are handled in drawing shapes. This class also provides overloaded but implemented methods for setting a rectangular frame to control shape drawing. Furthermore, there are subclasses dealing with concrete shapes. One of such classes is *Ellipse2D* for elliptic shapes including circles. This class is also responsible for implementing the methods inherited from root class *Object*. However, *Ellipse2D* is yet another abstract class to allow implementations of handling different value precisions to vary. *Ellipse2D.Float* and *Ellipse2D.Double* are two concrete subclasses, which are inner static classes of *Ellipse2D*. Java doesn’t support standalone static classes. Use of an inner static class might be based on considerations such as:

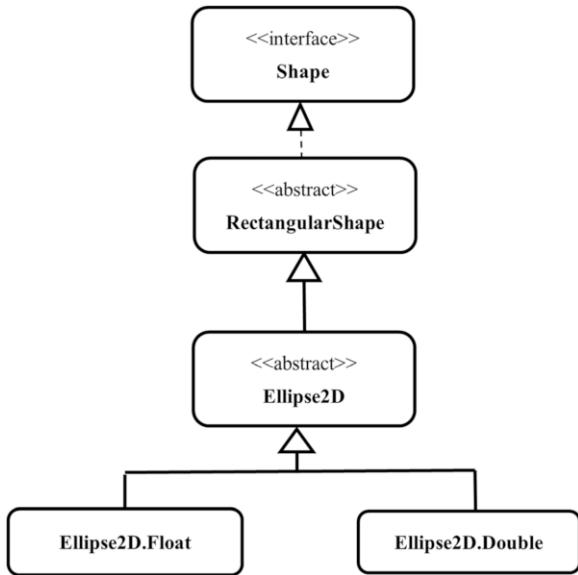
- The implementation task is relatively minor.
- Both inner class and outer class can be public in the same code file (whereas this is only one public standalone class allowed in a code file).
- It doesn’t require an instance to invoke a static member of the class.

Figure 3.6 is a branch of this shape hierarchy. This hierarchy first appeared in *JDK 1.2* in the package `java.awt.geom`, which was a significant upgrade to the shape drawing mechanism in the package `java.awt`. The redesign of Java’s geo-shape hierarchy is a very complex endeavor with many necessary design considerations. For example, designers must consider not only how graphics are rendered by the graphics pipeline rendering process associated with the hardware devices but also potential application contexts when images must be resized or transformed into a different representation (such as PDF) as resolution changes, including some extreme situations when extraordinarily large images are needed in certain applications. Besides, designers apparently also considered how different shapes can be drawn in a consistent manner, which may determine how abstract classes can be designed. Some of the decisions can be top-down such as graphics rendering details that apply to graphics of all kinds (2D or 3D), while others might be bottom-up like a consistent way of drawing a shape’s boundary.

It might be interesting to observe that a design can be motivated by the design effect we want to achieve. In *JDK 1.1*, the (abstract) class *Graphics* defines different versions of drawing methods to be implemented on various computing devices, such as:

```
graphicsObj.drawOval(100, 100, 150, 100);
```

Fig. 3.6 Partial hierarchy of shape types



The abstract `Graphics2D` class (a subclass of `Graphics`) provides more sophisticated controls over geometry, coordinate transformations, color management, and text layout. But much of the redesign might be based on a much-improved, more object-oriented drawing process:

```
graphicsObj2D.draw(new Ellipse2D.Float(100, 100, 150, 100));
```

Thus, drawing a sequence of shapes becomes a matter of a simple loop:

```
for ( Shape s : list ){ graphicsObj.draw(s); }
```

The success of the `java.awt.geom` package is partially because of the user feedback on `java.awt` (and perhaps also of the improved underlying API support for the graphics rendering process). In other words, we simply know more and better about what we need to do and can do. Unfortunately, we often design interfaces without a full understanding of the extent the abstractions would be used. For example, if we know an application uses only rectilinear shapes with no further information, then we might think of a top-level interface `Shape` like:

```
interface Shape{ void draw(Graphics g); }
```

If we know later that simple geometry calculations are also needed, we might consider more interfaces like:

```
interface ClosedRectilinearShape extends Shape{
    double getPerimeter();
    double getArea();
}

interface OpenRectilinearShape extends Shape{
    double getLength();
}
```

However, if we knew in advance that rectilinear shapes are used for puzzle games, then it would be even more helpful to understand how games are played, what the rules are, and the game constraints. Likely, none of the interfaces above would be appropriate. The more we know about an application, the better the chances we have to design sustainable inheritance hierarchies.

3.4.3 Section Summary

Abstract classes are often used to provide intermediate implementations between interfaces and concrete subclasses to support code reuse. They are also often used to provide application frameworks to support a family of (business) applications. When appropriate, we use abstract classes as data types too. For example, we might define a variable to be of type *Ellipse2D* (which is an abstract class), rather than *Shape*, to be more specific, relevant, and convenient without much side effect as the role of this abstract class is limited and its implementation is stable:

```
Ellipse2D oval = new Ellipse2D.Float(0, 0, 1, 2);
```

When abstract classes are used as a transition between interfaces and concrete implementing classes, we might face these design choices:

- Use a single layer of abstract classes, but each abstract class may implement multiple interfaces and provide multiple service methods to be used in subclasses. The tradeoff is a diminished value of the abstract classes as data types.
- Create multiple layers of abstract classes, each with leaner and more focused implementation. The tradeoff is a potentially a deep inheritance hierarchy that can be vulnerable to changes.

Inheritance hierarchies with abstract classes are often at least three layers deep with abstract classes in the middle. To keep hierarchies shallow, we often need to assess the

tradeoffs of introducing abstract classes. When abstract classes are used purely for code reuse, we might face the temptation of introducing new abstract classes when needs surface, which can lead to deep hierarchies. When code reuse is not significant or not well understood, we might think again about the need for abstract classes.

3.5 When to Avoid Inheritance

At the first sight, inheritance may look attractive. Consider the following simple example, where class *GeoCircle* extends *Circle*, and hence reuse the code.

```
class Circle{
    int radius; //constructors omitted
    void getArea( ){ Math.PI * radius * radius ; }
}
class GeoCircle extends Circle{
    Point center; //constructors omitted
    void moveCenter(int x, int y){ if(center != null) center.translate(x, y); }
}
```

Other benefits of inheritance include improved modifiability with more structural separation between a subclass and its parent, improved behavioral clarify with subclasses, and instance substitutability. Issues with inheritance include tight class dependencies causing structural rigidity; less efficient code performance (compared with code without inheritance); indirect code referencing with often negative connotations in code review, inspection, and testing; and potential unstable deep hierarchies. However, unless the hierarchy above is justified with additional information, we might just use the following class without inheritance with better clarity:

```
class GeoCircle {
    Point center; //constructors omitted
    int radius;
    void getArea( ){ Math.PI * radius * radius ; }
    void moveCenter(int x, int y){ if(center != null) center.translate(x, y); }
}
```

To implement additional behavior of a subclass correctly, we need full understanding of the implementation in a parent class. Extending an unfamiliar or less familiar class, due to hidden implementation details, can lead to unreliable or even erroneous object behavior with subtle mistakes. For example, the *Properties* class in *java.util* package is a subclass of *HashTable*. This subclass can be used to represent a persistent set of hash table-based properties as key-value pairs that are strings to be saved to or loaded from a stream. However, it is warned in the documentation that use of certain inherited methods, such

as *put* and *putAll*, is strongly discouraged because keys and values other than strings are not expected. As a result, the *store* or *save* method may fail for entries that are not string pairs.

Textbooks often describe the relation between a class and a child class as *is-a* or *is-a-kind-of* relation. Such a relation is often subjective and may largely depend on the names of the classes we choose to use, with no semantic indication on behavioral compatibility between a class and its parents. For example, *CircularBoundingBox* may sound like *a kind of BoundingBox* despite the possibility that the internal box of *BoundingBox* may be rectangular. *Point3D* is a *Point* except that the latter might model a two-dimensional point (x, y) . A perfect real-world relation doesn't necessarily translate into a good inheritance hierarchy either. For example, a square is a rectangle. But *Square* may not be a good subclass of *Rectangle*. The reason is that we would restrict the behavior of a rectangle (four sides must be the same) to make it compatible with a square with no apparent benefit from the inheritance relation.

In general, when we try to create a subclass that would require certain restriction on instance variables of a parent class, it may well be an inheritance relation where issues overshadow the benefits. As another example, consider *FirstYearStudent* as a subclass of *Student*. While the *is-a* relation appears fine, first-year student is not a lasting concept and may become invalid when certain properties change (such as the number of credit hours exceeding a certain number). Instead, we may use a property “seniority” in the class *Student* to allow the appropriate identification of first-year student, junior student, etc., although the value of this property may require an update when the number of credit hours changes. A better alternative is to use a method to compute “seniority” based on defined “number of credits” for each status. Similarly, to avoid creating a subclass for part-time employees, we can use a method to update the number of workhours of an employee. When the calculated number of workhours is below a certain threshold, the employee may receive reduced benefits.

Circumstantial properties of an object often can be better modeled with object's state than with an inheritance hierarchy. If an object may behavior differently, we may ask if the behavior can be addressed by applying polymorphism. In other words, we would rather extend a hierarchy “horizontally” by introducing new classes with overridden behaviors than “vertically” with subclasses so that the relations can be more rigid and brittle.

Our intuition about the relations between concepts to be modeled is often coarse and inaccurate. It can lead to the immature design of inheritance hierarchies that result in behaviorally incompatible objects, which often are subtle causes for software faults and failures.

We used the term code inheritance earlier to refer to inheritance practices to extend existing classes primarily for code reuse to introduce more specialized objects. While there are certainly plausible cases of code inheritance, it is often more plausible if we can avoid code inheritance given the fact that it often leads to more rigid dependency relations. Traditional modularity is still a safer choice for code reuse without much relational complexity. If reuse is needed between classes, object aggregation often is more beneficial than inheritance. Consider classes *Point2D* and *Point3D* that model 2D and 3D points,

respectively, with properties and operations appropriate for each type. To reuse the code of *Point2D*, we aggregate an instance, as shown below, instead of writing a subclass:

```
Class Point3D{
    Point2D pt;
    float z;
    void translate (float a, float b, float c){ pt.translate(a, b); z += c; }
    //other potential code omitted including constructors
}
```

For example, if *Person* aggregates an instance of data abstraction like *SocialStatus* and may take instances like *StudentStatus* or *WorkerStatus* when the person is either a student or an employee during different times, thus, we might be able to avoid subclasses of *Person* like *Student*, *Worker*, etc. The benefits of reuse through object aggregation include:

- Code reuse without forcing a more rigid class dependency
- Code reuse without exposing unused properties and operations to potential misuses
- Replaceable aggregated objects to allow behavioral flexibility without affecting the client code
- Complexity reduction of the enclosing module

Code reuse with aggregated objects can be viewed as “black box” reuse that a client is unaware of such a reuse. In contrast, inheritance for code reuse would be “white box” reuse, as a client may have (often needlessly) access to all inherited operations and properties. To elaborate further, consider the design of a movie collection with operations defined in the interface *IMovieCollection*. Furthermore, assume *List* is an interface and *ListImpl* is an implementation of *List*. We may either extend *ListImpl* like:

```
MovieCollection extends ListImpl< Movie> implements IMovieCollection{ .... }
```

or aggregate a list instance like:

```
MovieCollection implements IMovieCollection{ List<Movie> lst = new ListImpl(); .... }
```

The former has the benefit that an instance of *List* is replaceable by an object of *MovieCollection*. But the negative effect of exposing operations in *ListImpl* to client code (with potential undocumented misuses) may well overshadow the benefit. The implementation of *MovieCollection* would be more tightly related to that of *ListImpl*, leading to certain degree of rigidity in client code (that may access operations of *ListImpl*). In comparison, the latter approach also allows reusing the code of *ListImpl*. However, unused operations of *List* would not be exposed to client code. Furthermore, we could swap the instance of *ListImpl* with an instance of another implementation of *List* without affecting the

client code. In other words, the relation between client code and an implementation of *List* would be loose. The tradeoff is the loss of substitutability with an instance of *MovieCollection* when type *List* is expected. In general, if polymorphism is not a factor of the decision, using aggregated objects for code and behavioral reuse is almost always more appropriate than using inheritance.

3.6 Subtyping with Consistent Object Behavior

Subtyping refers to any process of creating subtypes ensuring behavioral compatibility between a subtype and its supertypes. Mechanically, subtyping is the same process as code inheritance, which is also known as subclassing. Subtyping seeks behavioral soundness of an inheritance relation, whereas subclassing may not. If we need a subclass of another class, we are not constrained by anything other than the syntactic rules of the language. This is the nature of subclassing. However, if a subclass is intended to be a subtype, this data type must be behaviorally compatible and consistent with its supertypes.

3.6.1 Representation Invariants

The static behavior of an object is determined by its static properties and methods and is uniform across all objects of the same type. An object’s instance methods, when invoked, determine how object behaves as the object’s state being altered. An object’s state is defined by the current values of its instance members representing object’s data. This collective value state is generally constrained based on the entity being modeled. For example, if class *InsurancePolicy* has an instance variable “age,” depending on a policy, an appropriate age range, such as between 18 and 75, would be part of a data representation constraint and must be preserved throughout the lifetime of an object. This constraint is called the (representation) invariant of a class or simply *class invariant*. Instance operations must not invalidate the class invariant to ensure that an object always behaves expectedly and predictably. A class must maintain not only its own class invariant but also all its ancestors’ class invariants to ensure behavioral consistency and compatibility of an entire type hierarchy.

Invalidation of class invariant may potentially cause an application to malfunction. For example, if *Square* inherits from *Rectangle*, its class invariant must include “length = width.” Methods inherited from the parent class, such as *setLength*, may invalidate the class invariant of *Square*. A subclass is subject to the same invariant constraints as its parent, but possibly more due to additional data introduced. To ensure the validity of class invariant at all times, we may group all data constraints in a method to make an assertion about the constraints. An assertion should be made before and after a mutator operation, as demonstrated below.

```

class Student {
    String name; float gpa; int enrolledCreditHours;
    void aMutatorMethod(){
        if ( !assertInvariant() ) throws new RuntimeException( "Invariant assertion failed at method entry");
        // details of method body ...
        if ( !assertInvariant() ) throws new RuntimeException( "Invariant assertion failed at method exit");
    }
    private void boolean assertInvariant(){
        return (name != null && !name.equals("")) && (gpa >= 0.0 && gpa <= 4.0) &&
               (enrolledCreditHours > 0 && enrolledCreditHours < 20);
    }
}

```

3.6.2 Liskov Substitution Principle

Protecting class invariants is, however, only one aspect of ensuring subtype soundness. A subtype must also respect all specifications of the parent operations when these operations are overridden. In other words, a specification we impose on an overridden method must include the specification of the inherited method, though possibly more. The *Liskov Substitution Principle* (LSP) provides the essential guideline for subtype soundness (Liskov, 1988):

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 , then S is a subtype of T .

To elaborate, suppose method q , as follows, defines a property of an object type:

```
void q(T x){ /* implementation */}.
```

If one invokes the method with the statement $q(y)$, where y is an instance of S (which is a subtype of T), the LSP says that $q(y)$ must remain valid with full predictability. Practically, when property q is invoked with an instance of S , a user should not be surprised about its outcome. This casual interpretation of LSP is known as *Principle of Least Astonishment*. For example, consider the class *Rectangle* below:

```

class Rectangle{
    int length, width;
    void setLength(int l){ length = l;}
    void setWidth(int w){ width = w;}
    int getArea(){ return length * width; }
}

```

Let *Square* be a subclass of *Rectangle*. The incompatible operations of the parent must be redefined in the child. For example, the setters must be overridden in *Square* to ensure the validity of objects, such as:

```
public void setWidth(int w){
    length = w; width = w;
}
```

Now, consider the following method of *getVolume* that returns the volume of prism:

```
class Prism{
    //other information is not important
    int getVolume(Rectangle r, int h){
        r.setLength(5);
        r.setWidth(4);
        return r.getArea() * h;
    }
}
```

As “expected,” the method returns number $20h$, which would be the case if r is an object of *Rectangle*. However, *getVolume(s, 1)* would produce $14h$ when s is an instance of *Square*. Thus, the substitution with an instance of a subclass results in an inconsistency with the “expectation”—a surprise to a client of *Prism*. This violation of the LSP is essentially caused by incompatible class invariants between the subclass *Square* and its parent. In fact, behaviorally questionable inheritance may easily lead to LSP violations. For example, suppose class *Point* is to model a two-dimensional point (x, y) with operations like *translate*, *scale*, etc. If class *Point3D* is a subclass with an additional instance variable *z* to model a three-dimensional point (x, y, z) , operations *translate*, *scale*, etc. would be overridden to be consistent with point transformations in a higher-dimensional space. As a result, if an object type uses an instance of *Point* in its operations, these operations may likely break if the instance of *Point* is replaced by an object of *Point3D* due to its behaviorally incompatible operations of point transformation.

Most often, however, violation of the LSP is caused by incompatible pre- and postconditions between an overridden method and the original version. We may change the specification of an overridden method that we might not be consciously aware, but the preconditions can only be weakened, whereas postconditions can only be strengthened. For example, consider the following method of a parent type with pre- and postconditions:

```
/* @pre { a > 0 }
   @post { b < 0 } */
int m(int a){ ... }
```

For an overridden method m in a child class, possible pre- and postconditions must satisfy:

$$@pre \{ a > r, r \leq 0 \} \quad @post \{ b < s, s \leq 0 \}.$$

The precondition is weakened in the sense that it is satisfied whenever the original precondition is. The postcondition is strengthened because it implies the original postcondition. For another example, consider the following skeletal method where A and B are object types:

$$B\ m(A\ obj)\{ \dots \}$$

To override m in a subtype, a weakened precondition means that the argument type could be type A or a parent type. This condition is known as *contravariance*, meaning that the inheritance relations between the methods and their arguments appear to be reversed. A strengthened postcondition of an overridden method means that the method could return an instance of B or a child of B . This condition is termed *covariance*, i.e., the inheritance relations between two methods and their returned object appear in the same direction. A postcondition also includes exceptions thrown in a method. For example, if a method throws an instance of *RuntimeException*, the overridden version of the method could throw an instance of a subtype such as *ArithmetricException* or *ArrayStoreException*. Throwing an instance of *Exception*, which is a supertype of *RuntimeException*, would be a violation of covariance and hence a violation of the LSP. In particular, an overridden method shouldn't throw an exception if the original one doesn't. Contravariance and covariance can generally be enforced by a compiler, though most compilers still only require that argument types be invariant. Interestingly, covariance is generally enforced by a compiler. To summarize, when we override a method in a subclass,

- Preconditions cannot be strengthened, and postconditions cannot be weakened.
- Invariants (i.e., all invariants of supertypes) must be preserved.
- Supertype behavior may not be modified, as doing so may compromise the compliance of LSP by the derived classes.

3.6.3 Design by Contract

In general, LSP compliance is still maintained mostly by hand. Thus, when we design an inheritance hierarchy, we must also consider the implications on LSP compliance. As a hierarchy extends deeper, class invariants become more complex. The class invariant at a leaf type of a hierarchy is the union of all class invariants of its parent types. A violation of the LSP at any level of an inheritance hierarchy affects all its branches. As a hierarchy becomes deeper, maintaining a class invariant becomes more difficult. Here are a few good practices to keep class invariants clean, less complex, and manageable:

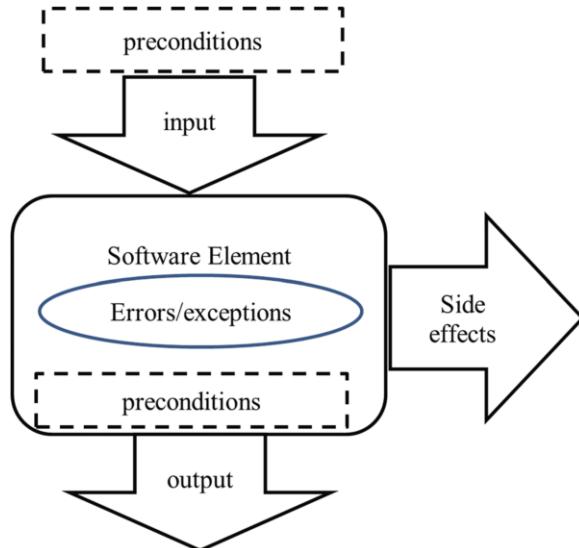
- Introduce only necessary instance variables.
- Keep variables local to a method whenever appropriate, as elevation of local variables to be instance variables increases the complexity of the class invariant.
- Discourage variable sharing across methods.
- Distinguish private instance variables from protected variables to keep inherited invariants at a minimal.

LSP compliance is part of a broader notion: *Design by Contract* (or DbC), coined by Bertrand Meyer in connection with his design of the Eiffel programming language (Meyer, 1986). The central idea of the notion is a metaphor, i.e., how elements of a software system collaborate with each other based on mutual obligations and benefits is like module creators and users being “suppliers” and “consumers” in a business environment.

As illustrated in Fig. 3.7, DbC applies to the design of a software element in the following way:

- The element must be implemented to protect preconditions.
- The input must satisfy the preconditions.
- The output must satisfy the postconditions.
- Potential exceptions, error conditions, and side effects must be explicitly stated.
- Contractual information must be documented.
- The class invariant (and similar behavioral properties) should be assumed on entry and guaranteed on exit.

Fig. 3.7 Design by Contract illustration



DbC has its roots in work on formal verification, particularly a formal specification known as the *Hoare's Logic* (Hoare, 1969), which formalizes module obligations as $\{P\}C\{Q\}$, where P and Q are pre- and postconditions and C represents “command” (or module execution). This “triple” describes how the execution of a piece of code changes the state of the computation and can be used to provide axioms and inference rules for all the constructs of a simple imperative programming language. This logic can be summarized with the following three questions that the designer must repeatedly answer in a contract:

- What does the contract expect (preconditions)?
- What does the contract guarantee (based on what a command does)?
- What does the contract maintain (postconditions)?

Many programming languages have assertion statements to provide support for *defensive programming* against potential defects. However, DbC considers these contracts being so crucial to software correctness that they should be part of the design process. Specifically, DbC advocates writing appropriate code comments to make an implementer aware of the contract, developing the assertions first, and making contracts enforceable by a test suite.

3.7 Lazy Object Creation Allowing Delayed Decision-Making

To reinforce the understanding of an object-oriented design technique we introduced in the last chapter, we revisit the technique with more elaboration in this section. Encapsulating difficult design decisions in modules was a design principle of the structured design, popular in the 1970s. By extension, if high-level modules depend only on modules hiding vulnerable decisions and details, they are more likely to sustain. The design technique we revisit here is a similar principle that underpins object-oriented design.

A high-level functional module may rely on low-level details about business decisions, data processing logic, or computational algorithms. These details often appear in a conditional or selection statement with multiple branches. Consider the following code to determine the prices of a commercial product line.

```
interface IProductPricing{
    double getSalePrice(double unitCost, double fixedCost, boolean cond);
}
```

Changes to business requirements, business process, or business policies may result in changes in price calculations and processing logic. Such changes can also lead to addition of new products that need price calculations or new business models that require the

products to be priced differently. Such multi-branch conditional statements create code rigidity that makes code modification (due to changes in details) an error-prone process. We have seen that when high-level modules can be constructed using abstractions, they tend to be more adaptable and extendable. For the example above, we can design a data abstraction to abstract away the details of price calculation, so that the multi-branch conditional statement is replaced by a polymorphic method invocation with an appropriate object dispatched at runtime. As mentioned earlier, this technique of *lazy object creation* is about avoiding object creation until it is needed. To achieve that, we create an abstraction such as:

```
class PricingFactory{
    static IProductPricing getPriceSetter ( int productID ) {
        if( productID == ITEM_C){
            return new ProdCPriceSetter (productID);
        }
        else ...
    }
}
```

To dispatch a delayed object, we use a “factory” method as follows:

```
salePrice = PricingFactory.getPriceSetter(productID).getSalePrice(unitCost, fixedCost, cond);
```

The multi-branch conditional statement earlier is then replaced by one line with an interface-inheritance structure:

```
if( product == ITEM_C ){
    price = unitCostC * (1 + 0.3)
    if( addProcessingCost ) price += processingCost * 1.2;
}
else if( product == ITEM_D ){
    price = unitCostD * (1 + 0.4)
    if( addShipping ) price += shippingCost * (1 - standardDiscount);
}
else if( product == ITEM_E ){
    price = 151;
    if( needInsurance ) price += insuranceCost + PROCESSING_FEE;
}
```

The following code is an implementation of the interface above, which can be delayed until the details become certain. For complex business objects, we may create a base class to absorb the commonality of all potential subclasses.

```

class ProdCPriceSetter implements IProductPricing {
    private int prodID; //other variables, properties, methods omitted
    public setProdId (int prodID){
        productID = prodID;
    }
    public double getSalePrice(double unitCost, double fixedCost, boolean cond){
        double price = unitCost * (1 + 0.3)
        if( cond ) price += fixedCost * 1.2;
        return price;
    }
}

```

High-level “if-else” structures often require upfront decision-making on details. We can “invert” this dependency, as shown in the example above, using an interface inheritance to allow delay of detail-oriented decisions. Conditional statements didn’t disappear; however, they were buried into a “factory” method. A factory method must be modified when products are added or removed. But since factory methods are structurally simple with a sole focus on object creation, modifications can be done safely. This is also a tradeoff. We trade more rigid, vulnerable code statements in high-level modules for more sustainable statements supported by interface inheritance, leaving vulnerability to the bottom code layer where changes and modifications can be safely made. Added structural complexity may also cause slower execution. However, not all high-level conditional statements need such object-oriented treatments; only the vulnerable ones do.

3.8 Object-Oriented Design in the Large: Design Principles

Object-oriented design can be summarized as five design principles, acronym-ed *SOLID*, which stands for:

- Single Responsibility Principle (an object type should model one thing)
- Open-Closed Principle (software elements should be open for extension, but closed for modification)
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

These five principles have been mostly discussed in this and the prior chapter either explicitly or implicitly. Open-Closed Principle is a more abstract design principle and philosophical in nature. It is, in fact, behind much of the discussion we have had so far about what object-oriented design is to achieve. Though we can’t generally make code closed to modification in any absolute sense, we have shown how to make code change and modification more manageable and less harmful.

Dependency Inversion Principle (DIP) is also behind our discussions of creating data abstractions to “rewire” the dependencies. This principle means the following:

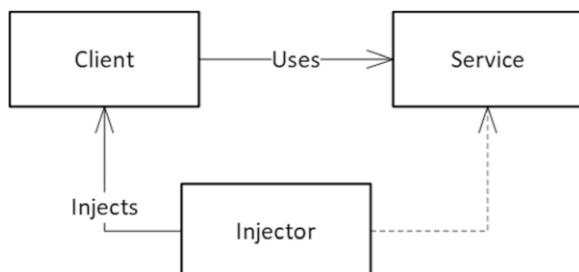
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details (i.e., concrete implementations) should depend on abstractions.

High-level program modules are those that are directly responsible for the software features, whereas low-level modules are those implementing business rules, algorithms, data structures, and software tools to be eventual executors of the features. When details are vulnerable, dependency of high-level modules on details makes code rigid and brittle when the code must be modified due to vulnerable details (a violation of the Open-Closed Principle). The technique of lazy object creation is used to invert the dependencies. The corollaries of DIP include:

- Instance variables should only be declared using interfaces or abstractions whenever appropriate.
- All concrete class packages must connect only through interface or abstract class packages.
- No class should be derived from a concrete class.
- All variable instantiations should require implementations of creational patterns such as the factory method or other factory patterns (including use of DIP frameworks).

When we access a software service (a module, a class, or a combination of such elements) directly, it means that the client code would be coupled with the service detail. This direct code coupling would make change or modification of the service difficult and is a violation of DIP. This direct coupling can also be understood as high-level modules being directly “controlled” by the low-level details in service code. This “control” needs to be “inverted.” Thus, DIP is also known as *Inversion of Control* (or IoC). To properly apply DIP or IoC, we need a related design technique named *Dependency Injection* (DI). To do so, we introduce an *injector*, which is like a “broker” that connects client and service through service abstractions. The concepts associated with DI are described below and illustrated in Fig. 3.8:

Fig. 3.8 Relations in dependency inversion



- Client: an entity that depends on a service abstraction
- Service: service implementations that depend on the service abstraction
- Injector: a container that “injects” concrete services into a client (i.e., an injector invokes a client application after connecting the application to the correct services required)

The following skeletal code demonstrates, in principle, the use of the three DI concepts described above using an example we had earlier:

```

class Client{
    IproductPricing priceSetter;
    //instance data needed like: product id, unit cost, boolean control, fixed cost
    void setService(IproductPricing service){ priceSetter = service; }
    void doApp() /* use of the service in the application */}
}

class Injector{
    Client c;
    Injector(){ c = new Client(); }
    void doCoordination(){
        //input for product id, unit cost, boolean control, fixed cost
        c.setService(PricingFactory.getPriceStter(c.productID));
        c.doApp();
    }
}

```

Frameworks can be developed to implement DI strategies. The main player in such a framework is a DI container (or injector) to facilitate the process of service injection and disposal of dependencies when appropriate. More specifically, this DI container includes the following elements (Fowler, 2004):

- Registration: an entity that registers a set of service types, creates the dependency between a client and service types it uses, and implements a way of dependency mapping
- Resolution: an entity that includes methods to resolve the dependency between clients and specified service types, create objects of specified service types using factories, and inject the required dependencies
- Disposal: an entity to manage the lifecycle of dependent objects and dispose the dependencies when necessary

There are many open-source or commercial DI (or IoC) containers available for commonly used web application development platforms.

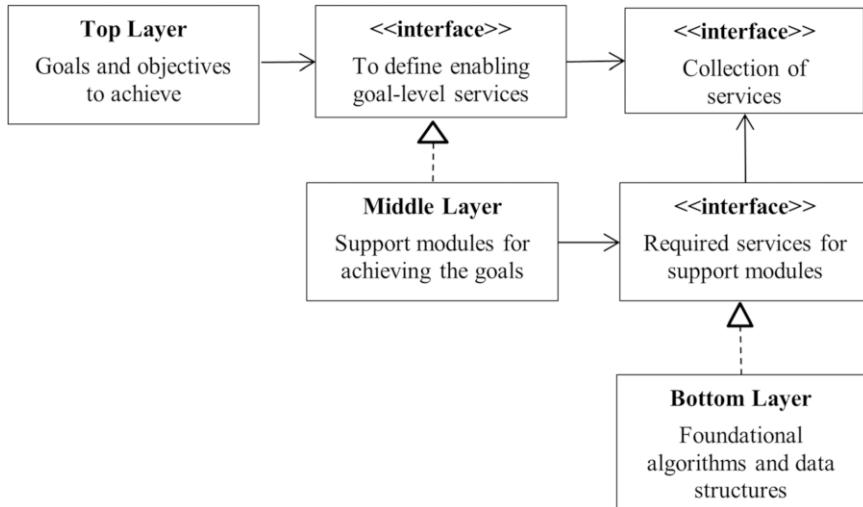


Fig. 3.9 Software layers and their dependencies

Use of (micro)services has been a design trend in the last decade to address the mobility of software components. Guided by the Interface Segregation Principle, services are designed to be nimble and replaceable. A software application generally has at least three implementation layers. The top layer is to specify what to do and the goals to achieve. The middle layer addresses how to achieve the goals and what is needed to do so. The bottom layer contains “workhorses.” Figure 3.9 illustrates the three layers and their (inverted) dependencies. The box at the upper right corner represents a collection of services. With themed and segregated interfaces, these services can be implemented separately or jointly.

Abstractions are also a means for communication between processing code and the support it needs from various software services defined by interfaces and implemented by low-level modules. These communications invert potential controls between high-level modules and low-level details. In many business applications, this three-layer software structure, centered at service abstractions, allows for loosely coupled code and design of small services to maximize software extendibility. It’s not difficult to see that the SOLID is a coherent set of object-oriented design principles to ensure design sustainability as software evolves.

3.9 Summary

The core of object-oriented design is to create effective abstractions that support interface inheritance. Concrete classes, as implementers of data abstractions, are mostly used for details of software operations and stay at the lowest level of any hierarchy. Abstract classes

can be used to provide common code base across all implementations of interfaces. But they are often used to provide generic controls of software processes that underlie overall software structures. Design of inheritance hierarchies often has significant implications on software sustainability. We have discussed in this chapter the benefits and potential issues with inheritance hierarchies and ways to use them effectively. Effective inheritance hierarchies not only support design sustainability but also keep structural complexity of the software at a manageable level, which would significantly contribute to software reliability.

As an effective alternative mechanism (to subclassing) for code reuse, using aggregated objects provides a more flexible, loosely coupled way for objects to collaborate. The soundness of a type system is key to software correctness and reliability, for which the LSP provides a primary guidance. However, LSP violations can be much less of a concern if we appropriately map software requirements to object types with well-defined behaviors and relations using interface inheritance.

Finally, dependency inversion is an underpinning principle of object-oriented design, though it was based on modular design principle that high-level modules should rely only on modules that encapsulate processing details. Finally, how to apply SOLID is generally not a “case-by-case” practice. It’s a collective guiding principle we apply when we set our design goals and priorities, do analysis to map software requirements and constraints to software elements, and weigh on design tradeoffs when we make design decisions. Despite the promises of object-oriented design, a design we seek is always a balance of the type of software we build, the technologies (including the programming languages) we use, and a development process we adopt.

Exercises

1. Find a reference for floating-point arithmetic, which is how data type “float” is represented in a computer. Then, describe how the “float” type (as in Java) fits the definition of abstract data type.
2. List the drawbacks of using a concrete class as a data type.
3. What is the difference between a generic type and subtype polymorphism?
4. We can display a graph of $f(x)$ vertically by displaying $f(x)$ number of spaces and then a connection character such as “*” for each x value in a defined range. Write a static method that displays a graph of any given function f . (Note that if $f(x) < 0$ for some x values, you can graph $f(x)-L$ instead, where L is a lower bound of the function values.)
5. Write three interfaces modeling customer, server, and cooker in a restaurant operation. Then, write a method (static or instance method) to simulate a process from customer taking an order to paying for the meal using only the interface methods.

6. In an application about class enrollment, suppose a list of student objects is needed. There are different kinds of students on campus—full time, part time, non-degree, exchange students, etc. The application must apply registration restrictions on each type of students and charge tuitions differently too. However, it is desired that high-level conditional statements be avoided.
 - (a) What might be an appropriate abstraction to model such a student entity or hierarchy?
 - (b) Write some code about a registration process that uses the abstraction.
7. The Java collections framework has interfaces that cover (almost) all commonly used linear data structures. But there is not an interface for stack data structures.
 - (a) Speculate a reason why.
 - (b) If you were to add an interface *Stack*, where would you put it in the hierarchy, and what might be possible classes in the existing hierarchy that would implement the interface?
8. Write a static method that can sort an array of elements of any data type with a sorting order that can be customized.
9. The following interface is designed for simulating a card game. Divide the following interface into multiple interfaces and provide reasons for the division.

```
interface CardGame{  
    initiateGame();  
    playGame();  
    shuffleDeck();  
    dealCards();  
    playerPaly();  
    declareWinner();  
}
```

10. How to model entities may depend on the roles they play in software operations and cannot be predetermined. Figure 3.5 is a hierarchy for modeling employees of an organization without much background discussion.
 - (a) Think of a scenario that may support the hierarchy.
 - (b) Also think of a scenario that the abstract class modeling a part-time employee is not needed.
 - (c) If you use only an interface to model employees (and nothing more), list pros and cons.

11. Given the following classes

```
abstract class Product {  
    protected float price;  
    public Product(float p){ price = p; }  
    public int getPrice() {  
        return price;  
    }  
    public int getSalesTax() {  
        return (int)(getPrice() * 0.06);  
    }  
    abstract Product recommend(Product prod);  
}  
class ProductForSale extends Product {  
    private float discount;  
    public int getPrice() {  
        return (int)(super.getPrice() * discount);  
    }  
    //This class is incomplete.  
}
```

- (a) Write a constructor for class *ProductForSale*.
- (b) Implement (in *ProductForSale*) the abstract method *recommend* in such a way that it returns an instance of *ProductForSale*. Can you change the parameter type, or return type, or both to *ProductForSale* when overriding? Why or why not?
- (c) What is class invariant for *ProductForSale*?
12. In theory, an instance method can be invalidated by an overridden method of a subclass (e.g., by throwing an unsupported operation exception). Is this a violation of the Liskov Substitution Principle? Why or why not?
13. We design software modules based on software requirements. Considering the notion of “Design by Contract,” describe the relationships between software requirements and modules’ contracts.
14. Multi-branch if-else or switch statements are common in scientific applications, and those conditions are often scientifically formulated. Should these conditions be replaced by an interface inheritance hierarchy and bury the conditions in implementing classes? Why?

References

- Fowler, M. (2004). Inversion of control containers and the dependency injection pattern – Forms of dependency injection. <https://www.martinfowler.com/articles/injection.html#FormsOfDependencyInjection>.
- Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–580.
- Liskov, B. (1988). Keynote address – Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5), 17–34.
- Meyer, B. (1986). *Design by Contract*. Technical Report TR-EI-12/CO. Interactive Software Engineering.



Design of Methods

4

4.1 Overview

A module is an independent program or code unit that can be referenced by certain means. A *method* (as a programming unit) is a module, so is a group of methods if a language has a construct to reference such a collection. A class is a module, as is a (Java) package containing a group of classes. In this fashion, a module can also be a combination of methods and classes. A method, with a name to represent either a *procedure* (with no “value” returned) or a *function* (to return a “value”), is usually the smallest unit of a module that can be independently referenced. Thus, design of methods is essential to module design. A method encapsulates a computational task using the information from the caller’s code environment. A method typically has parameters to deliver the information or data the method needs to complete the task. A method can be independent, part of a bigger module, or associated with an object. When a method is associated with a class, it is often an instance method. An instance method is often a *void* method (like a classic procedure) with a task resulting in modifications to an object’s state (consisting of current values of instance data fields). An independent method, often with a modifier *static*, can be a function that returns computed information through a return statement or out-mode parameters (if the language supports). An independent method can also be a void method to update values of variables in an external scope.

Despite different forms of a method, discussion in this chapter about various aspects of method design applies to methods of all kinds. In principle, a method should accomplish a well-defined task with precisely stated pre- and postconditions. A well-designed method is not only cohesive but also loosely coupled, reusable, and of good readability. A method that requires only a few lines of code hardly needs a design. However, when a method implements a complex task, it often must handle the task collaboratively with other methods and with a caller’s code environment. Methods are essential units we use to

decompose a complex problem. Decompositions result in software elements where methods are embedded. A good design determines element compositions in such a way that manages the complexity, facilitates element integration, and makes the code more sustainable.

We will start with a discussion about the essential characteristics of a method. We will then proceed with method cohesion and coupling. We will provide a brief discussion about refactoring (or redesign) of methods as a tool to make design improvements. Methods are essential work units, and their correctness can be critical to software correctness and reliability. A method's specification is part of procedural abstraction and can be crucial to a method's correctness and verifiability. Therefore, we will discuss later in the chapter about ways to make method specifications more accurate. Finally, we will show the importance of functional reliability of a method through a case study.

4.2 Essential Characteristics of a Method

Our ability to decompose a complex problem depends on our ability to abstract away the details that can be considered later. Methods provide a low-level abstraction to encapsulate process details such as business rules, proprietary practices, algorithms, details for data source connections, or basic data structures. A method can also use other methods to delegate some of its implementation complexity. Thus, methods can also provide higher-level abstractions to work collaboratively with lower-level counterparts. To appropriately design a method to balance its role between detail encapsulation and high-level modular abstraction, we need an understanding about general module attributes expected and the design goals we intend to achieve.

4.2.1 Procedural Abstraction and Modularity

Often, we design a method to meet not only our current needs but also potential needs we may perceive. For example, if we want to use a method to construct an URL containing a query string such as <https://example.com/path/to/page?name=ferret&color=red>, we might design the method header like:

```
static String urlString(String prefixStr, String separator, delimiter, Map<String, String> queries)
```

This design is based on what's changeable in such a URL—a prefix string, a separate, the exact content of a query string (with multiple key-value pairs), and a delimiter used. However, if we add two more parameters to make the method more general, such as *String fontName* and *int fontSize*, the method header would be less understood with generality of little practical value.

We use the term *procedural abstraction* to describe the notion of abstracting away variations of method implementations with appropriately designed parameters. Thus, an invocation of the method only represents an execution of the method with a particular set of parameter values. Procedural abstraction has the following characteristics (Liskov & Guttag, 2000):

- Parametric abstraction: Parameters of a method abstract away the input data the method uses, so that the method invoked repeatedly with different values of parameters completes different variants of a task.
- Specification abstraction: A specification of a method abstracts away the implementation details. It says what the method is to do, but not how to do it. Thus, implementers are free to choose what might be considered the most appropriate way to implement or alter an existing implementation without necessarily affecting the applications where the method is used. A method specification typically includes a description of the effect of a method invocation, as well as pre- and postconditions of the method. A method has the described effect only when its preconditions hold. What a method does can be extended, but only to properties that can be derived by, or inferred from, its postconditions.

It may appear that method design is about design of an implementation. However, an implementation is inevitably related to design of the parameters, which may in turn have implications on method cohesion and coupling (to be discussed later). An implementation must also satisfy the method specification. Thus, design of the specification, which defines the desired attributes of a method, is also part of a method design. Therefore, a method design is a comprehensive design “package.”

We use a module to group conceptually or semantically related elements. If conceptually helpful, a collection of modules can also be viewed as a module in a structural decomposition. Historically, *modular programming* generally refers to decomposing a program into modules, whereas *modularity* is a design concept to describe the degree to which system’s components, consisting of modules, may be separated and recombined. Modularity can also refer to both the tightness of dependencies between modular components and the degree to which the mixing and matching of components adhere the system’s architecture. Modularity encourages procedural abstraction, method reusability, and separation of method implementations from their uses. In general, implementation of a complex method may affect readability of the method, our ability to debug and modify the method, as well as the ways the method is used in other modules. Thus, modularity may also mean to decompose a complex task into modules of manageable sizes with manageable dependencies between the units.

4.2.2 Design Attributes of a Method

A method is to abstract a highly concentrated software action. The term *method (or module) cohesion* is used to characterize the degree of interaction within a module, i.e., the relatedness of code instructions of the method. It is easy to see that a highly cohesive method is easier to maintain and that a single task method is likely cohesive. It does not follow, however, that a method with multiple tasks would necessarily be less cohesive or incohesive. Cohesion is not about the number of tasks in a module.

Decomposition into modules means modules to work together collaboratively. The further a decomposition is, the more dependencies are between the modules. Thus, the term *method (or module) coupling* is used to describe the degree of dependency of a module with other modules. Functional decompositions mitigate software complexity and improve code comprehension, but decompositions result in module coupling too. Good coupling is intentional and well thought. Ad hoc coupling may lead to inappropriate module dependencies that can increase the vulnerability of a module to potential changes and reduce the reusability of the modules involved. It might be intuitive that tighter coupling makes modification to a method more difficult due to dependencies that changes in one collaborator may affect others operationally. Practically, cohesion and couple cannot be achieved optimally (modules are all highly cohesive with least amount of coupling); good modularity is to seek a balance between module readability, reusability, and, more importantly, the sustainability of the modules. In a balanced design consideration, we often do give loose coupling a priority, however, to maximize extendibility of a program. As Fig. 4.1 illustrates, for a given problem, there are a number of ways the problem can be decomposed into tasks, which would work together possibly in a messy way in one “giant” module to provide a solution. A good modular design is to seek appropriate groupings of

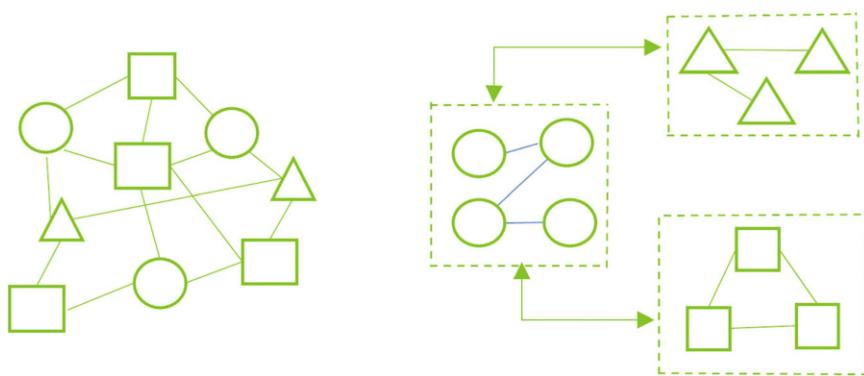


Fig. 4.1 Illustration of cohesion and coupling

the tasks to ensure that the cohesion among of the tasks is at an accept level while minimizing the level coupling between the groupings.

To modularize a program, we use a traditional principle for module organization: *separation of concerns*. Intuitively, unrelated parts of a system should be separated, and related parts should be grouped together to facilitate code reuse, make module updates easier, and allow for independent module development. Separation of concerns also decreases the chances that modifying one thing would adversely affect another. Appropriate separation also makes it easier to reason about the individual code units because there is less to be concerned about and more to be focused on. Besides, appropriate grouping also facilitates review, testing, debugging, and code refactoring. A well-designed grouping of parts in a module should ideally result in a well-defined, easy-to-understand, and sustainable interface (i.e., a collection of method headers). Separation of concerns is also behind some emergent design practices such as use of microservices that aims at design of functionally nimble and structurally mobile software modules.

The name of a method may imply the appropriateness of a method in terms of the role it plays in a larger module and its task specification. Therefore, naming of a method can also influence method design. For instance, name *computeSalesTax* conveys clearly what the method would do, whereas a method named *processOrder* leaves ample room to imagine about possible activities to be potentially involved. A method name is generally a verb or a verb phrase as it represents an action. The more specific the name is about the task, the less likely the task would represent an inappropriate abstraction. There are a few cases we can name methods with different naming styles. For example, a name like *isVisible* might be more appropriate if the method returns a Boolean value, whereas a name like *size* would be more conventional (to represent a property) than *getSize*. A third-person singular form of a verb might sound better when a method is invoked by an object such as *list.contains(a)*. But in any event, if we seem to struggle to find an appropriate name for a method, it might be that the method does more than it should and needs perhaps a breakup.

4.3 Cohesion of a Method

A method containing one simple task is likely cohesive, whereas a method containing multiple unrelated tasks is likely incohesive. But most methods we write lie between these two extremes in terms of cohesion. Indications of potential low cohesion of a method include:

- A method uses a relatively large of number of parameters.
- Only a part of the implementation uses the parameters.
- A significant number of code statements do not use declared local variables.

To quantify a cohesion assessment, we could construct a flowchart of a method and determine the number of independent paths that go through different statements. If a module is of high cohesion, most of local variables and parameters would be used in statements in most paths. Thus, the highest cohesion may be achieved when all independent paths use all variables (including parameters) in a method. But practically, a mental inspection is often adequate in assessing the level of cohesion.

Module cohesion can be divided into seven levels depending on the qualified amount of relatedness of the tasks. The two extreme situations described above would fit the descriptions of *functional cohesion* (the highest cohesion level) and *coincidental cohesion* (the lowest cohesion level). The other five categories are described below in the order of improved level of cohesion (though not linearly):

- *Logical cohesion* refers to a series of actions to be performed, but only one is selected at a time by the calling module with a parametric indicator evaluated in conditionals.
- *Temporal cohesion* means a series of actions to be performed, but all related in time.
- *Procedural cohesion* is like temporal cohesion with a difference that all actions are related by a procedure or an algorithm.
- *Communicational cohesion* strengthens procedural cohesion with an additional characteristic that all the actions operate on the same data.
- *Sequential cohesion* strengthens communicational cohesion further with an addition that one action uses the result of another.

Intuitively, a method that executes all tasks would be of higher cohesion than one that invokes only some of the tasks in any given call instance. A method with multiple tasks in which one task relies on another in a sequence without exception would be of higher cohesion, as the tasks are inseparably related, than a method where tasks merely operate on the same data. In other words, whether a method is cohesive is measured by the relatedness of the tasks, not by the number of tasks or by the level of complexity of the code. Thus, we can characterize the cohesion levels above in terms of relatedness of the tasks in a method as follows:

- *Coincidental cohesion*—Tasks are not at all related.
- *Logical cohesion*—Tasks may be unrelated, but each invocation executes only one task.
- *Temporal or procedural cohesion*—Tasks are bundled together in time or based on a procedure, though tasks may not be related to each other.
- *Communicational cohesion*—Tasks are not necessarily directly related to each other yet share the same theme and use the same data and are all executed in any single call.
- *Sequential cohesion*—One task can't be completed without another, typically in a sequential fashion.
- *Functional cohesion*—There is only one task (of possibly high complexity with support modules).

Often, cohesion of a method may be predictable simply based on the name of the method, the parameters, and the value returned (which are collectively called a *method's profile*). Consider the following two methods with only the headers shown. We can reasonably imagine that the first method would have tasks for verifying graduation requirements, computing grade point average, etc. These tasks are independent of each other yet integral parts of the output based on the same input. Thus, it fits the description of communicational cohesion. The second method may likely have tasks that are sequential such as computing object's trajectory first and then using the result to compute the object's potential location. Thus, the method matches the description of sequential cohesion.

```
AuditResult transcriptAudit(StudentInfo info)
float computeTargetLocation(RadarInfo info)
```

Similarly, the following methods are likely of high cohesion, and their implementations are predictable:

```
double nthRoot(double value, int n);
Set intersectionOf(Set s1, Set s2);
int[] Sort(int[] array);
void displayTimeOfArrival (String flightNumber);
float computeGrossPay (float hoursWorked, float payRate).
```

Logical cohesion is common, and sometimes, it might be the only viable choice. By design, a method may be of logical cohesion if it does multiple things alternatively depending on a control parameter. These conditional blocks may or may not be related, and their presence in the same method could be because of a logical grouping, a technical consideration, or simply an ad hoc act of convenience. However, when blocks are related by a process, such a method may be reasonably cohesive. For example, a payment processing process may pass an error code to a method for further actions depending on the result of payment verification. A user could be asked to provide more information, use a different payment method, or save their purchases to check out at a different time. In the following method, the method's profile may suggest a low cohesion because of a less descriptive name and a cryptic argument. But the detail shows that the branches are based on the same user input and produces data used by the last statement. Thus, the method seems to fit the description of sequential cohesion. These examples suggest that the cohesion characterization presented earlier may not always be accurate. When a profile suggests a possibility of low cohesion, but an evaluation speaks otherwise, we may try to improve the profile.

```
void processInput(String input){  
    String requestedInfo = "";  
    if( input.equals(expected) )  
        requestedInfo = retrieveJSONStr();  
    else  
        requestedInfo = retrieveErrorMessage();  
    postResponse(requestedInfo);  
}
```

Class constructors are a typical example of temporal cohesion. The role of a constructor is primarily to initialize instance variables. But a constructor can do a variety of other things as well like setting up a graphic frame, making a data source connection, or placing some default data items in a data structure. A constructor sets an initial state when an object is created. Temporal cohesion is appropriate for describing the cohesion level of a typical constructor where things may not be related to each other but need to be done at that “moment.” A generic control process often needs a method to initialize the operational environment, which therefore does a variety of things to ensure a meaningful start of an application. The details of such a method can be trivial, unimportant, ad hoc, and messy. Hiding such details in a single method may improve overall code readability even though cohesion of the method may likely be low (but independent). Again, design is a tradeoff.

4.4 Method Coupling

Functional decomposition inevitably leads to module coupling as details are extracted into modules at different levels of a design. Module coupling generally improves code reuse, code clarity, and hence code readability. Module coupling also supports appropriate code separation to allow independent code development and update. Coupling is also a way to tame code complexity by delegating complex tasks for other modules to handle. However, too much coupling, especially harmful coupling, can make the code more rigid, easy to break, and more difficult to change. Ad hoc coupling may fragment the code with little long-term benefit. Therefore, module coupling should always be appropriately devised and assessed.

4.4.1 The Phenomenon of Coupling

When tasks in a method are delegated to other methods, the method is coupled with other methods. To illustrate this process, consider the following code to compute the (unbiased) standard deviation:

$$S_x = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}},$$

where \bar{x} is the mean of the dataset $\{x_i\}_{i=1}^n$:

```
static double std(double[] data){
    double sum = 0.0;
    int n = data.length;
    for(int i = 0; i < n; i++){
        sum += data[i];
    }
    double mean = sum / n;
    sum = 0.0;
    for(int i = 0; i < n; i++){
        sum += (data[i] - mean) * (data[i] - mean);
    }
    return Math.sqrt(sum / (n-1));
}
```

If we introduce another method for computing the mean as follows:

```
static double mean(double[] data){
    double sum = 0.0;
    int n = data.length;
    for(int i = 0; i < n; i++){
        sum += data[i];
    }
    return sum / n;
}
```

then the code can be simplified as follows:

```
static double std(double[] data){
    double mean = mean(data);
    double sum = 0.0;
    int n = data.length;
    for(int i = 0; i < n; i++){
        sum += (data[i] - mean) * (data[i] - mean);
    }
    return Math.sqrt(sum / (n-1));
}
```

Thus, the computation of *std* is simplified by coupling with a method that computes the mean. The method for computing mean is completely independent and can be used to compute a mean or a sum of a sequence in many other applications. Without reading the code for *mean*, the formulation of the standard deviation is still clearly identifiable in the method *std*. Therefore, the coupling is loose and good. If we want to implement the method to compute the mean in a different way such as the following functional approach, the client code is not affected. In other words, in situations of loose coupling, we can treat a method like a “black box”; we only care about what it does, not how it does the task.

```
static double mean(double[ ] data){
    return reduce(lambda x, y: x + y, data) / data.length;
}
```

Had we extracted the second loop of the method *std* into another method, say, *sumSquares*, which might be useful in certain statistical computations, the computation in the method *std* would become fragmented, affecting the comprehension of the method.

Coupling may become tighter when we cannot treat a method as a “black box.” Consider the following method:

```
static double doOneThing(int flag, int default){
    int var = compute(default);
    if(flag == 1) doThis(var);
    else if(flag == 2) {
        var = compute(var);
        doThat(var);
    }
    else { var = computer(var); doAnother(var); }
}
```

The details of the three methods inside the conditionals may be unimportant, but it is important to realize that a caller must know what the three methods do to pass in a correct “flag” value. This is the coupling effect between a method and the enclosing operational environment of a method invocation. The implications of tight methods coupling include:

- Reusability of a tightly coupled method is low because we cannot treat the method as a “black box.”
- It can be difficult to specify preconditions without “leaking” the implementation details. But any linkage of a precondition to implementation details is a violation of the principle of specification abstraction.

Ideally, parameters of a method should be transparent and predictable as input. When they are not, a caller must know how exactly the parameters are used in the method to pass

correct values. In other words, the enclosing code, in which the method is called, is not completely independent of how the method is implemented, resulting in tighter coupling. Consider the following two method calls for drawing a circle (suppose the two methods have almost identical implementation):

```
drawCircle( radius, true );  
drawCircle( radius, enum.DRAW_SOLID_CIRCLE );
```

The first call is slightly more coupled with the enclosing code than the second because of the uncertainty of what “*true*” might imply, whereas the second call is completely transparent with explicitly defined parametric constant.

4.4.2 Effects of Coupling

Tightly coupled modules tend to cause one or more of the following effects:

- A change in one module may result in a ripple effect of changes in other modules.
- System integration may become more complex due to strong or tight inter-module dependencies.
- Reuse of modules may become harder because of a tight dependency on other modules.
- Unit testing becomes more difficult due to complex testing environment setups to address module dependencies.

It’s easy to see the effect of bad module coupling when an algorithm is fragmented into ad hoc pieces, buried in separate methods, resulting in code rigidity. As mentioned earlier, a design of parametric abstraction also has implications to module coupling. To make a module more independent, reusable, reliable, sustainable, and clearly understood, design of parameters is typically based on these considerations:

- Provide adequate input information an implementation needs.
- Facilitate module reuse.
- Keep preconditions simple and easy to verify.
- Use intention-revealing names.

Code extraction from larger modules is a common way to introduce new methods. As a result, ad hoc parameters can be (easily) introduced, especially when a process is so fragmented that it becomes incomprehensible without reading the details of its support modules. Fortunately, such situations can be felt, and ad hoc parameters can be easily spotted—indications of a modularity redesign.

Making more information available to a method than it needs (through parameters) may cause tighter coupling too. For example, in the following code, the method has a parameter of a two-dimensional array yet uses only one row in the implementation.

```
double[ ] determineFinalGrades(double[ ][ ] gradesMatrix, int rowIndex){
    double[ ] courseGrades = // extract the required row with rowIndex
    //do some processing on courseGrades
    return courseGrades;
}
```

Clearly, a caller would need to ensure the correct index is passed, thus cannot treat method like a “black box.” Besides, passing more information than necessary is a hazard in the sense that it provides additional possibility for accidental data modification and complicates preconditions at the same time. The correction, however, is straightforward by extracting the appropriate row before calling the method to make the intent of the method more transparent. Passing more information than needed is often for the sake of convenience but can easily create parameter vulnerabilities.

Instance methods, by design, are often modifiers to instance data, but expectedly and predictably. Nonetheless, they use “side effect” to modify shared instance data. This sharing may create unwanted method coupling, however, if we use an instance variable not intended to be part of an object’s state. Consider the following code with an instance variable *currentPosition*. This variable may be assigned a value after *search* or *findMax* operation to provide some convenience when other methods may need the value for their own operations. The issue is that a call to *search* may invalidate the value produced in *findMax*. The coupling effect is that unless we use *currentIndex* immediately following a call to *findMax* or *search*, we are unsure about the validity of the value. Methods that use the value are coupled with methods that produce it, and their interactions must be carefully coordinated. Modification to these methods may potentially invalidate the coordination—this is an example of code vulnerability.

```
Class List{
    int[ ] list;
    int currentPosition;
    boolean search(int item){
        .... if(found) currentPosition = i; ....}
    void findMax(){
        .... currentPosition = maxIndex; }
}
```

As mentioned earlier, local variables should generally not be “elevated” to become instance variables to complicate the class invariant. The above example shows that doing so also makes instance methods unnecessarily coupled that may lead to code vulnerability.

Correction is straightforward, however; both methods should return an index value (or an invalid index value as appropriate) and use no “side effect.”

Similarly, sharing variables in a larger scope (such as using “public” instance variables or global objects retrievable through static methods) makes all modules that modify the variables coupled. Invocations of the modules that use shared data must be carefully coordinated to ensure the validity of the data. Incorrect ordering of module invocations may invalidate the shared values, potentially causing software to function incorrectly. For example, the function below uses an external variable *externalVar*, which, when the method is called in succession, makes this seemingly trivial comparison false: *funWithSideEffect(3) == funWithSideEffect(3)*. This example is deliberate, but it shows the fact that coupling through external variables can make program verification difficult.

```
static int funWithSideEffect(int param){  
    return param + externalVar++;  
}
```

Older programming languages may support static and dynamic scoping for procedures, so nested modules could be coupled through variables declared in enclosing scopes, with static variables visible to all code in static scopes. Such scoping rules might facilitate the undisciplined use of externally defined variables—a common source for software bugs and faults. To avoid unnecessary coupling through variable sharing, we should:

- Treat value-returning methods as traditional functions to void using side effects.
- Limit data modification by instance void methods to instance data only.
- Assess the coupling effect of each data sharing and document the situation to raise the awareness.

4.4.3 Categorization of Coupling

In the literature (Stevens et al., 1974), module coupling is categorized with five levels of coupling tightness as follows (listed in a decreased level of coupling):

- *Content coupling* occurs when one module uses the code of another module when the content cannot be independently separated. Code statements or blocks can be labeled in some programming languages, and one can use traditional “GOTO” construct to create jumps between distant statements, or jump in or out of code blocks, causing code contents to couple. If we separate a coherent piece of code into modules with no clear intent how each module would be used independently, content coupling may also occur.
- *Common coupling* occurs when several modules have access to the same data in a larger scope. As discussed above, this level of coupling can lead to uncontrolled error propagation and make debugging very difficult.

- *Control coupling* occurs when a module's code logic is controlled by controlling parameters from a caller as discussed in detail earlier. Control coupling and logical cohesion are connected to each other.
- *Stamp coupling* occurs when modules share a composite data structure and use only parts of it, possibly different parts. A commonly seen situation of stamp coupling is to pass a composite data structure to different methods that modify different parts of the data structure.
- *Data coupling*: Data coupling occurs when modules share data through parameters. Each data parameter is used in its entirety. There are no other forms of information sharing between the modules. The computation of standard deviation above using a method to compute the mean is a good example.

This categorization of coupling was suggested in an era when structured design was the primary design paradigm. When factored in object interaction, there are other ways modules can be coupled. As discussed earlier, inappropriate instance variables may cause unwanted coupling. Inheritance hierarchies introduce coupling too between a child class and its parents. This coupling can be tight when the hierarchy is deep. We will continue the discussion in the next chapter.

Cohesion and coupling are two different aspects of module design. The two concepts are intimately related (such as the connection between control coupling and logical cohesion). The more independent and cohesive a module is, the less likely the module is tightly coupled with other modules. A sustainable module is necessarily cohesive and loosely coupled. Highly cohesive modules are more likely to offer better opportunities for designing loosely coupled collaborators.

But design, in many ways, is about tradeoff. When appropriate, we may design modules that might be less cohesive or more tightly coupled, with a full intent to minimize the impact, to trade for better software operational conditions or meeting stringent software constraints. Module coupling affects system sustainability and extendibility; thus, minimizing module coupling is always a high priority of a design. Maintaining a highly coupled system is difficult. However, evolving such a system is even more difficult and costly because decoupling the system first may be the only way to ensure the software's long-term success.

4.5 Module Redesign and Code Refactoring

Code complexity is not always unavoidable. Sometimes, it might take redesign of an algorithm to simplify the code. Other times, redesign of modules may also reduce the code complexity by improving module cohesion or loosening module coupling.

Nested loops are not desirable and often a source for code complexity. But they are often caused by inappropriate data encapsulation. Consider the following code with a nested loop:

```
for(Order : OrderList)
    for(item : order.getOrderItemList()) totalSales += item.getUnitPrice();
```

This code violates the data encapsulation principle of a data type by exporting a list of order items. Thus, the inner loop can (and should) be avoided by providing more service methods in the object type *Order*. To do so, a method *getTotalValue* should be introduced in *Order* to absorb the inner loop above. The original code now becomes the following with much improved readability:

```
for(Order : OrderList) totalSales += order.getTotalValue();
```

Sometimes, we can also reduce code complexity by creating new object types. In the following code, the nested loop computes grade point average for each student with grades from a two-dimensional array where each row represents grades of the courses a student has completed:

```
for( int i = 0; i < grades.length; i++ ){
    for( int j = 0; i < grades[i].length; i++ ) gpa[i] += grades[i, j].getGradePoint();
    gpa[i] = gpa[i] / grades[i].length;
}
```

If we define a new object type, say, *StuAcademicRecord*, to encapsulate a list of course grades and offer a service method *getGPA*, we can simplify the above code with much improved readability:

```
for( int i = 0; i < students.length; i++ ){
    gpa[i] = stuRecords[i].getAveGPA();
}
```

Module redesign involves tradeoffs too. Extracting code into new classes introduces additional code layers to the program structure and new code coupling (with potentially less efficient code). In this example, code complexity with poor readability is traded for some structural complexity with much improved module comprehension and (arguably) reduced code complexity.

Module redesign can be costly too. A monolithic program may be structurally simple, but evolutionarily difficult to maintain often with modules sharing common data and using parameters that are often data structures. To address the problem, there must be a good

assessment about the extent of a redesign effort, as decisions are consequential. Changes based on immature design decisions can be costly.

Professionals use the term *code smell* to refer to a surface indication that may suggest a deeper problem in the code or the design. Use of public or global variables, nested loops, long list of parameters, data-only classes, and making extensive comments on a piece of code, among others, can all be code smells. Code refactoring is a mechanism for code improvement or for retrofitting the code to implement changes to the software requirements (Fowler, 2018). The two examples above are demonstration of how code refactoring works. There isn't a simple answer to when or how often we should do code (or design) refactoring. When code smells surface, an impact assessment is generally necessary before a refactoring process takes place.

Today, most integrated software development environments (IDEs) have built-in support for code refactoring to facilitate commonly needed refactoring activities. These activities are meant mostly to improve the code design, including renaming, introducing variables, replacing a variable reference with a method call, extracting code into a new method, making public properties and methods private, and pulling methods into a supertype, among others. Software-assisted code refactoring can be very helpful when changes are pervasive.

Design paradigm may also have ramifications on module redesign or code refactoring. Structured design decomposes processes into smaller ones with functional modules to reduce complexity and increase reusability. It addresses the behavioral aspect of a software system separately from its data. Because of the separation of data from functional modules, module redesign or code refactoring can be less consequential. Adding or removing modules likely affects the design and code only locally if essential decompositions remain unaltered. For object-oriented design, however, adding or removing object types may require regrouping of data and methods that are currently in different layers of the design, resulting in possible changes to the program's structure and inheritance hierarchies. Such changes must be done manually. Therefore, sound initial design decisions based on solid analysis and modeling are fundamentally important to make module redesign and code refactoring meaningful.

4.6 Method Specification

A module's specification is the basis for understanding the module's functionality and correct construction of the module. Module specifications can also be used to reason about system decomposition and functional modularity. Software requirements derive module specifications, which, in turn, validate the requirements in a requirement engineering process. A specification is also a technical contract, based on which we engage with software estimation, project planning, and business negotiations.

4.6.1 The Nature of a Module Specification

Given all the stakes, a module specification must be not only correct but also precise. Precision of a specification means the following (Parnas, 1972b):

1. The specification must provide to users all the information that they will need to use the method correctly and nothing more.
2. The specification must provide to implementers all the information about the intended use to complete an implementation and nothing more (other than a possible contextual impact when a call is made).
3. The specification must be sufficiently formal to avoid any potential inconsistency, incompleteness (of all possible uses), and potential loss of any desirable properties of a specification (such as clarity, conciseness, specificity, etc.).

By extension, a precise specification of a method should invalidate any potential behavioral variations of the method. Preciseness means not only “nothing more” but also “nothing less.” For example, if the statement $x > 0$ is precise, then the statement $x > 2$ would be a stronger statement (if it holds, so is the original statement) permitting less values of x , whereas $x > -2$ would be a weaker statement (implied by the original statement) allowing more values. A stronger or weaker specification can be understood in a similar way. A stronger specification would provide some safeguard for users (including client code) while making implementation harder. In contrast, it can be easier to implement a method with a weaker specification, but risk potentially client’s applications if they were built according to the original specification. A deviation from a precise specification might do no harm under normal circumstances, but cause software difficulties when the system is running under boundary conditions. In the real world however, modules are implemented not based on how a specification is written, but how we understand what is written.

Often, a specification may appear adequate, but implicitly make assumption on what a reader may have known. For example, this statement may sound adequate: The method searches the instance member array for a match of the given argument and returns true if match is found or false otherwise. It nevertheless assumes that a reader would know what that “array” is, what “match” means, and how to deal with the situation when array is “null.” Omission and incompleteness may easily evade the originators of a specification who may take what people might know for granted. There is only one way to understand a specification—the expected understanding. But a specification with a natural language often bears an unmitigated risk that there are other ways to understand or interpret the specification. A specification narrative using a natural language with a free style is often full of imprecision even for describing some seemingly straightforward operations. For example, the following skeletal method is to sort an integer array and return the sorted array:

```
static int[ ] sort(int[ ] A){ ... }
```

Even though sorting is a common operation that a casual description above might be adequate, there are much more to specify:

- What is the order of sorting (as we cannot assume “ascending” is always the order)?
- Are the argument array and returned array necessarily the same array?
- Can the argument array be empty or a null reference?
- Can the argument array be altered in any way (e.g., repeated items are removed)?
- Is the argument array necessarily fully filled?
- Does the operation have side effect?
- Would efficiency of sorting matter?

A method specification involves pre- and postconditions. Preconditions of a method are assertions (i.e., statements that are either true or false) generally about:

- Operational constraints that must hold true at the time the method is called
- Conditions for data validity, where data includes the arguments and external values the method may use

Preconditions are what a proper invocation of the method requires of a caller. If preconditions are not verified, the method, upon execution, cannot be assumed to behave in any definitive way. Neither can we assume that a method would verify its preconditions, though an implementer may do so if technically convenient. To be certain, it is always the responsibility of a caller to verify the validity of the preconditions. In comparison, postconditions generally contain:

- Impact on the operational environment when a method call is completed
- Descriptions about the data or information that has been altered or created when method execution ends

Postconditions may specify what the method returns, possible error conditions, external variables the method modifies (when side effects are used), and the effects of these modifications.

JavaDoc (a document generator tool of Java for generating standard documentation in HTML format) uses directives particularly designed for elements of pre- and postconditions such as @requires, @returns, @effects, etc. These directives precede the header of a method.

4.6.2 Method Specification with Some Formalism

It is a challenge to use a natural language to describe pre- and postconditions with preciseness. An area of software engineering called formal (specification) methods is a study about mathematically based techniques used for software specifications with accuracy and preciseness. These formal methods offer logical and relational constructs to construct specification statements. They can be used for specifying modules and systems, including object-oriented systems. One of the methods is Z (pronounced as Zed to mean an “ultimate” specification language). Z (Jacky, 1997) has been popular since its inception in the 1980s. But because of the learning curve of the language, the use may have been limited to large organizations with mission-critical software systems.

Some “minimal” formalism of software specification might add value without being too technical, however. For example, if we view an array as a discrete function with an integer domain $\{0, 1, 2, \dots, \text{array.length}-1\}$ and the range consisting of array elements, then the sorting method above can be more formally specified in Table 4.1.

Software deals with data that may be encapsulated in objects, embedded in data structures, or accessible through input or output. As a result of software operations, data are created, read, modified, or removed. Often, we can inject certain formalism into a specification if we can define a more formal way to describe data. Set theory is minimally mathematical and, in fact, quite “intuitive” (with many online learning materials). It is adequate for more formal software specifications to use sets to describe data and hence pre- and postconditions. To describe a relation between datasets, we can use a discrete function

Table 4.1 Specification of method sort with some formalism

<i>static int[] sort(int[] A)</i>		
Assumptions: For easy reference, let returned array be B.		
Precondition	Predicate	Note
Parameters	$A \neq \text{null} \&\& \text{card}(\text{dom } A) = A.\text{length}$	“dom” for domain, which is a discrete set of numbers “card” for cardinality, or size of a set
Environmental Constraints	None	Method does not have any environmental constraints.
Postconditions	Predicate	Note
Returns: array B	$\text{dom } A = \text{dom } B \&\& \text{ran } B = \text{ran } A \&\& \text{any } i, j \text{ in dom } A, \text{ if } i > j, \text{ then } B(i) \geq B(j) \&\& \text{if } A(i) = B(j) = v, \text{ then } A^{-1}(v) = B^{-1}(v)$	“ran” for range, X^{-1} denotes preimage of function X, thus, $X^{-1}(v) = \{i : X(i) = v\}$ is pre-image set of v. The predicate ensures B contains the same values of A including multiplicities of the values, and values of B are sorted in an ascending order.
Throws (exceptions)	None	There aren’t known error conditions in a sorting algorithm implementation.
Modifies	Nothing	The method has no side effects.
Effects	True	Predicate value “true” means method “can have any effect” on what can be modified. In general, assertions that appear in “effects” constrain the changes of what can be modified; more assertions are imposed, more limits changes can be made.

(a discrete function has a domain of a discrete set). A useful extension of a discrete function is the concept of relation, i.e., a collection of ordered pairs between two sets. A discrete function is a relation, whereas a relation may not be a function. For example, consider a set of all courses (with set *COURSES*) offered in a college and a set of all classes (with set *CLASSES*) offered in a semester. The set $\{(course, class) \mid course \text{ is in } COURSES, class \text{ is in } CLASSES\}$ is a relation, but not a function because one course has multiple classes. Intuitively, a function is generally a many-to-one correspondence (with one-to-one correspondence as a special case), whereas a relation is generally a many-to-many correspondence. An array can be viewed as a function from a set of non-negative integers (i.e., indices) to a set of array elements. If needed, an array can also be viewed as a sequence (denoted, say, by $\langle a_1, a_2, \dots, a_m \rangle$) ordered by array index. We could then apply operations like *concatenate*, *append*, etc. commonly seen in languages that support sequence data structures. When helpful, we may also view an array as a list to apply operations commonly seen in a functional programming language. Dictionary or map data structures can be treated as functions too with domain and range consisting of keys and values, respectively.

As another example, consider a module *EnrollmentManager*, which includes operations like *addClassEnrollment*, *dropClass*, etc. We define a relation between a set of academic classes (of a semester) and a set of enrolled students. It is a relation because one class can be enrolled by many students and one student can enroll multiple classes. Table 4.2 is a specification on one of the operations.

A specification of an operation is independent of any implementation. The relation *class_enrollment* defined in Table 4.2 is only conceptual for the sake of specification and has no implication on how the operation or the module should be implemented. In fact, we could use a different way to describe enrollments with a function to map *courseSecId* to a list of identification numbers of the enrolled students. If we name this function *enrollment*, then the postcondition of the method *addClassEnrollment* on effects of execution becomes:

```
(card enrollment(courseSecId) < capacity(courseSecId) &&
      enrollment'(courseSecId) = append(enrollment(courseSecId), stuid) ) ||
(card class_enrollment(courseSecId) = capacity(courseSecId) &&
      class_enrollment' = class_enrollment)
```

Again, function *enrollment* was defined entirely independent of any implementation, though an implementer might be motivated by it with a hash map to represent this function in memory.

While complete formalism is not always necessary, it takes adequate formalism in a statement to avoid ambiguity. Consider the following method that verifies whether argument *part* is a sub-list of *src*.

```
static <E> boolean isSub(List<E> src, List<E> part)
```

Table 4.2 Specification of method *addClassEnrollment*

Operation: <i>addClassEnrollment</i>		
Assumptions:		
$STUDENT_ID$: a set of student identification numbers used by school $CLASS$: a set of identification numbers of the classes offered in an academic term $Relation\ class_enrollment: STUDENT_ID \leftrightarrow CLASS$ (ordered pair of ($student_id$, $class_id$)) $Function: capacity: CLASS \rightarrow Integer$ (a mapping from a class identification to class capacity)		
Precondition	Predicate/assertion	Note
Parameters	$stId$ is in $STUDENT_ID$ $courseSecId$ is a class section ID number	
Environmental Constraints	Appropriate data stores the operation interacts with are initialized.	
Postconditions	Predicate/assertion	Note
Returns	None	
Throws (exceptions)	None	There aren't known error conditions in this operation.
Modifies	$class_enrollment$	
Effects	$(card(class_enrollment^1(courseSecId) < capacity(courseSecId) \&\& class_enrollment' = class_enrollment \cup \{(stId, courseSecId)\}) \mid\mid (card(class_enrollment^1(courseSecId) = capacity(courseSecId) \&\& class_enrollment' = class_enrollment))$	$enrollment^1(courseSecId)$ is a (pre-image) set of student id's of currently enrolled students. Relation $class_enrollment$ with a prime denotes the post state of the relation after method execution. Symbol \cup represents union operation between two sets.

To explain what “sub-list” means, we might use “portion,” “segment,” “partial match,” etc. But none would eliminate ambiguity entirely without further clarification. If, instead, we assume $part = [p_1, p_2, \dots, p_k]$, $H = [h_1, h_2, \dots, h_i]$, and $T = [t_1, t_2, \dots, t_j]$, where H or T or both can be empty, then part of the postcondition may be:

$(src = H + part + T \&\& return true) // (no such H or T exists \&\& return false)$

Academics were hopeful that the development of a formal specification would provide insights and better understanding of the software requirements and design, facilitate software prototyping and correctness verification, and make it easier to build software tools to assist software development and testing. These hopes might have largely faded away primarily because the payoff by adopting formal methods was not immediately obvious, but the impact of formal specification has continued. Some constructs of the formal method Z were used in the examples discussed earlier. Learning a few such formal constructs to describe data involved in pre- and postconditions can help improve our ability

to specify modules and operations with the preciseness that we may not be able to achieve with a natural language.

4.7 A Case Study: Overriding “Equals”

This case study is about the complexity of a method’s specification. The effect of an instance method affects an object’s behavior, which may become interestingly delicate when a method is overridden. Method *equals*, defined in Java’s root class *Object*, is such a method:

```
public boolean equals(Object obj)
```

The method asserts whether “this” object “equals” the argument object based on the implemented criterion. Pre- and postconditions of the method appear not complicated, but their hidden complexity is often overlooked. Suppose the method is overridden in class *A*. Practically, it might make sense to implement only the following version, as we might want to compare only objects of the same type.

```
public boolean equals(A obj)
```

If the latter version were to replace the original version of *equals*, it would be a violation of the *Liskov Substitution Principle* as preconditions can only be weakened in an overridden method. However, compilers would typically treat the latter as an overloaded method coexisting with the original method. When an actual argument is not of type *A*, the original version in *Object* would be used—a reliability concern if the originally version is not appropriately overridden.

As a method in Java’s root class, *equals* method has hardly any preconditions. The argument can be any object with an implementation to be meaningful to any implementer. For appropriate postconditions, the method implements a mathematical equivalence relation, which means:

- The relation is reflexive: *x.equals(x)* is true for any *x* not null.
- The relation is symmetric: If *x.equals(y)* is true, so is *y.equals(x)*.
- The relation is transitive: If *x.equals(y)* is true and *y.equals(z)* is also true, so is *x.equals(z)*.

Object class provides only the most discriminating possible equivalence relation on objects, i.e., *x.equals(y)* is true if and only if *x* and *y* are aliases. Here are other conditions we must consider also:

- For any non-null reference value x , $x.equals(null)$ is false.
- Multiple invocations of $x.equals(y)$ would consistently be true or consistently be false, if x and y do not alter the information used for comparison.
- Considering potential multiple overrides of *equals* in an inheritance hierarchy, all overrides of the method must be behaviorally compatible governed by the *Liskov Substitution Principle*.

The implementation of *equals* can affect other overridden methods from *Object*. For example, the hash code of an object determines the location of the object in a hash map; thus, it is reasonable to expect that if $x.equals(y)$ is true, then $x.hashCode() == y.hashCode()$ (though reverse is not necessarily true).

For the sake of discussion, let class T be the following with *equals* overridden according to the discussion above:

```
class T {
    int value;
    T(int v){ value = v; }
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if( super.equals(obj) ) return true;
        if( !(obj instanceof(T) ) return false;
        T other = (T) obj;
        return this.value == other.value;
    }
}
```

With slight modification, the overridden method in a subclass S of T can be implemented as follows:

```
class S extends T {
    int newValue;
    S(int v){ newValue = v; }
    public boolean equals(Object obj) {
        if( obj == null) return false;
        if( this == obj ) return true;
        if( !super.equals(obj) ) return false;
        if( !(obj instanceof S) ) return false;
        S other = (S) obj;
        return this.newValue == other.newValue;
    }
}
```

In these overrides, aliases must be explicitly checked. If objects fail to equal under *super.equals* (note that *super* now represents an instance of T), they fail too with an object of type S . When s is an object of S and t is an instance of T , comparison becomes uncertain.

Since s is also an instance of T , $t.equals(s)$ appears valid with no behavioral surprises. However, the comparison $s.equals(t)$, which is based on the overridden version in S , would return false as t is not an instance of S . Therefore, the symmetry rule is violated. We could change the override code in class S to allow obj to be an instance of T so that the symmetry rule would be restored. However, transitivity rule would still be violated. To see how, consider two instances of S , say s_1 and s_2 with equal numbers for *value* but unequal numbers for *newValue*, and t being an instance of T . Then, $s_1.equals(t)$ and $t.equals(s_2)$ would both be true, but $s_1.equals(s_2)$ would be false. Thus, there appears to be no perfect solution to override *equals* in each subclass of a hierarchy if comparison involves objects of supertypes. Commonly used resolutions include the following:

- Provide some safeguard in the caller code to allow comparison only between objects of the same type:

```
if( s1 instanceof S && s2 instanceof S ) boolean result = s1.equals(s2);
```

- Override *equals* only once with no further overrides in subclasses if that makes practical sense.
- Override *equals* as above in every subclass and always make sure that a subtype instance invokes *equals*, so the correct version would be used if violations of equivalent relation are not consequential.
- Make a superclass an abstract class to prevent objects from being created; thus, the super-class version of *equals* can only be invoked using *super* reference in a subclass.

For some of these workarounds, we would be unable to treat the *equals* method as a “black box,” which may have an unintended implication on coupling between the calling code and *equals* method. The reliability of an overridden *equals* can be much improved if we ensure that an abstract class provides the first override of *equals* and no subclasses of a concrete class are created.

4.8 Summary

A method represents a domain or a service action. A method is also a procedural abstraction, with its functionality specified independent of any potential implementation. We have discussed some fundamentally important attributes of a method in this chapter, including module cohesion and coupling—the two important attributes that can affect software reusability and sustainability in some essential ways. However, decisions on cohesion and coupling are also based on many other factors that must also be considered in achieving design goals.

The discussions we provided in this chapter apply to methods of all kinds, which do share some common characteristics. Standalone methods use data in public scope; thus, the

design often focuses on the unique roles they play in computation, their profiles, and their collaborators. In contrast, instance methods control how an object behaves. They are generally small in size, use fewer parameters, and operate on instance data. Because an object can collaborate with other objects, instance methods may have distinctive ways of coupling with other methods, which we will further explore in the next chapter.

A method specification is essential to the reliability of the operation. The more precisely we can describe its functionality and operational conditions, the more reliable an implementation will be. We provided a discussion about some possible ways we can specify a module more formally with minimal number of constructs of formalism.

Though much of the discussion in this chapter was within the context of method design, the design principles, practices, and techniques are applicable to the design of modules of any size. However, objects are a special kind of modules that are only present during runtime; thus, design of objects deserves separate exposure, which is the primary focus of the next chapter.

Exercises

1. We often use flowcharts to describe how algorithms work. For lack of better tools, we might also use a flowchart to convey how a system works often from a user's perspective. Parnas (1972a) argued that modules do not generally correspond to steps in the processing; thus, it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart.
 - (a) Choose one of the scenarios below to describe, with a flowchart, how the system might work, and then identify some software modules of the system as if you were asked to design such a system.
 - Withdrawing cash at an automated teller machine
 - Filling a gas container at a gas station
 - (b) Verify Parnas' argument above.
2. Parnas also proposed that one should begin with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others (Parnas, 1972a).
 - (a) Use this Parnas' criterion to redesign modules of the system you chose for Question 1.
 - (b) Are the two sets of modules similar? If not, which set of modules would be more stable going forward?
3. Parnas also argued that to achieve an efficient implementation, we must abandon the assumption that a module is one or more subroutines and instead allow subroutines and programs to be assembled collections of code from various modules (Parnas, 1972a). The argument made half century ago is still very relevant today. Can service-oriented software design and construction be considered as an application of this argument? Why or why not?

4. Given appropriate information about the polynomial, design and implement a (static) method that returns a string representing any given polynomial (of real coefficients) like $2x + 3.2x^3 + 1.2x^8$.
5. Design the header of a (static) method that sorts any portion of an array of objects of any type with any order defined by a user (no implementation is necessary).
6. Design the header of a (static) method that searches any portion of an array of objects of any type with a given key (no implementation is necessary).
7. For many languages, *main* method is the entry point of a program execution. When we use the same main method to test multiple implementations that are unrelated, it may become messy quickly and difficult to keep track of various test cases. Use modularity (without using a unit testing framework) to suggest a way to better organize the testing code in a main method for better readability, easier expansion of the scope of testing, and effective communication when tests are reviewed by others.
8. Write procedural abstractions for the following tasks:
 - (a) Credit card validation.
 - (b) Verify whether a cash withdrawal from a bank account exceeds the withdrawal limit.
 - (c) Grant access to a file with an access permission of certain form.
 - (d) Update a patient's record with a new medical prescription.
9. Check printing is a software process with an input of a numeric money amount and an output of a check required of the money amount printed in words. Use procedural abstraction to determine the modules and methods needed for the process.
10. Write a function for temperature conversion to be used for either converting a temperature in Celsius to one in Fahrenheit ($F = 9/5 \times C + 32$) or vice versa ($C = 5/9 \times (F - 32)$).
 - (a) Discuss pros and cons in terms of cohesion, usability, and extendibility (to include conversions to and from temperatures of another type).
 - (b) Propose another approach that may overcome the drawbacks of the above single function approach.
11. Given monthly sales data, consider a method to compute and print average monthly sales, average weekly rate of change in sales, maximum monthly sales, and minimum monthly sales.
 - (a) What would be the level of cohesion of the method, and why?
 - (b) What would be a strategy to return all computed results if only a single value can be returned?
12. To combat complexity, it was suggested (Ousterhout, 2018) that classes should be deep, meaning to do something significant, be a significant part of the computation, or offer some significant services. A deep class implies that some of its methods are deep. For example, linear regression is a linear model for the relationship between a dependent variable, like sales, and one or more independent variables, such as investment in product design, development, and promotion. A method to compute the coefficients of the linear model using the input data points is necessarily deep.

- (a) Reason that a deep method is necessarily of high cohesion to be viable.
 - (b) Discuss what “deep” might result in if the method is of low cohesion.
13. Describe separation of concerns in your own words.
 14. Functional decomposition, when overdone, may fragment algorithms and processes that can negatively affect software integration. Develop some strategies to avoid excessive decompositions.
 15. A computer program is said to be portable if there is very low effort required to make it run on different platforms. How may coupling (in the context of this chapter) affect software portability? Provide examples to support your answer.
 16. Discuss module testability in relation to module design. In other words, how might different aspects of module design affect testability of a module?
 17. Objects are modules, and they can share static variables. Develop some dos and don’ts in terms of objects sharing static variables effectively and safely.
 18. Refactoring with software assistance improves code organization in (mostly) non-essential ways. However, it is not unusual that certain regular refactoring activities (such as adding methods to or removing methods from classes) may trigger new ideas that may require substantial changes in code structures. Whether we should proceed with the changes likely depends on many factors such as the impact of the change, the extensiveness of the change that can affect delivery schedules, or simply whether the idea was adequately evaluated. In other words, “code refactoring” can be consequential and may need a process to manage the activities and decision-making.
 - (a) What might this management process entail?
 - (b) Develop some guidelines for making significant code changes and structural alterations.
 19. Method design can often be less strategic based on spontaneous thoughts, immediate benefits to a code vicinity, or even coding conveniences of certain ad hoc nature. In contrast, strategic design of a method is based on considerations on design impact, code readability, taming the complexity, and module reuse, among others. Think about, and then develop some guidelines for avoiding making less informed design decisions and becoming more strategic in method design.
 20. Suppose the following method copies the values of array *A* into array *B* in the same order, assuming array *B* has the same length as array *A*.

```
void copyArray(int[] A, int[] B)
```

Write a specification of the method in a more formal way with techniques demonstrated in this chapter.

21. Similar to Table 4.2, write a specification of the operation *dropClass* in the module *EnrollmentManager*. The method is to perform withdrawal of a student from an enrolled class.

22. Suppose a social networking site needs a module to manage users' posts categorized using hashtags. The module has the following operations (among others):

- *addPost(String post, String hashtag)*, which adds a post to a collection of posts represented by the given hashtag. If the hashtag doesn't exist, the method creates a new pair formed by the hashtag and a collection of posts with *post* being the only item in the collection.
- *retrievePosts(String hashtag)*, which returns the collection of posts with the given hashtag or nothing if the given hashtag doesn't exist.
- *changeHashtag(String newTag, String oldTag)*, which replaces an old hashtag (*oldTag*) with a new hashtag (*newTag*) with the collection of posts intact.

Write specifications of these three operations in a more formal way by defining an appropriate discrete function or relation between hashtags and a collection of posts.

23. Consider the following situation: a class *A* does not override *equals* method (other information about *A* is not relevant to the question), and class *B* extends *A* with an additional instance variable *x* of integer type and a constructor to initialize *x*. Class *B* overrides *equals* like it was typically done as described in this chapter (including the criterion that two objects of *B* are equal if their respective *x* values are equal). However, the following code would display (a contradictory) "false."

```
B b1 = new B(5);
B b2 = new B(5);
System.out.println(b1.equals(b2));
```

What might be the cause of the issue? How can the issue be resolved?

24. The *equals* method of Java's library data structure *List* behaves like this: Two *List* objects are equal if and only if they are either aliases or contain the same elements in the same order. Write a subclass of *ArrayList* (in Java's util library package) by overriding *equals* method in such a way that two lists are equal if and only if they are either aliases or have the same elements regardless of the orders of the elements. Examine the potential issues that this override might lead to.
25. It was mentioned in the last section that if we modified the code for overriding *equals* in subclass *S* to allow argument to be an instance of supertype *T*, the symmetry property of the *equals* operation would be satisfied. Reimplement *equals* in class *S* to verify that.

References

- Fowler, M. (2018). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley Professional.
- Jacky, J. (1997). *The way of Z: Practical programming with formal methods*. Cambridge University Press.

- Liskov, B., & Guttag, J. (2000). *Program development in Java: Abstraction, specification, and object-oriented design*. Addison-Wesley Professional.
- Ousterhout, J. (2018). *A philosophy of software design*. Yaknyam Press.
- Parnas, D. (1972a). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12).
- Parnas, D. (1972b). A technique for software module specification with examples. *Communications of the ACM*, 15(5).
- Stevens, W., Myers, G., & Constantine, L. (1974). Structured design. *IBM Systems Journal*, 13(2).



Design of Objects

5

5.1 Overview

Objects are instances of object types. Design of objects is, thus, about design of object types. To create object types, we use classes (abstract or concrete) or interfaces. Therefore, this chapter is about design of classes and interfaces or, more appropriately, about design of data abstractions. A data abstraction is a separation between specifications of the operations and their implementations. Thus, object design is primarily about design of the operations an object is responsible for and collaborating objects that support the operations. In other words, design of an object is to design its behavior and its structural relations with other objects.

We will start with a brief discussion about object design in relation to other activities in a development process and about its own design process. We will then discuss object design essentials. Based on the foundation of object design, we will take a closer look at design of objects of a particular kind—control objects that is often of architectural importance. We will extend the discussions to design guidelines in the areas of entity cohesion and coupling before we conclude the chapter.

5.2 The Context and Process

An analysis of software requirements leads to the creation of various potential software entities or elements. As design advances, we refine the entities and elements, their responsibilities, their collaborators, and their relations with external entities and elements. Figure 5.1 illustrates the design of objects in this context. Thus, design of objects is part of a software analysis process.

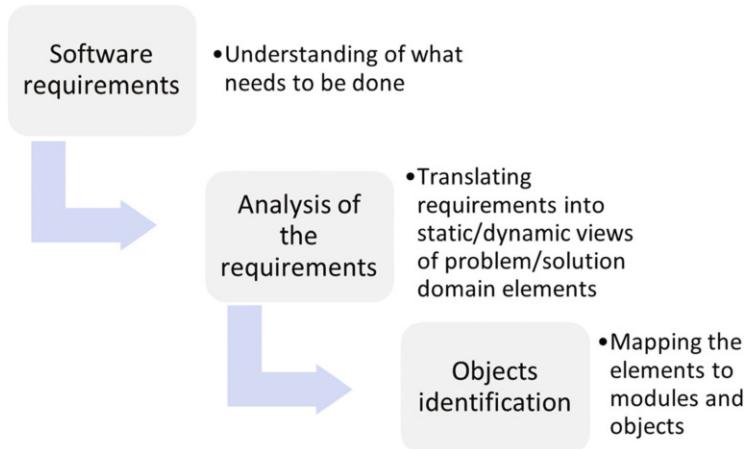


Fig. 5.1 Process of identifying software objects

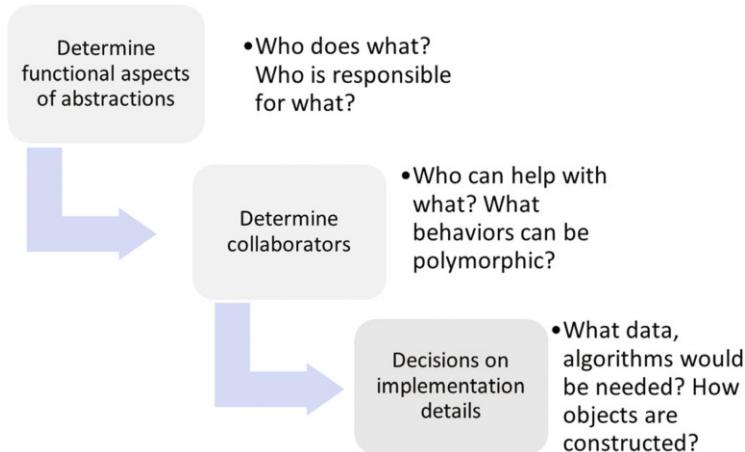


Fig. 5.2 Design of objects in the context software construction

Software analysis is a high-level design activity, leading to a structural organization of software entities and elements we call it software architecture. However, design of object types is generally not an isolated event. For example, to design a domain object type, we must consider not only its relations with other domain object types but also collaboration with structural objects within the architecture. Design of objects results in an object model of the system. Thus, object design also includes design of objects' representations and behavioral details. Figure 5.2 illustrates this design context.

An important aspect of object design is to design objects with polymorphic behaviors. Thus, part of object design is also about recognizing functionally distinctive yet

conceptually connected entities. This would require analysis on identified entities and elements to understand their conceptual commonality and behavioral differences.

Finally, design of objects, like most of other aspects of software development, is iterative. During a requirement engineering process, requirements validation and evolution may lead to changes in requirements, which necessitate object design iterations. These iterations, in turn, may trigger design refactoring loops to update the object model. In this iterative process, new object types can be introduced, and existing object types can be modified, combined, or removed. While this iterative process may continue as the software is delivered incrementally, the workflow of object design may gradually diminish as the object model stabilizes.

5.3 Essentials of Object Design

Objects are computing or processing software entities, and their interactions manifest computing and problem-solving. Objects may play different roles. Some are structural entities, some collectively model software domain behavior, and some are essentially data structures or data handlers. Domain elements are typically modeled with objects, compared to computational processes and services, which can be either static or dynamic entities depending on situations. Discovery of objects may start with behavioral and functional decompositions of a system. These decompositions with conceptualization are the basis for designing procedural and data abstractions. Conceptualization of entities may benefit the most design of behavioral polymorphism, but it can also impact design of data representation, collaborative task completion, and the tradeoffs.

5.3.1 Object to Model One Thing

One design principle that applies universally is the single responsibility principle. In the current context, an object type should model one thing and do it well. This design principle is consistent with the principle of interface segregation to promote conceptual separation and independence. Often, we also have various design concerns; thus, the principle of separation of concerns also applies to design of objects. To create an abstraction, it is likely that we identify a task first (as a result of functional decompositions) and then the name of a potential entity responsible for the task. We may name a single-task entity relatively easily such as *UserRequestDisplay* or *BookOrderDelivery*. If we find that the responsibilities are bit diverse, we might raise the level of abstraction so that a name might still be appropriate, such as *DataHandler* or *InfoProcessor*, to allow a wider range of interpretations. However, a name alone cannot determine whether responsibilities of an object are appropriate.

Functional decomposition is used to decompose a system into subsystems or a module into smaller modules. It is still a tool to use to design objects. Decompositions lead to software tasks, simple or complex. Each task may further be decomposed as necessary

before tasks are assigned to appropriate objects. To identify non-domain objects, we may use decompositions of information, a concept, or even a concern.

A “thing” can be of any nature. It can be (conventionally) a domain entity, a data structure, or a data connector. If something can potentially vary, we might design something more abstract to allow variations. If there is a design concern, say, about structural feasibility in the solution domain, we might design “structurers” to address the issues. A naïve strategy to identify initial set of objects is to identify objects by looking for nouns in software requirements and user stories. While we recognize the names in software requirements or user stories that may indicate potential objects, we perform appropriate analyses to discern between nouns and software needs based on decompositions. In fact, entities we design are mostly fabrications, purely based on analysis and modeling needs, and they often lack counterparts in the real world. Even for domain entities, we may design properties and responsibilities that we do not observe in their real-world counterparts.

5.3.2 Diverse Object Design Possibilities

A primary concern of object design is to design ways objects collaborate. An object uses or collaborates with other objects for operational support because a collaborator may:

- Hold the information to be needed
- Provide required services
- Maintain data connections
- Be responsible for delegated actions
- Act as a messenger
- Deliver structural support
- Provide software tooling
- Function as a data container

To facilitate brainstorming about potential object models, professionals suggested using a notecard, known as Class, Responsibility, and Collaboration (or CRC) card. A CRC card consists of three sections as shown in Table 5.1. The top section is the name of a potential

Table 5.1 Illustration of a Class, Responsibility, and Collaboration (CRC) card

(Candidate Type) Name	
Responsibility:	Collaborators:
Operation 1	Collaborator 1
Operation 2	Collaborator 2
...	...

object type. Operations the entity is responsible for are placed in the lower left section with possible collaborators in the right section. A CRC card may also include a description of the purpose for the object type and its role in the overall design. As modeling advances, development may require more elaborated modeling considerations where CRC cards may be less effective if the written notes on such cards (taken during modeling sessions) are not documented properly to reflect design updates to establish a new baseline.

The *problem domain* of a software system generally refers to (or defines) an environment where the solution will come to work, whereas the *solution domain* refers to an abstract environment where the solution is developed. For example, *course* is likely a problem-domain entity, whereas *mediator* is likely a solution-domain entity to play a role of object “mediation.” These two domains overlap partially, and it is perhaps harmless if we consider them as the same thing. But generally, objects in the problem domain are directly responsible for delivering the functionality of a system, whereas objects in the solution domain are often associated with the control mechanism of the system (determined by the architecture). However, objects we design are fabricated entities; they may deliver certain functionality while “structuring” other objects.

Consider a bank account, which may include owner’s name, account number, verification information, a balance, etc. There are different ways we can design objects based on these simple account attributes. For example, a programming textbook might suggest using a single object type, say, *BankAccount*, which also assumes responsibilities such as “deposit,” “withdraw,” etc. But we can argue that it’s appropriate for a bank account object to assume responsibilities, like “update account information,” “increase balance,” etc., but inappropriate to assume responsibilities of performing transactions. Thus, we might design another type, *Transaction*, to handle “deposit” and “withdraw,” among other possible account activities. *Transaction* is a problem-domain concept, but object type *Transaction* may “structure” *BankAccount* in a way appropriate for a given architecture. Yet, there is no way to prevent someone from designing a number of object types like *Account*, *Name*, *Validation*, *AccountData*, *Balance*, *AccountDebit*, *AccountCredit*, etc. to be more “object oriented.” As a result, object interaction to accomplish the same task would be very different. Consequently, a separate control object would be needed to coordinate object interactions to deliver required functionality. Therefore, design of objects in the solution domain may depend on the design of a domain model. CRC cards can be effective to “document” such a process in terms of what has been contemplated, discussed, and analyzed when we face a diverse set of design options and possibilities.

Despite many design possibilities, “ideal” objects would behave substantially and distinctively, work collaboratively, and function polymorphically. Objects we try to avoid are those that do not behave much beyond simple data updates. Such objects would encourage highly centralized controls that are often a characteristic of monolithic programs.

5.3.3 Prototyping Object Interaction

How an object behaves is perhaps best understood in ways it interacts with other objects. Once we identified objects and their responsibilities, we want to observe how they interact. This process is called casually “object modeling,” and certain diagrams are typically used. But often, we can model interaction directly in code to prototype the interaction. Prototyping object interactions in code allows instant feedback and “just-in-time” design refactoring.

Consider a task of making reservations at a hotel chain. The software requirements may suggest that making reservations would result in subtasks such as making reservations at local hotel branches, updating room availabilities, and adding customers to hotels’ customer notification lists to keep customers informed before their actual stays, etc. With initially identified objects (such as *HotelOpsController*, *HotelHeadquarter*, *HotelBranch*, *Reservation*, etc.) and initial assignments of the responsibilities, we can write some code to describe the interaction of the objects involved:

```
HotelBranch branch = hotelHQ.findBranch(reservationRequest.getDestination());
Reservation reservation = branch.makeReservation(reservationRequest);
if (reservation != null ){
    branch.updateRoomAvailability(reservation.reservationDetails());
    branch.addCustomers(reservation.getCustomer());
    branch.sendReservationConfirmation(reservation);
}
```

As we will see in the next chapter, we typically use UML sequence diagrams to show object interactions. But prototyping object interaction with code has the following advantages:

- It describes interactions more accurately.
- It allows more effective design refactoring.
- It promotes the practice of early testing.
- It facilitates incremental delivery efforts.
- It makes transition from design to code construction smoother.

However, we may need an architecture to guide prototyping interactions to make sure that prototypes would not potentially fragment the code.

5.3.4 Designing Objects Around a Structural Style

As will be discussed in detail later, we can use an architectural style as a guide to structure software elements. We can then design objects based on the chosen architectural style. A

widely applicable structural style known as *Model-View-Controller* (or MVC) is to structure software entities and elements around three themes—model, view, and control. This structural style was first introduced in the late 1970s and was even integrated into the object-oriented programming language Small Talk. Model is the “engine” of a software application, where data is processed. View manages the communication between the software and users or between data sources and information display. Control drives software internal operation and coordinates between the model and possibly multiple views. As an example, consider a simple address book application. The view would be a user interface for user input and information display. In an execution of a search operation, for example, the model would be a data structure managing an address book with editing operations and capability of using externally stored address data. A control would coordinate interactions among a user, a view, and the model. Objects of other kinds may still be needed such as objects handling data between the in-memory data store and an external data source (these are called boundary objects as they are part of a solution and communicate with problem-domain objects).

Because of its applicability across many application domains, the MVC style, or a variant, can often be used as a “default” style for the organization of software elements. Figure 5.3 illustrates how the style may be used in web applications. Today, popular web application development frameworks have typically integrated MVC into their component building mechanisms. In such a framework, communications between different aspects (model, view, and control) are built in. The framework provides templates for view

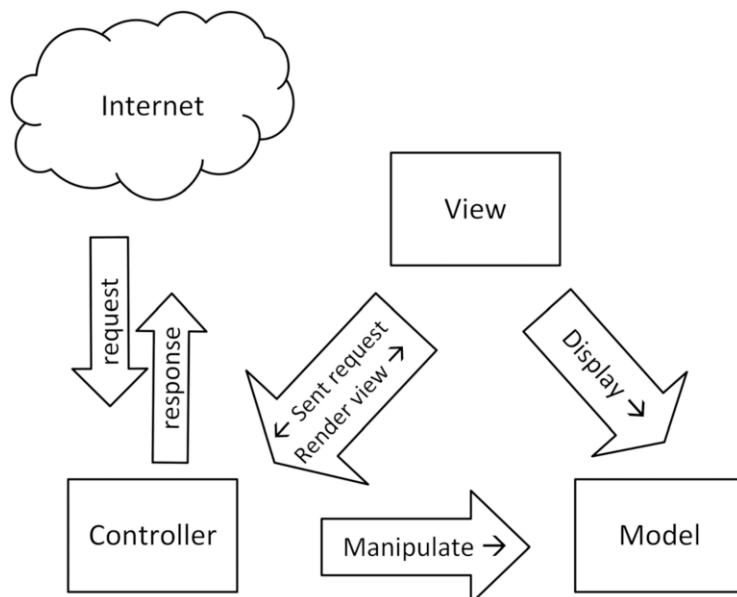


Fig. 5.3 Internet applications with a Model-View-Controller architecture

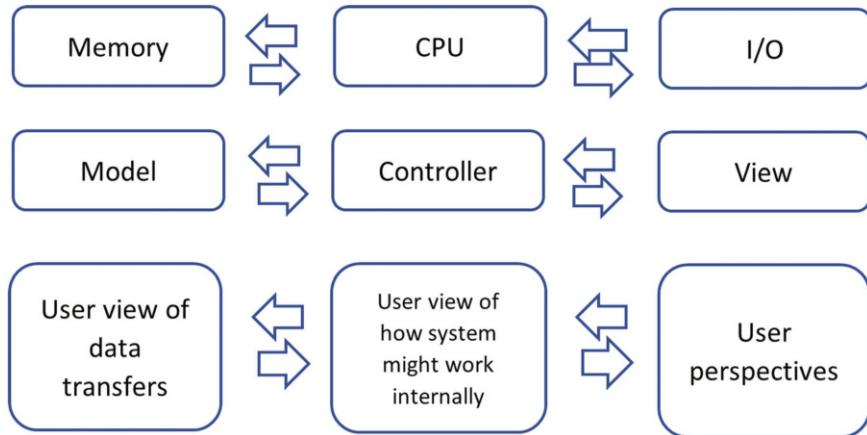


Fig. 5.4 Alignment of computer, software model, and user's mental model

components, coded in certain intermediate markup language, to be rendered by the framework. These templates also provide placeholders for customized code to be added. A model represents processes that transform data. A controller handles user (HTTP) requests from views and communicates with the model to deliver the responses the model produces based on user requests. With the support from such a framework, object design becomes much simplified with a focus on the model design.

Figure 5.4 illustrates the alignment of the MVC style with how a computer works and users' perspectives. Software systems are virtual machines; thus, the style appears to align well with the architecture of a computer. Software users have their (varied) perspectives about software operations they experience through interactions. Their viewpoints form a basis for their mental models about the software system as they perceive how the system might work and where data might come from. Object-oriented paradigm, as some may have argued, makes a design better in alignment with users' mental models when objects we design simulate physical objects that users observe. But the reality is quite different. Users may see data as represented in its observable raw forms, whereas computing "sees" data in terms of data transformations. A simulated object may often be given fabricated roles to transform data. It is an object design that creates a way how data is distributed (among objects) and transforms through object interaction. The underlying attractiveness of the MVC style is perhaps a simple conceptual interpretation of a system that makes data distribution into objects much easier and effective, regardless of how a user might view the system.

Design of a problem-domain object can be motivated by explicit entities we observe in software requirements. However, an object we design often has very different responsibilities even in cases of a "perfect match." For example, in a card game, a player draws a card from another player's hand, but that's equivalent to software operations that

the other player returns a random card and then the controller places it in the first player's hand. The following are a few more heuristics that we may use in the design of objects:

- A software action may require multiple collaborating objects; often that is because none of them seems to be appropriate as a host of all collaborating actions needed.
- A complex process may need data encapsulated in other objects; as a result, its process can be fragmented with distributed intelligence across multiple objects.
- Data might be the only stable aspect of a system; everything else can change; thus, design stability is always a concern.

5.3.5 More About Object Discovery

To take full advantages of object orientation without creating unnecessary object layers, an object needs to be rich in behavior or substantial in its intelligence to contribute to an overall computing process. Objects with simple or trivial behavior often only create unnecessary complexity and runtime overhead with little computational benefit. YAGNI (You Aren't Gonna Need It) is a mantra used to help developers avoid pitfalls in designing objects. The statement was originally used for discouraging introduction of features we presume the software would need in the future without much reliable information to support the decisions. It is intended to promote simplicity in software design. If we suspect whether a potential object might help, YAGNI probably will apply. Removing bad objects through code refactoring can be costly. Meanwhile, objects that have significant computing or processing power do not necessarily mean they are complex. Software complexity, as opposed to algorithmic complexity, typically comes from entity or data dependencies and obscurity in contrived or convoluted code due to improper object communications. Good object design with powerful abstractions can help avoid complexity in high operational layers and push complexity down to an appropriate level—ideally the bottom layer where objects of workhorse, data structures, and application service modules reside.

When we might need an object type is perhaps a question too broad to address effectively. But a few heuristic examples are possible. Consider postal codes. It can be adequate to represent postal codes with strings if we don't expect postal codes to be anything but data. However, if postal codes need to be linked to their geographic locations, counties, and states and searchable for general information about postal code areas, an object type modeling postal codes and related services can be helpful. Similarly, a string for an email address might be adequate in many applications. But, if we want an email address that can self-verify its validity, "report" the domain, check email aliases, etc., then an object type modeling an email address can also be useful. In applications involving single currency, data type "decimal" may be adequate. But if applications involve transactions with multiple currencies and money expressions that must be consistent with the conventions of the currency with various formatting options such as inclusion of commas, then an object type modeling money can be convenient.

Scientific applications often have verification checks of whether values used in computation are inside expected ranges. To be more general, we might consider a tool functionally captured by the following interface.

```
interface IDomain{  
    boolean isOnBoundary();  
    boolean isBoundaryIncluded();  
    boolean isInDomain();  
    boolean intersects(IDomain d);  
    float distanceFromBoundary();  
}
```

This abstraction can have a variety of implementation possibilities; among simplest are value or time intervals (discrete or continuous). It can also be used to model a domain of any defined shape or dimension.

In addition to understanding our computational needs, there are other things we can do to explore opportunities for the design of object types. Working with multiple lists of decompositions can be another effective strategy for identifying powerful object types. A functional decomposition may provide an initial list of operations; and subsequent requirements iterations may update this list as necessary. We keep a list of unassigned operations and assign them to existing or new object types when opportunities arise. A concept decomposition may produce a list of concepts to be associated with potential objects. System variability decompositions may result in another list of things to be abstracted with object types. There may also be emergent needs to warrant new object types (though such needs must be adequately assessed). For example, we might see a need for a lightweight object as a “messenger” to collect and then deliver information among objects that do not communicate directly.

Focusing on specificity early may also limit opportunities of creating more meaningful and intelligent object types. In an object design process, the more general a concept is that we focus on, the more responsibilities an object type, related to the concept, may assume, resulting in more substantial behavior. For example, “collection” is a more general concept than “list”; thus, a collection object generally assumes more responsibilities than a list object does. Generality applies to the design of operations too. The concept of “federal financial aid” could be too restrictive for students who are part-time. But if we used a more general concept of “financial assistance,” then either *CommunityStudent* or *ParttimeStudent* would be able to provide meaningful implementations to “applying for financial assistance” with polymorphic behaviors.

When facing multiple lists of “things” to “mix and match,” divergent thinking probably works the best. We run simulations of possible object interactions in our minds but expect them to fail (quickly) before more plausible ideas emerge. When modeling object’s behavior, we consider not only interactions with its collaborators but also the potential to be used as a collaborator by others. Mental modeling and evaluation can quickly identify

inappropriate alignment between concepts, responsibilities, and priorities to avoid pervasive and often costly consequences if responsibilities are missed or misplaced. Prototyping with code fragments, as shown earlier, can be used interweaving with a mental process as necessary to confirm or abandon the ideas. For example, an object modeling a student can be “shallow” without much behavior beyond simple state updates. But it can also be responsible for loading data student’s academic and financial aid data to design substantial behavior as appropriate. Or if a student can work for school in certain capacity, we might design a student object to assume certain responsibilities as an employee with polymorphic behavior. These are diverse design possibilities that mental simulations can be particularly effective.

Design tradeoffs are always expected in the design of objects and must be carefully assessed. Divergent thinking often provides a quick and effective assessment on the pros and cons of potential tradeoffs in terms of the impact and the feasibility.

5.3.6 Design to Ensure Objects’ Behavioral Correctness

Pre- and postconditions are to ensure that we correctly understand the operations of an object. The representation invariant (or class invariant) of an object is to ensure its behavioral correctness. Pre- and postconditions specify object’s behavior and thus should be part of the design of object’s responsibilities. We explicitly include pre- and postconditions in the design of instance methods for good reasons:

- Allow accurate evaluation of the appropriateness of the responsibilities associated with the concept being modeled.
- Allow appropriate identification of collaborators.
- Allow accurate representation of object’s properties and class invariant.
- Support early prototyping.
- Support validation of the software requirements.

When we create abstractions, we focus on their responsibilities, not their representations. As a result, we may not be able to specify class invariants accurately (without precise representations). But describing invariants in more abstract terms is still possible. For example, assume we want to create a problem-domain entity *Student*, and a student’s academic record and current financial aid status may be needed as instance data in designed operations. We may state the invariant abstractly to raise the awareness of needing the precise invariant later, such as “A student’s academic record must be consistent with a good academic standing defined by school’s academic policies” or “The financial aid a student received must be awarded in compliance with government policies and regulations at all levels.” When operations are mutators to alter an object’s state, we can also describe postconditions in more abstract terms that are consistent with the abstract descriptions about object’s states. It is not straightforward to frame class invariants and

pre/postconditions in abstract terms without placing unnecessary restrictions on representation details. But any reasonable effort in doing so would add value to the reliability of object design.

5.4 Design of Control Objects

A control of a software operation delivers the outcome of the operation by coordinating task completion in a defined process and by delegating tasks to other entities as necessary. A control strategy determines responsibilities of the subordinate objects and how data flows between software processes. Monolithic applications in the past worked well with relatively simple-structured, centralized controls using procedures and functions sharing common or global data. Such a control is often a “call-return” hierarchy of modules. Each procedure or function is either a “worker” to deliver a result or a “broker” to further delegate a task and assemble the results returned from task executors. In some extreme cases, a control may become completely centralized when the control delegates the work directly to “workers.”

Object orientation promotes distributed controls. Nonetheless, an object may still be a centralized control as illustrated in Fig. 5.5. A centralized control object may allocate tasks or sub-controls to other objects as appropriate but maintain the control as the overall decision-maker. Figure 5.5 also describes the opposite to centralized control—decentralized control. Delegated control is a particular kind of decentralized control and is common in the design of control objects. A delegated control differs from a centralized control in that the primary control object is more of a coordinator to delegate decision-making (or at least a part of it) to subordinate objects. The term “dispersed control” is used to describe a “highly” decentralized control. A dispersed control is much like taking a vacation using a travel agent. We would flight to where we wanted, visit where we desired,

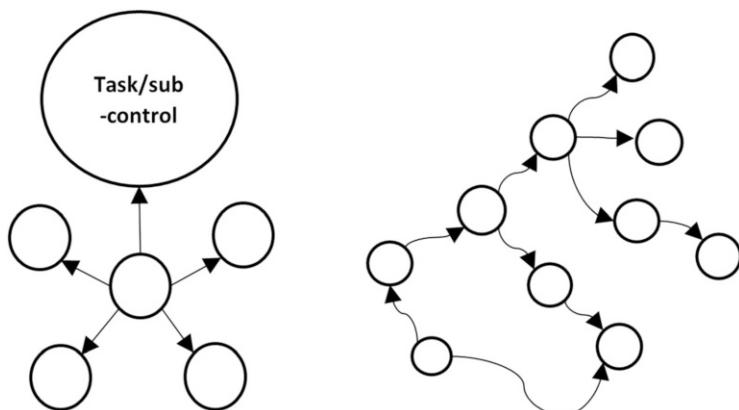


Fig. 5.5 Illustration of centralized and decentralized controls

and stay where we were told to, but we would have little idea who might have made all the travel decisions. Local small travel agents probably would make eventual decisions about where we stay, where to tour, and means of transportation. The travel agent that we would contact only knows their immediate travel contacts and have no knowledge either how eventual decisions are made that would satisfy our travel needs.

As we have discussed, design choices for a control object are much more diverse than they are with traditional approaches. Design of controls with objects is generally also of architectural significance and has ramifications to the design of other objects. For example, we might design a control using more moderately intelligent objects with relatively independent modifiability. When appropriate, we might instead use a more concentrated approach with fewer, but more intelligent, objects with decreased modifiability. Therefore, design of control objects is perhaps among the most consequential design decisions we make.

5.4.1 Highly Centralized vs. Coordinated Controls

Because of diverse possibilities of designing collaborated controls, it is possible that we may not completely understand the structural impact of a distributed control until much later when issues start to emerge. This challenge might prompt some to settle on control objects that mimic more traditional, highly centralized approaches with task delegation hierarchies. But the risk is that such a control can be susceptible to making the whole system monolithic (thus less structurally modifiable). Besides, lack of appropriate control abstractions may force high-level control logic to be connected to low-level details, making the control structurally rigid and brittle.

An analysis often identifies a control center at a high level, which may become a focus of design. A control center can be highly centralized if a task delegation hierarchy is shallow, but widespread with task executors. To design a control that is less centralized, we can use a control style with multiple task coordination layers. For example, a computer game may have a control center that implements a game loop using some abstract processes, but the control center is primarily a manager or coordinator to delegate appropriate decision-making to objects modeling players, environmental obstacles, and various other game elements to render their responsibilities based on the game rules. For example, the control center does not play for players (to move them around or make them jump), but players play for themselves. However, a player may further delegate play to a “device” it controls to move around or make jumps. Thus, the collaborating objects may be controllers too in their respective areas of concerns with their own coordination intelligence. In this fashion, a coordination-oriented control forms a tree of task coordinators with leaves being pure task executors. Each task coordinator is responsible for a significant portion of an overall task, thus achieving distributed control with an appropriate balance between the number of control layers and the flexibility of the coordinated controls.

5.4.2 Process Control with a Framework

As an important control mechanism, we look at software frameworks again (though we will formally discuss in Chap. 7). A framework is a set of structurally related, domain-customizable modules to implement an aspect of a system, a subsystem, or even an entire system. A framework depends on customizable modules, when implemented, to become a working program. Design of a framework is based on a design strategy to implement what's stable and abstract out what's changeable. The abstracted processes, captured by abstract operations, are “placeholders” for concrete implementations of the operations when a framework is used for a concrete application. A context in which a framework is used can also vary. For example, a framework for a self-check-in process can be used at an airport or a railway station. Contextual variations can generally be addressed by interfaces.

As an example, consider a framework for an animation application with user interventions through user interface widgets to pause, resume the animation, or change the animation content. Because of our optical illusion, an animation is created by presenting a sequence of still images or objects in quick enough succession. This sequence of images or object profiles is called animation key frames, which must be changed or altered in an animation process. This process is stable because the underlying animation mechanics is the same regardless of a particular animation scenario. We can then design an interface to abstract away changeable details of an animation key frame with two essential, but abstract, methods:

- A method to initialize images or objects
- A method to update a key frame

Besides, abstract methods and other interfaces can also be designed if we want to capture minor variations of an animation process. Chapter 10 contains a case study that illustrates this process of a framework design.

Developing a framework is perhaps the most effective way for software reuse by maximizing its applicability to possibly an entire system. For a particular application instance, we can simply focus on implementations of required interfaces and abstract methods. Frameworks are all about process controls. For similar processes even with very different application scenarios, developing frameworks remains a viable option to streamline software development and shorten development cycles with consistent development quality and software maintainability.

5.4.3 Controls for Event-Driven Systems

Event-driven applications achieve distributed controls because of an event-driven programming model consistent across programming languages (such as Java’s *awt* model). In such a model, a control object (known as an event listener) is registered with an event-enabling

component. The system responds to an event in a callback operation embedded in a listener class, which implements an interface appropriate for a designated event. Thus, each event-enabling widget works with a dedicated event-handling object—a control object often created with an anonymous class or an inner class. For a program that involves many user-interaction widgets, too many scattered small event-handling classes may negatively affect code comprehension, refactoring, and extension. There are a few alternatives for reorganizing event-response controls:

- Use a combined control object by implementing all necessary interfaces. This option may effectively reduce cluttering of small control classes with a less cohesive overall control class.
- Extend a widget class with implementation of an event interface to create component objects with an event handler “built in.” This approach can be ideal if a widget’s internal data is adequate for event handling. Otherwise, object coupling would be a tradeoff.
- For events of the same kind such as button push, we can use an “integrated” control by “consolidating” responses polymorphically if details of the responses to different event sources can be abstracted way, as shown in the following code:

```
class A implements ActionListener{
    public void actionPerformed(ActionEvent e){
        ...
        String t = widget.getTask();
        Response r = ResponseFactory(t);
        r.respondToEvent(e);
        ...
    }
}
```

Code in an event-driven application can be messy and structurally rigid. Thus, a decision on restructuring control objects often has a priority to improve the clarity, readability, and some structural flexibility of the code.

5.4.4 Different Control Roles

For some applications, a centralized control is how things work. A control of elevator operations is a good example. An elevator control directs elevator movement and must be able to determine the direction the elevator travels and the floor to stop. Thus, the control must know the current states of the floor and elevator buttons. When appropriate, an elevator control might even manage elevator door operations. Thus, the role an elevator control plays is global and central. It is much more often, however, that we design objects

that play control roles in a “local” capacity to control dynamic processes. Here are a few scenarios of “local” control:

- Control information transfer from one object to another in either a synchronized or asynchronous fashion. Such an object can be useful to reduce object coupling.
- Control an assembling process, much like a (virtual) pizza making process by “picking” the ingredients and passing the “composed pizza” to a “baking” process.
- Control operational contributions from collaborators, much like a general contractor who does home remodeling by contracting the tasks to subcontractors. Such an object may be called a “structurer.” Using the analogy, this general contractor would “structure” subcontractors for the right tasks done in a right order at the right time. The difference of a structurer from a coordinator can be subtle, but a structurer is about presenting a coherent whole by coordinating the disparity of the pieces.
- Control an adaptation process. For example, consider a method $a.op(x, y, z)$ where parameter z is no longer practical or meaningful to use. To resolve the issue, we create an object b , collaborating with a , to offer users the method $b.op(x, y)$ instead. Object b may be called an “interfacer” or “adaptor” to bridge a “gap” between entities. A “gap” can also be a mismatch between two processes, and work to “bridge” the “gap” can be substantial and complex.
- Control a decision-making process such as channeling incoming messages or user requests to the right handlers. An appropriate analogy would be at a service store of a service company or a government service office where a person directs customers to the right service windows.

5.4.5 Objects Are Designed to Control

Design of distributed controls is a core characteristic of an object-oriented design. Arguably, if an object is not a domain “workhorse,” it provides a control in two perspectives. One is to provide a centralized control with its adequately encapsulated data, and the other is to manage a process to provide certain control in its computing vicinity. The first kind is explicit and relatively easy to identify such as a control of elevator operations. The second kind probably comprises most of objects we design. They participate in a control process collaboratively. Thus, appropriately identifying an object’s role in such a collaborative process is key to object design.

A complex application typically has multiple control centers due to complexity in multiple application aspects like event handling, complex sub-processes, data visualization, or access to external systems. The appropriateness of a distributed control style for a given software process (or the way objects participate in an overall process control) is a design decision often based on a chosen style of the system architecture.

Design patterns (to be covered in Chap. 8) provide certain control structures for objects to participate, locally or globally, in control processes of software operations. There are three groups of patterns depending on their roles in process control:

- Control object creation
- Control process transfer (such as bridging to external processes, functioning as proxies, adapting to, or interfacing with existing processes)
- Control over object’s behavior (with behavioral strategies, by reacting to events, or through coordinated communications)

In closing of the section, appropriate distribution of an overall system control and computational intelligence over objects is frequently a more challenging aspect of an object-oriented design.

5.5 Object Cohesion and Coupling

Objects collaborate through defined dependency or association relations. Making an object do an appropriate amount of work with an appropriate level of help from collaborators is a concern about object’s cohesion and coupling. “Good” cohesion with the “right” amount of coupling makes an object easier to design, maintain, and extend.

5.5.1 What Are the Issues?

Like cohesion of a module, object cohesion is a measure of relatedness of its behavioral operations. When modeling a concept or a role with an object type, we design the operations that are related to and appropriate for the concept or the role. More importantly however, objects should be “experts” in the role they play or in a conceptual area they serve. They should provide every needed responsibility if they have the encapsulated data resource to do so but assume nothing beyond what they can deliver without using external information. A less cohesive object assumes responsibilities in less related areas.

Like module coupling, object collaboration is generally necessary. But collaboration introduces coupling with other objects, which can make updates to coupled objects more impactful and consequential. It can also encourage temptation to make an object do more by coupling with more objects, thereby reducing the level of cohesion, readability, and maintainability. More coupling does not, however, necessarily mean lower cohesion, as complex responsibilities necessitate substantial object collaborations. Issues about cohesion and coupling often appear together for objects.

Incoherent responsibilities can be apparent such as responsibilities that appear to be associated with modeling concepts in different domain themes. But often some analysis would be needed to understand whether a responsibility is coherent with the role of an

object. For example, if we suspected that an *OrderShipment* object might be responsible for the action *doOrderInspection*, we may further look at what inspection might entail and how this action might be related to other responsibilities in terms of behavioral relatedness, data sharing, code reuse, etc. What we find would help us determine the appropriateness of this responsibility and the possibility of a collaborating object type, say, *OrderInspection*.

Incohesive objects are often due to inappropriate information sharing. Sharing instance variables with “public” visibility may encourage overreaching responsibilities, making all objects involved less cohesive. Public instance variables can also jeopardize the validity of class invariants. More often however, cohesion breach is caused by using externally encapsulated data. Consider the following instance method (say, in an object type *OrderManager*), which retrieves an order list encapsulated in object type *Order* to compute the value of an order.

```
float totalSales(){
    double totalSales = 0;
    for(Order : OrderList)
        for(Item : order.getOrderItemList())
            if( !item.isReturned() ) totalSales += item.getUnitPrice(); }
    return totalSales;
}
```

But the total value of an order should be a calculation *Order* is responsible for. This use of an external order list makes *OrderManager* less cohesive because of the calculations not supported by its encapsulated data. This also creates unwanted code coupling (with the inner loop vulnerable to potential changes in *Order*). This code coupling may also make order items vulnerable to unintentional modifications when retrieved as aliases. The issue can be easily corrected, however. With an additional method of *Order*, say, *getOrderValue* (), the inner loop in the code above can be eliminated with better clarity and improved cohesion:

```
float totalSales(){
    double totalSales = 0;
    for(Order : OrderList)
        totalSales += order.getOrderValue();
    return totalSales;
}
```

Creating getters in an object type may potentially compromise the principle of information hiding. It can be attempting to use getters of an object to access data for a task that an object may not be responsible for. The following code demonstrates this side effect to certain extreme, where an object of bank headquarters retrieves the balance of an account from one of its branches.

```
Balance balance = centralControl.getBank(bankInfo).getBranch(branchInfo).
    getCustomer(customerInfo).getAccount(accountInfo).getBalance();
```

This code is simply rigid; a change to the behavior of a getter in any of the objects involved may potentially break this code. A contextual object of the code is tightly coupled with the objects involved. Besides, it is also a statement difficult to read. The proper way for an object to work with a collaborating object is to “ask for” what is needed so the collaborate provides the “missing” responsibilities. For the example above, each an object should communicate only with an explicit collaborator to get the balance. Thus, a statement in the top layer may look like:

```
balance = centralControl.getBalance(bank, branch, cust, acct);
```

Object *centralControl* has its own version of retrieving an account balance with a collaborating bank object by providing required information, and so do a bank object, a branch object, and a customer object:

```
bank.getBalance(branch, cust, acct);
branch.getBalance(cust, acct);
customer.getBalance(acct);
```

In much of the same way, to turn on power of a television, we avoid a statement like:

```
myTelevision.frontPanel.powerSwitch.powerOn();
```

Instead, a television object communicates only to a front panel object, which communicates to a power-switch object to turn the power on in this sequence:

```
myTelevision.powerUp();
frontPanel.turnPowerOn();
powerSwitch.powerOn();
```

These are small “wrapper” methods that do nothing but delegate a request. This is a tradeoff between somewhat inefficient codes with much improved readability and, more importantly, improved object cohesion and coupling.

Code vulnerability to potential changes due to inappropriate coupling translates into code rigidity—making code refactoring difficult. Inheritance can be another source for tightly coupled objects (thereby code rigidity) when changes to a super class may trigger a chain of changes that must be made to its subclasses. The deeper a hierarchy is, the more rigid the code can be. This is commonly known as “fragile base class problem” when working with instable inheritance hierarchies. The closer the instability is to the base, the worse the problem is. Consequently, leaf object types in a deep hierarchy can be

particularly brittle. This fact just gives us another reason not to extend concrete classes to introduce deep inheritance hierarchies.

By far, violation of information hiding leads to coupling that makes code rigid (cohesion is a victim as a result). If the violation occurs at multiple places, even across the layers, code coupling can be so severe that object communications become contrived and convoluted. Consequently, code refactoring becomes difficult, as does code comprehension. Ultimately, it may significantly hinder system maintenance and extension. Often, the only way to extend a system is to decouple what has been improperly coupled, and corrections can be difficult and costly.

5.5.2 Law of Demeter

A design principle known as *Law of Demeter* (or the Principle of Least Knowledge) was intended to address inappropriate level of “intimacy” among software units. Proposed in 1987, the Law describes what would be considered appropriate knowledge a software unit possesses:

- Each unit should have knowledge only about units that are “closely” related to the current unit.
- Each unit should only talk to its immediate friends, not strangers.

Software units are variables (including objects) and methods. When implementing an object type O , we should, implied by the Law, access variables and methods only visible to O . In particular, if we implement an instance method m of O , we can only access:

1. O ’s instance and static variables, internal constants, or global variables O has access to (within the scope of m)
2. m ’s parameters
3. Any object created in m
4. Any object returned by an m ’s sibling method
5. Any combination that falls into above categories

The Law may be casually stated as “use only one dot” (in terms of operation invocation by an object). Thus, the code $a.b.method()$ may break the Law. The reason is that if a is visible to O , $a.b$ is not; therefore, b is a stranger to O . But this “formula” should not be used blindly. For example, if we use an iterator to access some service objects, a statement like $iterator.next().service()$ is appropriate, but $iterator.next().data$ is not because “ $data$ ” is not visible to the client of $iterator$.

Use of an excessive number of “dot” operations in a code statement is almost always a sign of questionable cohesion and coupling. When this occurs, it may mean either that an object tends to overcontrol beyond the territory of its data or that some data may be

misplaced. Similar statements are likely to occur also in the neighborhood and in other parts of the program. Relations among objects become contrived or convoluted, significantly affecting code comprehension and verifiability.

A more colloquial version of the Law is: “Tell, Don’t Ask” (to “tell” what is needed, not “ask” for more than that). It is otherwise also known as the “Shy Principle” (to stay shy of what you might be able to access). In other words, a collaborating object should offer all needed operations that use its encapsulated instance data, as opposed to exporting data for other objects to consume. An object should be as intelligent as they can be with its own resources to maximize what they can do.

5.5.3 Objects with No Overlapping Behavior

To minimize overlapped behaviors, object types should be behaviorally “orthogonal” or as much different behaviorally as they can be. For example, if both *Order* and *Shipping* have the access to an address list of customers, the two types may contain competing operations. Objects offering similar or competing operations, if used as collaborators, may potentially introduce inconsistent behavior in a hosting object because the overlapping operations of the collaborators may not be created or updated consistently.

DRY Principle (Don’t Repeat Yourself) is meant to avoid code redundancy, encourage code reuse, and discourage cut-and-paste coding practices. But this principle may apply more broadly to the design of objects to avoid overlapping properties and behaviors to minimize the possibility of inconsistent object behaviors. Code refactoring could make objects less orthogonal overtime as types are split or subclassed. The following heuristics may help avoid creating evolutionarily less “orthogonal” object types:

- Keep only a core set of essential methods of an object type.
- Create static methods (such as extension methods in C#) in collaborating objects, when appropriate, to avoid subclassing, and provide safe extension of objects’ capabilities with minimal side effects.
- Use polymorphism to control functional variations.

5.6 Iterative Design of Objects

Object design is also an iterative process. Software requirements provide essential ingredients for the design of object abstractions. The initial design of objects will likely undergo subsequent updates and changes when the process of analysis and modeling proceeds deep into developing a domain model and an architectural structure.

5.6.1 Initial Design of Domain Abstractions

A domain model (i.e., problem-domain object model) is a model of the entities within a business context. It is to provide an “overview” of the interrelationships among the business entities and their functional responsibilities. As appropriate, a domain model also provides a high-level description of data encapsulated in domain entities (separate from a data model). Some domain entities are of high level such as a business control that might be consistent across businesses of the same kind. Some are of low level such as entities that involve business policies and rules, which may be company specific.

Software requirements provide initial input to conceptualizing candidate domain objects. Requirements are typically stated in three different forms: user requirements, system (functional) requirements, and non-functional requirements. User requirements are based on ways a system can be used, which contextualize the functional requirements. User requirements, which may include elaborated interactions with the system, often serve as initial input to a domain design. These requirements are stated from a client’s viewpoint, which are further elaborated from viewpoints of an analyst or modeler. Design considerations based on user requirements elaboration may include:

- Design goals
- Design heuristics
- Possible ways for the system to respond to a user interaction
- Similar features developed in the past
- Ambiguities and things that might appear ill-defined
- Potential technical challenges
- Implications of non-functional requirements

Consider a user requirement about online bill payment of a banking system. The requirement might be stated in a usage scenario that specifies how a user may schedule a payment by completing a sequence of steps interacting with the system (known as a use-case scenario). By analyzing and elaborating the steps (with additional information from the system requirements), we discover and reason about potential objects the system may need. At the same time, we might also think about any past experiences when a similar software was developed with closely related features. A particular kind of user interface, if preferred, may also place certain constraints on the design of domain objects that need to be taken into consideration. Here are a few other design considerations for online bill payment:

- Ways to maintain transaction records
- Possible use of an external payment API
- Possible linkage to other accounts when fund transfers are necessary

- Scheduling multiple payments concurrently
- Ways to track payment activities
- Options for past payments search

These considerations, however, must be derivable from the requirements and consistent with design goals we want to achieve. When modeling an entity, the more specificity a name contains, the better we understand about the entity's responsibilities and possible representations. Consider these names: "account," "bank account," "savings account," and "high-yield savings account." While the concept of "account" may offer better opportunities for designing polymorphic behavior, "high-yield savings account" would help us better identify the exact responsibilities of an object.

Objects are fabricated entities; anything that can be affected by the system or connected to the system can be potentially an object. If we view software requirements as telling a story, we might see objects as characters in the story, object types as concepts to which the characters belong, and responsibilities as how characters behave in the story. If we analyze software requirements as we do a story, we would not be content with the stated value of the story. We may want to explore what might be behind the story and how we might be able to construct similar stories with extended roles of the characters or extended structure of the story.

5.6.2 Subsequent Design Validation and Refactoring

As the process of requirement engineering progresses, candidate object types are reviewed, validated, refactored, or removed. Additional object types can be discovered and added. To facilitate validation of objects' roles and responsibilities, we can group object candidates based on their stereotypes, application themes, layers of abstractions, user requirements they satisfy, and their dependencies. To update existing entities, we look for symptoms that may suggest design flaws such as instance variables of similar nature, overlapped operations, or trivial operations with little behavioral value added. This process might be termed "static validation of the candidate objects." A "dynamic validation" is to simulate how objects would work together to complete certain tasks, implement certain usage scenarios, or provide certain services or structural supports. We use modeling languages (to be covered in the next chapter), our minds, or (when helpful) code prototypes to perform dynamic validation, allowing quick decision-making.

When software requirements evolved, so does our understanding about the software structure and potential new risks. Based on our improved understanding, we proceed with object design iterations that may result in minor design tweaks, moderate design refactoring, or even a complete redesign. This iterative process may be proceeded

concurrently with code construction and software testing. We ask very different questions in design iterations such as:

- Are the roles objects play unique in the design?
- Is there a fundamental shift of our design goal (if the software structure has been significantly tweaked)?
- Are all the software requirements mapped by the design of the software entities?
- How stable are the requirements going forward?
- Are there other tradeoffs to be considered?

Questions we ask during the initial phase of object design are typically around the appropriateness of the object types, responsibility assignments, or effective object collaboration. These questions become less frequent if at all reoccurring in design iterations. In other words, the initial design focuses more on the static aspects of the candidates, whereas subsequent design iterations are more likely to focus on the dynamics of object interactions.

As an example, consider user login, which is a domain concept for many software applications. Initially, we may associate the concept with an entity that has merely two universal attributes—a user identification and a password. Suppose the application is about an e-commerce platform that has three different kinds of users—shoppers, sellers, and advertisers, each with its own application scope, access restrictions, and resources to use. Upon successful login, a customer of a specific kind would be redirected to a separate interface for service activities. With these requirements, an initial design of the abstraction may look like:

```
abstract class Login {  
    private String userId, password;  
    boolean validate(String userId, String password);  
    void redirect();  
    void recoverLoginCredentials();  
}
```

Whether or how we would refine this abstraction might depend on further analysis on the requirements, constraints, and software resources. Questions we may ask could include:

- Should all three kinds of customers be subject to the same process of login validation?
- Is there an appropriate third-party process to help with user credential validation and recovery?
- What is the persistent data store required to use?
- Are there existing login APIs we might take advantage of?

Non-domain objects are design objects. Design of such objects is often based on the behavioral and structural needs of the system. As discussed earlier, these objects exhibit certain control characteristics to direct, manage, or structure operation flows of the software. Some are designed when the architecture is being formed; some are added when needs emerge.

5.7 Summary

A design process begins with software analysis with the goal to create a set of object types that share computational complexity and intelligence with a balanced distribution of software tasks. Object design is a process of conceptualization, validation, confirmation, asking questions, and raising focused concerns at different times of the iteration cycles. This process is interwoven with other aspects of design throughout the whole development phases. We ask “what if” questions and think of “what might be possible” when we do concept modeling. We explore unfamiliar areas or aspects of a design when we do validation and perform iterations. We exercise creativity when conventional approaches fade.

An object is to model a “thing” of any nature. We discussed in this chapter an object design process, design guidelines, practices, and techniques. Some in-depth discussions were given to object discovery and design of objects that participate in an overall control mechanism of the software either as collaborators or as hosts to delegate control tasks. Object collaboration is essential in the design of objects. However, violations of information hiding in object collaboration can lead to highly coupled, rigid code relations that can be difficult to refactor and correct. Combined with the discussions on method coupling in the previous chapter, we may safely say that minimizing coupling is a common priority of software design with any design paradigm and methodologies.

Exercises

1. To simulate restaurant operations, you need a design of entity *Server* (to model waiter/waitress operations). List the entity’s responsibilities and collaborators.
2. Write an interface to model a quadratics function $ax^2 + bx + c$, where a , b , and c are real numbers, with operations related to a quadratic function, including an abstract form of the representation invariant (i.e., class invariant).
3. Argue both for and against (with perhaps different perspectives) an object type *Money* that would include currency conversion services.

4. Identify design flaws of the following two classes and suggest possible corrections.

```

class Email {
    private final String Signature = "Regards";
    public String constructEmail(User recipient, string body) {
        String str = "";
        str += getNiceUserName(recipient);
        str += "\n";
        str += body;
        str += "\n";
        str += Signature;
        return str;
    }
    private String getNiceUserName(User user){
        return user.firstName() + user.lastName();
    }
}
class User {
    String first, last
    public String firstName() { return first; }
    public String lastName() { return last; }
}

```

5. Given the following code:

```

class O {
    private B b;
    C n(){ return new C(); }
    void m(A a) {
        b.hello();
        a.hello();
        GlobalClass.getStaticInstance().hello();
        new Z().hello();
        n().hello();
    }
}

```

- (a) Identify any statements in the code that violate Law of Demeter.
 (b) Write two more code statements involving use of objects that are also supported by the Law.
 (c) Write a code statement that violates the Law.
6. Someone might argue that a statement like `book.pages().last().text()` would be fine because there is no direct access to externally encapsulated data, despite that it appears to be a violation of Law of Demeter. Reason against this argument.
7. Suppose object type *A* aggregates an instance of object type *B*.
- Describe cohesion and coupling implications if *B* is a concrete type.
 - Do part (a) again if *B* is an interface.

8. A payment through a credit card reader may appear simple in terms of the payment process we observe. But there are many creditors involved in collecting their shares of a payment at the point of sale. These are:

- The cardholder: the person making a payment (to get a fraction as payback).
- The merchant: the business selling a service or product.
- The acquiring bank: a financial institution that lets the business take money from payments.
- Payment gateways: the services that connect with credit card companies to make it easy for the business to accept payments. (The payment gateway collects payment details from the transaction and routes the information to a payment processor.)
- Payment processor: the system that connects the merchant's bank account, the card network, and the card issuing bank for payment processing.
- Issuing bank: the consumer bank that determines whether the cardholder can fund the transaction. If approved, it'll release the funds for payment.
- Card associations in credit card networks: the entities responsible for setting the interchange fees and standards for compliance (such as transaction fees incurred when a payment is processed in a different country).

Design an object type (using a CRC card) that controls or coordinates an entire credit card payment process from collecting card information to completing a payment transaction with all parties getting their shares of a payment.

9. For banks in a business district of a large city, business is heaviest during lunchtime and toward the end of the day. Thus, it is useful to simulate customer flows at a service location to deploy adequate staffers to keep average waiting time at a desired minimum. Consider such a simulation to compute the average wait time at the end with three simulation parameters: (1) an average number of customers who may arrive during any minute (or a probability that a customer may arrive during any minute if more appropriate), (2) average transaction time regardless of the nature of a transaction, and (3) the number of staffers deployed. Total simulation time (say, 3 h) is an input, but not a simulation parameter.

(a) Design (and implement) the simulation with two design options:

- A monolithic program with no objects (if Java is used, that means the program has one required class containing the main method; all methods and non-local variables are static)
- An object-oriented program

(b) Describe the pros and cons of each program.

10. Suppose you need an object type *CourseSchedule* to track course schedules of a school. A course schedule contains only information about the course number, semester, day (of a week), starting time, and ending time. Users shall be able to:

- Add, remove, or update a course schedule
- Check any schedule conflict
- Search for a course schedule given a course number
- Display, as a string, all saved course schedules

Besides, you can assume that there is an in-memory storage with data from an unspecified persistent data source. Design this object type and write some code to show how you might use it in an application scenario and make changes to the design as needed.

11. A simplified student course registration system allows a student to register through console input. To start, a user enters a major name, and the system then displays available classes with their schedules for the major. The system then prompts user to enter a course number and a class section letter. Other requirements include:
 - Each student can register up to four classes (per term).
 - Each class is limited to 20 students.
 - A class can't be registered if it has a time conflict with a registered class.
 - A student can be waitlisted if the class is full.
 - A student can remove a registered class.Furthermore, the program can also print the roster of a class given a course number and a section number; for each registered student, a display includes the name, major, seniority, and email.
 - (a) Design object types for this system.
 - (b) Develop a prototype of the system that satisfies the requirements. It's a "proof of concept" program to show things are working together structurally, though many details can be left out.
12. Design object types for a program used to manage an address book that uses a text file to store address entries, each containing a person's first and last names, address, city, state, zip, and phone number. Figure 5.6 is a potential user interface of the program. The interface shall use a File menu with menu items New, Open, Close, Save, Save As, and Quit with respective functions explained as follows:

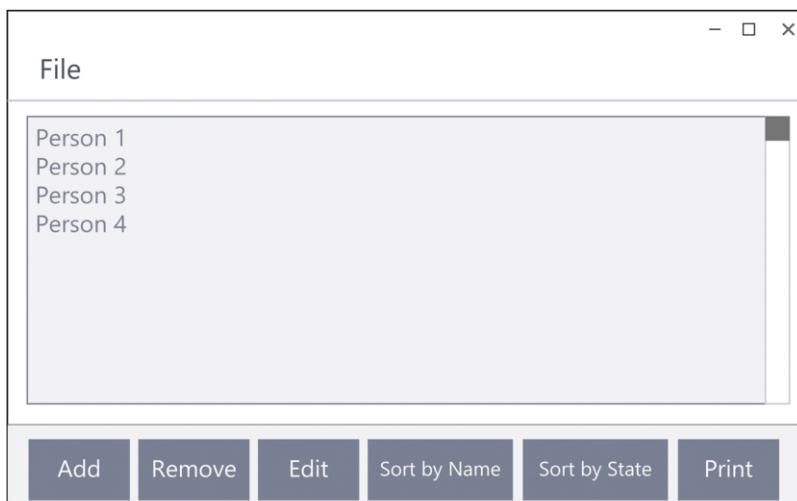


Fig. 5.6 A sample user interface of an “address book” application

- New operation is to *create* a new (empty) address book.
- Open operation is to open an existing address book by requiring a user to enter the name of a file.
- Close operation is to close an address book with changes automatically saved.
- Save operation is to save an address book by updating the existing file.
- Save As operation is to save the (in-memory) addresses into a new file.
- Quit operation is to save and then close an opened address book instance and terminate the program.

The program shall use the file name of the currently opened address book as the title of the main frame. When a new address book is initially created, the frame shall be titled “Untitled.” When saving an “untitled” (currently open) address book with Save or Save As operation, the user is required to enter a file name before “save” button becomes clickable. In addition, the program shall be able to:

- Add a new person to the address book
- Remove a person (when selected)
- Edit existing information about a person other than the person’s name (when selected)
- Sort the entries by last name or by state code
- Print as console output all the entries in the address book in a “mailing label” format

Based on these features,

- (a) Design object types.
 - (b) Implement a prototype of the application as a “proof of concept” to test the workability of the design. For example, the program can load data with one File menu item and one button that worked.
13. An automated teller machine (ATM) is illustrated in Fig. 5.7. Such a device has a card reader, a customer console (keyboard and display) for interacting with the machine, a slot for depositing envelopes, a dispenser for cash, a printer for printing customer receipts, and a key-operated switch to start an operator or stop the machine. The keyboard includes buttons for cancel, clear, and enter, along with the number keys. There are also physical buttons beside the screen for the selection of a transaction type such as cash withdrawal, deposit, and inquiry about account balance (although more recent ATMs have touchscreens for such actions).

The ATM operations are explained below:

- A customer can:
 - Make a cash withdrawal (typically in multiples of \$20) from any suitable account linked to the card. The bank must approve the withdrawal before cash is dispensed.
 - Make a deposit, consisting of cash and checks in an envelope, to any account linked to the card. The customer will enter the amount of the deposit into the ATM, subject

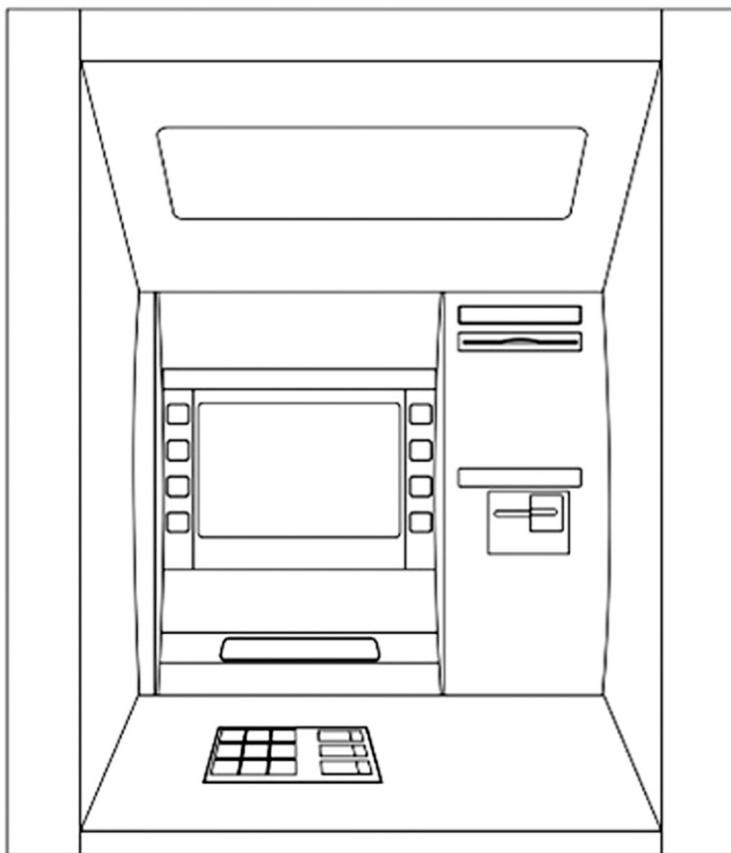


Fig. 5.7 Illustration of an automated teller machine (ATM)

to manual verification when the envelope is removed from the machine by an operator. The bank must approve the deposit before the ATM physically accepts the envelope. After the customer deposited the envelop, a second message will be sent to the bank indicating that the customer has deposited the envelope. The customer must deposit the envelope within the timeout period to make a valid deposit.

- Make a transfer of money between any two accounts linked to the card.
- Make a balance inquiry of any account linked to the card.
- Abort a transaction in progress by pressing the Cancel button.
- The system shall service one customer at a time.
- A customer shall be required to insert an ATM card and enter a personal identification number (PIN), both of which will be sent to the bank before the customer is able to perform one or more transactions.
- The card shall be retained in the machine until the customer indicates (when prompted) the end of a session, at which point the card is returned.

- The system shall allow a customer to try a total of three times if the bank determines that the PIN entered is invalid. When all three attempts are failed, the system permanently retains the card, and the customer must contact the bank to retrieve the card.
- The system shall display an explanation of the problem and then offer the customer a choice to do another transaction if a transaction fails for any reason other than an invalid PIN.
- The system shall provide a customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and available balances of the affected accounts.
- The system shall also maintain an internal log of transactions to facilitate resolving ambiguities that may arise from a hardware failure in the process of a transaction. Entries will be created in the log when the ATM is initiated and shut down, a message is sent to the bank, cash is dispensed, or an envelope is received. Log entries may contain card numbers and dollar amounts but will never contain a PIN.

(While not part of the simulation, an ATM also has a key-operated switch that allows an operator to start the machine and stop servicing customers when the operator needs to remove deposit envelopes and reload the machine with cash, blank receipts, etc. The lengthy problem description above intends to give design a good context of the requirements. Nonetheless, internal communication complexities are not simulated. For example, communication between an ATM and a bank is represented in a trivial manner of message exchange without functional substance.)

- (a) Design object types for this simulation.

(Hint: Some useful considerations:

- A controller to control an AMT user session.
- Is it useful to have an entity to represent a usage session of ATM?
- An abstraction “transaction” may represent any kind of transaction ATM supports with polymorphic transaction behavior.)

- (b) Implement a prototype for one of the transactions—withdrawning cash with an adequate ATM interface. Here is a sequence of user actions to further clarify the requirements:

- (1) Insert (symbolically) a bank card.
- (2) Enter a PIN (in response to a system prompt for entering a PIN).
- (3) Select transaction type of “withdraw cash” (in response to a system prompt upon successful verification of PIN).
- (4) Enter amount (in response to a prompt for amount entry).
- (5) Take the cash bills dispensed from the machine.
- (6) Enter finish (when prompted for the next transaction) and take the receipt.

Further Reading

- Cioch, F. (1990). Teaching software design using both functional and object-oriented decomposition. *Computer Science Education*, 1(3), 223–236.
- Jackson, D., & Rinard, M. (2000). *Software analysis: A road map* (pp. 133–145). Proceedings of 22nd International Conference on Software Engineering, Limerick, Ireland.
- Jorgesen, P. (2009). *Modeling software behavior – A craftsman’s approach*. CRC Press.
- Skrien, D. (2009). *Object-oriented design using Java*. McGraw-Hill.
- Wirfs-Brock, R., & McKean, A. (2007). *Object design, roles, responsibilities, and collaborations*. Addison-Wesley.



Software Modeling Languages and Tools

6

6.1 Overview

Software analysis and synthesis lead to software models. While design can often be a mental process, we use analysis and modeling languages and tools to help us reach our design goals and our communication responsibilities, which is the focus of the chapter. We start the chapter with a discussion about the relation between software analysis and modeling—their connections and subtle differences. We will then proceed to introducing analysis and modeling tools and diagramming languages. We will first illustrate software analysis with use cases, followed by an introduction to commonly used diagrams. Most of the diagrams covered are part of the Unified Modeling Language (UML). We will close the chapter with a section about opportunities of using customized diagrams in design. With many online resources and published book about use cases and the UML, we will focus on the correct and effective authoring of analysis and modeling artifacts using the tools. We will also provide discussions on tool selection.

As a word of caution, modeling languages and tools are not methodologies. Simply because we are using them to produce design artifacts does not necessarily mean the artifacts we created are of value. Like other languages of communication, misuse of modeling languages and tools can and often happen. To produce useful design artifacts, it is essential to not only correctly understand the semantic meanings of the constructs we use but also appropriately apply them to describing, accurately, our thoughts and ideas.

6.2 Software Analysis and Modeling

Software development starts with an analysis on software requirements—what the software is required to do and functionally behave. Analysis, in simple terms, is a process of breaking a whole into more manageable, more easily understood parts. The goal of software analysis is to understand software structure and behavior by identifying appropriate software abstractions, their relations, and a mapping between software operations and requirements they may satisfy. As applicable, software analysis is also to capture system operations in response to user interactions. On the one hand, by taking requirements “apart,” analysis may discover requirement ambiguities, inconsistencies, omissions, or even mistakes. In fact, software analysis plays a critical role in a requirement engineering process to provide essential ingredients for software specification, which constitutes the basis for the design of software elements and software construction that both the client and the developer can agree upon. Requirement validation, change, or evolution provides subsequent inputs for software analysis, leading to updates of the requirement specification and the software model. On the other hand, software analysis also produces an initial set of software elements and their relations in the problem domain. When requirements are solid, such as those for scientific applications, there are fewer analysis cycles needed to identify omissions, ambiguities, or mischaracterizations of the requirements. In such situations, analysis may turn out to be mostly a modeling process focusing on computational details.

Synthesis, meanwhile, is a process of putting things back together. In other words, if an analysis is a decomposition process, then synthesis would be a composition process. Analysis, with decomposition of the requirements into software elements, explores the possibilities the software can be built with different choices of software elements (i.e., what a system can be), whereas synthesis, by modeling how the elements work together, is to show how the system may actually work to satisfy the requirements (i.e., what the system will do). To use the card game example again, analysis would look into how a game is decomposed into card objects, a deck of cards, players, and a game controller, or other decomposition possibilities, whereas synthesis would show a software model of game playing with each design option using a prototype, modeling diagrams, or a combination.

We have used the term “modeling” on a few occasions. According to the Object Management Group (an international, open-membership, not-for-profit technology standards consortium founded in 1989), “modeling is the designing of software applications before coding.” As discussed earlier, design is a “fluid” term; it can mean an act, an artifact, or a thought process. It may also mean modeling.

Thus, modeling encompasses both software analysis and synthesis (as both are about the designing of a software application). We know what software analysis is about, but we don’t usually use the term “software synthesis.” Would it be more “conceptually helpful” if we characterize modeling as primarily a synthesis process? Analysis helps us design abstractions and their relations, whereas modeling helps us understand the adequacy of the abstractions, their representations, and their interactions to deliver software operations that satisfy the software requirements. This cycle of analysis, modeling, and validation is

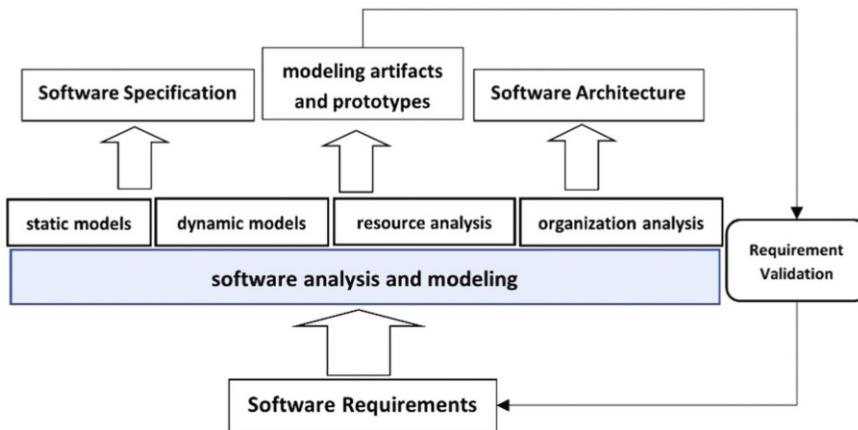


Fig. 6.1 The “input” and “output” of software analysis and modeling

repeated as needed in an incremental software development process (though these cycles might be performed only in our minds, leaving code and traces of refactoring to be the only “proof” that such iterations have ever existed). As a high-level design activity, analysis is where we weigh design possibilities with divergent thinking. Modeling can then be understood to be the process where we apply convergent thinking, leading to a software model. In other words, software modeling fills the gap between abstractions laid out in the architecture and a software model designed to satisfy the requirements.

Software prototyping is then a modeling technique to build a part of or the whole system to remove or mitigate uncertainties in complexity, architecture, or use of new technology. A prototype of the software is like a “proof of concept.” Once it has fulfilled its purpose, we may choose to discard the prototype or to further develop based on the prototype when it was reasonably well designed.

With software requirements as input, analysis leads to software specification and architecture—the two essential outcomes for “what the software is,” which also constitute the basis for non-architectural design and code construction. Figure 6.1 summarizes the roles of software analysis and modeling. This process is iterative, as necessary in requirement validation cycles.

If we view software analysis and modeling as two separate workflows of design, Table 6.1 summarizes what each workflow may produce if we want to understand more accurately the “scope” of each workflow.

Another way to look at the differences between analysis and modeling is that analysis tends to answer “what” questions, like:

- What do software requirements tell us?
- What are appropriate abstractions?
- What is a domain model?

Table 6.1 “Scope” differences between analysis and modeling

Process	Domain		
	Problem domain	Solution domain	System domain
<i>Analysis</i> , focusing on:			
• “What” • High-level abstractions and architecture • Understanding about requirements	Domain analysis produces: • Domain abstractions from conceptual perspectives • Structural relations of the entities	Solution analysis produces: • Decompositions (conceptual, behavioral, and functional) • Non-domain architectural elements	System analysis produces: • Software specification • Software architecture
<i>Modeling</i> , more of a synthesis process based on analysis, focusing on:	Domain modeling produces: • Entity representations • Understanding of domain entities from behavioral and interaction perspectives	Solution modeling produces: • Architectural validation • Infrastructural support elements	System modeling produces: • Modeling artifacts • Understanding of the software operations supported by the architecture • Prototypes

- What might be domain-model elements and their relationships?
- What might be an appropriate software architecture and a set of structuring elements?

Modeling is to provide answers to “how” questions, like:

- How are software elements represented?
- How do elements behave and interact?
- How does a software process, a subsystem, or the system behave?
- How do elements fit into the architecture?

Analysis and modeling may also differ in terms of fundamental software concepts and elements each deals with primarily. In an analysis, we often deal with use cases, requirements, abstractions, interfaces, relations (including hierarchical relations), and structures. For modeling, we may frequently encounter things like data, objects, modules, events, threads, concurrency, external processes, and conditions under which modules and objects interact to exhibit software system’s behavior.

In practice, however, it is often unhelpful to recognize such “scopes” between analysis and modeling in terms of what we do. Analysis can be understood as high-level modeling. High-level and low-level modeling activities are intertwined, and thus there is no clear separation between analysis and synthesis as design is opportunistic. In a design thinking process, it is often impossible to focus on one thing without “peeking” another. Modeling is a refinement and thus a continuation of an analysis. Therefore, analysis and modeling,

despite their respective focuses we have tried to characterize, are inseparable processes. We may answer some questions above in forms of mental simulation realized only in code. We may also answer some questions with assistance of analysis and modeling tools such as the UML, especially for the questions that we must communicate the answers to the stakeholders. More likely however, some of the answers, with relevant artifacts, will go into software documentation to provide guidance for software construction and future development.

6.3 Developing Effective Use Cases

Software requirements can be expressed in a variety of forms. A requirement is a general statement, but it can be contextualized in different ways. A commonly used way is through use cases. A use case is a specific case of using the system that can be implied or derived from software requirements. Thus, a software requirement may be mapped to multiple use cases. A use-case scenario (or narrative) describes how a user or an external entity may use or interact with the system to show how a use case may play out in the real world. Use-case narratives help identify potential software elements and their behaviors.

6.3.1 Use-Case Scenarios

A use case is a case that a user may use certain function or feature of a software system. For example, we use an ATM machine to check account balance, as “user shall be able to check account balance” is a function, feature, or requirement of the software that controls ATM operations. As a user, we swipe an ATM debit card, enter an ATM pin, select (in some way) “Balance Enquiry,” and finish the transaction. Each user action would trigger a response by the ATM (controlled by the software). Respectively, the ATM would verify the card, verify the pin, display the balance, and clean up the temporary storage in response to user interactions. These (software) responses are useful, based on which we can think about software elements such as a proxy bank account (to be able to handle multiple ATM transactions), entities that communicate with display and printing mechanisms of the ATM, and a controller that coordinates entity interactions to complete required transactions.

Consider another software system that compares sorting algorithms with randomly generated data. To use the system, one simply selects (with certain user interface) an algorithm and a data pattern (such as unsorted, reversely sorted, almost sorted, etc.), and the software manages the rest and displays the result. This is a use case too, but it lacks substance. The software is still a “black box” with little to analyze based on this use case. A valuable use case is rich in system response to reveal some observable system behavior. Such a use case helps us identify and design software elements that can exhibit the software behavior we have observed. In other words, good use cases provide important contexts for

Table 6.2 A use-case template

Section	Content description
Use-case identification	Use case identifier, reference number, modification history, responsible personnel, etc.
Preconditions and system's pre-state	The conditions and the state of the system at the beginning of the use case
User goal description	Goal to achieve and software requirements that the use case provides a context for
Actors	List of entities interacting with the system
Normal flow	Interactions that are most common between actors and system that contribute to achieving the user goal
Alternate and/or exceptions flows	Any alternate flows relative to normal flow or exception flows for error handling
Postconditions and system's post-state	The conditions and the state of the system at the closing of the use case
Assumptions and other constraints	These are situations other than pre- and postconditions and the system states
Notes	Known issues and potential impact

us to understand software requirements. These contextual descriptions (i.e., scenarios) about software requirements are used as inputs for software analysis and modeling.

For good use cases, we need effective narratives to describe them to capture as much operational intricacy of the software as possible. A formal presentation of a use case generally has sections listed in Table 6.2. Other possible attributes of a use case, such as a name, an identification number, author, priority level, etc., may also be present, but not critical. The most important sections of a use case are the following:

- A normal flow (or a main success scenario)—the most likely interaction scenario under normal operational conditions to achieve an observable user goal
- An alternate flow (or an alternate scenario)—a less frequent interaction scenario under normal operational conditions or a scenario with a less common operational condition
- Exception flow (or an error scenario)—known error conditions due to mistakes made by an actor or caused by system difficulties

The following is a use case describing how a user can view and print an itinerary of a scheduled trip (i.e., a user goal) at a commercial travel portal.

- *User goal:* Print the itinerary of an existing trip.
- *Requirement the use case is based on:* User shall be able to view and print the itinerary of an existing trip.
- *Precondition:* User has logged into the portal system (through a “Login” use case).

- *System state at the beginning of the scenario:* The use case starts immediately after “Login” use case ends without error conditions.
- *Normal (interaction) flow:*
 1. System defaults user to page “email setting” (for email information update).
 2. User updates email information if necessary and then selects link “My Itineraries” (among other possible links).
 3. System displays the page with recent itineraries listed.
 4. User selects the itinerary (to be printed).
 5. System displays the details of the trip.
 6. User selects “Print Version.”
 7. System displays a page that contains only the information related to the trip.
 8. User clicks “Print” button.
 9. System sends the page to an output device, and the use case ends.
- *Alternate (interaction) flow A:*
 - 6-A. User selects “Email Itinerary.”
 - 7-A. System displays a textbox containing default email address to allow entering a different address if needed.
 - 8-A. User enters an email address and activates “Send” button.
 - 9-A. System displays message “Itinerary Sent,” and the use case ends.
- *Exception flow Ea:*
 - 9-Ea. System identifies incorrect email address and allows user to reenter.
 - 10-Ea. Continued with step 8-A, or user chooses to exit, and the use case ends.
- *Postcondition:* User has successfully printed the itinerary, or had it sent to an email address, or encountered an error and aborted the operation.
- *System state on completion of the use case:* The system remains at the last page visited.

A use case must describe accurately “who is responsible for what” to be useful for identifying potential software elements and their responsibilities. Ambiguities, uncertain actions, and complex language constructs can all contribute to ineffective use cases. The following guidelines may help develop accurate and useful use cases:

- Every statement should contribute to the understanding of software operations. A statement like “user retrieves a printer to be ready” is useless as it conveys nothing about a software operation.
- Always use a simple sentence for a use-case statement in an active voice for easy and unambiguous understanding.

- Avoid details if a more abstract statement makes sense. For example, compare the statement “system displays ‘thank you’” with “system acknowledges completion of the process.”
- Information that may add clarity but potentially complicate the structure of a statement (may thereby affect comprehension) should be extracted and footnoted.
- Each statement should be definitive. If alternatives exist, they should be handled as alternate flows. For example, we shall state “user provides a credit card” (which probably happens most frequently), not “user decides whether he or she wants to use a credit card or debit card.”
- Each step must succeed before advancing to the next step; otherwise, there is abnormality to be handled as exception scenarios.
- Use appropriate labeling in alternate and exception follows to allow easy association with the main flow.
- Pre- and postconditions should include the state of the system before and after the use case. These conditions can be stated separately for better clarity.
- An “actor” can also be an input or event if it can trigger an interaction with the system.
- A use case captures no algorithms, business rules, or user interface details, as these things may change overtime and have little implication on an architecture. For example, statement “user provides required personal data” is more appropriate than one that describes the exact form of the required personal information.
- Because of our cognitive limitations, we want to keep the number of exchanges in a normal flow under a manageable level, say, less than 10, for effective comprehension, management, and utilization. To reduce the complexity of a use case, collaborating use cases can be used when possible and appropriate.

6.3.2 Use of Use Cases

Use cases are useful in several ways. Most apparent is a possible mapping between software responsibilities (identifiable in a use case) and potential software elements and modules to assume the responsibilities. For the example earlier, the following is an annotated version of the scenario with potential system implications:

1. System defaults user to page “email setting” (for email info update)—*The last statement of the login module is a “redirect” statement, which loads a page (upon a successful login).*
2. User updates email information if necessary and then selects link “My Itineraries” (among other possible links)—*The page contains a user interface for email update and buttons with server-side modules to handle button events; one of the buttons shall be “My Itineraries.”*
3. System displays the page with recent itineraries listed—*On page load, login information is retrieved from previous page (as primary key) and used to retrieve itineraries from*

customer database; the titles of the itineraries are populated into a list container with each itinerary item selectable; details are saved into itinerary objects and maintained by an in-memory/in-browser collection.

4. User selects the itinerary (to be printed)—*Upon selection, relevant object is retrieved from the collection.*
5. System displays the details of the trip—*Selected itinerary information is retrieved from the object and populated into an html container with two buttons for printing or emailing itinerary.*
6. User selects “Print Version”—*Itinerary data is passed to a page to be loaded (this can be done in a variety of ways such as using a query string, context object, etc.).*
7. System displays a page that contains only the information related to the trip—*Upon page load, itinerary information (from previous page) is retrieved and displayed in an html container, with a button for printing.*
8. User clicks “Print” button—*An appropriate handler at the server handles the printing.*
9. System sends the page to an output device, and use case ends—*A printer prints the page content.*

The annotated text describes one way the system could be designed to implement the use case. The narrative allows identification of pages, server-side modules, temporary storage needs, potential objects, and communication mechanisms. Given the nature of a web application, the mapping between the dialog and software elements in this case is relatively direct. For software of different nature, such mappings can be indirect, in which cases, further functional decompositions may be required for further analysis in the solution domain. Subsequent use-case iterations may add, remove, or combine use cases, which may also necessitate updates of the steps in existing use cases. As a result, existing modeling elements and their communication schemes must also be validated, modified, or redesigned when necessary. These iterations gradually stabilize the use cases and allow the adequate validation of the requirements and the current architecture.

If use cases are developed first, software requirements can be extracted from use cases. The requirements are validated and updated as use cases are updated, added, or removed. The process of mutual validation between software requirements and use cases can be an effective mechanism in a requirement engineering process. Besides, use cases also provide direct test scenarios for integration testing.

Agile software development often uses “user stories” as a form of software requirements. For example, a user story might be like “As a user, I want to view my travel itineraries so that I can select one and print.” The annotated texts of the use case above might provide a context to develop acceptance criteria of the user story. In other words, use cases can also provide contexts for user stories and help us understand operational implications of user stories. Regardless of an approach to software requirements, use-case analysis is always an important tool for software analysis.

When there are a small number of well-developed use cases to cover most of the requirements, development of the software may adopt a use-case-driven process. This

(small) set of use cases are chosen based on risk-driven analysis and criticality to the overall software project through an analysis-modeling process. The process then is divided into two phases. The initial phase includes the following activities with interleaved interactions:

- The narratives are “elaborated” to identify major abstractions of all kinds (classes, interfaces, processes, components, subsystems, etc.).
- Design the initial architectural elements to layout the abstractions.
- Prototype, test, and validate the architecture to detect issues and flaws.

As requirements evolve, so may use cases (or vice versa). Thus, there must be a process to validate, correct, improve, and advance the current architecture. This is the second phase of a use-case-driven process. We carry out this second phase iteratively to re-evaluate and refactor the current set of abstractions with each update of the use cases. Each iteration follows these steps:

- With newly updated use cases, reassess risks and rationale of the process adoption and update the risk mitigation strategies, the architecture, and the guidelines.
- Identify additional abstractions and make necessary architectural changes.
- Update the prototype to reflect the changes.
- Update the design artifacts as applicable.

As the iteration slows, the abstractions and architecture may become stabilized. We utilize use cases as guiding test cases to test the prototype and its subsequent updates as more details are added. The updated versions of the prototype may provide further validation and feedback on the requirements and the use cases.

6.3.3 Development of Use Cases

Use cases are professional products; therefore, it is probably rare that the client alone provides use cases in addition to software requirements. A plausible scenario is that software professionals write use cases in collaboration with the client based on an initial set of software requirements. When an application is user interface driven, keeping the number of use cases at a manageable level is important. It might not be rare that one thinks 3 use cases might be adequate and another comes up with 30 for the same software. E-commerce systems and those of similar nature are highly interactive and thus use-case-rich. In such a system, user goals are clearly identifiable—searching for items to buy, placing items in a shopping cart while browsing, checking out with payment, arranging shipping, etc. These user goals are candidates for use cases. We may identify more use cases during use-case construction. For example, while developing a use case “checking out with payment,” we might find that the use case can be significantly simplified with a

separate use case on “processing payment.” Here are a few heuristics that may help develop a manageable set of robust use cases:

- A “shallow” use case is one that doesn’t reveal significant system processing behavior or logic. Often, “shallow” use cases can be either abstracted away or dissolved into other use cases.
- A “deep” use case, in contrast, is one that reveals significant system processing behavior or logic (e.g., a use case to allow user to specify data cleaning criteria and the system would deliver a dataset that satisfies the criteria). A “deep” use case is not necessarily large (with many steps), but necessarily contains “deep” system responses.
- Excessive number of use cases can lead to fragmented abstractions, misplaced operations, and duplicated processes, making analysis more difficult.

Arguably, any software operation is a use case, as we use the operation to accomplish (directly or indirectly) a goal even if there is no user (or external) interaction with the system other than initial data inputs. Such use cases are not much different from algorithms (with only system steps). A “workhorse” operation is not likely of architectural significance and thus should not be an analysis focus. However, a control operation of a system or one of its subsystems can be a good use case despite the possibility of lack of interaction with an actor because it is likely to contribute to our understanding about the architecture. Thus, whether an operation constitutes a good use case depends on its architectural value and the possibility to advance our understanding about the requirements. For use cases where interactions are not significant, we may use a tabular form, such as one shown in Table 6.3, to describe it. This use case describes an operation “Scrub data” of the system “Analytika.”

For initial use-case development, consider a health club web system that manages its daily operations to replace its current manual process. Expectedly, club employees would help draft an initial set of use cases based on club’s daily operations and departmental responsibilities that can be digitalized. Table 6.4 lists some candidates of meaningful use cases. Meanwhile, the software requirements would probably be developed based on the tasks that are currently done manually. To ensure consistency between the use cases and the requirements, we use requirement validation loops to add, eliminate, and refine the use cases.

Like modules, use cases may relate to each another in certain ways. Appropriately defined relations among use cases can help simplify the steps and improve the effectiveness of a use case. A use case, such as “withdraw cash,” can use another use case, such as “process user login,” in its dialog construction. We might also be able to identify common dialog blocks in multiple use cases and define them as separate use cases to share. Reuse of use cases helps improve the clarity of the narratives. The commonly used relations between use cases are essentially two kinds:

- “Include” relation means one use case uses another. For example, “System prompts user to Login,” where underlined “Login” is a different use case. “Include” relation is

Table 6.3 A tabular description of a use case

Analytika: Scrub data	
Actors	User of the system Analytika
Description	<p>A user may use the function to perform data cleaning with the following options:</p> <ul style="list-style-type: none"> • Remove rows where values in any of the specified columns are missing • Remove rows where numerical values in any of the specified columns are outside of a specified range • For a numerical column, replace missing values with the column average • Convert categorical data to numerical, and then replace missing data with the column average
Data	<p>Input data can be in one of the three forms:</p> <ul style="list-style-type: none"> • A CSV file (a comma delimited text file) • Excel • Pandas's dataset <p>User can choose one of the above three forms for an output data file</p>
Stimulus	User activates a command with a visual component (such as a button)
Response	System displays a user interface for user to specify data cleaning options specified in "Description." The interface also allows user to upload a data file specified in "Data." Upon user activation of the operation, system displays the scrubbed data and provides options for user to save the resulting data in a form specified in "Data"
Comments	According to the current requirements, a data file must be uploaded from a local drive. But data loading from a remote site with a URL address may also be implemented if desired

Table 6.4 Use-case discovery of a health club management system

Actor	Use case
<i>Current club member</i>	Renew membership
	Cancel membership
<i>Past member</i>	Resume membership
<i>Potential member</i>	Complete membership registration
<i>Membership services department</i>	Request for membership
	Send promotional offers
	Send membership renew notices
<i>Marketing department</i>	Create promotions
<i>Club management</i>	Create reports for promotion, sales, etc.
	Create new programs, partnerships, etc.
	Manage memberships
	Manage club maintenance and updates
	Manage front-desk operations

functionally equivalent to “use” relation, though the latter may mean a mandatory inclusion.

- There are situations where a use case can be extracted to reduce the complexity of another use case. This kind of extractions is not for reuse; rather, an extracted portion may apply only in specific circumstances to not clutter the use case with unusual or exceptional behavior. For example, in an airport check-in process, it might be helpful to separate the process of dealing with luggage that exceeds the allowed limit by either dimension or weight, as such situations do not happen often, and the details may unnecessarily clutter the check-in process, affecting clarity of the use case. A common practice is to insert into the use case an “extension point” (such as “measure/weight luggage”) with an “extension condition” like “either the luggage weight or dimension exceeds the allowed limit,” where the use case expands to a separated use case if the extension condition is satisfied. Such a relation between two (or more) use cases is called “extension relation,” for which separate use cases extend or augment, as appropriate, the primary use case.

6.3.4 Use-Case Diagrams

Use cases address high-level software operations and features. They help identify an architecture to structure the elements we create through use-case analysis. Primary use cases may directly be responsible for the formation of an architecture, whereas alternate flows may provide architecturally significant “corner” or “boundary” cases. Such corner and boundary cases may also provide insights into an appropriate architectural style and possible non-domain entities. To facilitate architectural validation with use cases, we can group use cases to form a summary-level use case that accomplishes a larger user goal. For example, “Purchase a book online, have it shipped to the user with an ability to cancel while in transit” is a summary-level use case. For the health club example earlier, “User registers for club membership with possibility to cancel, temporarily suspend, or resume the membership” is also a summary-level use case. Summary-level use cases appear more comprehensive with multiple boundary or corner cases (e.g., certain restrictions may apply if canceling a book order while being shipped may be a boundary or corner case) and thus beneficial to architectural design and validation.

Summary-level use case or use cases of an entire system can be visualized using UML use-case diagrams. Figure 6.3 is a use-case diagram about airport check-in operations. A use-case diagram uses the following graphic constructs:

- A stick figure represents an actor interacting with use cases. For a non-human source of interaction (such as a data source), an appropriate visual image with a label can be used.
- An ellipse with a name inside represents a use case.
- A line segment represents an association relation, connecting an actor to a use case with which it interacts.

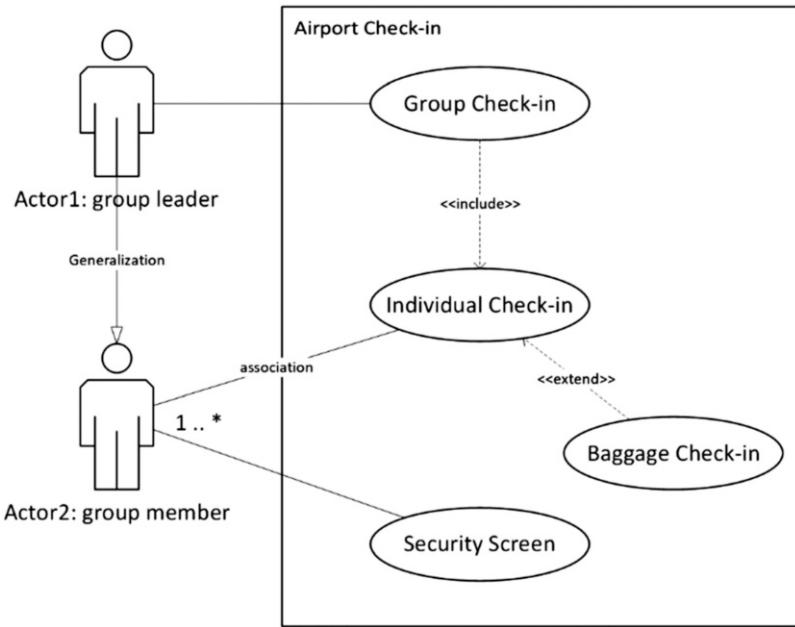


Fig. 6.2 A use-case diagram for airport check-in operations

- A directional dotted arrow indicates a relation between use cases with appropriate stereotype name “include,” “extend,” or “uses.”
- A labeled rectangle is a confinement of use cases, like large rectangle labeled “Airport Check-in” in the example Fig. 6.2, indicating a boundary of a subsystem or a summary-level user goal.
- While not common, an inheritance-like arrow can be used to indicate generalization/specialization relation between actors.
- While also not common, multiplicity notation of a relation between an actor and a use case can be used such as “one or more group members” would use the same “Individual Check-in” use case in the example Fig. 6.2.

When use cases cover most of the requirements, they represent a system decomposition based on user goals. This decomposition can be visualized with use-case diagrams to facilitate the validation of architectural suitability in terms of use-goal accommodation. Use-case diagrams may also be an effective means to communicate with the stakeholders about the system.

A common mistake that novices often make is to use a use-case diagram to describe the steps in an operation. For example, for a use case of “inquiry of account balance at an ATM,” instead of using a single oval to represent the use case, a novice may use ovals to represent “insert debit card,” “verify debit card,” “select transaction type,” “display

balance,” and “complete transaction.” This kind of mistakes can be a source for a large number of “shallow” use cases.

However, a use-case diagram is merely a graphic summary of the use cases to provide a static view of the user goals. It is not an analytical tool. Therefore, use-case diagrams cannot replace use-case scenarios. If beneficial, we may use a table (like Table 6.4), in lieu of a diagram, to summarize use cases with perhaps another column for the identification of the relations among the use cases.

6.4 Other UML Diagrams

The Unified Modeling Language was born in 1995 because of a merge of object-oriented methodologies by separate efforts and later adopted by the Object Management Group (an international, open-membership, not-for-profit technology standards consortium, founded in 1989) as a standard for its members. The language was published in 2005 by the International Organization for Standardization (ISO) and has since undergone multiple reviews and revisions. The most recent version was released in 2015. As object-oriented mythologies became popular in the early 1990s, the UML has been the standard diagramming tool suite for the visualization of software operations and their structural relations. Although the UML can be used for non-object-oriented design scenarios, we will benefit more from the full strength of the UML in an object-oriented development process (such as the Unified Process). With agile development practices, formal use of UML artifacts as part of a software specification document has become less popular. Nonetheless, the UML remains an important analysis and modeling tool for software design.

This section is to introduce four most frequently used UML diagrams. Given a large volume of printed and online tutorials on these diagrams, we will focus on the correct construction, appropriate use, and possible misuse of these diagrams.

6.4.1 Class Diagrams

The UML class diagram is probably the most frequently used diagram because of its simplicity and straightforward communication power. A class diagram describes the static model of a system or subsystem by showing its object types and their relationships. Table 6.5 is a visual representation of a single class with a rectangular box divided into three sections:

- The top section is for class name.
- The middle section contains variables and constants in the form of var : type.
- The bottom section is for instance and static methods.

Table 6.5 A graphic representation of a single class

Circle
<u>- radius: double</u>
<u>- center: Point</u>
<u>~ numInitiations: int</u>
<u>~ PI : double</u>
+ getArea(): double
+ getCircumference(): double
+ setCenter(Point)
+ setRadius(double)
<u>~ count()</u>

Visibility is represented with symbols: + (for “public”), – (for “private”), # (for “protected”), and ~ (for package view). Static members are underlined.

Figure 6.3 depicts a more comprehensive class diagram about a hypothetical library system. In general, an object type can be represented with a stereotype name wrapped in angled brackets like <<interface>> or <<enumeration>>. Stereotype notations can also be used to indicate the nature of a type such as <<domain>>, <<entity>>, etc. We may choose to include none, some, or all members in a class box depending on our viewing interests. In Fig. 6.3, only selective members are included in each of the class boxes.

Relations are the most important part of a class diagram. Inheritance relation has two forms: subclassing (to inherit members) and interface realization (to inherit operations’ functional prototypes). A subclassing relation is denoted by an arrow with a closed arrowhead pointing to the superclass and a solid tail connecting a subclass (e.g., the inheritance relation between *Book* and *BookItem* in Fig. 6.3). The notation for interface realization is very similar except that the arrow tail is dotted (e.g., between *Catalog* and *Search* in Fig. 6.3). There are two other commonly used relations between object types: dependency and association. If an object type *A* depends on another object type *B* if some operations of *A* use operations of *B* to function. There are a variety of situations such as a method of *A* uses a parameter object of *B* or has a local instance of *B*, etc. In other words, *A*’s operations depend on *B*’s operations to function. Symbolically, a dependency relationship is denoted by a dotted arrow with an open arrowhead pointing to *B* (e.g., between *Librarian* and *Search* in Fig. 6.3). Dependency relation is also called “use” relation (e.g., *A* “uses” *B* to function). An association relation (denoted by a line segment) between two object types means one type has a member that is an instance of the other type (e.g., between *Book* and *Author* in Fig. 6.3). The association relation is like one between two data tables in a relational database. Thus, cardinality (one-to-one, one-to-many, or many-to-many) between two object types can also be used if helpful. A short description of the relation (like an author “writes” a book) can be attached to a relational line if we want to be more specific about the relation with a short arrow to indicate a “reading order” of the

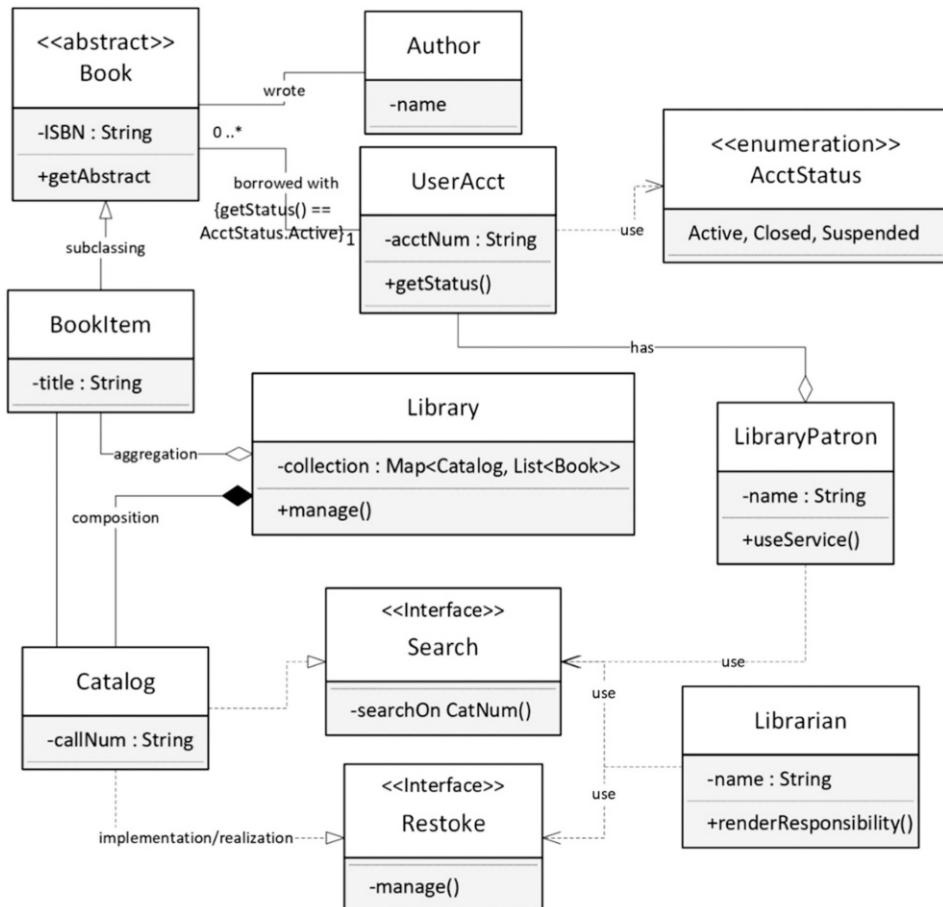


Fig. 6.3 Class diagram of a hypothetical library system

relation, pointing to the receiver end of the relation. However, unless it is necessary, such descriptions can clutter a diagram.

Often, association may refer to a whole-part relation (e.g., books are part of a library). Such an association is narrowly called “aggregation” indicated with an open diamond-shaped arrowhead. Furthermore, if aggregated objects cannot exist on their own (e.g., library catalog numbers must be associated with libraries), then instead, the relation is even more appropriately called “composition” with a filled diamond arrowhead. Finally, a constraint, included in curly braces {}, is a condition that every implementation must satisfy (e.g., a library account must be “active” for a patron to borrow books).

Despite the defined symbolic notations, construction of a class diagram can be rather liberal. Visibility symbols may be omitted altogether, as may a variable’s type. A “dependency” or an “association” relation can be simply recognized as “use” with a solid open

arrow pointing to the object type being “used.” Class diagrams are often used with certain focuses. For example, we might want to see a more complete diagram with all data fields and operations to understand better about structural entities. While omissions of attributes or methods are common in practice, use of defined symbolic notations is generally expected. To not clutter class diagrams, we may organize object types by subsystem, domain, theme, or service category and present them in separate diagrams. It is also helpful to keep related boxes close to avoid too many relational arrows crossing each other. If we use non-standard notations in class diagrams, we should explain their meanings with graphic legends or lookup tables and use them consistently.

6.4.2 Sequence Diagrams

UML diagrams are influenced by classical diagrams to various degrees. However, sequence diagrams appear to have the least influence from the past because of its pure object orientation nature. A sequence diagram shows a sequential flow of method invocations (or message passing if calling a method is understood as sending a message) that captures the dynamics of object interaction to accomplish an operational goal. Objects that exist at the time of the operation and participate in the operation are listed at the top of a sequence diagram. A sequence of method calls represented by arrows is drawn from top to bottom in a diagram, indicating the order of the calls. Figure 6.4 is a sequence diagram about a much-simplified course registration process to help with the illustration of the constructs explained below.

- Each object shown in the diagram is represented by a rectangular (object) box labeled with the instance name (optional) and a colon, followed by the name of an object type. Every object box has a dotted line (called a lifeline) with a possible “X” at the end of the line when the object no longer exists at the completion of the interaction. The latest version of the UML introduced three new object constructs. A boundary object (such as a data connection object) is represented by a circle with a left foot. A counterclockwise circular arrow represents a control object. For an object of any other kind, use a circle with a bottom foot. Whether it’s worth the switch to the new object constructs is a personal or organizational choice.
- A labeled horizontal arrow represents a method invocation (or originating object sending a message to a receiving object). An arrow with a filled head represents a synchronized invocation (meaning the call follows the sequence indicated). If an actor send a message “op_A” to an object of A, which in turn sends a message “op_B” to an object of B, the following code illustrates what that means:

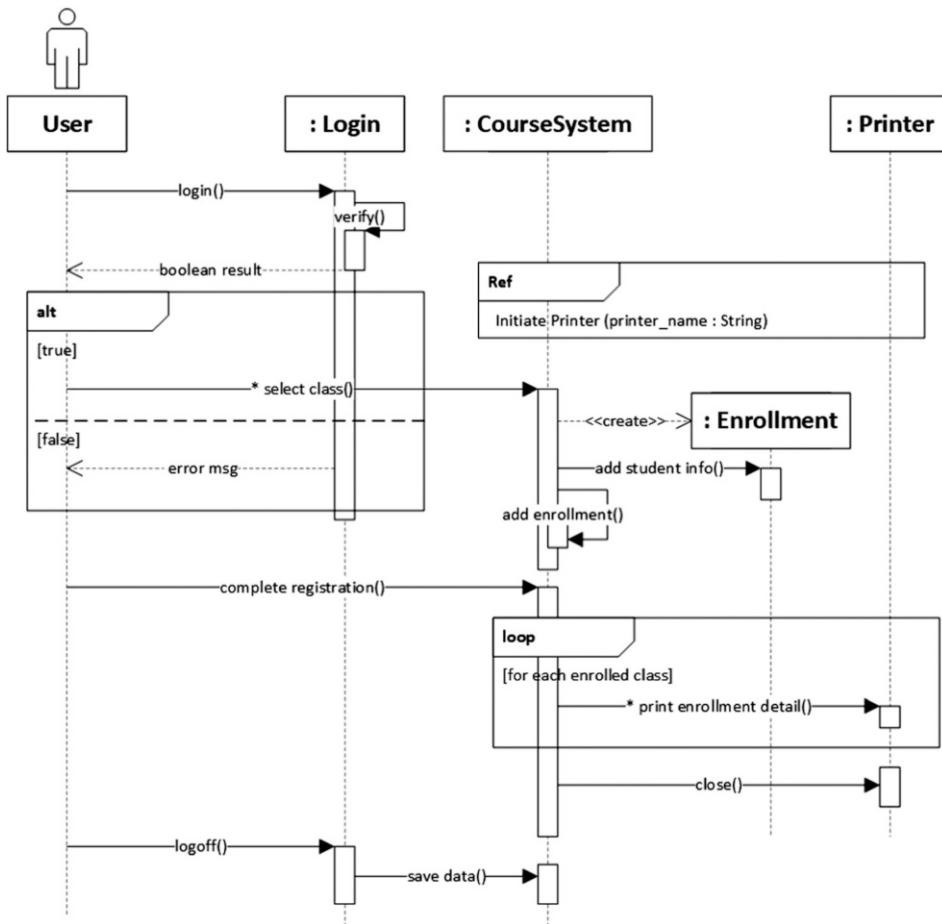


Fig. 6.4 A sequence diagram about class registration

```

class Actor{
    void main(){ //objects creation
        a.op_A(); }
    }
    class A{ //objects creation
        void op_A(){ b.op_B(); }
    }
    class B{ //objects creation
        void op_B(){ ... }
    }
}

```

An object at the receiving end of an arrow is responsible for the operation. An asynchronous call, executed at an unspecified time, is represented by an arrow with a stick head. A message arrow may point to an invocation box (an elongated opaque box) on an object lifeline. The length of an invocation box should be comfortable to accommodate all message arrows that may originate from the box (i.e., these are method calls made inside the method). It is common that activation boxes are omitted for less cluttered lifelines. A dashed arrow with stick head represents returned data (if an operation has a return statement) or simply the end of the call if we so wish.

- A dotted arrow pointing to an object box represents the creation of an object (with a stereotype <<create>> attached if we wish).
- When activation boxes overlap, it means an object invoking another method of its own (like operation *verify* in Fig. 6.4).
- A sequence diagram can be framed and labeled with a name (and parameters as appropriate) for easy reference.
- A message originated from (or returned to) an actor or a frame's left edge is called a "gate." A "gate" that starts or ends a sequence diagram is called an "entry gate" or "exit gate."
- A framed block with a label "ref" at the upper left corner represents a reference for an "interaction occurrence," which can be diagrammed elsewhere. The block covers the lifelines of the objects involved. In Fig. 6.4, there is some interaction between *CourseSystem* and *Printer* (but not represented in the diagram). If "ref" blocks are to be used, object boxes need to be carefully arranged for easy stretching over multiple objects' lifelines. It's also possible that a "ref" block contains some code context to add readability and clarity.
- Framed blocks are also used for conditional flows and loops. A conditional is labeled "alt" with bracketed conditions and alternative call sequences separated by dashed lines. An option block, labeled "opt," is applied to a situation appropriate for a "switch" statement. A loop block is labeled "loop" with bracketed loop conditions below the label. Messages in the block (i.e., in the loop body) are prefixed with symbol *, meaning there are one or more such messages. We can also prefix a message with * to mean repetition caused by other means such as an actor's will. For example, a student actor may select a class multiple times to register in Fig. 6.4.
- Newer UML versions have significantly enhanced the capabilities of sequence diagrams by introducing block constructs for grouping message flows. Blocks can be used to model interactions that include not only conditional branches and loops but also concurrency, parallelism, and asynchronous messages. In fact, we can use a block for any set of messages that we want to emphasize such as a block of messages with proxies of external entities or a block of mix-natured interactions. However, too many blocks can make a diagram look cluttered, affecting comprehension of message flows. Despite significant representation enhancement of more recent UML versions, people may be complacent about their existing and defined diagramming practices. Thus, modeling

using older versions of the UML is common, as is a mixed use of constructs from different versions.

- By default, messages in a sequence diagram are executed from top to bottom in the same process. However, with possible asynchronous messages and multiple blocks of different kinds, the order of message execution can be unclear. Thus, it is also common to number the messages in a sequence diagram to make the order of execution explicit.

Sequence diagrams are more complex than they might look. Incorrect sequence diagrams are simply misrepresentations of our mental models and can be detrimental to our modeling efforts. The most frequent mistake novices make is perhaps incorrect message originator or receiver. Originator sends a message to a receiver, who is responsible for invoking an operation related to the message (often, the message is the name of the operation). In other words, originator object depends on a receiver's method for its own operations. This understanding is consistent with our observations in real-world situations. For example, person A sends a message to person B for help; person B, as the receiver, is responsible for the implementation of the “help.”

Sequence diagrams are often more effective if they are used to describe a high-level operation such as an overall control of the system or a subsystem. At a more abstract level, messages may not necessarily represent the precise operations receivers are responsible for. Thus, invocation boxes can be omitted. More accuracies can be added when diagrams are subsequently refined if necessary. High-level sequence diagrams also use less, if any, framed blocks and are thus easier to comprehend and more helpful in architectural modeling.

A sequence diagram can be too cluttered to be unambiguously comprehensible if it includes too much low-level detail with multiple framed conditional and loop blocks involving multiple objects. However, even at a higher level, modeling interactions with code (or pseudo-code) can be easier and more accurate when accuracy is important. For example, to design an abstraction for a generic sorting process with two degrees of freedom, one for algorithm selection and the other for data pattern selection, we can model the abstraction directly in code with precise statements as follows:

```
AbstractSort as = new ConcreteSort();
as.setAlgorithm(new SelectionSort());
as.setData(DataFactory.generateTestData("random"));
as.sort();
as.displayTimeInMillisecond();
```

Figure 6.5 is an equivalent sequence diagram. Even though a sequence diagram is appropriate to use in this case, it is not straightforward to construct due to an extra cognitive load to process about the diagram steps in relation to the code statements we are more used to thinking about.

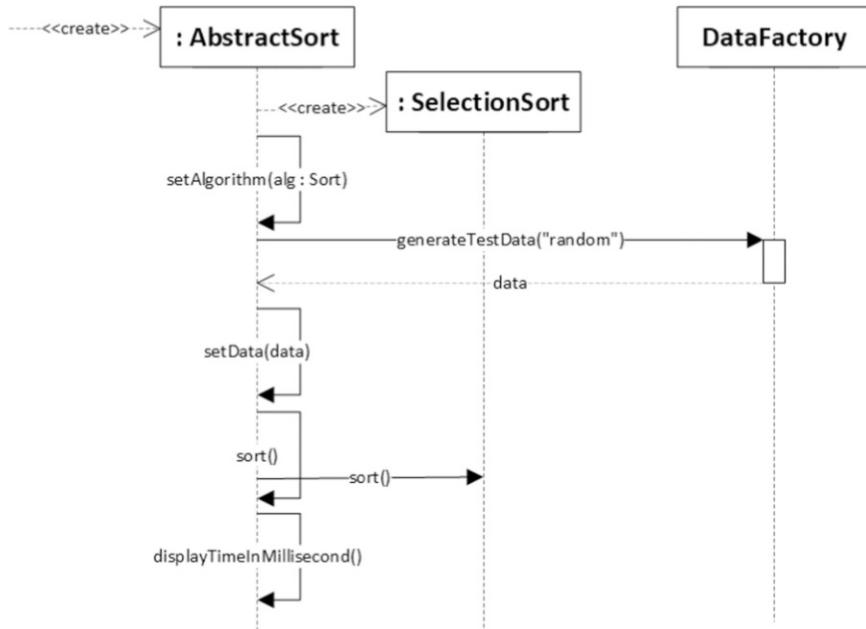


Fig. 6.5 Sequence diagram for a generic sorting process

Sequence diagrams may not be effective when used for modeling low-level details for multiple reasons:

- Because of the complexity of details, a sequence diagram can be easily imprecise even with the best effort.
- Details can change, making sequence diagrams obsolete easily if not updated in time.
- It is not consistent with the goal of software analysis—focusing on high-level software aspects that contribute to the formation of an architecture.
- For documentation purposes, reverse engineering tools can produce precise sequence diagrams based on existing code.

Studies have shown that sequence diagrams are most likely used to model entity interactions when they are extracted from requirements and use cases (Dobing & Parsons, 2006; Agarwal & Sinha, 2003). The following are other situations where sequence diagrams can also be useful.

- Document interactions of software entities in an existing (or a legacy) system. This documentation is useful to provide critical architectural information for further development or as a baseline for a new development of a similar product.

- Use sequence diagrams to communicate how a business process works by showing how various business objects interact. These business objects may not be software entities, but they work in much the same way in sequence diagrams when they interact and send messages to one another.
- Sequence diagrams work for modules too, where a module is a collection of independent methods. Modules interact when methods in one module use methods in another module.

6.4.3 State (Machine) Diagrams

UML state (machine) diagrams are based on classic finite state machines and state charts—both are powerful computation abstractions. To use a state machine diagram to its full potential of modeling power, there is conceptual complexity associated with constructions of useful state diagrams. Given that, it might be helpful if we look at state diagrams from historical, conceptual, and technical viewpoints.

A Historical Viewpoint

Computation can be seen as a sequence of data or information transformation (due to computing operations) from one (data or information) state to the next starting with data or information inputs and finishing with data or information outputs. Thus, a computing process can be viewed as a conceptual machine with a finite number of states. This machine has the following properties:

- A set of inputs (denoted by I)
- A set of outputs (denoted by Z)
- A set of defined states (denoted by Q)
- A state transition function that maps an event and a state in Q to another state in Q (in other words, this function defines a set of state transition criteria)
- An output function defined over I and Q with an output in Z

Figure 6.6 is a graphical representation of such a conceptual (finite) state machine, which computes whether an input string of 0s and 1s (i.e., a binary number string) contains even number of zeros. In other words, the output is a Boolean value *true* (if a string contains even number of 0s) or *false*. There are two defined states S_1 and S_2 . The transition criterion is that if a zero is read, a transition of state occurs (from S_1 to S_2 or from S_2 to S_1). When all characters are processed and the computing process ends at state S_1 , one gets the output *true*. In this sense, state S_1 is also called an accepting state (with a double circle). If the reading of input ends at state S_2 , the output would be *false*.

This is a state machine built specifically for the defined operation. Using the notations defined earlier, this state machine is represented as follows:

Fig. 6.6 A simple (finite) state machine

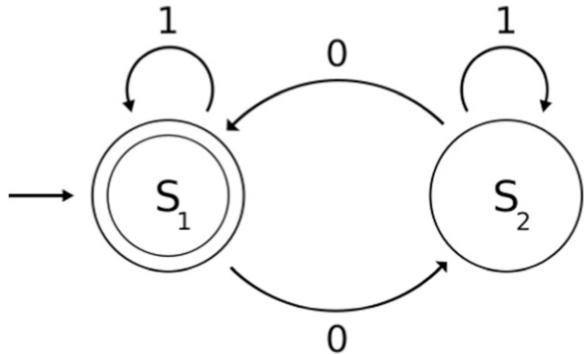


Table 6.6 State transition table of the example

state	event	transition criterion	new state
S_1	1	$t(1, S_1)$	S_1
S_1	0	$t(0, S_1)$	S_2
S_2	1	$t(1, S_2)$	S_2
S_2	0	$t(0, S_2)$	S_1

- Inputs $I = \{\text{strings of any combination of 0s and 1s}\}$
- Outputs $Z = \{\text{true}, \text{false}\}$
- Defined states $Q = \{S_1, S_2\}$
- A set of state transition criteria = {1. An input reading of 0 (an event) will cause a state transition. 2. An input reading of 1 (also an event) will cause the computation to remain at the current state.} (Thus, a state transition function t can be defined as $\{t(0, S_1) = S_2, t(1, S_1) = S_1, t(0, S_2) = S_1, t(1, S_2) = S_2\}$.)
- An output function $f = \text{true}$ if an input string contains even number of zeros or false otherwise based on the input and state transition criteria.

As an example, consider the evaluation of $f(1010) = \text{true}$. The following Table 6.6 describes the sequence of state transitions defined by this finite state machine, assuming reading of characters is from left to right (note that the initial state is S_1 because an empty string would be acceptable for output true):

In theory, any computing operation, simple or complex, that can be represented by a finite number of states can be represented by a finite state machine. Because of the assumption of a finite number of states, the model of finite state machines may have less computational power than some other models of computation such as the Turing machine. Finite state machines are studied in a more theoretical field of computing—automata theory.

Despite its power to model computing, a finite state machine requires the creation of distinct nodes for every valid combination of parameters that define a state and thus may

have a large number of states even for a relatively simple operation. State charts were developed by David Harel in the late 1980s (Harel, 1988) that provided more flexibility and hence addressed the weaknesses of finite state machines as a modeling tool. In a state chart, states can be nested (and hence hierarchical), thus allowing states to be distributed into layers of varying granularity. State charts are complex, however, with many defined language elements for events, conditions, and actions. This complexity is necessary for building a computing engine to translate a state chart into execution steps. Without the complexity of the language elements (if we are not building executable models), a state chart is, in fact, not much different from a UML state diagram. Figure 6.7 is a state chart about gas pump (software) operations, though the graphic constructs are more in line with those of a UML state diagram.

A Conceptual Viewpoint

UML state (machine) diagrams evolved from graphic representations of finite state machines and more directly from state charts. Like the case for classic state machines, software modelers define the states of a software operation or system. An object, when in a state, takes some action while waiting for an event and may remain in a state for a finite amount of time. A state has several properties:

- A state has a unique name (though an anonymous state may also be used as appropriate).
- Actions can be executed during a state.
- There may be self-transitions without causing a state change.
- A state may have substates with nested structures of sequential or concurrent of substates.

An object has its data states, which can often be used in a state diagram. We may use our intuition to identify an initial set of system states. As an example, consider an online banking application; we might see, as users, the states such as “being authenticated,” “selecting a task,” “transaction in process,” and “transaction completed and logging off.” Would these states be useful from the viewpoint of developing the software? They may or may not. The reason we want to use a state diagram in a modeling process is to view software operations not only in the context of a set of states but, perhaps more importantly, in the context of state transitions formed by system events, guard conditions, and resulting transitional actions (note that output is based on input and state transition criteria in a classic sense). Therefore, if we cannot observe a rich set of transition criteria or protocols, a state diagram is likely ineffective to use in modeling.

An object can be viewed as a “machine” on its own operating on its internal data states, which can be altered by instance operations. Operational conditions and external or internal events may trigger method calls, causing possible object’s state changes. Therefore, state diagrams can be useful in object design to understand roles operations play in state transition, as well as in maintaining a state. For example, to design an object that simulates engine operations, a state diagram can be useful modeling engine operations with a rich set

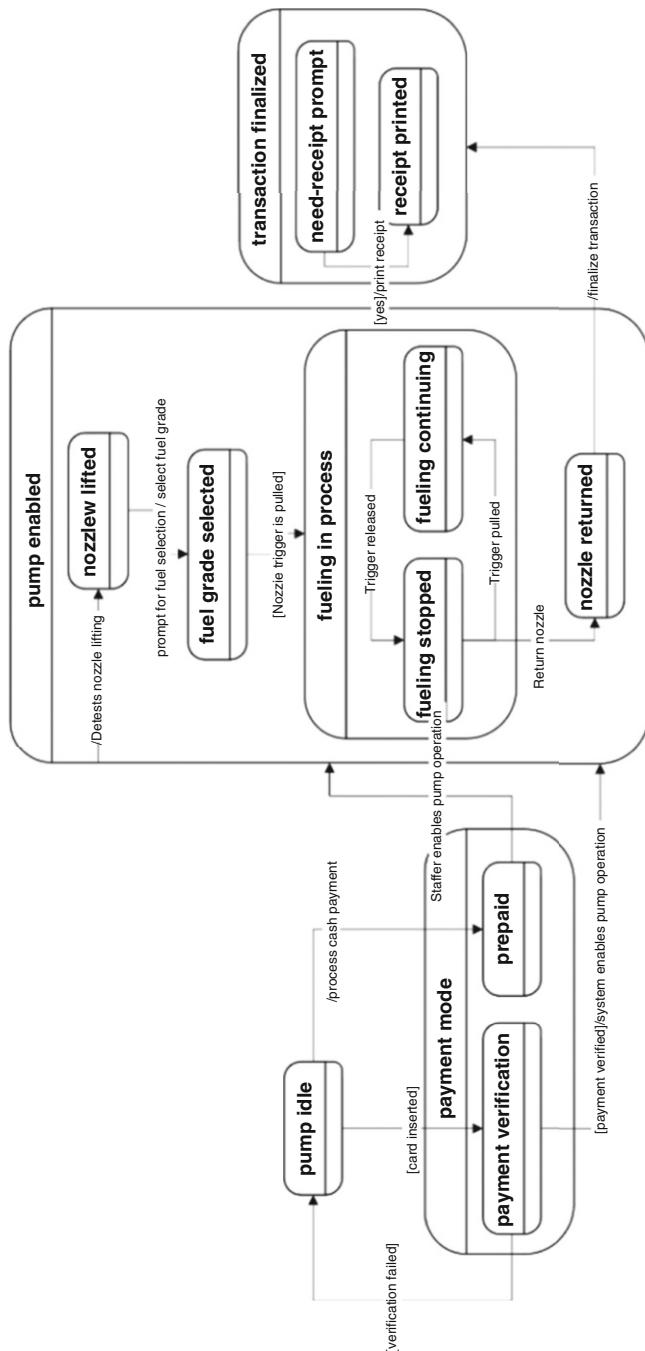


Fig. 6.7 A state chart for gas pump software operations

of (observable) operational states of an engine (like “idling,” “accelerating,” “running at constant speed,” etc.), trigger events (like “external resistance,” “external assistance force,” etc.), conditions (like “speed < max speed,” “running without additional engine power,” etc.), and transitional actions (like “apply break,” “accelerate,” etc.). Consider another object that models elevator operations. Observed states of an elevator may prompt us to define an enumeration type with values, such as *MOVING_UP*, *MOVING_DOWN*, *ARRIVED_AT_FLOOR_X* or *WAITING*, etc., to represent various operational states of an elevator. Events like pushing an elevator or floor button may cause an elevator’s state to transition. For example, when the elevator is requested at floor x , while in transition, the elevator may stop at certain floor before floor x as appropriate. Therefore, a state diagram can also be effective in modeling elevator operations.

In many applications, system states may not be explicitly observable. Consider modeling the process of a class enrollment. It might not be immediately clear whether a state diagram would be useful. Suppose these are the designed operations (possibly in different object types) related to enrollment:

- Add an enrollment.
- Withdraw from enrollment.
- Cancel a class.
- Reserve seats.
- Create (more) seats.
- Open a new section.
- Wait for a potential seat to open.
- Add enrollment from a waitlist.
- Update enrollment roster.

Except the last operation, all operations are related to operations of a waitlist entity we need to design and hence must correctly understand its behavior in relations to the operations above. Specifically, adding a student to or removing a student from a waitlist may depend on whether a class is still “open” or “closed.” Thus, a state diagram can be useful. For example, an enrollment object may have an instance variable *numSeats* and a constant like *ENROLLMENT_CAPACITY*. Thus, whether $\text{numSeats} < \text{ENROLLMENT_CAPACITY}$ would indicate if the class were still “open” or “closed.” A state diagram would include all operations (from all relevant object types) with a view of their roles in state transition, which help us understand the behavior of the system.

A set of states we define are often based on a “vantagepoint” we want to use to visualize the inner workings of a system’s operations. States are defined by the modeler to contextualize system’s operations, events, and global conditions. Consider elevator simulation again. An elevator’s physical movement defines its natural states—no move, moving up, or moving down—but these states alone may not be very helpful in understanding software operations. There are more useful states like “elevator is moving up towards floor x while a request is made on floor y ,” “elevator arrives at floor x ,” etc. Such states would allow

certain operations to be understood in a context such as “controller sends a message to elevator to stop at floor y if the stop is along the path or queue the request otherwise.” This context provides conditions and operations that constitute a transition to the next state. Consider another example about a bank’s ATM operation. An ATM may have these states that everyone can observe, “idle,” “in use,” or “out of service,” and be in one of these states at any given time. However, these states are switched without software intervention and thus have little value to observing the operational behavior of an ATM. To expose the system behavior of an ATM transaction, a more useful set of states might be “initiating transaction,” “in transaction,” and “completion of transaction.” However, another set of states can also be useful with a shifted vantagepoint: “verifying user information,” “processing a transaction,” “terminating a transaction,” and “completion of ATM use.” A different set of states is associated with a generally different set of state transition protocols, allowing operations to exhibit behaviors with a different operational context. This perhaps also explains why one state diagram can be more useful than another for modeling the same system.

Since analysis focuses on high-level system operations, we may start with some states that sound more abstract, like “credit card payment being validated.” Such a state may be termed a “composite state” as it may imply multiple more concrete substates, like “card information (being) validated,” “card credit limit (being) checked,” and “card use authorized.” Composite states can be more effective in modeling high-level software behavior. As needed, we can decompose a more abstract state into more concrete states and create more elaborated state diagrams. These sub-diagrams can also be further refined, forming a hierarchy of state diagrams to allow us to understand the system’s behavior at varied levels of granularity.

When visualized with a state diagram, operations can be divided into two kinds: those that are associated with state transition and those that do not. Operations of the latter kind are then executed at different times when a system is in a defined state:

1. Some operations might be executed on entry of a state. For example, elevator door opens on entry of the state “elevator arrived at floor x.”
2. Some operations might be executed during the state such as an elevator controller counting elapsed time for closing elevator door.
3. Some operations are executed on exit of the state such as an elevator controller determining the next state of the elevator.

However, a strict separation of the operations around possible execution times is not important. For the example of class enrollment earlier, the operation “update enrollment roster” should be executed in both “open” and “closed” states but is irrelevant to the exact time of an execution.

Fig. 6.8 A state box in a state machine diagram

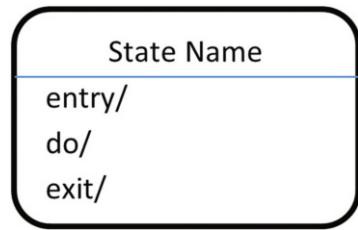
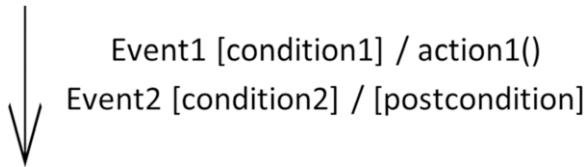


Fig. 6.9 Representation of a (protocol) transition



A Technical Viewpoint

Figure 6.8 illustrates a typical state box in a state diagram with two sections—a name section and an operation section to list operations invoked at appropriate times as discussed above. The initial state (a dummy state to start the software process) is represented by a (filled) circle. The final state (also a dummy state representing the exit of the process) is represented by a partially filled circle (or a double circle).

A transition (with a transition criterion or protocol) from one state to another is represented by an arrow. A state transition is generally associated with:

- Trigger events—Such events may include external events, call events, a passing of time, or a change in another state, though a transition may lack trigger events (or event triggers).
- (Guard) conditions (wrapped in brackets)—A guard condition is evaluated after a trigger event and is often represented by a Boolean expression.
- Actions (or postconditions, separated with slash symbol “/”)—An action is an executable computation that cannot be interrupted by an event. Actions may include method calls, checking availability status of another object, or sending messages to other objects. Often, an action is consequential due to trigger events and guard conditions; it is a transitional operation.

Figure 6.9 illustrates the visual representation of a transition. Here is an example of a (protocol) transition: *left-mouse down [mouse coordinates in active window] / link = select_link(coordinates); link.follow();*.

An effective way to model a system with state diagrams is to start with a few high-level composite states with a coarse state diagram and refine each of the states until all events, conditions, and operations are contextualized. Thus, state diagrams are often nested, composed, or layered. Figures 6.10 and 6.11 are state diagrams at different levels of

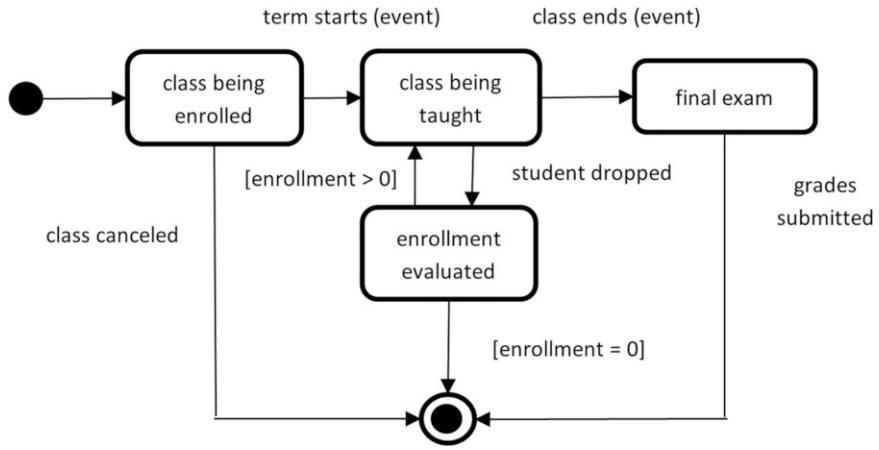


Fig. 6.10 A state diagram modeling course operations

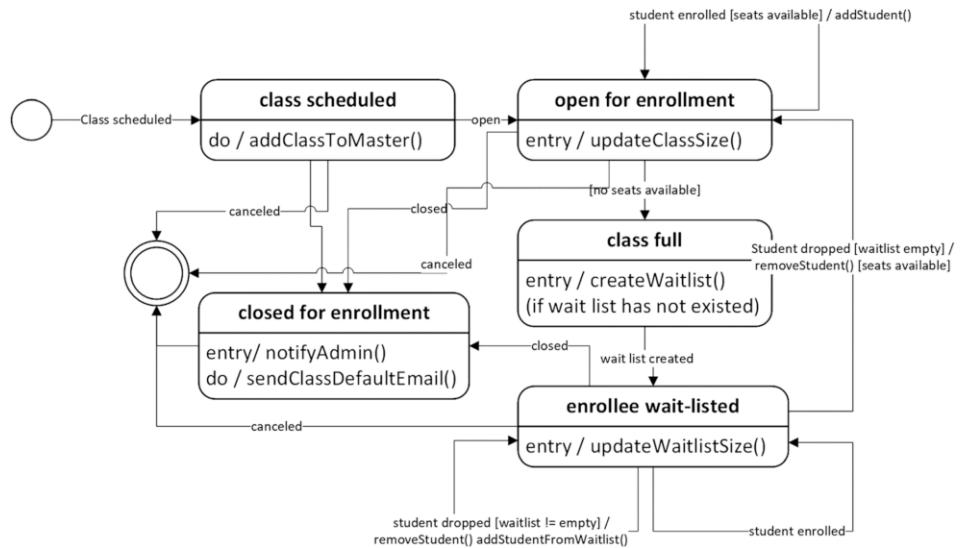


Fig. 6.11 A refinement of composite state: “class being enrolled”

granularity. The first diagram describes the course operation at a high level. The second diagram is a refinement of the state “class being enrolled.”

Complex state transitions can be tabulated, like Table 6.6, to allow better clarity. For example, if we eliminate the state “enrollee waitlisted” in the diagram Fig. 6.11, the transition protocols become more complex. Table 6.7 describes transition protocols and next states, assuming the current state is “open for enrollment”:

Table 6.7 State transitions as the result of combining “*class full*” and “*enrollee waitlisted*”

Event	Condition	Resulting action	Next state
<i>student enrolled</i>	<i>capacity reached</i>	<i>waitlistStudent()</i>	<i>class full</i>
<i>student dropped</i>	<i>waitlist not empty</i>	<i>removeStudent()</i> <i>addStudentFromWaitlist()</i>	<i>class full</i>
<i>student dropped</i>	<i>waitlist empty</i>	<i>removeStudent()</i>	<i>open for enrollment</i>

In general, the more elaborated the states are, the simpler the transition protocols are. Despite different choices of a set of states, states diagrams are all “equivalent” if they can all contextualize operations through transition protocols, though some can be much more effective than others in revealing an underlying architecture.

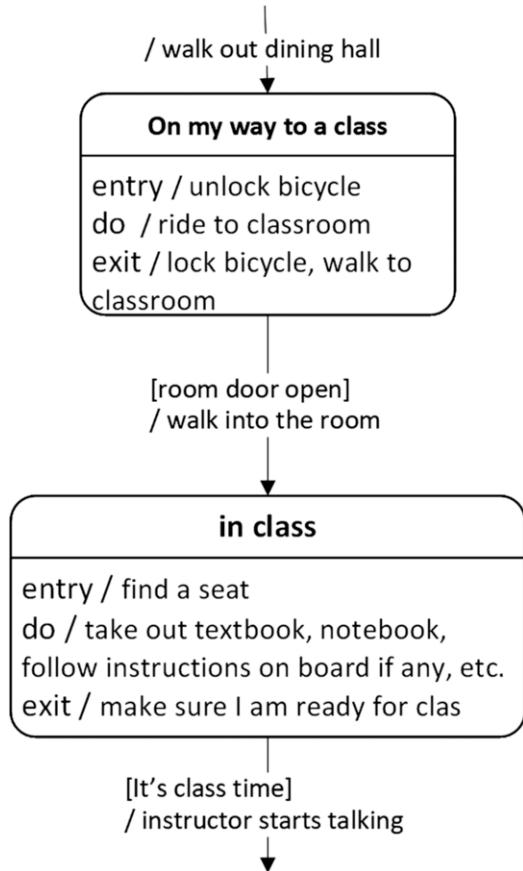
State diagrams are semantically simple and thus more flexible than finite state machines and classic state charts. However, a modeler must define a set of states and, more importantly, a transition function that maps “values” in a space of $\{\text{events}\} \times \{\text{conditions}\} \times \{\text{operations}\}$ (in the sense of Cartesian product) to $\{\text{states}\}$. Novice learners often miss state transitions altogether. A likely cause is to base states on a set of actions. For example, the process of approving a credit card use may be to collect card information, verify the information, verify the status, and authorize (or deny) the use. A novice may simply translate the actions into “credit card information collected,” “credit card information being verified,” “credit card status being validated,” and “credit card use authorized (or denied).” Asking these questions may help create “correct” and meaningful state diagrams:

- Does a state identified have a lasting effect or an enduring impact?
- What are the transition protocols between states?
- Would the diagram still make sense with small adjustments if you combine some states?
- Are there operations to put into state boxes?

The state “credit card information collected” would be appropriate if, for example, we imagine a separate window being activated (an event), required information being filled out properly by a user (a condition), and submitted (an action), which, in combination, would form an appropriate transition to the state. In addition, a possible operation to be executed while in the state is to verify if the name of the card owner is on the list of compromised credit cards before exit of the state. A set of states probably does not serve well for a useful state diagram if it cannot contextualize system operations in an operational environment.

A good practice on producing semantically sound state diagrams is to diagram one’s daily life around such states like “awake,” “ready to leave for breakfast,” “having breakfast,” “on my way to a class,” “in the class,” “having my class lesson,” “class finished,” etc. Figure 6.12 is a partial state diagram to demonstrate the use of these states. Can we describe our daily lives around a very different set of states like “it’s morning,” “it’s afternoon,” “it’s

Fig. 6.12 A partial state diagram about one's daily life



evening,” “it’s midnight,” etc.? Certainly. These states are more abstract, meaning the modeler might want to have a coarse view of daily lives, whereas the previous set of states might suggest that we desired to view our daily lives with greater granularity. States we design are always associated with a viewing intent (or a vantagepoint). It is thus not helpful to think about the “most appropriate” set of states to view software operations. Modeling with state diagrams can be powerful with its unique strength not shared by diagrams of other kinds, even though creating useful state diagrams is not straightforward.

6.4.4 Activity Diagrams

Activity diagrams model system behavior in terms of operation flows with constructs we are more familiar with. They are semantically much simpler than sequence and state diagrams yet still capable of modeling software processes of virtually any kind. The flow of an activity diagram may begin at the conclusion of an operation with the availability of

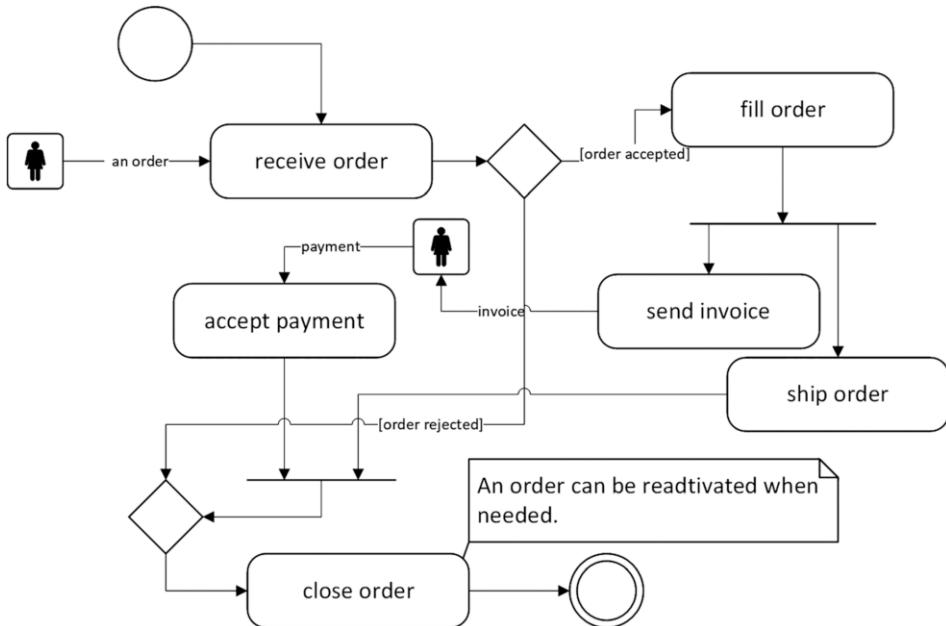


Fig. 6.13 An activity diagram about processing an order

certain objects and data or with some trigger events external to the flow. The constructs for activity diagrams include a few shapes connected with arrows (Fig. 6.13). They are:

- A (rounded) rectangular box represents an activity (i.e., an operation) and should be labeled using a verb phrase.
- An arrow indicates an order in which activities are executed.
- A diamond represents a diverge of an activity flow based on bracketed conditions or a merge of activity flows.
- A bar represents a split of a flow or a join of concurrent flows.
- A (filled) circle represents the start of a workflow.
- A double circle represents the end of a workflow.
- A note box could be attached to an operation if needed.
- An optional message (in terms of data or a condition) can be passed between activities.
- When appropriate, an actor can be used as a source or receiver of an activity.

Activity diagrams, in many ways, resemble classical flowcharts often used to depict algorithms if we view pseudo-code statements in flowcharts as activities. The difference is that a flowchart depicts a single algorithm that typically resides in a method, whereas an activity diagram generally shows a multi-method process. Compared with sequence and

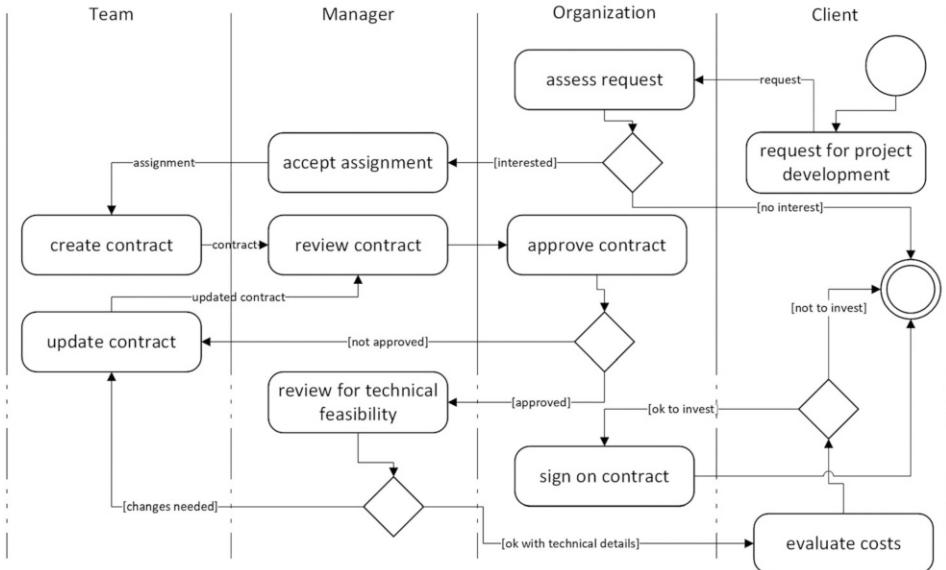


Fig. 6.14 An activity diagram with object “swim lanes”

state diagrams, activity diagrams are semantically simpler, thus easier to learn. Newer constructs such as a “fork” or “merge” bar, borrowed from classic Petri net (another mathematical abstraction that can be used to describe computing processes), have expanded the capability of activity diagrams to model computational processes of virtually any kind. Activity diagrams are generally used to describe the static behavior of a system or process. However, with introduction of “swim lanes” (vertical or horizontal strips each representing an object as illustrated in Fig. 6.14), an activity diagram can also show dynamic behavior of a process. When “swim lanes” are absent, all operations visible in a diagram may be independent or belong to a single object type. For example, all operations in Fig. 6.13 may be defined in an object type, say, *OrderManager*.

When transition protocols are missing, a state machine diagram looks much like an activity diagram. Novice learners can often be uncertain about the difference between the two. They may simply change the labeling of an activity box to make it sound like a state and use otherwise the same diagram as a state diagram. For example, a novice might simply change an activity “verify payment” to “payment being verified.” Learning state diagrams earlier might help in overcoming some of the confusion. Using analogies, an activity diagram is like “chaining” activities with “twists and turns,” whereas a state diagram creates “clusters” of activities around states. These are very different diagrams in terms of their structures, semantics, and modeling purposes.

6.5 Use of UML Diagrams

We provide some discussion in this section about miscellaneous aspects of using UML diagrams. The discussion is not based on any list of “frequently asked questions,” but it does reflect the difficulties and issues learners may encounter when learning the UML and diagrams of other kinds.

Diagramming rules can be imprecise, and individuals have varied interpretations of the rules. There can be gaps between our thoughts and their expressions in diagrams. This gap can be narrow or wide depending on modeler’s technical and cognitive skills. Consequently, modeling artifacts must receive timely updates to remain relevant and useful, especially when they are used in documentation. These updates are not only about the changes made in design but also about corrections of the mistakes made when diagrams were created. It is appropriate to say whether a diagram is helpful, but it might not to say whether a diagram is “incorrect” because there isn’t a “correct” one to check against. However, a diagram can incorrectly represent a mental model. Therefore, diagramming is also an iterative process to ensure an accurate expression of our modeling thoughts and evolved ideas.

As discussed earlier, diagrams are most helpful when they are used to model software operations at more abstract levels. People create diagrams primarily for the following reasons:

- To visualize modeling thoughts and ideas individually or in collaborative modeling sessions, where modeling artifacts were brainstormed, created, discussed, and forgotten after they have served the purpose
- To create aspect views for communicating to the stakeholders about the software architecture and some critical aspects of the system
- To be part of a software specification document

Often, informal uses of diagrams facilitate discussions in team modeling activities. For informal modeling, people may use a mix of formal graphic constructs and informal notations of their own choosing. If casual diagrams are to keep for future reference, they need to be reviewed for misrepresentations and reconstructed when necessary to minimize any inaccuracies such diagrams are often susceptible to. However, there are good reasons why formal uses of diagrams may also be necessary:

- Keep stakeholders informed with accurate information about the software.
- Provide accurate description about the architecture to facilitate software maintenance and evolution.
- Serve as updated baseline descriptions about the architecture as software prototypes evolve.
- There may be some critical low-level details that must be documented for future references.

There are other UML diagrams that are not as widely used as those covered in earlier sections. For example, UML communication diagrams use a free-form linkage between objects (as opposed to an ordered form of interactions used in sequence diagrams). To maintain the ordering of messages in such a free-form diagram, messages are labeled with chronological numbers. Objects communicating with each other tend to stay close, which can be more discernable than they would in a sequence diagram. Nonetheless, a communication diagram shows much of the same information as a sequence diagram does. It seems that people always find workarounds in their modeling practices when they occasionally encounter the limitations of the diagrams in their regular toolkit. With a couple of more diagrams that we will discuss later, the set covered in this book should usually serve well for our modeling needs.

We sometimes do encounter situations where a single kind of diagrams seems inadequate. Combined uses of UML diagrams of different kinds are appropriate and beneficial in a modeling process, especially when processes are asynchronous. Figure 6.15 is an

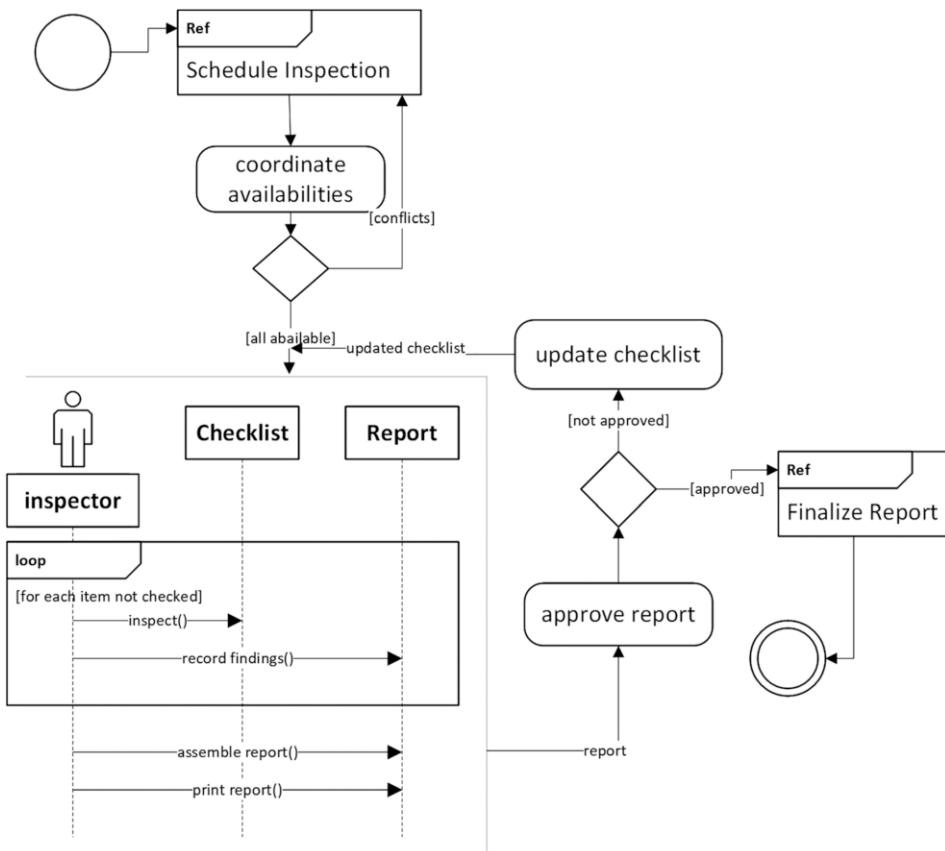


Fig. 6.15 Mixed use of UML diagrams

example of mixed use of diagrams to depict an inspection process that involves multiple sub-processes: scheduling an inspection, coordinating participation, the actual inspection process, post-inspection report approval, and finalizing the inspection. In this diagram, activity and sequence diagrams are used together, with some references to other (possibly asynchronous) processes to be modeled elsewhere. The processes in this diagram are asynchronous that could be modeled separately. But keeping the diagrams close with clearly drawn boundaries help better understand the operational connections of the processes (and more effectively validate the architecture). Processes and their refinements may also be described using diagrams of different kinds. For example, a high-level activity in an activity diagram might be elaborated in a state diagram, or a process in a high-level dataflow diagram (discussed in the next section) is detailed in a sequence diagram about how the data is transformed through a sequence of object interactions.

The kinds of diagrams you choose may affect the effectiveness of modeling to achieve modeling goals. The following heuristics may help with diagram selection in an initial modeling process:

- For object-oriented modeling, sequence diagrams are more common to use and particularly effective when there are multiple objects involved.
- The form of requirements may also affect the selection of a modeling tool:
 - Sequence diagrams are typically used when requirements are use-case driven.
 - Activity diagrams are likely more useful if requirements are process driven.
 - For data-oriented requirements, dataflow diagrams can be more beneficial.
- State diagrams are paradigm-neutral and effective when modeling with mixed paradigms. They are more appropriate to use when events and global conditions have significant impact on system's operations or when system states are explicitly observable.
- Despite its emphasis on object orientation, the UML was created to be a general-purpose modeling language, as many of its constructs have classical roots. With some necessary and appropriate adaptation, UML diagrams are suitable for modeling with any design paradigm or a mix of paradigms.

Finally, we reiterate that the UML is a tool for communication, not a piece of methodology that might turn a poorly designed software model into a better system. Diagrams that misrepresent our mental models can be detrimental to an analysis process. A diagram may be an aid to analytical thinking, but never a replacement. There is no “correlation” between the amount of diagramming work we do and the quality of a modeling effort. Using UML diagrams is neither necessary nor sufficient for a quality modeling process. The UML, when used appropriately, adds value to a development process in terms of effective communication with stakeholders, effective modeling collaboratively, and effective documentation that provides a critical guide to software maintenance and evolution.

6.6 Dataflow Diagrams

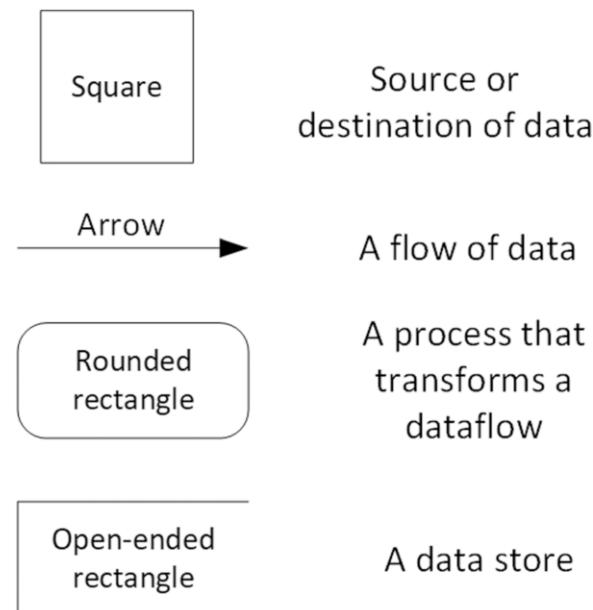
Dataflow diagram (or DFD) is perhaps the best-known classical diagram, which is still widely used today. DFDs are data oriented from a viewpoint of how data are transformed by the processes represented. Processes are connected by a “flow” of data. A process uses data in one form as input and produces an output possibly in a different data form, which in turn is used as an input to the next process. DFDs are easier to construct as they use only four visual constructs shown in Fig. 6.16:

- A rounded rectangle represents a process or an operation.
- An arrow indicates a dataflow (data or information passed between processes).
- An open box represents a data store such as a database, a data file, or some combination.
- A (double) square represents an external agent or entity—an actor or an external process that interacts with the system being modeled.

There are only a few simple rules for constructing a DFD.

- A dataflow always starts from an external source and ends with an external agent or entity or at a data store. In other words, a flow of data cannot start or end at a process.
- A process must have both data inflows and outflows. While it is common to have a single inflow with a single outflow, multiple inflows and outflows are possible.

Fig. 6.16 Constructs of a dataflow diagram



- All (arrowed) dataflows must be labeled with data using descriptive nouns only (not attached with events or conditions). Arrows are not to be left unlabeled.
- Process names are simple verb phrases.
- A data store must have at least one data inflow.
- A dataflow is not to connect between an external agent and a data store, between two external agents, or between two data stores unless there is a process in between. In other words, a data source or a data destination must be connected to a process.
- External agents and dataflows can be repeated connecting different processes to avoid lines crossing, but processes cannot.

Because DFDs are structurally simple with clearly defined construction rules, it is expected that diagrams be constructed with full compliance with the rules. Incomprehensible dataflows are common mistakes learners can make. For example, a process may have data inflows but not outflows, or vice versa. Or a data inflow and outflow appear to belong to different data streams. Since the focus of a DFD is on dataflows, processes in a DFD can be any “processes” that transform data, not necessarily in the forms of methods or functions. Traditional processes include various computational processes, processes for decision-making based on business rules, processes that split dataflows or perform data filtering to provide various data views, or processes that provide contexts of data transformation.

DFDs may look like activity diagrams with shared primary constructs of boxes and arrows, which may challenge the learners who are new to both. The primary differences are:

- DFDs use no notations for conditional branches. There are reasons for that. First, if all branches produce dataflows, they are likely represented by parallel dataflows. Second, DFDs are artifacts at more abstract levels where conditional branches are fewer, if any. Third, a Boolean data inflow or outflow is possible when appropriate.
- Arrows in a DFD indicate data transformed by the processes, and they must be labeled appropriately to add some comprehension to not only the dataflow but also the processes that transform the data. An arrow in an activity diagram is part of a control flow simply pointing to the next activity in the flow of execution. An arrow may be labeled (with a message of any nature) in an activity diagram if it helps explain the rationale for the next activity.
- A DFD is a decomposition of data transformation, whereas an activity diagram is a decomposition of activities/operations.
- Activities in an activity diagram are generally operations represented by methods, whereas processes in a DFD are processes of any nature.

Although DFDs and activity diagrams may be based on different notion of decomposition, a functional decomposition may lead to both dataflows and activity flows. Thus, it is

possible that one can construct a dataflow diagram based on an activity diagram with necessary tweaks when we want to shift the focus on data and its transformations.

To use DFDs in a modeling process, it is common to start a DFD in a high-level context. Thus, it is not unusual that the first process in a DFD is an entire system with initial dataflows showing the data passage between the system, data stores, and external entities. This initial diagram is then refined and elaborated to include more process details of the system until the data transformation is well understood or until processes are appropriately mapped to low-level operations. Software internal and external events could also trigger dataflows, and they can be linked to processes that either trigger or react to the events. Thus, keeping a list of events may facilitate the refinement of DFDs.

Figure 6.17 is a DFD about a library operation at an initial context level where subsystems simply exchange data with external actors or data stores. Figure 6.18 is a refined DFD for “process inter-library loan,” though some of the boxes can be further

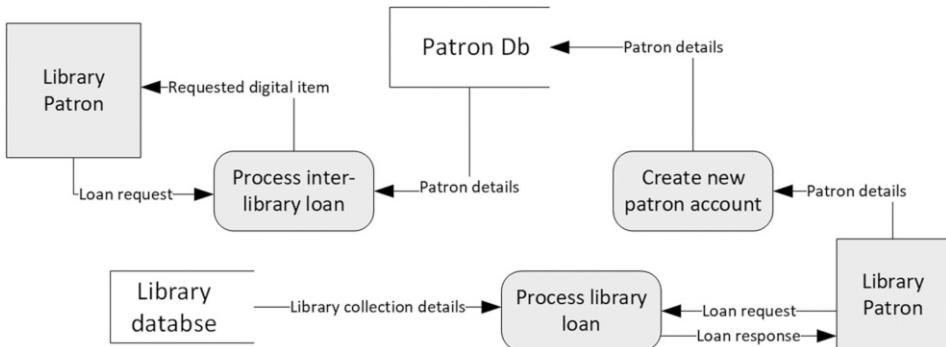


Fig. 6.17 A context-level dataflow diagram

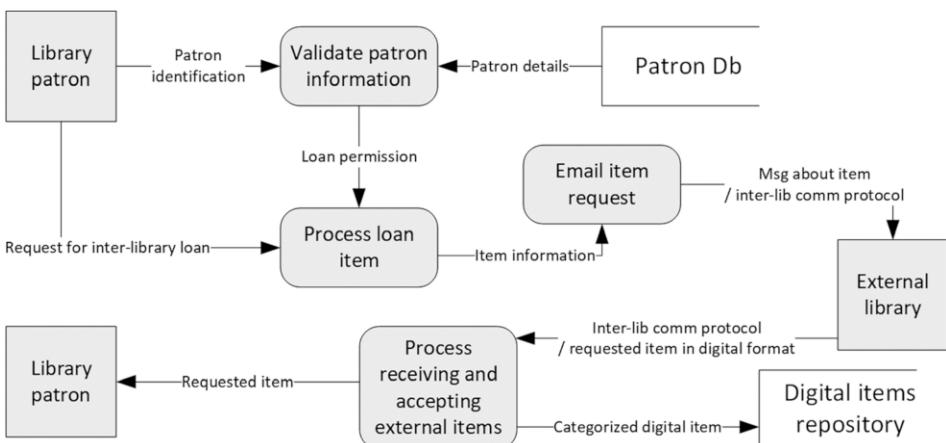


Fig. 6.18 A level-two dataflow diagram to refine “process inter-library loan”

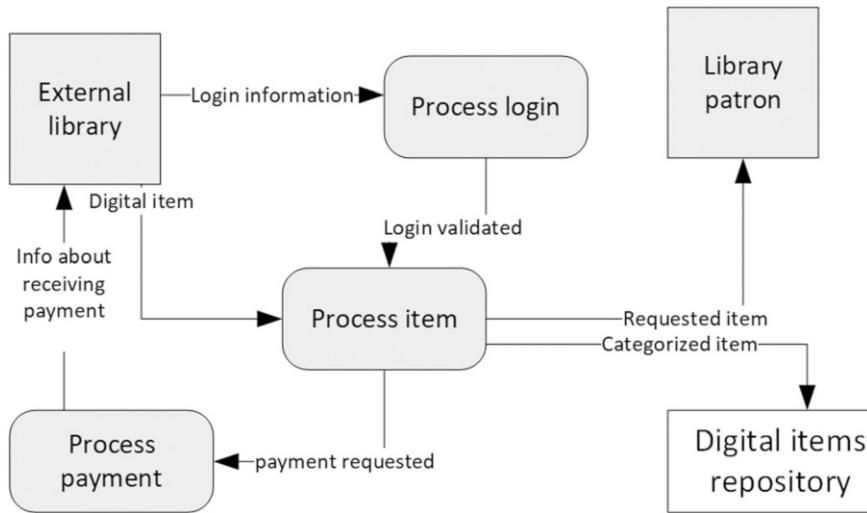


Fig. 6.19 A level-three dataflow diagram to refine “process receiving and accepting external items”

refined and elaborated as necessary. For example, Fig. 6.19 is a DFD that provides dataflows for “process receiving and accepting external items.” This process of refinement can continue until all processes are appropriately mapped to primitive operations. As desired, processes can also be numbered to indicate an order of execution.

Object interactions result in system behaviors, but not necessarily dataflows, especially when a system is interaction oriented. Popularized in the late 1970s, an era characterized by structured software design methodologies, DFDs appear to be “dated” when object-oriented methods are primarily used. However, if “processes” in a DFD are understood as any processes that rely on and produce information, DFDs can be applied to a variety of contexts. In the past, DFDs were used in describing dataflows in business processes. Today, DFDs are also appropriate for modeling web systems where pages can be viewed as processes, and user interactions lead data exchange between pages. Each page may have its sub-processes interacting with user or data from different sources, resulting in further elaborated DFDs. Figure 6.20 is an example about a web-based process of course registration at a relatively high level.

As software development is increasingly becoming more multi-paradigm oriented with systems that are rich in both data and user interaction, structural data modeling with DFDs along with UML diagrams modeling software behavior is becoming common in modeling practices. When processes in a DFD are instance methods, we might adopt “swim lanes” (used for activity diagrams) to contextualize a dataflow in terms of object interaction. Alternatively, we may use a modified process box shown in Fig. 6.21 with three sections for process name, a number indicating an order of process invocation, and object type the process is associated with. With the modification, a process can be either static or dynamic, improving the applicability of DFDs.

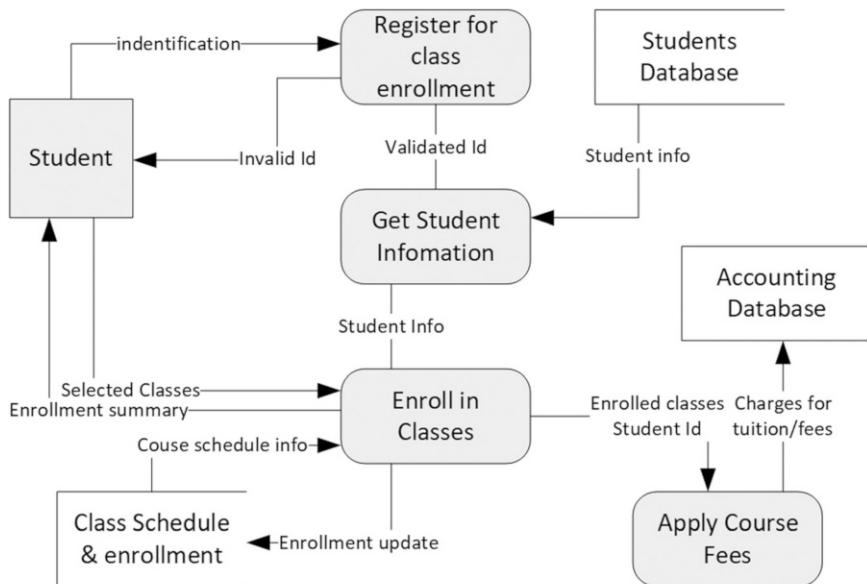
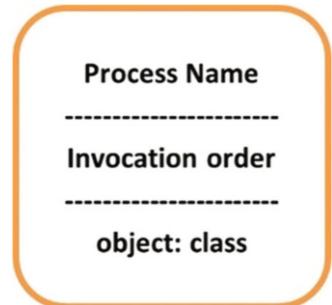


Fig. 6.20 A DFD about a web-based course registration process

Fig. 6.21 An alternative DFD
process box



DFDs were created and used when professionals were facing challenges of accurately understanding the processes involved in designing information systems. These processes tended to be complex, highly variant, and understood very differently. Design of an information system must not only capture objectively the process functionalities but also resolve the conflicts between business process design (based on business goals) and system's technical design based on technological constraints. DFDs were among the techniques that were introduced to reconcile, through dataflow analysis, between ambiguous understandings of system processes from business viewpoints and an information system's structural processes of data handling. Data we use today are very different from

they were half century ago in many ways, as are today's software processes that are increasingly cloud based. Challenges and conflicts between business processes and data handling are still with us today, though they may be taking different forms. As data has increasingly become integrated in business decision-making processes, DFDs will remain a viable modeling technique.

6.7 Modeling with Customized Diagrams

We can be rather liberal in diagramming if constructs we use are either self-explanatory or readily understood. Figure 6.22 is a diagram about card validation process before a user selects a service. The validation involves two stages with a card validation followed by a user validation. Some of the processors are conditionally accessible, while others are associated with events. After card information is provided, action “request PIN” collects an entry of a user PIN while validating the card. Card validation results in an event (or a condition) leading to the exit of the validation process. An event or a condition can be wrapped in brackets or appears in an action box as a precondition. Data, like “acct number” or “PIN,” is connected to processes with separate flows represented by dotted line segments. Data descriptions could be emphasized even with a different color.

This validation process can certainly be depicted with UML diagrams discussed earlier. When one wants to focus on an understanding about a process and all the events and conditions around it, customized diagrams can be appropriate with self-explanatory notations and constructs. Informal diagramming can be effective in a modeling session to facilitate communication about thoughts and ideas without being confined by formal

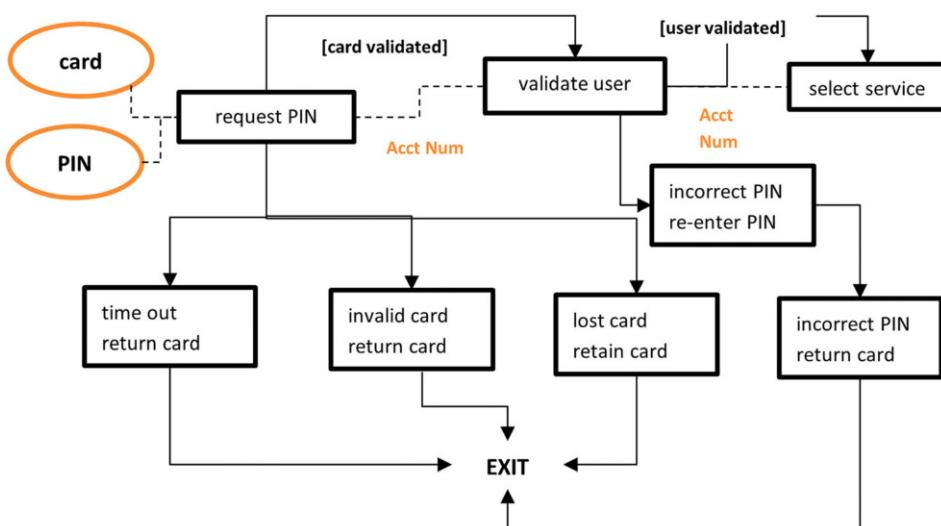


Fig. 6.22 A diagram of card validation process

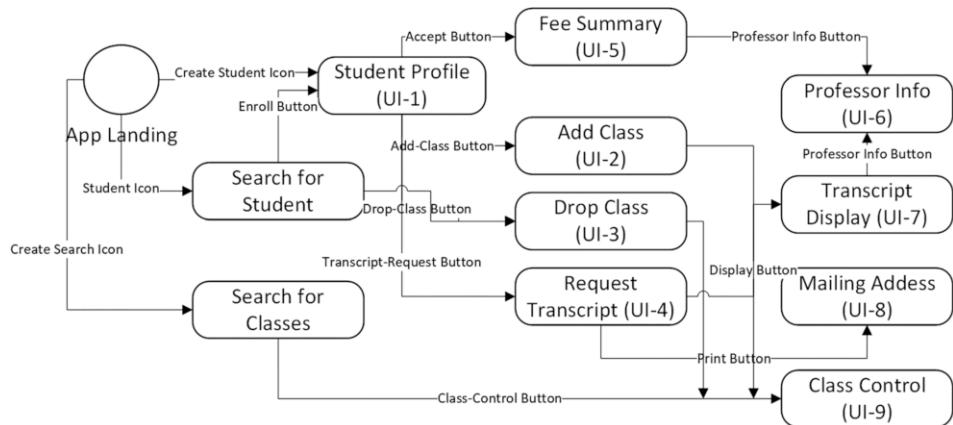


Fig. 6.23 A UI flow diagram

diagramming rules. However, if the artifacts are to keep and endure, informal notations, non-standard graphic representations, and ad hoc usages must be appropriately documented.

Software systems today are often interactive. Thus, software modeling may not be complete without user interface (UI) modeling. UI design, however, is a very different kind of design that involves design of not only interaction and visual presentation but, more importantly, an information architecture to help users understand where they are, what they've found, what's around, and what to expect. Although the UML and traditional diagrams were not designed for modeling UIs, some can be used to model interactions with appropriate modifications. For example, Fig. 6.23 is a UI interaction diagram constructed like a DFD with button flows replacing dataflows and UI boxes for process boxes. A UI box includes a reference number to indicate an activation sequence of the UIs involved. This diagram, with matches between UI boxes and the trigger buttons, provides a straightforward overview of the site features.

User experience (UX) design is about design of visual presentations based on an information architecture. UX modeling is purely user-focused, and detailed visual presentations are often important. Figure 6.24 is an example of UX design for a chat system. There are commercial or free software products to help facilitate the creation of UI interaction and UX models. UI/UX modeling may affect system modeling as software processes, data, and communications must be designed and structured appropriately to accommodate the realization of the UI/UX design. This is an example of “priority-focused modeling.” Similarly, a design can be prioritized on scalability, runtime efficiency, service orientation, etc. Priority-focused modeling would start with priority areas and expand outward to form a coherent whole of a design.

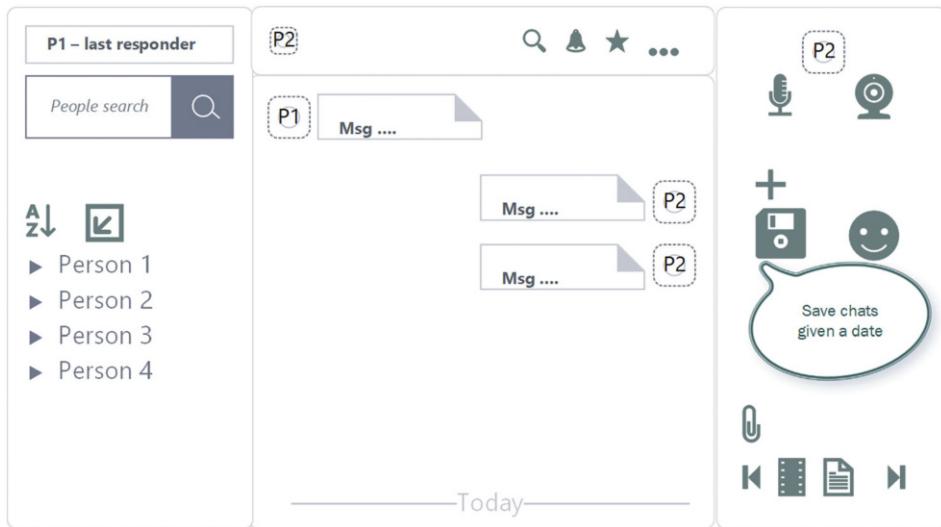


Fig. 6.24 A UX design for a chat system

6.8 Summary

Software analysis focuses on analyzing software requirements to design essential software elements of software models. Software modeling is a process that uses the outcomes of software analysis to elaborate a software model. We have drawn some distinction between analysis and modeling (which we consider it being a synthesis process). Such a distinction is, however, practically not important because modeling is a continuation of analysis and analysis is high-level modeling. We then discussed some modeling tools and, more importantly, a most useful set of UML diagrams.

Use-case analysis is an effective tool to support the analysis of software requirements. Use-case scenarios, detailing interactions between actors and the system, are an effective means of a use-case analysis. Use-case scenarios are used to identify software entities, their responsibilities, and their structural relations in an operational neighborhood. Use-case diagrams are used to provide overviews about system features and operations.

If any, class diagrams are the most likely UML artifacts we create to represent object types and their relations. Class diagrams offer effective static views about a system. For dynamic views of a system or process, we typically use sequence, activity, or state (machine) diagrams depending on the nature of software dynamics we want to visualize. Modeling artifacts are most useful when they describe high-level software activities, behaviors, or structures. Most of the discussions in this chapter about diagrams, however, were about correct and effective use of the diagrams, as diagrams can be of little value if they are not constructed to correctly represent our modeling thoughts.

Data-oriented visualization, represented primarily by dataflow diagrams, can work well in paradigm-neutral modeling, as can self-defined diagramming approaches. With some modifications, the diagrams introduced in this chapter may also be used for modeling user interfaces. However, user interface modeling may affect the design of other aspects of the software and must be proceeded with priority.

Software modeling in the real world is carried out in a variety of forms, with varied practices, and using modeling languages and tools oriented to a local development culture. In collaborative modeling practices, we represent visual models using, most likely, informal diagrams to model software features with a nimble, just-in-time set of modeling practices. Informal modeling artifacts may evolve with short development cycles, but they can also become obsolete in subsequent development cycles if not updated.

For documentation or communication with stakeholders, we use artifacts that are constructed formally to represent the design of the software accurately and consistently. Such artifacts must be kept current by all means. In more formal modeling practices, we use a fuller set of diagrams especially when software systems are complex that may require a balanced process model (such as the Unified Process Model). Formal modeling, however, must be accompanied with a reliable management process to update and evolve the artifacts to ensure they are of lasting value.

Exercises

1. Describe what a software model is based on your understanding.
2. Describe the relationships of software modeling with software requirements, software analysis, and software design, respectively.
3. Traditionally, software is modeled in three dimensions: data, process, and control. If object-oriented paradigm is primarily used, data and processes may become integrated to a certain extent. Are three dimensions still relevant to object-oriented modeling, and why? If not, what would be alternative dimensions to model in?
4. Modeling produces a software model. Would it be a complete model of the software? If not, to what extent can we still call it a software model?
5. Write a use-case narrative for each of the following software cases to capture user interaction with the system with one normal scenario, one alternate scenario, and one exception or error scenario:
 - (a) A user interacts with a stream service (such as Netflix, Disney Plus, Hulu, etc.) using a remote control.
 - (b) Fill gas tank at a gas station (to capture gas pump operation powered by software).
 - (c) Self-checkout at a grocery store.
 - (d) Self-check-in at an airport.
 - (e) Withdraw cash at an ATM.

6. For each of the cases in Question 5, describe the software process using:
 - (a) A sequence diagram
 - (b) An activity diagram
 - (c) A state machine diagram
7. The following instance method of class *GradeBook* computes the average score of the assignments a student completed. The implementation of the method *computeScore()* is polymorphic because assignments (like homework assignments, projects, presentations, etc.) require different ways to compute scores. Variable *classAssignmentRecords* is an instance of *Map* type.

```
public double getAverageScore(int studentId){  
    List<Assignment> assignments = classAssignmentRecords.get(studentId);  
    double total = 0;  
    for(int i = 0; i < assignments.size(); i++) total += assignments.get(i).computeScore();  
    return total / assignments.size();  
}
```

- (a) Based on your understanding of the code above, draw a class diagram that captures the design behind the code. The diagram should include all identifiable object types and their relations. The diagram should also include instance variables and methods identifiable in the code.
- (b) Construct a sequence diagram to describe the operation *getAverageScore*.
8. Construct a sequence diagram to describe the following method *processCheck* (code unrelated to the question may not be shown):

```
class Bank{  
    public void processCheck(Check theCheck, String acctNum){  
        Account acct = new Account(acctNum);  
        double amount = theCheck.getAmount();  
        double balance = acct.getBalance();  
        if( balance >= amount ) {  
            acct.addDebitTransaction(theCheck.getCheckNum(), amount)  
            acct.storePhotoOfCheck(theCheck);  
        }  
        else {  
            acct.addInsufficientFundFee();  
            acct.noteReturnedCheck(theCheck);  
            informCustomer(acct.getCustomerInfo());  
        }  
        fileCheck(theCheck);  
    }  
}
```

9. The following class is about processing orders of online purchases with only the details relevant to the question:

```

class OnlineOrderProcessor {
    private Map<String, Order> orders;
    private PaymentProcessor paymentHandler;
    private ShippingService delivery;
    public OnlineOrderProcessor(){
        paymentHandler = new PaymentProcessor();
        delivery = new ShippingService();
        orders = new HashMap<String, Order>();
    }
    public boolean processOrder(String orderId){
        Order order = orders.get(orderId);
        If( order != null ){
            paymentHandler.processPayment(order);
            if( order.isExpedited ) delivery.exprShipping(order);
            else delivery.regShipping(order);
            return true;
        }
        return false;
    }
    public void processReturn(String orderId, String kind){
        Order order = orders.get(orderId);
        if(order != null){
            if( kind.equals("damaged") ) processNoRefund(order);
            else if( kind.equals("past_return_window") )
                processPartialRefund(order);
            else processFullRefund(order);
        }
    }
}

```

- (a) Construct a class diagram to include all object types and their relations identifiable in the code.
 - (b) Construct both sequence and activity diagrams to describe the operation *processOrder*.
 - (c) Repeat part (b) for the method *processReturn*.
10. Suppose a piece of printer software has a class called *PrintJob* with the following information:

A set of methods:

{*readJobData()*, *submitJob*, *initiateJob()*, *displayJobdata()*, *checkConsistency()*, *computeJob()*,
placeJobOnJobQueue(), *saveTotalJobCost()*, *buildJob*, *saveWorkOrderNum()*}

A set of states:

{submitting job, building job data, forming job, computing job cost}

A set of state transition events:

{data input incomplete, delivery date accepted, job cost accepted, data input completed, job submitted}

A set of transitional conditions:

{all data items consistent, all authorizations acquired, customer is authorized}

A set of transitional actions:

{getDigitalSignature(), displayUserOption(), printJobEstimate(), printJobOrder()}

Construct a state machine diagram using the given information by placing the first set of methods in appropriate state boxes and identifying appropriate state transitions in the form of “event [guard condition] / transitional operation.” (The order of the items in the collections is arbitrary, not suggesting any order the items are to be used.)

11. Describe, using a state machine diagram, how to set time for a digital clock.
12. Pick a day of your recent memory, and describe how it went using:
 - (a) A state machine diagram
 - (b) An activity diagram
13. Use a dataflow diagram to describe the operation of a microwave.
14. Describe the software process of making an online purchase with an online retailer that you have experience with using:
 - (a) A state diagram
 - (b) A dataflow diagram
15. Think about a structured process (do some research if needed) such as:
 - (a) The process of purchasing a piece of real estate
 - (b) The process of a legislature bill reaching the president’s desk for signature
 - (c) The process of a trial that eventually reaches a verdict at a court

Then, use a diagram of any kind with constructs of your choosing to describe the process. The diagram should be as self-explanatory as possible.

References

- Agarwal, R., & Sinha, A. (2003, September). Object-oriented modeling with UML: A study of developers' perceptions. *Communications of the ACM*, 46(9), 248–256.
- Dobing, B., & Parsons, J. (2006, May). How UML is used. *Communications of the ACM*, 49(5), 109–113.
- Harel, D. (1988, May). On visual formalism. *Communications of the ACM*, 31(5), 514–530.

Further Reading

- Bell, A. (2004). Death by UML fever. *ACM QUEUE, March Issue*, 73–80.
- Christian, F., et al. (2006). In practice: UML software architecture and design description. *IEEE Software, March/April*, 40–46.
- Fowler, M. (2003). *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley.
- Jackson, D., & Rinard, M. (2000). *Software analysis: A road map* (pp. 133–145). Proceedings of 22nd International Conference on Software Engineering, Limerick, Ireland.
- Jorgensen, P. (2009). *Modeling software behavior – A craftsman's approach*. CRC Press.



Design of Larger Software Elements

7

7.1 Overview

Few software systems, if any, are built entirely from scratch. In other words, software systems, especially large ones, are always implemented, as much as they can be, using existing software elements, frameworks, and new elements yet to be developed with language's libraries and open-source code modules. Popular programming languages often have their own “ecosystems”—collections of libraries for generic use and for software applications in specific domains. These “ecosystems” have significantly improved our capacity of building software. As businesses are finding their business common grounds and identifying ways to standardize business practices across organizations, widely used enterprise resource planning (ERP) software systems can be built to address the common needs yet flexible enough to accommodate differences in business practices. ERP software systems are giant collections of software frameworks.

The truth is that software systems are often simply too big to be designed and developed in ways other than utilizing existing software elements of various kinds in a variety of ways and forms. Larger software elements that encapsulate changeable processes such as business rules and practices are typically replaceable elements so that a product can reach as a wide variety of users as possible. However, large-scale software reuse may become a casualty of ad hoc design methods and approaches, which often lead to elements that are difficult to reuse even in the same organization. Given this background, this chapter focuses on the design of larger software elements. We will first characterize larger software elements and then discuss about the principles and practices that are more oriented toward the design of larger software elements. Then, we will provide more elaborated discussions about the design of libraries, components, and frameworks. A brief introduction to microservices is given at the end. A large software element can be created using any combination of components, frameworks, services, as well as internal or external libraries.

Therefore, the discussion in this chapter is by no means complete and is pivotal toward the essential kinds of larger software elements used as “building blocks” in software construction.

7.2 Software Interfaces and APIs

The four kinds of larger software elements we focus on in this chapter are library elements, components, frameworks, and microservices, though the first three will be given more elaboration. A common characteristic among all larger software elements is that users utilize them as “black boxes” as their implementations are generally hidden. Users use the functionalities larger elements provide through their published interfaces. So before diving into the details, we need a clarification about the term “interface” we will be often using in our discussions. An interface could narrowly mean the programming construct that contains only abstract operations and constants. But an interface could also loosely mean a “communication medium” between a software element (say, a module) and use of the element, which may include a documentation about not only the technical details of the element but also instructions about the correct use of the element. A description about an element typically includes all the modules and data types involved in terms of what they do, the constraints to use them, and possible exceptions.

This term is given more nuances for a component element, however. Technically, a component has possibly two different kinds of interfaces:

- If a component provides certain functionality described in an interface, such an interface is either called a “provides” interface as the component provides it or a “provided” interface from a viewpoint of a user as the functionality is provided.
- A component may also require an interface to code against. For example, suppose a component has an instance method `saveData(ICustomer cust)` with an externally defined interface `ICustomer`. Thus, users of the component must provide an implementation of `ICustomer` before they can invoke the method. Similarly, the `TreeSet` class of the Java collections library requires an implementation of the interface `Comparable` if elements to be stored are not of a primitive type. Such an interface can be termed a “requires” interface as the component requires an implementation of the interface to function correctly or a “required” interface as a user is “required” to provide an implementation.

To be definitive, we will be using the term “provides interface” or “requires interface” for an interface a component either provides or requires (as explained above) in our discussions.

Components collaborate through their interfaces. For example, if component *A* requires an interface *Ia* to function and component *B* has the functionality adequate for implementing interface *Ia*, *A* collaborates with *B* either because *B* contains a direct

implementation of *Ia* or because a user of *A* provides an implementation of *Ia* using *B*'s interface *Ib* to access *B*'s functionality. In other words, for the latter case, a user would write some code to "glue" the two components.

We have used and will be using more often the acronym API (application programming interface) in our discussions. This acronym can be used in any appropriate context where we need to access and use external elements' functionalities. An API is for "programming" needs in the construction of an "application." Thus, an API can be more generally thought of as a contract of service between two programs. This contract defines how a client of an API should communicate with the API for "programming" needs. An API can also be viewed as a "catalog" of the services and operations it provides. This "catalog" is likely described in an API's documentation about specifications of the services and operations and their correct use. There are various kinds of APIs. For example, the OpenGL API (released in 1992) provides cross-language, cross-platform functions for hardware-accelerated rendering of two-dimensional and three-dimensional vector graphics. When we use certain language to construct software, we use the language's library API to support various programming activities. As software is increasingly cloud based, network communication APIs (such as SOAP APIs and REST APIs) and web-based service APIs are some of the popular APIs we use to facilitate web-based software applications. As APIs become more complex and sophisticated, owners of APIs may create necessary API management tools to help enterprise clients use the APIs to handle common tasks like user authentication, data analysis, and business management.

Clients can access an API using defined access protocols. For example, we use "import" statements to access Java library APIs. Publicly accessible elements of an API, like classes, interfaces, methods, etc., could be used as if they were defined locally. Use of APIs helps us simplify otherwise complex programming tasks. For example, webchat APIs allow us to develop chat applications with much reduced complexity. APIs not only help reduce the amount of code we need to create but also ensure implementation consistency across applications. Besides, we often also rely on APIs to gain access to hardware and software resources.

7.3 Characterization of Larger Software Elements

Older software systems were often one-of-a-kind systems with unique interfaces and implementations of the software elements. Modern software systems, in contrast, are families of systems consisting of many reusable software elements.

Component

Despite many possible definitions, it is widely accepted that a software component is a software element that conforms to a component model and can be independently deployed, replaceable, and composed according to a composition standard. Thus, by necessity, a software component has contractually specified interfaces (including all names,

parameters, and exceptions of the component operations) and explicitly stated contextual dependencies. By following well-defined component interfaces and their contextual dependencies, users can develop compatible components with customized functionalities. In particular, software vendors can develop composite components (i.e., components that are developed using other components) to facilitate component adoption. Vendors, as appropriate, may also be responsible for deploying independently the components they have developed.

A component can be as small as a module containing reusable code or as large as an entire system. Some components (such as those for data searching or sorting) can be used in a variety of contexts and composed without a particular order, while others can only be used for certain contexts and composed in particular ways (such as use of “order” and “shipping” components). Because components are replaceable, a components-based development creates a family of software products.

Framework

We have mentioned software framework a few times previously. A software framework is an abstraction with which software provides generic functionality of applications of a certain kind. As an abstraction, a framework provides implemented, yet generic, software operations by capturing operational commonalities across many similar application scenarios. User-written code is generally required to provide an application-specific use of a framework. A framework usually achieves controls of operations with abstract methods and interfaces, while users of the framework must provide their implementations. Recall an example mentioned earlier; a generic card-game framework might use an abstract class that contains a method that controls how a card game is played including shuffling and dealing the cards, starting a game, letting each player take turns to play, watching for winners, and ending the game when appropriate. Variable details are captured by abstract methods and interfaces. The control operation may be implemented against abstract methods such as *initiate game*, *play a round* (i.e., each player has played once), etc. and interfaces like *IPlayer* and *ICardDeck*. This control operation is like a “template” that can be applied to a variety of game scenarios with required details to be provided through the “placeholders”—the abstract methods and interfaces.

A framework can also be a software development platform (such as .NET or PHP) that includes support programs, compilers, code libraries, toolsets, and APIs that bring together platform components to enable and facilitate the development of software systems. In other words, such a development platform provides a generic control of program development, which is consistent with the general notion of a framework.

It might appear intuitive and plausible to apply a top-down approach to create “one-size-fits-all” software products. The history of software evolution, however, has suggested that the most effective frameworks turn out to be those that evolve from refactoring the common code of the enterprise. Software for accounting, human resource management, customer service, and educational learning management are some of the large software frameworks that have evolved from disparate software systems by identifying common

business operations, data-sharing attributes, and methods that share integral utility libraries and user interfaces. Clearly, frameworks also support the creation of families of software. A task-oriented framework (such as an animation framework that can be embedded in other applications) can be viewed as a component with customized code always expected.

Library

A library is a collection of software elements of non-volatile code, written in certain language, to be used in client programs with well-defined APIs. A large software library may include not only classes, interfaces, and implemented code modules but also configuration data, help documentation, message templates, etc. Typically, software libraries provide computing services, but users will determine how the services are used in their programs. Popular libraries typically include more advanced input and output features, commonly used data structures, math and statistics functions, data processing capabilities including data visualization, and implementations of algorithms of various kinds. Most compiled languages have a standard library. Supplementary libraries created by third parties constitute an “ecosystem” of the language, expanding the language’s capacities for software development. Libraries (and software services they offer) can be so extensive that most code in a software application could be written using the services from a standard library or from the language’s ecosystem with only some native “clue” code. When libraries are imported into code files, library modules would be called as if they were written locally. Thus, use of library code is independent among different programs.

A static library is one when library code is accessed when the hosting program is being built in a linking process to produce an executable binary image. In comparison, a dynamic library is loaded at runtime when library code is invoked only during program execution. However, library modules do not usually constitute larger “building blocks” with broader applicability. Customized code libraries can be helpful for a large software development undertaking such as the development of a software product line—a collection of software products. A customized code library can be viewed as a construction asset, along with other asset items like common requirements, common design, essential documentation, basic prototypes, components, and shared test cases.

Certain libraries might function like development frameworks such as a library for building graphic user interfaces that supports event-driven programming. Such a framework provides classes and methods for creating user interface widgets and abstractions of event models. However, such a framework is generally not an application framework in the sense that it offers various “puzzle pieces” but not a way to put some of the “pieces” together to make an application. In general, a library only provides code services, and users of a library determine how the code services are used in client code. If needed, library users generally can extend library elements in ways instructed.

Microservice

Microservices emerged in the wake of improving software maintainability and extendibility that traditional monolithic software applications lack. A monolithic

application is frequently a single-layered, self-contained software application in which the user interface, business logic, and data handling are combined into a single program using elements from a single platform. As such, a monolithic piece of software can be rigid to modify, difficult to scale, and slow to develop. Microservices promote an architectural approach to application development, which divides a large application into smaller, functional units capable of functioning and communicating independently to address the challenges and difficulties when applications are developed monolithically. Applications developed using microservices may have the characteristics of faster development, easier error detection and correction, better maintenance, and higher availability and scalability. Each microservice is designed to address an aspect or a function of the application such as logging, data search, external resource use, etc. Microservices can be separately developed internally or externally through defined service contracts. As a result, microservices can be combined in ways that allow solutions to larger complex problems to be formed relatively easily.

7.4 Design of a Library

Using software libraries has been perhaps the most popular form of code reuse since the computing pioneers attempted code reuse through saved punched cards or magnetic wire recordings—the early forms for storing computer code. We will first summarize the kinds of libraries people use to meet their needs for software development and then discuss library design principles and guidelines. Though there are high-level design principles that can be applied to the design of a library of any kind, specific guidelines may be needed for the design of libraries targeted at specific groups of users.

7.4.1 Software Libraries of Different Kinds

As mentioned earlier, a popular programming language often has its standard library to provide themed support for code writing. Typically, support categories of a library include data structures, input and output, common algorithms, concurrency, distributed computing, and various utility and tool functions. Component libraries are common too. Java’s graphic user interface component libraries *java.awt* and *javafx.swing* are popular for building event-driven applications. Another popular component library is React, a declarative, efficient, and flexible JavaScript library for building web-based user interfaces. React components implement a method *render* that takes input data and returns a displayable element using an XML-like syntax called JSX (which stands for JavaScript XML). Complex user interface React components can be composed from smaller and simpler components.

Algorithm-based libraries have helped advance the science and technology, as well as enterprise-level applications. As mentioned earlier, the high-performance graphics library OpenGL provides users with easy-to-use library routines for rendering two-dimensional

and three-dimensional graphics by hiding the low-level details that often deal with the underlying hardware architecture. The API provides all necessary functions (and named integer constants like GL_TEXTURE_2D) based on graphics pipeline—a conceptual model that describes a sequence of pixel transformations to render a desired graphic scene to a flat screen. The API abstracts away the underlying hardware graphics rendering mechanics and keeps programmers away from writing code to manipulate the graphics hardware accelerators. The API is implemented mostly in hardware to achieve high performance of graphics with many language bindings including C, Java, WebGL (more recent JavaScript binding for three-dimensional rendering from within a web browser), and bindings used on mobile platforms. For users without much graphics or multimedia programming experience, Java 3D (another graphics component library) provides APIs using a scene graph-based three-dimensional graphics model. It is a high-level, object-oriented view about three-dimensional graphics, in contrast to lower-level, procedural three-dimensional graphics APIs OpenGL provides.

Libraries can also be built to offer services. For example, Oracle’s cloud business has its management software called Enterprise Manager Cloud Control, which delivers centralized monitoring, administration, and lifecycle management functionality of an organization’s complete IT infrastructure, including systems running Oracle and non-Oracle technologies. The system has its own software library that enables clients to select Oracle-supplied entities and customize them or create client’s own custom entities. Once defined, these reusable entities can be referenced from a deployment procedure to automate operations (like patching, provisioning, and so on) in the deployment and maintenance of standard operating environments for databases, clusters, and user-defined software types.

Libraries are also critical for high-performance scientific computing. LAPACK (an acronym for Linear Algebra Package) is a standard software library, written in Fortran 90, for computations in numerical linear algebra. It provides routines for solving systems of linear equations, various matrix decompositions, and many other linear algebra numerical computations. LAPACK relies on an underlying implementation of Basic Linear Algebra Subprograms (known as BLAS) to provide efficient and portable computational building blocks for its routines. BLAS itself is also a library that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. BLAS implementations are often optimized for speed on a particular machine by taking advantage of special floating-point hardware that may support vector registers or other parallel computing architectures to gain substantial performance benefits. Apart from acting as a repository for software reuse, these libraries also play an important role in providing a knowledge base for specific computational science domains.

Substantial library support of a programming language may constitute a language’s ecosystem to support computation and software development in multiple application domains. Most noticeable example is perhaps the language Python with its own powerful syntax. With the library support for numerical computation, data visualization, and

machine learning algorithms, Python's ecosystem has been the most popular platform for applications in data analytics and artificial intelligence.

7.4.2 Characteristics, Benefits, and Risks of a Library

A frequent scenario of code reuse is a proprietary code library, as illustrated in Fig. 7.1, developed for a particular software project to handle essential low-level operations on fundamental business logic, data access, special authentication, and error handling, among others. An effective code library can be crucial for the long-term success of a software product. However, a good code library can be reasonably developed only when our experiences have supported a good set of requirements and use cases. Thus, for a completely new development, we might not have a good code support from a library until we have done some significant code refactoring that manifests a basis for a code library.

Design principles we discussed earlier still apply to the design of a library without much modification. However, libraries, not like other software products, have their basic characteristics that must be preserved in any library development. First, good software libraries are community standards and serve as ways for members of the community to communicate with one another technically. Therefore, a priority of a library design is to develop (or continue to develop) features and standards that can better serve the community's needs. Second, library routines and modules are context independent. For example, to design a component hierarchy in a component library, we only consider existing components the library already supports and potential use cases independent of

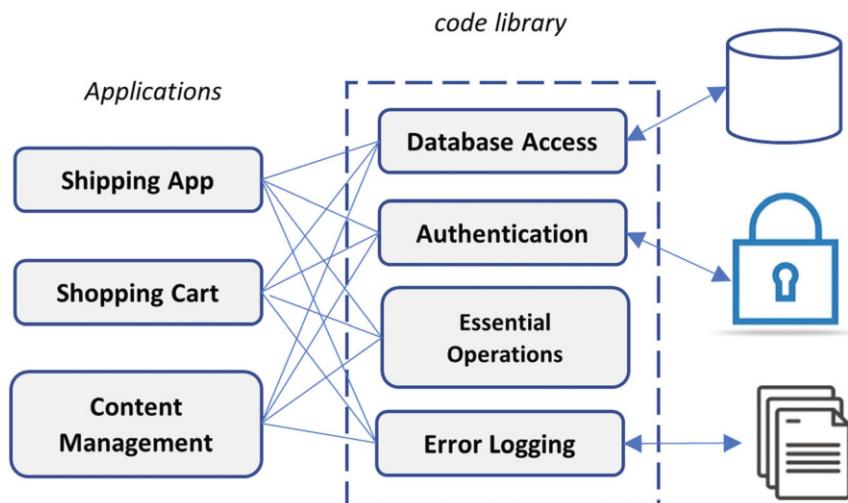


Fig. 7.1 Illustration of a code library

any specific applications. We design “requires” interfaces for a library only if implementations of the library use them in most general terms (like the Java’s *Comparator* interface). Since a library does not generally provide control mechanisms for external processes, abstract classes a library may use would be mostly for providing implementations that can be reused across supported concrete classes (like those in the Java collections library). Therefore, while applications may inspire the development of a library, capabilities of a library must be widely applicable, not biased toward any specific application domain. This is important for the stability of a library’s API and for seeking even near-optimal implementations. Third, library modules are generally concrete, highly cohesive, task oriented, and independent of each other in terms of their use (though they could be internally related in some ways). As a result, modules must have stable interfaces for their sustainability, though implementations can be modified, enhanced, or completely changed in subsequent library releases. In rare cases, modules could be deprecated if they no longer serve well the needs of the community. Such modules are not removed from the library, however, as the library must provide a continued support for all existing applications.

Code reuse is perhaps the most obvious benefit of using a library in software development. To effectively design a library, we also need a clear understanding about other potential benefits a library may bring:

- Reduction of regression errors (assuming library operations were tested thoroughly)
- Software maintenance becoming more manageable
- Better enforcement of standards across development teams
- Consistent module performances (using the same library support)
- Providing a basis for diversified uses to meet more specific needs (i.e., to create “library forks”)

Some of these benefits may provide additional motivation for a library, some may be used in a cost-benefit assessment, and some may even become impetus for changes in development focus or direction. Other benefits might emerge as use of a library has become part of a development daily routine.

Like any software endeavor, creating a library is a software development undertaking that has risks that we must fully assess before embracing it. Understanding the risks is as important as understanding the benefits. Some potential risks may have lasting impact on software quality if the library is not properly developed:

- Because of poor performances of library modules, users can be as much disadvantaged as they may benefit from a good-performing library.
- An unstable API, even with small changes, can cause widespread broken code.
- Use of an awkward library might require unexpected (often problematic) deviation from a design.

- A library was developed in a language that future development cannot use due to language differences. (Translating a successful library in a different language environment is always a problematic practice as disparities of the languages' constructs could easily cause mismatched behaviors in translated modules.)
- Library modules tied to volatile requirements may cause widespread troublesome coupling when requirements change in basic modules.

7.4.3 Initial Design of a Library

When we write certain code repeatedly, we may refactor the code by defining an appropriate module to share. But if such refactoring happens to some significant pieces of low-level code that has lasting stability and applicability, we might start to think about whether a library is warranted. Here are other considerations:

- Consider only justifiable generality and applicability with potential performance implications in mind.
- Consider people's needs from bottom-up (for initial set of requirements).
- Consider some code assets as the start point of a library with feedback on their usability.
- Consider independent testability of potential library modules.

But perhaps the most important consideration to start the development of a library is to ensure that a library's API must be stable in all possibility. Thus, to develop any library module, we need a mindset of getting it right in the first place with the following basic considerations for module sustainability:

- A module should be self-contained, rich in implementation details, and powerful enough to support faster completion of coding tasks.
- Design a module with extendibility in mind to minimize constraints and restrictions.
- Design a module with users in mind to maximize its applicability (in particular, taking advantages of powerful language constructs, such as generics, when appropriate).

Initial design activities for a library also include researching about appropriate range of possible themes, setting expectations for performances of the operations and memory usage, and prototyping the API. These are very different design considerations from those we discussed in early chapters because we must realize that design decisions that we make on a library are permanent. Library modules are not to be removed once they are put into use. Therefore, library design decisions must be applied more conservatively with abundance of precaution. These considerations may translate into the following practices:

- Solicit inputs and take them seriously.
- Gather requirements with healthy doses of doubt.

- Leave features out when doubts about their viability can't be resolved.
- Start with short specifications with a small but adequate API.
- Rather be more restrictive than based on unfounded generality.
- Expect to make mistakes, but always strive for minimizing the impact of the mistakes.

7.4.4 Design of a Library's API

Design of a library API has profound impact on a library's sustainability. The reason is simple. As library implementations are entirely hidden from users, an API can be liable for any confusion, incorrect use, or misuse of the library's functionalities. Thus, API documentation would be the first line of defense to avoid such liabilities. A documentation should include not only what an operation does and proper use of the parameters with effective examples but also the pre- and postconditions, units used (as appropriate), side effects, possible performance implications, and even design decisions. However, a good documentation cannot replace the clarity of an API given that users are not required to read the documentation carefully before they use the API. Therefore, an API should be self-explanatory to use and free from mysterious abbreviations and inconsistencies of any kind. For example, the operation *sublist(int fromIndex, int toIndex)* for any list-like container is self-explanatory for what it does if the index range of the extracted sub-list is consistent with the convention that the sub-list starts at *fromIndex* and ends at *toIndex-1*. A library's API design is a design process that may require extraordinarily delicate design decisions. Although discussions in Chaps. 4 and 5 still apply in principle, additional guidelines are necessary, including the following:

- Use the conventions of the language (such as its naming convention).
- Whenever appropriate, a library should maintain an operational symmetry. For example, if there is an *add* operation, there should also be a *remove* operation. If there is operation *search(collection, startIndex)*, there should be another operation *search(collection, endIndex)*.
- Modules should be easy to explain and amenable to functional splits and merges.
- Use “requires” interfaces whenever appropriate to always delegate specifics to customized client implementations.
- Provide reasonable defaults for attributes and parameters.
- Make objects immutable (when appropriate) to minimize potential issues, and design light object states when objects are mutable.
- Use short parameter lists whenever it is possible. However, for complex operations like plotting routines or machine learning methods, it is more likely that a library operation would consider all possible parameters the algorithm or a plotting process could use to meet an exhaustive variety of user needs. In such cases, reasonable defaults must be provided, and the characteristics of the parameters should be fully explained in the documentation with examples.

- Since performance-tuning parameters can leak implementation details into the API, design such parameters appropriately not to inhibit potential alteration of the implementations.
- Maximize information hiding to minimize chances of unintended access to implementation details.
- To avoid inappropriate client subclassing on concrete library classes, make them final.
- Avoid unexpected operational behaviors of all kinds. For example, throw exceptions in a library operation always with clear indication of the origin.
- Errors caused by client code should be made known at the earliest possibility, ideally during compile time.
- Provide overrides to `toString()` operation whenever appropriate.
- Avoid overloading using parameters with wide substitutability, such as `Object` or `Collection`; or to avoid ambiguity entirely, always require a different number of parameters for overloaded operations.
- Use primitive and interface types, when appropriate, for typing parameters and returned values of an operation to minimize possibilities of runtime type mismatches.
- For basic algorithms (like searching, sorting, making copies, checking equalities, etc.), a service operation with data of a primitive type is generally more efficient than using a more general approach with data of type `Object`.
- Always make exception handling optional, but potential exceptions known to the user.

To briefly summarize, because of the lasting stability, wide applicability, and expected powerfulness of a software library, the design of a library takes a very different approach with many delicate considerations. Frequent use of libraries improves our sense of what may constitute a good library. When we are puzzled by some of the library features or outcomes, we might be learning lessons and reflect our own practices. Frequent use of software libraries develops our sense of what a good library may mean and our own abilities to write them. It may also be very helpful to study about implementations of libraries to learn from.

7.5 Design of Components

Component design can be central to a development endeavor and have sustained impact on software evolution. To provide a relatively comprehensive coverage in this section, we start with related concepts first. We then will discuss component design principles. For component development, we will address component identification and composition. Finally, we will explain what a component-oriented architecture is about.

7.5.1 Component Concepts, Structures, and Models

Despite a formal description about “component” given earlier, we often use the term “component” loosely in programming. When we say a program has many components, we mean a program may have interfaces, classes, standalone static methods, containers of modules, etc. In other words, we may use the term to refer to any of the elements we can use independently in a program. But technically, a component must support not just use by one program, but many other programs too. In a stricter sense, a component is a “packaged” entity that programs can gain access to its functionality using a defined means. For example, we may place various code elements that share the same “theme” in a (Java) package. A program can “import” the package to gain access to the code elements.

A component is used as a “black box” (as implementation details are hidden from users) and thus must use interfaces to allow external synchronous access to its functionality (by other components or external systems). A component supports a family of software products in two ways:

- A component may provide different interfaces for different clients. Product variations use different “provides” interfaces to access business logic, services, or algorithms built into the component.
- A component may also use “requires” interfaces for its own operational needs, and product variations may provide different implementations of these interfaces.

In this sense, a component can be said to have a property of being “plug-compatible and interchangeable,” so long as component replacements satisfy contextual dependencies defined by the component. Thus, it also makes sense to use the notion of *component type*, a set of components that satisfy the same contractual interface and dependencies. This can add value to the formalism of component relations. For example, if c is a component member of a component type T and it uses two other components, we can simply denote $c / x : T_1, y : T_2 \mid : T$. This notion of component type is also helpful for better understanding component-based operations. For example, when a component c uses component x (of type T_1) and y (of type T_2), it means c uses components in the union type $T_1 \cup T_2$ for its implementation. In other words, c transforms operations defined by the components x and y (through their “provides” interfaces) into operations defined in c ’s “provides” interfaces. This is essentially how a component-oriented application works in principle.

The UML includes constructs for diagramming components and their interactions (Fig. 7.2). To represent a component, we use a rectangle with a smaller rectangular icon in the upper right corner to help distinguish it from a class. The name of a component is required, but a stereotype text is optional. A small ball (open or filled circle) refers to an interface that describes a set of operations the component provides for users to use its functionality. This interface, as explained earlier, is termed a “provides” interface. A socket (a half-circle) refers to an interface that the component is implemented against and hence requires an implementation (by client) for the component’s operations to work properly.

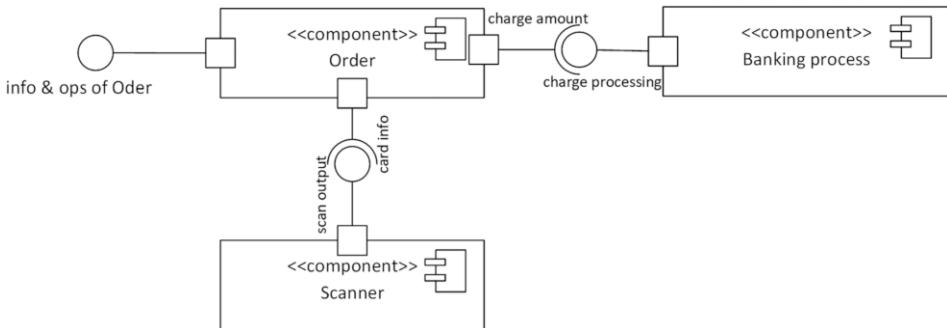


Fig. 7.2 A component diagram

This interface, as explained earlier also, is termed a “requires” interface. Users of a component must provide implementations to all “requires” interfaces of the component. To summarize, a ball and a socket, when they “meet,” refer to the same interface; the component on the ball side provides an implementation of the interface for the component on the socket side to consume. Components communicate, or they are composed, through interfaces.

In Fig. 7.2, the *Order* component may require two interfaces for its payment operations. Assume the two interfaces are as follows, one for obtaining customer’s payment information and the other for processing debit or credit payments:

```

interface IPaymentInfo{
    CardInfo getCardInfo();
}
interface IPaymentProcessing{
    boolean validatePaymentInfo(Info in);
    boolean chargePurchaseTotal(double amount);
}
    
```

This order component has no ability to implement these interfaces but must use the operations in its own implementation and thus “requires” clients to implement these interfaces. The *Scanner* component may have the functionality adequate for implementing method *getCardInfo()*. There are two situations, either the *Scanner* component has already implemented the interface *IPaymentInfo*, or a user of the order component writes an implementation using the functionality of the *Scanner* component to “glue” the two components together. Similarly, the *Banking* component either provides an implementation of *IPaymentProcessing* directly, or a user must do using the *Banking* component’s functionality. Meanwhile, the *Order* component has a “provides” interface for its own properties and operations to be used by other components or external processes such as a hypothetical one blow, where *Package* would be an internally defined interface.

```
interface Order{  
    void processShoppingCart(String cartId);  
    void processPayment();  
    Package getShippableOrder();  
}
```

A software layer may consist of one or more components, and a software system may be a composition with component layers. Figure 7.3 is an illustration of a component with its internal layers of components. However, this component (*Store*) could be, in turn, used by a larger system with multiple larger components (a square on an edge of a component box represents a port for exposing a “requires” or “provides” interface). The *Store* component defines its interfaces for external communications. In this example, *Customer* is an internal component with a “requires” interface *Account*, to be implemented by a user of the *Store* component. Components *Product* and *Customer* provide information and operations the *Order* component needs through their “provides” interfaces. The *Order* component has a “provides” interface for functionality it provides about “order entry” on behalf of the component *Store*. The two ports on the boundary of the *Store* component expose the two interfaces that describe the component’s contextual dependencies. These ports are like “gateways” for the *Store* component to communicate with external processes. An external process (i.e., a client of the component *Store*) must provide an implementation of the interface *Account* to use the functionality of *Store* defined in its “provides” interface. Finally, a dotted arrow in Fig. 7.3 is used to indicate a dependency relation of an exposed interface at a port and its origin. This dependency notation is necessary if a component has its own layered components. Ports are also used for components to communicate synchronously or asynchronously with each other or with input or output streams.

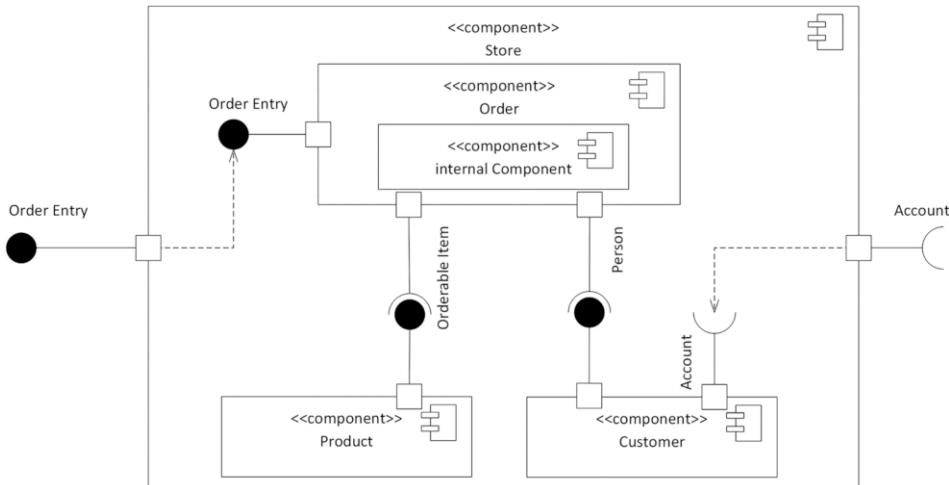


Fig. 7.3 A composite component

Component is also a form of a module. As a module, a component may consist of a set of collaborating classes, a set of standalone methods, or a mix of classes and independent methods. In other words, Fig. 7.3 can be zoomed in further to reveal each component's internal structures. New component can be created from existing components using various adaptation mechanisms. For example, Java's *javax.swing* components are extensions of those of *java.awt*.

The two interfaces exposed at the boundary ports of the component *Store* constitute the component's API. Consider another example of a component that provides an artificial intelligence service to predict the likelihood of a person developing heart disease in the future and provide health advice in multiple areas. Its API would be like a catalog of its service operations and resources it uses. This catalog specifies service operations like predicting developing heart disease in the next x years, advising for improved physical exercises or diet, etc. The catalog might also explain an interface the component uses that a user of the component must implement to provide personal but required health information. In contrast, if a component is about a process control such as a luggage check-in process as part of an airport check-in system, its API would communicate how the component can be used in conjunction with the required external systems such as a luggage scale, its setups, and possible ways to connect. Such an API can be technical and may require knowledge in hardware devices to fully understand. Like any API, a component's API should be generally stable. However, modification to component's internal structures and implementations for performance improvements and greater adoptability is appropriate and is part of component maintenance. Despite the fact that a component's API gives everything a user needs to know to use the component correctly, it gives no information or indication about the component's internal structures.

Before object-oriented component design became mainstream, components were understood mostly as subroutine libraries reusable in a broad range of scientific and engineering applications. Though these libraries provided well-defined, reusable algorithmic services, the application domains they served were limited. Contemporary components have much broader scope and service capabilities because they are developed with their own data structures, algorithms, and networking and data services. They are often developed with support from not only a powerful language but also its ecosystem. As a result, components of modern times are much more powerful and versatile, and their development can be streamlined based on consistent component models with the support of component infrastructures.

To facilitate component adaptation, we use the notion of component standardization to define component models. Established standards may provide guidance at any level of a component development. A standardization process may establish standards for component interfaces, documentation, composition, deployment, and component metadata (to contain information about the component, its use, its configuration, its customization, etc.), as shown in Fig. 7.4. For example, Enterprise Java Beans, .NET User Controls, and React components are different component models. These component models are defined for developing components under a specific software development platform.

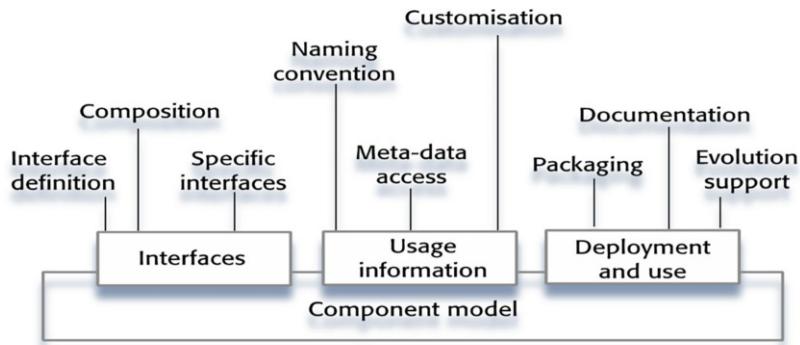


Fig. 7.4 Illustration of a component model

Unfortunately, because of the fundamental differences among these platforms, the component standards and usage assumptions are very different in these component models. As a result, components created under these models do not interoperate. Nonetheless, a well-defined component model is essential for third parties to provide support for the development and execution of components, as well as for component integration into a software development environment.

A component infrastructure is to provide support for larger endeavors of component development. It generally offers a range of standard services, even different compilation models, to facilitate a variety of component development scenarios. For example, Enterprise Java Beans and .NET are not only component models but also infrastructure frameworks that provide runtime environments for web-based software components. Such an infrastructure (.NET, e.g.) may also enable inter-process communication and interoperability of objects created with different programming languages.

When components are consumed remotely, a component infrastructure must also provide communication mechanism for component access over a network in some binary or textual form adhering to certain communication protocols. An interface description language can also be used in combination with a communication protocol to describe a component. For example, a web service (a form of software component) is transmitted over the Internet (i.e., HTTP) using Simple Object Access Protocol (SOAP) for exchanging XML-structured information. Listing 7.1 is the content of a file written in a Web Services Description Language (WSDL) to describe the service (which contains one operation that simply accepts a string “xxxx” and returns another string “Greeting, xxxx”). This file is an XML document in compliance with an XML Schema (to describe the structure of an XML document). A client program connecting to a web service can read the WSDL file to determine what operations are available with the remote component. Any special data types used are embedded in a WSDL file as well. The client can access the component using a local proxy object responsible for the communication with the remote component.

Listing 7.1 WSDL for a simple web service

```
<definitions name = "GreetingService"
  targetNamespace = "http://www.examples.com/wsdl/GreetingService.wsdl"
  xmlns = "http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns = "http://www.examples.com/wsdl/GreetingService.wsdl"
  xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

  <message name = "GreetRequest">
    <part name = "firstName" type = "xsd:string"/>
  </message>

  <message name = "GreetResponse">
    <part name = "greeting" type = "xsd:string"/>
  </message>

  <portType name = "Greeting_PortType">
    <operation name = "greet">
      <input message = "tns:GreetRequest"/>
      <output message = "tns:GreetResponse"/>
    </operation>
  </portType>

  <binding name = "Greeting_Binding" type = "tns:Greeting_PortType">
    <soap:binding style = "rpc" transport = "http://schemas.xmlsoap.org/soap/http" />
    <operation name = "greet">
      <soap:operation soapAction = "greet"/>
      <input>
        <soap:body encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/" namespace = "urn:examples:greetingservice" use = "encoded"/>
      </input>
      <output>
        <soap:body encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/" namespace = "urn:examples:greet" use = "encoded"/>
      </output>
    </operation>
  </binding>

  <service name = "Greeting_Service">
    <documentation>WSDL File for GreetingService</documentation>
    <port binding = "tns:Greet_Binding" name = "Greeting_Port">
      <soap:address location = "http://www.examples.com/Greet/" />
    </port>
  </service>
</definitions>
```

Component-based software development has fundamentally changed the landscape of software development. With the availability of an extensively wide variety of high-quality components, software reliability may also be fundamentally improved.

7.5.2 Component Design

A component design generally involves the design of data structures, algorithms, interface characteristics, and (internal and external) interaction mechanisms appropriate to a component. A component is a module, nonetheless. Thus, much of what we have discussed earlier about module design is still applicable to the design of components. For example, a control component is more likely to implement a high-level control process, and it should (according to the dependency inversion principle) depend on interfaces to connect to low-level details. Similarly, the essential cohesion and coupling principles remain applicable in component design:

- A component should be powerful, deep in functionality, and widely applicable, but remain focused. For example, it's appropriate for an order component to depend on multiple components for payment processing, shipping arrangements, and inventory updates, but it may not if the component offers operations for either shipping or inventory management.
- A component encapsulates only data and information that its operations directly depend on to render the functionality the component provides.
- A component may be a client of other components to reduce its complexity or strengthen the functionality it provides. But it is important to use only non-volatile functional resources.
- A component may extend its features or services, but only those that are closely related to the primary “theme” of the components.
- Powerfulness of a component doesn't mean a bloated component that would necessarily be more complex or resort more coupling than can be reasonably managed for its sustainability.

To ensure replaceability of components, the Liskov Substitution Principle may imply that if component *A* is replaceable by component *B*, then *B* must provide at least what *A* provides (but possibly more) and use no more than what *A* uses (but possibly less).

The term “topology” in mathematics refers to the study of geometric properties and spatial relations of mathematical objects that may be unaffected by certain geometric transformations. Objects we create become existent in a solution space of a given problem. When we impose relations among the objects and components we created, we tactically define a “topology” in a solution space. A system's state, which is a set of all objects' states, is a “geometric” property of the system's “topology.” The system's collective operations define the system transformation that, in turn, defines a relation between a state of the system and its image state under the system transformation. What is to expect between a system's state and its image state? We call this system transformation “structure preserving” if the relation between any system state and its image state under the system transformation is an equivalence relation. If we denote *s* to be any state of the system and *s_img* to be its image state, an equivalence relation between *s* and *s_img* means:

- The relation is self-reflexive, meaning there is a collective operation (i.e., a subset of all operations) I such that $I(s) = s$.
- The relation is symmetrical, i.e., there is collective operation INV such that $INV(s_img) = s$.
- The relation is transitive, which requires that if $A(s) = s_img$ and $B(s_img) = s_img_img$, then there is collective operation C such that $C(s) = s_img_img$.

Preservation of this equivalence relation is a characteristic for software reliability. When we design components and operations, we want to be mindful about “structure preserving.” For example, to maintain “transitivity” of the relation, if there is an *add* operation, there needs to be a version of *addAll* operation too. To maintain “symmetry” of the relation, there should be both *add* and *remove* operations, not just one or the other. It can be shown that if a supertype’s operations define an equivalence relation, so would its subtypes’ operations if the Liskov Substitution Principle is followed. In other words, we want to be consistent in defining objects’ behaviors and relations so that object’s operations define an equivalence relation between an objects’ state and its “image” state. If we do this for all object types and components we design, the resulting system operations would be likely “structure preserving.” It does not follow, however, that operations that may break this equivalence relation should be avoided. Code refactoring often introduces such “breaks” yet improves code in other areas or perspectives. When appropriate, design priorities may take precedence over the principles.

Operational symmetry may suggest a “design center” of the operations, with respect to which certain operations behave symmetrically. It is perhaps too abstract to define a “center” of a design in a conventional sense. But it might make sense, perhaps, to think of a “design center” as an operation, a component, or a set of entities that would determine the “fate” of everything else we design, for example, a card-game operation that controls game play, which would determine other object types we design accordingly. A “design center” is more likely a high-level unit, though it may not always be apparent what that “unit” might consist of. Furthermore, for a complex design, there could be multiple “design centers” at the top layer of a decomposition. Further decompositions may lead to smaller “design centers.” A component, especially a framework, is more likely a “design center” with “central” operations. There are multiple reasons for that:

- A component is large enough to cover a functional aspect of a decomposition.
- A component is often implemented against interfaces, which would be implemented in its “peripheral” environment.
- Component communications are more like a “center-to-center” interactions (possibly between different layers) than interactions of any other nature.

Functional, behavioral, or structural decompositions lead to component design decisions, as we design a component by drawing our attention to various operational focal points such as (layered) control, data handling, data structures, application domain,

interoperability with other systems, or connections with external systems and resources. These design focuses often illuminate “design centers.”

If this discussion about “structure preserving” and “design centers” can be called “component design in the large,” then “component design in the small” might be about the design of the internal structure of a component, for which a few design heuristics may help:

- An application often has its way or style for exception handling. Thus, exceptions a component can potentially throw might be better left for client to handle with complete information about exceptions in the component’s documentation. When a component is updated, its list of exceptions can be reduced or increased. Any changes to the existing exceptions, however, may have serious implications to client operations that use the component.
- To ensure component reusability, we want abstractions of a component model to be stable with hidden representation details. This may require good component requirements elicitation. As an example, for a learning management system, even though the fundamental operations involving students, class registration, course management, etc. are mostly consistent across college campuses, requirements elicitation focusing on the differences would be critical to developing stable abstractions associated with the common operations.
- There is a design tradeoff between reusability and usability. The more general the interfaces are, the greater the reusability there is, but the more complex and difficult it is to use. The cost for developing more general components may be higher than the cost for writing more specific equivalent software counterparts. Besides, generic components may be less space-efficient and have longer execution times than their problem-specific implementations.

Even though components are modules, we may need design practices that differentiate the design of components from the design of modules of other kinds. For example, the following design practices may be common in component design (but may not have been previously discussed):

- Remove application-specific methods.
- Name methods to make them more general.
- Add methods to broaden application coverage.
- Use abstract methods to capture the steps in an abstract process, applicable to a variety of application scenarios.
- Make exception handling consistent.
- Add configuration interfaces for easy component adoption.
- Reduce dependencies to an appropriate level that balances an object’s functional depth and its relational complexity.

Finally, the design of component access is somewhat unique to component design. Component access may take different forms. Some components can be accessed through static libraries, while others can only be retrieved through dynamically linked library at runtime. Remote component access, like web services we have seen earlier, often uses techniques such as object serialization or marshalling (the process of transforming the memory representation of an object into a data format suitable for storage or transmission) in some binary or textual form adhering to the standards of certain interface definition languages. Thus, the design of component access or delivery mechanisms must be appropriately determined, fully documented, and thoroughly tested with robust input validation capable of making user aware of potential issues.

7.5.3 Component Discovery

Component conceptualization is a process of components discovery and identification based on system's functional, behavioral, and structural decompositions. Components can be generally divided into three kinds:

- Control components for coordinating invocations of other components that exhibit how the system behaves
- Domain components for the implementation of the functional requirements
- Infrastructure components for supporting structural operations required in the solution domain

When we decompose a system into modules, modules like a login control, content navigation, data search, etc. may reside near the top of a decomposition hierarchy. A top-down process of component discovery in a decomposition analysis can be particularly appropriate for identifying major domain entities and control elements. This also means we can conceptualize and discover components early in an analysis-modeling process, including their relations, dependencies, and determining inter-component communication mechanisms such as global variables, shared memory, or communication protocols. For example, a learning management system may be naturally divided, from top-down, into large components for course management, class registration, class enrollment, grades reporting, student advising, etc., each addressing one aspect of learning in a broader perspective. A large component, like course management, can be further divided into relatively smaller components such as assignment, online discussion, grades calculation, documents management, and so on. In this decomposition process, a concept decomposition may facilitate the discovery of components. Figure 7.5 is a concept diagram about "course management." The diagram shows how this concept is connected to other (sub)-concepts in their respective ways.

Software requirements motivate analysis decompositions, thus directly contributing to the discovery and identification of larger components. Requirements are often the sources

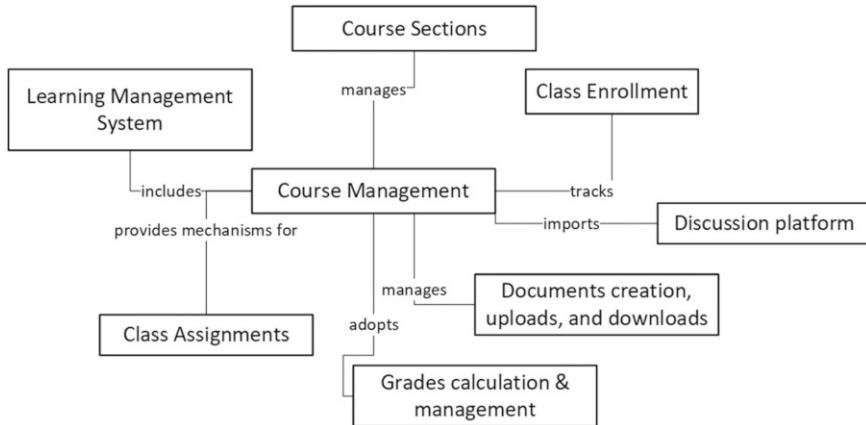


Fig. 7.5 Example of a concept diagram

for us to conceptually group classes, standalone methods, or potential components to satisfy the requirements. Use cases can also be a good source for finding components. Entities identifiable from a use case can be candidates for components or a component's internal elements. Operations identifiable in a use case can be candidate operations in a "provides" interface. In this top-down approach, "provides" interfaces are likely to be defined before "requires" interfaces, which may require further analysis on the nature of the details a component may need for its operations. This top-down process of finding components is again iterative. Each iteration would refine the component conceptions and re-evaluate the components' candidacies.

We may also take a bottom-up approach to component discovery. Such an approach may start with a module (a class or a collection of operations), which can be "elevated" to become a component. For example, a statistical analysis module may be "enlarged" to include additional features for data engineering, data validation, and statistical data visualization. A structural independence may also be a reason for creating components. For example, we might want certain events and their handlers to be managed by a particular communication mechanism and create a control component to do so. Components can also be created because of a consolidation or streamlining process. For example, a banking system may have fund transfers between accounts of different nature. While transfers can be implemented at local levels specific to the kinds of accounts involved with varied bank policies and procedures, it would be helpful to streamline the software process by developing a component for transferring funds. General processes like bank ledger calculation, audit trail update, account transaction, and transaction reporting can be implemented against abstract operations (in the "requires" interfaces) to be implemented with the concrete policies and procedures. It is also possible that modules that address specific concerns (for security, communication, logging, profiling, configuration, etc.) might grow into larger components to provide better-defined services.

Another source for creating effective components is a set of difficult decisions (which we want to postpone making them) such as handling software inter-communications with external systems of heterogeneous types or a set of decisions that are likely to change over time such as data representation, business processes, business policies, algorithmic details, and input and output mechanisms and formats. We can create and construct components to hide such decisions. A design process of creating such decision-based components is likely also iterative:

- Start a design with decisions that have global impact and then with those that are most likely to change.
- Design dependency hierarchies with support modules based on expected functionalities of the components.
- Evaluate the progress and decisions yet to be considered.
- Start a new iteration with the decisions left until all decisions are hidden in components and all designed components are easy to understand and reasonably independent to implement.

Components developed for a specific application can also be generalized with a broader scope. Thus, it is beneficial that we design the original components with extendibility in mind. However, component extension may also need reusability study that includes possible component feature extensions, their structural feasibility, and reuse design strategies.

In summary, there are many ways components can be discovered, identified, and developed. However, component development or extension is not simply a design effort; it often requires a collaborative effort with collective ideas. It also requires early testing to validate the design and the assumed functionality. As components are often designed to satisfy software requirements, component testing can be integrated into a requirement validation process. As development continues, updates to existing components or addition of new components may have architectural implications. However, when elements are structured into logical layers (for process control, business model implementation, data handling, tooling, networking, etc.), structural impact due to continued component development may be significantly reduced. Besides, layered component organization may also help component evolution along the axes of the layers and in better understanding of the roles that components play in the architecture.

7.5.4 Component Composition

Component composition is the process of assembling the components, according to their respective interfaces, to create certain combined functionalities of a system. Some “glue” code is generally expected in a composition process. For component-based software

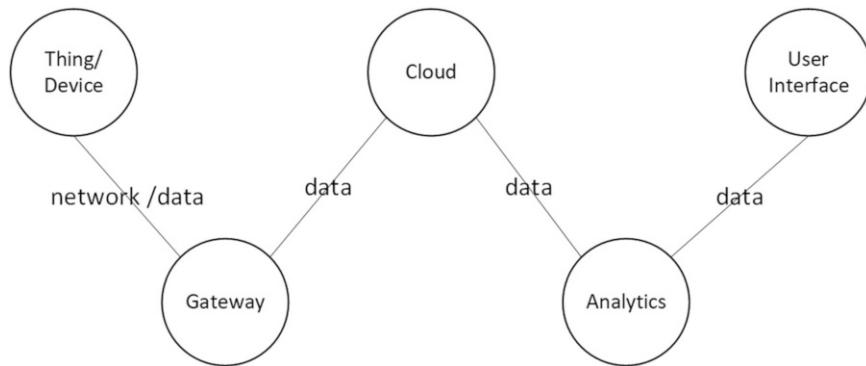


Fig. 7.6 Example of component composition (IoT)

development, component composition is a basic mechanism within a component infrastructure.

If $a: T$ and $b: T$ are two components we use to form a composition, then only the composition $a[b]$ or $b[a]$ may be valid. However, if both are meaningful and contextually consistent, we say the composition is symmetrical. For example, consider a search operation to filter first-year students who have taken AP Computer Science exams. It doesn't matter whether we first filter first-year students (with component a) and then those AP CS exam takers (with component b) or all AP CS exam takers and then first-year students among the takers. Both compositions, however, depend on the availability of whole student population of a targeted domain.

Figure 7.6 is an example of component composition in an application of Internet of Things (IoT). Components of the device connectivity layer involve smart sensors to continuously collect data from the environment and transmit the information to the next layer through a wireless or another kind of network. An IoT gateway component layer manages the bidirectional data traffic between different networks and protocols. An IoT cloud layer offers tools to collect, process, manage, and store a large amount of data in real time (which uses its own distributed data management components of IoT cloud data services). An analytics component layer consumes data from the cloud and turns it into insights to provide solutions for IoT system improvement. A component layer of user interfaces is tangible to users of IoT applications. The composition of these components must follow an order defined in each of the component layers.

For component composition, communication protocols must be defined between components and between component layers including, when appropriate, data format and information about component deployment and operational environment. Component communication protocols define how components interact, which may take a variety of forms such as (synchronous or asynchronous) method invocation, message-driven interaction, data stream communication, or other protocol-specific interactions. These protocols are

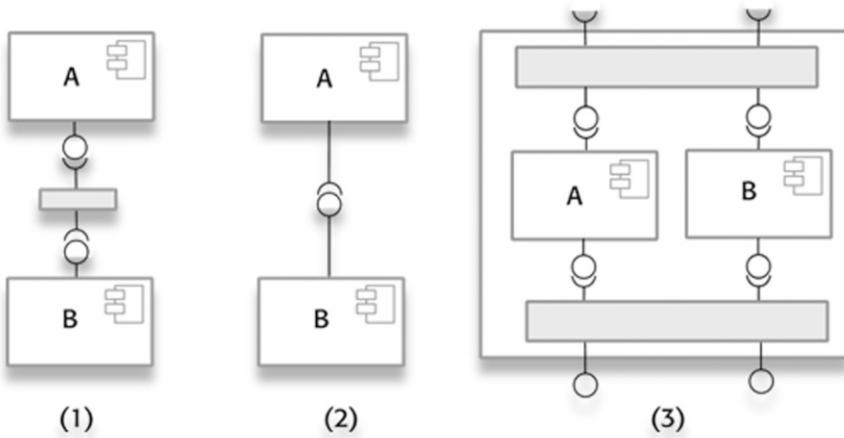


Fig. 7.7 Illustration of component composition

determined by ways components are composed and must take interface disparities into consideration.

Components can be composed to create a larger component, a subsystem, or the system itself in three different ways as shown in Fig. 7.7 (where horizontal bars represent development work that may be required for the composition):

1. A sequential composition refers to composed components being executed in sequence, with the hosting element providing “requires” interfaces the composed components use. For example, if component A provides real-estate multiple listing services and component B is a collection of services a real-estate platform offers, the reason to compose A and B is to cater the platform’s services to benefit both sellers and buyers (which might also challenge the value of traditional real-estate agents). The host, which is a platform element, would need to implement designed “requires” interfaces using the “provides” interfaces of A and B to access their operations. By accessing the functions that A and B provide, the host would be able to implement features to provide users not only searches for real-estate properties (A enables) but also services such as hiring a real-estate agent online (B enables). Minor amount of development may be needed based on a design of sequential consumption of the components involved.
2. A hierarchical composition is one where one component (with “requires” interfaces) calls on the services of another (with “provides” interfaces). Some “glue” code may be needed too in such cases. For example, an order component (A) is composed with a credit card verification component (B), where the order component can be viewed as a parent component.
3. An additive composition is to put the interfaces of two components (or more) together to create a new component. As shown in Fig. 7.7(3), the integrated component, on the one

hand, implements the “requires” interfaces of the components *A* and *B*. On the other hand, the integrated component creates “requires” interfaces to consume features of *A* and *B*. In other words, “provides” and “requires” interfaces of an integrated component are designed based on interfaces of constituent components. Some significant “glue” code may be necessary, as may some new development if the integrated component provides additional features. An example of an additive composition is to create a new catalog component based on two or more commercial catalog components.

Glue code is particularly important to resolve interface incompatibilities and provide necessary adaptation mechanisms. Interface incompatibilities between the “requires” and “provides” operations include mismatches in terms of the operation names, functionalities, parameters, and data types. For a simple code example below, the second line is a piece of “glue code” that strips post code from an address string from a component method before calling another method from a different component to display average home price in the postal code region.

```
address = addressFinder.location (phonenumber);
postCode = getPostCode (address);
realEstatePriceFinder.displayAvePrice(postCode, "detail")
```

More composition patterns are, in fact, possible such as partial sequential and partial hierarchical. These differences in component composition are critical to distinguish in practice to understand the amount of work involved in component integration. Consider the example earlier about a banking fund transfer component. If bank policies and procedures are implemented as separate components, then the original example suggests a hierarchical component composition where policies and procedures are child components. If we must apply policies before procedures, then use of these components would become a sequential composition. However, if the two components were used in an additive composition, we might need some transformed “requires” interfaces to provide necessary details that existing policies and procedures would rely on and expose new policies and procedures that would be based on existing policies and procedures.

7.5.5 Component-Based Architecture

Software construction can be component based, meaning components are the primary “building blocks” of the software. To support such component-based software construction, we need a high-level structural layout of software elements to establish relations and protocols of communication among the elements, i.e., an appropriate architecture that supports component-based development. As will be discussed later, in an architectural

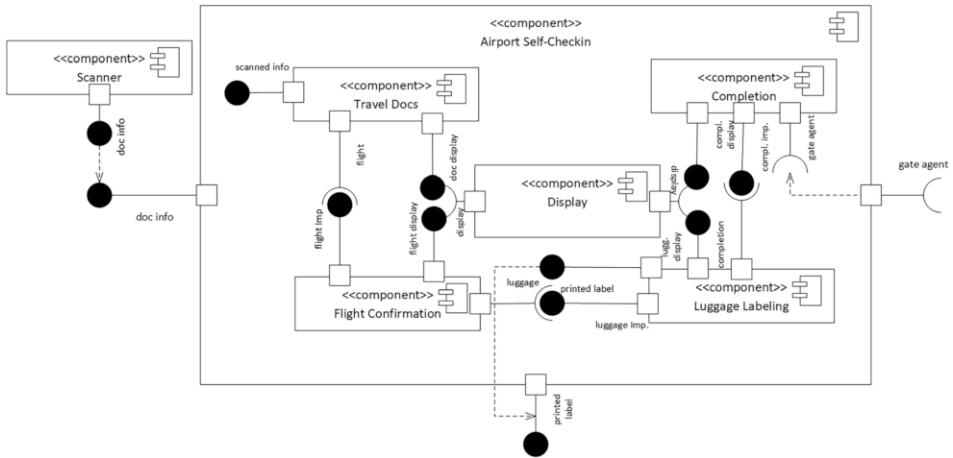


Fig. 7.8 Illustration of component-based software construction

design, we choose an architectural style (to layout elements) and design structuring elements based on the chosen style. Such architectural decisions may also include:

- Decisions on components, their roles, and collaborations
- Decisions on compositions and communications among the components and with the rest of the system

Figure 7.8 illustrates how components may interact in an airport self-check-in system, each component managing a step of a check-in process. Since the system is built using mostly components, the diagram also provides some indication how an architecture of component-oriented software might look like. These components communicate with each other, with a display component, and with other external processes for document scanning, printing luggage labels, or access to operation that controls monitor display. To keep the illustration less complex yet adequate for the purpose, the system described may be only a component in a larger system that manages airport gate operation including a customer data handling component. There are three types of connections with components in the diagram:

- “Standard” ball-and-socket connection between two internal or two external components
- Ball-to-ball or socket-to-socket delegation through ports at the boundary of a component
- Connections with external devices for input or output

Connectors are separate pieces of code (including “glue” code), which may have different “sizes” depending on the nature of a connection. As discussed earlier, ways

components are composed may affect connectors to be written, especially when components need to be configured. A component-based software development defines an architecture that specifies how the components are related and composed. This architecture also includes structuring components that must follow system's constraints and behave consistently with runtime behaviors of the components, if any.

If we design a system using mostly components, we construct the system by simply connecting the components with possibly some “glue” code. There are only a few ways two components can be composed. In addition to three commonly seen compositions explained

Fig. 7.9 Column view of component-based architecture

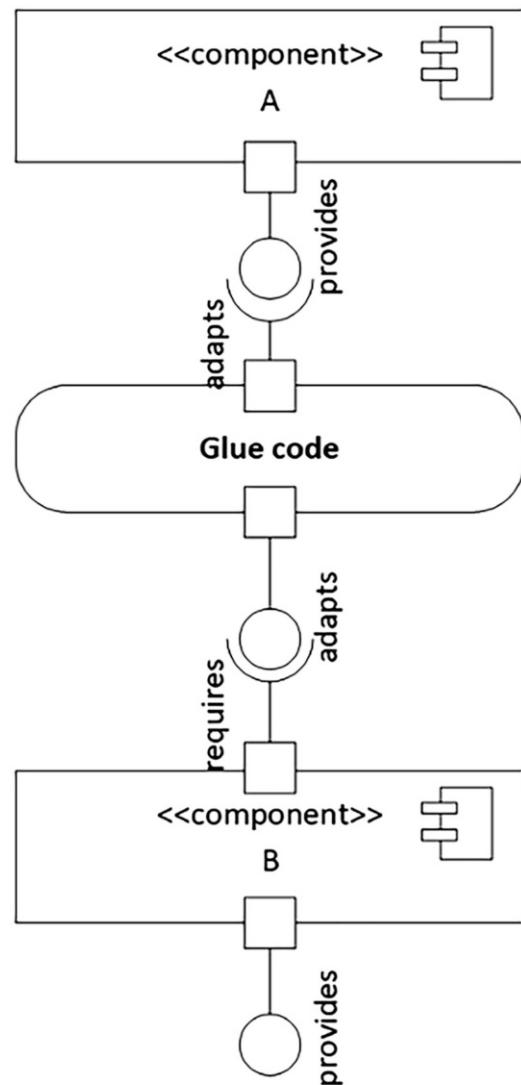
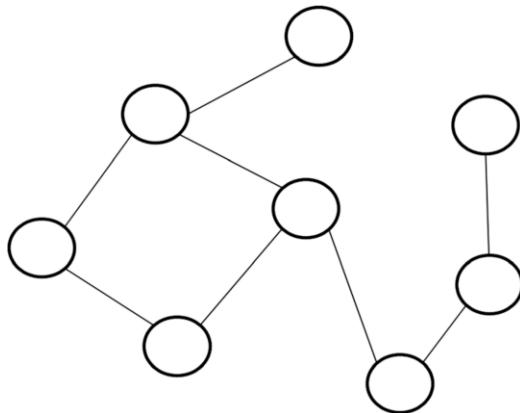


Fig. 7.10 Analogy of a component-oriented program structure—a workflow graph with component “nodes”



earlier, Fig. 7.9 depicts another situation in a hierarchical composition to use one component’s functionality to implement a “requires” interface of another, where some “glue” code would be needed.

If we viewed components as “vertices” (or “node”) in a (mathematical) graph (a larger composite component with arbitrary internal component layers would be viewed as a single vertex as well), then a “path” between two “vertices” would be a connection between the two components. We might even associate a “weight” with a “path”—the amount of “glue” code we must write (with appropriate qualitative measurement scales). Thus, an architectural structure of a component-oriented program might look like a weighted graph illustrated in Fig. 7.10. The shape of this graph (in terms of possible connection “paths”) depends on a workflow of the software, i.e., a flowchart that connects all required software processes in a logical order and maps each process to a group of components with which the process can be implemented.

7.6 Design of Application Frameworks

As explained earlier, software frameworks include development frameworks such as .NET and PHP. Such frameworks are necessarily large, complex, and multi-functional with very different design principles and practices. In this section, we will only discuss application frameworks that aim to provide generic solutions to families of applications.

7.6.1 Characteristics of an Application Framework

Design of an application framework is generally a much larger design endeavor than design of a software component. Design of a framework generally aims to facilitate software development by effectively implementing software control structures that can be reused

across many different application scenarios, thereby reducing overall software development time and improving software reliability at the same time. Thus, framework design is a high-level design activity focusing on the design of abstractions. A large application framework may include code libraries and tools, along with its APIs to facilitate use of the framework. As mentioned earlier, an enterprise resource planning (ERP) application framework provides organizations with generic software solutions to manage day-to-day business activities such as accounting, procurement, risk management, customer relation management, and supply chain operations. A complete framework (called an ERP suite) may also include software performance management and support software that helps organization plan, budget, and produce reports. ERP systems often also have abilities to eliminate data duplication of organization's shared transactional data (from multiple sources) to provide data integrity with a single reliable data source. To many companies today, ERP systems are critical for managing their businesses that could be distributed around the country and in different parts of the world. Commercial ERP frameworks are periodically updated by the vendors as the underlying technology advances or significant improvements are implemented. As a result, using an ERP framework requires continued learning. As a large framework evolves, legacy capabilities are generally retained to support existing applications, which may impose additional challenges in maintenance with increased complexity of system configurations.

A framework that provides a generic solution often implements a generic control of software operations that can be customized to satisfy different application scenarios. Customization can be provided through "requires" interfaces and the abstract methods of the primary control. Users of the framework must provide required implementations pertinent to a particular application of the framework. In this sense, we can view an application framework as an abstraction of a software solution that abstracts away the application details through interfaces and additional process-related abstract methods. By extension, a framework supports a family of software systems by contextualizing the abstraction with all relevant details of a concrete application scenario, realized in appropriate implementations of the interfaces and abstract methods a client provides. For example, platform games involve a player character running and jumping across platforms overcoming obstacles with different difficulty levels. An application framework could be developed to implement a control of how one can play but leave platform conditions and various kinds of obstacles customizable. By comparison, a component supports a family of software systems through component replacement with altered software behavior. For example, to better support the framework application, we can design a platform as a component with customizable platform gaps, heights, and terrain conditions. Often, frameworks and components are complementary in supporting the development of software families.

An application framework consists of "stable parts" and "flexible parts." "Stable parts" define an overall architecture of the system consisting of the control mechanism, the essential components, and their relations. These parts are unchanged in any instantiation of the framework. "Flexible parts" are software parts where users of the framework must

add customized code to provide the functional details specific to their own application scenarios. With an object-oriented design, a framework generally consists of a set of abstract classes. Each abstract class implements certain control that defines how objects collaborate with supporting processes captured by abstract methods. Operations that must be contextualized by concrete application scenarios are placed in “requires” interfaces, which must be externally implemented. To build an application using a framework, users would extend the abstract classes, configure other framework objects, and provide required implementations. When necessary, however, users may also modify existing business processes appropriately to be able to adopt the framework and hence shorten the development cycle significantly.

7.6.2 Framework Design

Frameworks are software components; thus, discussions in the last section about component design are also applicable, in principle, to framework design. But whether we need a framework is often a very different decision to make (compared with decisions on a component). It is possible that requirement analysis, particularly with a major use case, may lead to the conceptualization of a potential framework by analyzing variability of the application. Larger processes are decomposed into smaller processes to better identify similarities and differences. This process of top-down exploration is generally iterative with design refactoring interspersed, as we discover different ways to look at the problem.

However, frameworks are more likely to be identified in a bottom-up approach by examining concrete software instances. Procedural abstraction is essential for framework design. If operations vary in non-essential ways such as use of information, we introduce appropriate parameters to address the differences and hence consolidate the operations. Furthermore, we can view operations “essentially the same” if we can capture the differences with appropriate procedural abstractions. But a good framework is an evolutionary product. As a framework being developed, we continue to examine application scenarios we can collect to confirm existing framework operations or expand their applicability. To design an effective framework, the following understanding about the system we are designing is essential:

- High-level commonality in different control processes is stable (though it may be complex).
- Variability can be effectively captured through abstractions (interfaces and separate abstract methods) to allow adequate business-specific details to be provided.
- Customizable “parts” can be constructed independently without requiring any knowledge of system’s inner workings.

One way to explore the viability of a framework is to identify two or more applications in the targeted application domain that are structurally similar. It’s also possible to refactor

an existing application while envisioning the potential that the refactored application can be applied to other application scenarios. Iterations are generally required when new application scenarios are discovered or new ideas emerge. Though such iterations may not change much the essential structure of the process control (because of the fundamental similarities of the applications' control structures), they can significantly change the design of framework abstractions.

Prototyping is important to test the applicability of a framework to a new domain of applications. There are two ways to prototype. If the newfound applications are within the scope that the framework can be extended, prototyping can be straightforward. However, if modification is required, we should only prototype the possibilities using a copy of the framework before applying the modifications permanently. It is generally true that the more general the framework is designed, the easier it is to extend.

Good examples are crucial for defining the scope of a framework and for identifying effective abstractions, framework features, and, more importantly, solid bases for generalization. Effective practices in seeking good examples may include:

- Study a solution domain based on the problem domain
- Study real programs (if feasible) to identify commonalities and variabilities with a wide range of carefully chosen test examples
- Look for common patterns of interaction
- Parameterize variations when feasible
- Research ways to decompose a framework candidate into components

7.6.3 A Case Study

Managing a product inventory and managing financial assets are two different processes. But how much commonality do they share? Consider a financial management firm that manages a collection of financial assets like stocks, bonds, mutual funds, etc. The main characteristics of an asset collection are the following:

- Each client of the firm has a portfolio of assets like mutual funds, stocks, and bonds, including their transaction records.
- These portfolio assets can change daily in any number of times due to purchases of new assets or selling existing assets (through various vendor platforms).
- The value of an asset item may change at any moment in a trading day.
- The current total asset value of a client's portfolio should be updated daily after trading is closed.

Consider a grocery store inventory that has a categorized collection of food items like meats, vegetables, bakery, beverages, etc. The characteristics of the inventory are “behaviorally” similar to those of an asset collection despite a very different application domain:

- Each branch location of the store has a similar collection of food items and its sale and replenish records.
- Food items can change daily due to customer purchases and store's reorder operations (with a list of vendors).
- An item may change its sale price daily (even by hour) due to a variety of factors that can affect an item's price.
- Each branch location should update its collection every night to monitor triggers of a reorder if items are low in stock.

Although items (financial assets and food items) are very different, they have similar containment structure, similar transaction pattern, similar daily updates, and even some similar accounting practices. Common software elements appear to include:

- A controller: to monitor transaction events, save transaction instances completed in a certain time interval (say, a day), and update item accounts at the end of a time interval
- An item account: responsible for updating its account details using the saved transaction records with potential abstract methods for customized calculations when account attributes are updated
- A transaction entity: to record buy or sale of items (the controller may manage a collection of transactions)
- Client account: containing a collection of items to be updated daily

Representational and operational similarities, however, are just some of the aspects we assess the suitability for a framework. Similarities in other areas also need to be assessed such as quality assessment and control, reporting, and customer service. In the case of inventory, for example, quality management means decreased management costs, better business forecasting, ease of supply chain management, improved profit margins, multi-venue sales performance, customer satisfaction, and efficiency of order fulfillment. Quality management is a process of decision-making that must be informed by appropriate business analytics. For example, decisions must be made on ideal inventory amounts to carry, optimal storage of products based on sales at various locations and venues, the frequencies of product reorders, appropriate times to offer discounts, etc. For the case of inventory, Fig. 7.11 describes a conceptual quality control framework that integrates a business intelligence module to help with a decision-making process.

In comparison, the quality of a financial management service may be measured by some similar quality attributes such as good profit margins and investment returns. However, the quality a financial service is also measured by attributes that are very different such as risk tolerance of financial management. Thus, whether a control process can be shared between the two systems needs to be further assessed. An assessment would examine possible “requires” interfaces that can abstract away the differences of the quality attributes in the two cases and a consistent “provides” interface useful in both cases. Similarly, customization of data representation also needs to be assessed to see the possibility of a business

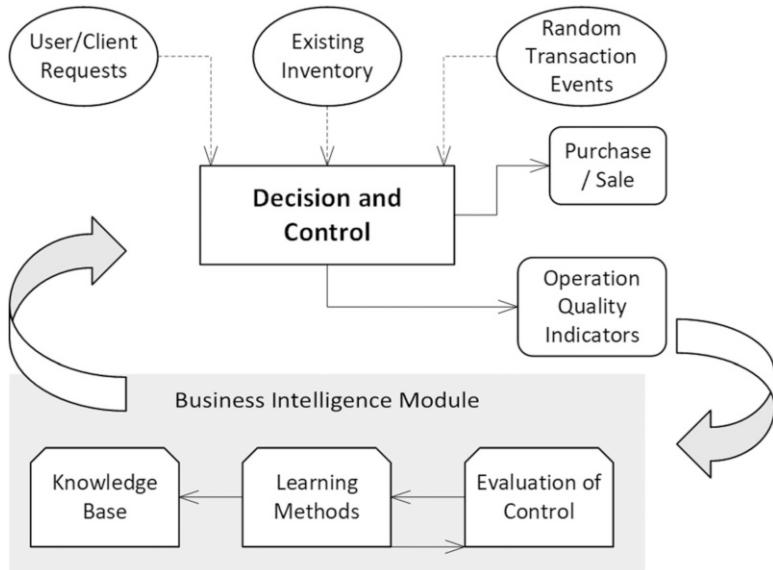


Fig. 7.11 A conception diagram for inventory management control based on data analytics

intelligence component to support business data analytics that can be used in both business scenarios. Similar analyses are also needed to assess other areas of a potential framework such as reporting and customer service.

7.7 Microservices

Microservices are relatively a “newcomer” in the world of larger software elements. Despite the prefix “micro,” the size of a software service is relative to the purpose it is created for or to the context it serves in. A service for data sorting is necessarily (much) smaller than one for a “shopping cart.” A microservice element attempts to address a single concern, such as a data search or a login function that can be independently deployed to facilitate software maintainability. Using microservices, code updates would not result in refactoring in other parts of the system or redeployment of the software. A microservice can indeed be small or even “micro” as necessary. Regardless of its potential size, a microservice also uses collaborating elements. In that sense, a microservice is a module or component. Interestingly, when we talk about microservices, we generally mean a service-enabled architecture, not individual services. It is practically more important to find or build an architecture with a support infrastructure to facilitate use of microservices in software development.

7.7.1 Some Background Information

A monolithic application is one built as a single unit even at an enterprise level. For example, a web application was often built in a client-server architecture with a relational database. The client-side processing consists of HTML and JavaScript code with execution cycles managed by client's browser software. The server-side process (or the backend of the system) handles HTTP requests, executes domain logic, retrieves data, and constructs HTML views to be sent to the browser (illustrated on the left side of Fig. 7.12). Such an application is monolithic in the sense that it results in a single executable file. Any changes to the system would involve building and deploying a new version of the backend and all the logic for handling a request running in a single process. Software delivered as a monolithic program has the advantage of being relatively simple, allowing use of basic design principles to decompose the application into namespaces, classes, and standalone methods. Such systems are also easier to refactor and shovel parts of the modules as their boundaries are often not clearly defined (or judiciously protected). It is a familiar approach for many developers, who can run and test the application on their laptops with proper care in version control before a new deployment of the application. With an appropriate load balancer, monolithic applications can scale "horizontally" running many instances concurrently. However, the "bottleneck" of a monolith is "vertical" scalability when frequent modifications or new additions are required, which would cause code structures to erode and the application to become increasingly difficult to maintain.

Consider a conventional e-commerce web application with three subsystems for customer accounts, merchant inventory, and shipping arrangements. A monolithic approach would place all three subsystems in the business layer sharing perhaps two databases, one for inventory and the other for customer accounts, orders, and order shipment. A service-

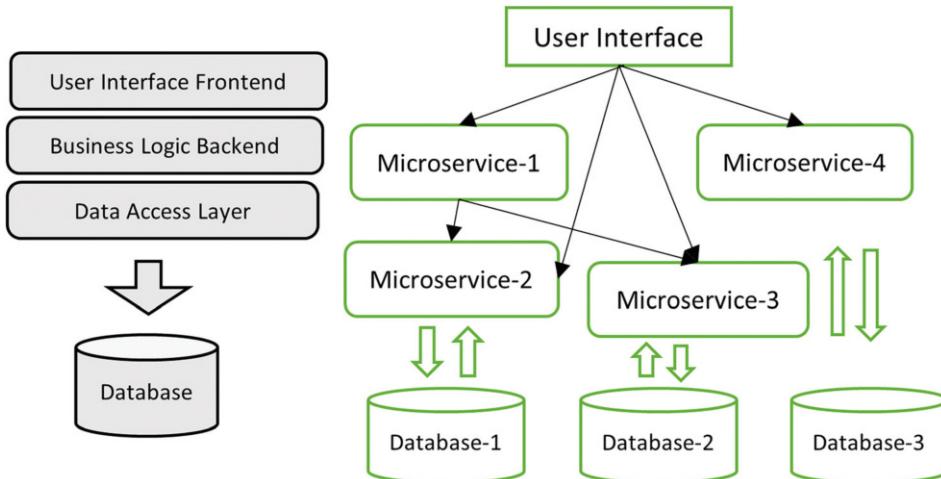


Fig. 7.12 A monolithic application vs. a service-oriented application

oriented approach would build three services for customer accounts, inventory, and order shipping with their own databases (each service may have its own background services like credit card validation, shipping logistics, etc.). The store-front web application and a corresponding mobile application access and use the same services (with assistance from potential gateway APIs). The service-oriented approach makes development nimble, updates less impactful, and further development more manageable (illustrated on the right side of Fig. 7.12).

Web services are easy to use and can be developed independently—the two primary reasons for their popularity in the early 2000s. Web service was the primary form of remote software services using Simple Object Access Protocol (SOAP) as a communication mechanism to transmit information or data in XML format over HTTP for an application to access a remote service. As discussed earlier (when web services were viewed as remote software components), interfaces of a web service are described in a machine-processable Web Services Description Language (WSDL). Applications use web services through local proxies to deliver required data and retrieve the results over the network. Web services are typically standalone applications, such as a service about weather forecast or mortgage calculation. They can be applied to a variety of application contexts with software systems operating in heterogeneous hosting environments. Around the same period, the notion of service-oriented architecture (or SOA) also gained popularity with, however, a much broader understanding. In loose terms, SOA is an architectural style that supports service orientation, a way of thinking about software in terms of services it uses, their outcomes, and service-based software development. A service is a logical representation of a repeatable (business) activity that has a specified outcome (e.g., verifying a credit card, providing weather data, or handling hotel reservation). It can be self-contained, composed of other services, or coupled with external components. An SOA structures system operations, implemented with services, to represent (business) processes. A client uses software operations with no knowledge about services underlying the operations. Thus, to a client, a service is a “black box.”

In SOA, services communicate with each other. They do this over an enterprise service bus (ESB), a notion mimicking the communication between a computer’s CPU and its memory. An ESB is a software platform used to distribute work among connected components of an application. It is designed to provide a uniform means of distributing work among connected components, offering applications the ability to connect to the ESB and subscribe to messages based on simple structural and business policy rules. The communication mechanism of ESB can be like that of web services, say, using SOAP and a service description language in XML format over HTTP connection. But ESB can use other communication means like JSON over HTTP to send requests, read, or change data. However, the focus of an ESB architecture is to decouple systems and services from each other while allowing them to communicate in a consistent and manageable way with good scalability. Therefore, an ESB architecture has its core integration principles. There are many EBS (middleware) vendors, and they follow the same building principles with subtle differences.

7.7.2 The Promises of Microservices

Microservices are to provide viable solutions to the “bottleneck” of monolithic applications in terms of “vertical” scalability. Like web services, they are generally distributed, highly independent software units. They are “micro” compared with a “comprehensive” system. Nevertheless, they are generally larger software elements, though the actual size of a microservice can vary significantly depending on what it does. Professionals have suggested that a service size should be kept as small as appropriate for a roughly a five-person team to be able to develop (and maintain) the service. In contrast to web services, microservices are nimbler, task specific, and more flexible about communication protocols. These services are built around business capabilities and a bare minimum of centralized management. They are independently, fully deployable with an automated deployment process. Software processes built around microservices help organization make rapid changes to applications within very short production schedules. By simultaneously leveraging microservices and an automated deployment process, software development can be well streamlined to accomplish things that were impossible in the past, such as code deployment of hundreds or even thousands of times per day. With the characteristics, clearly, microservices are not distributed modules of an essential monolithic application.

The architecture that supports microservices is a subset of SOA with the following characteristics:

- It uses componentization with the ability to replace parts of a system independently.
- It uses organization around business capabilities instead of technologies.
- It supports services that carry their own communication logic with a light carrier structure that transmits communication messages (aside from use of an ESB).
- It supports decentralized data management with one database for each service as necessary.
- It has infrastructure automation capable of continuous delivery of the services.

The term “microservice architecture” refers to an architectural style that supports applications primarily built using microservices with the characteristics listed above. It is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight communication mechanisms. This architecture also supports microservices that are designed to make failure points more independent of each other, creating a more stable operation environment. A cluster of services can be designed in such a way that if one fails, another picks up the operation (with an orchestration tool). Thus, the failure is self-healing without human intervention.

The technical approach to microservices may vary in organizations. There are language-dependent frameworks for building applications using microservices. Java-based framework Spring Boot appears to be a frontrunner with a popular language and a familiar development platform Eclipse, integrated with a popular software project management tool. The framework allows for rapid application development and easy deployment on

various platforms. Spring Boot is a mature, open-source, and feature-rich framework with a vast development community to turn to for technical support. There is also a strong community support for learning with training of various kinds from introductory to advanced certification levels.

7.7.3 Internal Operations of a Microservice

Design of services conforming to an architectural style is a critical step in a service-oriented development. Figure 7.13 illustrates conceptually the internal operations of a microservice:

- A service has an API that consists of operations (invoked using some combination of synchronous protocols and asynchronous messaging) including:
 - Commands, such *createOrder()*, *cancelOrder()*, etc., resulting in data mutation
 - Queries, such as *findOrderItem()*, *getOrderHistory()*, etc., without data mutation
- Part of a service API is also about published events, such as domain data being updated, deleted, or emitted by an aggregate after an event is created. A service publishes events to a message channel implemented by a message broker.
- A service has a processing logic core, the heart of a service. This logic core implements the API's operations and publishes events. The processing logic invokes the operations of other services and subscribes to their events. A service is also responsible for (persistent) data handling using the service's dedicated database.
- A service has a private database with sometimes data replicated from other services. To ensure loose coupling, a service does not share its database tables and allows data access only through its API.

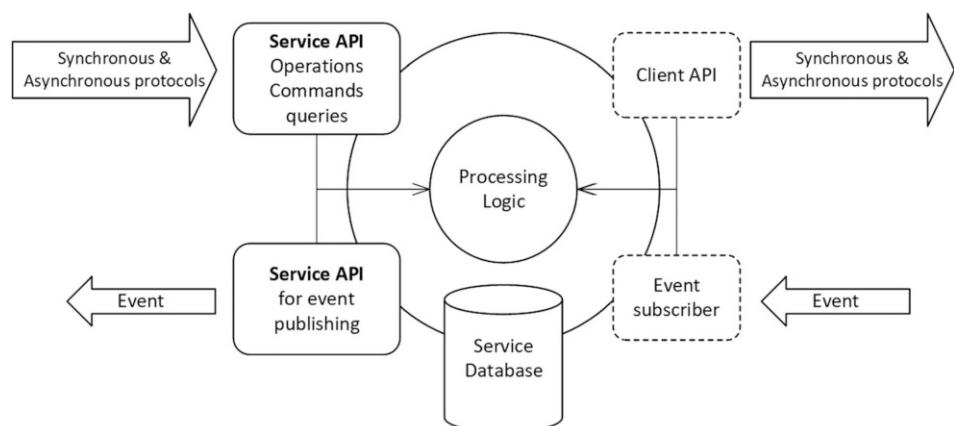


Fig. 7.13 Operation flow of a microservice

In comparison with monolithic applications, applications built using microservices may not have the freedom of using moving parts between microservices, as services are highly independent with clearly defined boundaries. Microservices can be built using different platforms and languages, which is another reason for the difficulty of sharing and swapping microservice parts. Other challenges using microservices include:

- Microservices have all the associated complexities of distributed systems.
- Communication between different services can be fragile and unpredictable.
- With minimized central management mechanism, it can be difficult to manage many services.
- Developers need to deal with some environment issues like network latency and load balancing.
- Testing can be complex over a distributed environment.

7.8 Summary

Successful development of a complex software system typically relies on larger software elements as building blocks. This way of building software is not only an effective software engineering strategy but also a means to ensure software's long-term viability and success.

Use of a software library is perhaps the most popular way to use implemented algorithms, data structures, software services, and generic software components to develop software features and operations with less time, more reliability, and a better streamlined code construction process. Software libraries for high-performance computer graphics and scientific computation are critical for the advancement of science and technology. There are many factors that can impact the success of a library. Among the most important are a community support, context independence, task orientation, and stable interfaces without limiting implementation possibilities. Design of a library should consider the benefits and risks and focus on computing power it offers, its extendibility, and its applicability. Good design and implementation practices are essential. A successful library is also characterized by an accurate and a complete documentation.

A software component is an independently deployable and replaceable unit. Software components facilitate software construction though composition according to defined composition standards with contractually specified interfaces and explicit context dependencies. There are essentially three kinds of components for software control, domain representation, and infrastructural needs. Component-oriented software development is an effective way to ensure software reliability and ease of software maintenance. Design of components includes decisions on interface stability, implementation flexibility, functional cohesion, external coupling, access protocols, and, mostly importantly, a component model about the interfaces, usage information, and deployment. Components may be identified through analysis based on behavioral, functional, and structural decompositions. They can also be identified through concept development and abstractions over a set of difficult

decisions. Besides, operation extensions and grouping may also lead to the discovery of new components. Component composition may follow a sequential, hierarchical, or an additive pattern, but it can also be any combinations of such patterns. Thus, in abstract terms, a component-oriented architecture organizes components into a graph structure with “paths” connecting component “vertices” through appropriate compositions with user-written “glue” code when necessary. An application’s workflow defines the structure of this graph of component composition and integration.

A software framework is an abstraction in which the software provides generic functionality, which can be customized with user-written code, resulting in specific software applications. Building frameworks is an effective way for building and deploying applications by providing a reusable software environment with universal values in an application domain. Developing frameworks is also a standard way to provide a software development platform with supporting infrastructures, language compilers, and component APIs. Design of an application framework consists of design of the stable parts and changeable parts of the framework. Thus, abstract classes and interfaces are commonly used to capture what’s changeable and customizable of a framework. A framework conceptualization often starts with application or code examples with structural similarities.

Microservices are software units highly maintainable and testable, loosely coupled, and independently deployable. They are organized around business capabilities, where traditional monolithic systems often fail to address. Service-oriented development also helps an organization evolve its technology stack. An appropriate architectural style is required to structure an application as a collection of microservices (according to their roles in applications or in infrastructures) to enable rapid, frequent, and reliable deliveries of large, complex applications. A streamlined software development process with microservices also helps shorten a software lifecycle process from development into production by breaking down barriers between software development and technology operations. Essential aspects of design of microservices include decompositions of business capabilities, componentization, light-structured communication logic, and decentralized data management.

Exercises

1. Explain “requires” and “provides” interfaces of a component with your own examples.
2. Explain the similarities and differences between a software component and a software framework and between a framework and a software library.
3. Library APIs can be deprecated for a widely variety of reasons, such as those listed at this site: <https://docs.oracle.com/javase/10/core/enhanced-deprecation1.htm#JSCOR-GUID-23B13A9E-2727-42DC-B03A-E374B3C4CE96>. As this website shows, the creator of a library also generally makes great efforts to help users minimize the potential negative impact of the deprecated library APIs. Based on your reading of this website, answer the following questions:
 - (a) As a user, how would you know an operation in a library API is deprecated?

- (b) What would you do if you know some library operations you used in your code became deprecated?
 - (c) What kind of tool would you need to help you find code lines where deprecated operations are used?
4. Suppose you are building a login component for a student course registration system.
- (a) Give an example of a “provides” interface of the component.
 - (b) Give a scenario where a “requires” interface is needed.
5. Give an example that software components are identified not from scratch (by analyzing software requirements), but because of the evolution of existing modules.
6. A button is a reusable component, but a panel of buttons is a larger component that can be used in a variety of software applications. Assume a button panel contains ten number-digit buttons. Besides, it must also have a “backspace” button to backtrack characters as necessary. Other buttons may vary depending on an application. The function of a button panel is to allow a user to provide required information in an application. Given only these assumptions, design “requires” and “provides” interfaces for a button panel component.
7. Use a program you wrote in the past to reorganize, as appropriate, the program elements as components. Then, use a component diagram to describe how the program works in terms of component communications.
8. Operations of an automated teller machine were described thoroughly in an exercise of Chap. 5. Design a simulation of an ATM using component-oriented approach with the following tasks:
- (a) Identify the components needed and describe ways they can be composed.
 - (b) Use a diagram to show the workflow of ATM operations.
 - (c) Describe how the components fit into the workflow, and then incorporate the components into the diagram.
9. User login is required to access software applications of virtually all kinds. A user must provide a form of user identification and a password correctly before he or she can access the software. However, different applications may require additional information to safeguard a login process. For example, they may require a verification code sent by certain means (text, email, etc.) or answers to some questions that the user provided the answers at registration. Different applications also have different ways to allow the recovery of login credentials. Given only these assumptions, design a framework component that could be used by commonly seen registration and login processes by completing the following tasks:
- (a) Design some abstract classes that control the registration and login processes.
 - (b) Design needed “requires” and “provides” interfaces.
 - (c) Use a diagram (of any kind) to describe the control processes.
10. In a restaurant, customers order food with a server, and the server has the food prepared. In a service store of a service company (such as AT&T or Spectrum), a customer may use a touchscreen to specify a reason to come to the store (by selecting

an item from a given list) and provide the name before receiving a service-order number. Later, they are called on a first-come-first-serve basis to receive specified services. Is it possible to design a framework to be used in either case and possibly in more business settings? Answer the question by completing the following tasks:

- (a) Study the similarities and differences of the two scenarios.
 - (b) If you think possible, design some potential abstract class(es) to provide a control of the process that can be customized in either service scenario with needed interfaces. (Try some pseudo-code to see how the control might work.)
 - (c) If you doubt the possibility, provide a few reasons for your doubts.
11. Consider self-serving processes such as self-checkout at a grocery store or self-check-in at an airport. Is it possible to design a framework to be used in such self-serving processes?
- (a) List similarities and differences of such processes.
 - (b) If you think possible, what might be possible abstract classes and interfaces?
 - (c) If you doubt, provide reasons to support your judgement.
12. Simulations on “wait” lines are useful in many situations, for example:
- A bank wants to know whether additional tellers would be needed during peak times.
 - An owner of a restaurant wants to know whether additional spaces would be needed to accommodate a significant increase of diners.
 - An airport operator wants to know whether additional runways would be needed to shorten or eliminate a landing queue of planes in the sky waiting for availability of a runway (which can be a hazard).
- A simulation is desirable in a decision process because some of the modifications to the business can be quite costly. Explore the possibility by answering the following questions.
- (a) Is a framework helpful for such simulations?
 - (b) If it is, study the commonalities and variabilities of such simulations, and then design (and implement) an abstract class that provides the control of a simulation with necessary abstractions.
 - (c) If not, reason why.
13. Suppose you need a software process for an outdoor event catering service such as wedding events, graduation ceremonies, outdoor movie shows, or any such outdoor gathering that requires seats, tents, and stages to be set up. Other potential arrangements might be a music band, food catering, etc. The software can be essentially a calendar application (web or desktop) with the following features:
- To enter information about the services needed for an event (and staffers would make phone calls, send emails, etc. to arrange any physical setups)
 - To allow updates of event status
 - To search an event (given an event identification number) for event details and arrangements status

- To provide weekly or monthly event views (only brief information is shown for each event in a view)
- To remove events given an even identification number

Identify the software elements needed (components and frameworks). For each component, design its interfaces and collaborating elements. For a possible framework, design potential abstract classes and their collaborating elements.

14. Suppose you want to develop the application in Question 11 using microservices. Identify potential services and their relations based on the essential characteristics of microservices.
15. Docker is a platform designed to help developers build, share, and run modern applications. It handles the tedious setup, so users of a “docker” can focus on the code. Do some search to learn how a “docker” can be used as a tool in running microservices.

Further Reading

- Bloch, J. How to Design a Good API & Why It Matters. *InfoQ Audio* <https://www.infoq.com/presentations/effective-api-design/>
- Elgabry, O. Component Based Architecture – Revamping the architecture thoughts. [medium.com](https://medium.com/omarelgabrys-blog/component-based-architecture-3c3c23c7e348).
<https://medium.com/omarelgabrys-blog/component-based-architecture-3c3c23c7e348>
- Hu, C. (2003, December). A framework for applet animations with controls. *ACM SIGCSE Bulletin*, 35(4), 90–93.
- Newman, S. (2021, August). *Building Microservices* (2nd ed.). O'Reilly Media.
- Parnas, D. L. (1976). On the design and development of program families. *IEEE Transactions on Software Engineering, SE-2: March*, 1–9.
- Sommerville, I. (2011). *Software engineering* (9th ed.). Addison-Wesley.



Software Design Patterns

8

8.1 Overview

Since the seminal work of the four original authors, commonly known as “Gang of Four,” the design patterns have been widely known and used for roughly three decades (Gamma et al., 1994). These patterns provide solutions to some commonly recognized design needs in such a way that supports software variability, generality, or extendibility. They are idiomatically named so that users can communicate easily about potential solutions without going into granularity of a pattern’s structure. The influence of the design patterns in modern software design, especially with object-oriented methodologies, is undeniably significant and lasting. The study of design patterns has become an indispensable aspect of learning object-oriented software design.

These design patterns reinforce object-oriented design techniques we discussed in earlier chapters, particularly on the use of interfaces to hide specific object types the code uses, so long as objects adhere to the interfaces. Thus, code can be written using abstractions, leaving (often uncertain or vulnerable) details to dynamically bound objects at runtime through polymorphism. Many design patterns are formed based on the topology of object aggregation and composition. Thus, design patterns often support “black box” code reuse as the presence of composed or aggregated objects, and all their topological details are hidden and invisible in the code using them. All design patterns are interface-driven and can be applied to any concrete application scenarios.

Twenty-three design patterns in the Gang-of-Four book are documented in three categories—creational, structural, and behavioral patterns. The ideas behind these patterns are based on some of the most important, yet nonetheless, classical design principles—chiefly encapsulation of variations in modules to improve code reusability and sustainability—which can be traced back to the 1970s when structured design paradigm was prevalent. This chapter introduces arguably the most useful Gang-of-Four patterns.

However, because of a large volume of online references to these patterns and numerous books about the subject published in roughly the next 15 years since the publication of these patterns, our discussions about these patterns are geared toward understanding the common design ideas behind the patterns, comparison and contrast between the patterns, variations of the patterns when useful, and possible alternatives when applicable. Patterns do provide elegant solutions to some common design problems. But it is also important to realize that we do not design problems for the patterns we want to use simply because we want the “elegance” of a solution. Thus, we will also discuss when use of patterns is appropriate and when we might avoid.

8.2 Creational Design Patterns

It is always desirable that objects are created only when they are needed, not before, to allow potential behavioral alteration. All creational design patterns support delayed (or lazy) object creation and encapsulation of an object creation process. However, high-level implementations should not be affected by lazy object instantiation when code is written using interface operations (or abstract classes when appropriate). We have already used one of the creational patterns earlier—the Factory Method pattern—which will be skipped in our discussions.

8.2.1 Singleton

Business processes are often unique such as a resource management object in an inventory application. The use of the Singleton pattern would ensure the uniqueness of a process (at runtime). The pattern is probably better explained with the code below. A singleton class has a private static self-referential object (i.e., an object of the type being defined) to allow referencing the object in a static manner (as opposed to using an object). To prevent creating different instances, a private constructor is used. To ensure uniqueness of an object, no public constructors are present. We can only access the internal object through a static “getter” method. In general, any object that requires a “global scope” for its unique presence is a Singleton:

```
public class SingleObject {
    private static SingleObject instance;
    private SingleObject(){}
    public static SingleObject getInstance(){
        if (instance == null) instance = new SingleObject();
        return instance;
    }
    public void actionMethod(){ ... }
}
```

The use of Singleton can cause some complication in a multi-threading environment, where multiple Singleton aliases are present; when one gets a lock, all others are locked out. Besides, concurrent access to a Singleton object from separate threads should also be either prevented or synchronized. To be thread safe, method *getInstance* can be implemented to allow synchronized retrieval of a Singleton object.

The Singleton pattern should not be confused with static information sharing among objects, however. For example, in the following code, the static instance *donated* would be shared by all *Employee* objects to record donations. This sharing is limited to places where type *Employee* is accessible. We can create a separate Singleton class, say, *Donation*, for the same purpose but can be used in any context. Therefore, it is generally a design decision whether we want a single class or simply a shared static (resource) variable:

```
public class Employee {  
    private static Map<String, String> donated;  
    //other variables ...  
    public Employee (){ donated = new HashMap<String, String>(); ... }  
    public static void donate(String item){  
        donated.put(getId(), item);  
    }  
    public String getId(){ ... }  
}
```

Finally, because of the global scope of a Singleton object, there is a negative connotation associated with the pattern. As discussed earlier, global variables are often an unwelcome source of code coupling and prone to errors that can be difficult to detect.

8.2.2 Abstract Factory

This pattern encapsulates a group of processes, each creating a product. Products the pattern creates often share a common theme. For example, we may use the pattern to create a group of buttons each with different look and feel. The details of product creation are left to a client. To describe the pattern and its use, consider an airline company needing to create two different flight departure-arrival schedules at airports, one for domestic airports and the other for international airports where military time must be used for flight departure and arrival in addition to a different format. Figure 8.1 illustrates this scenario in a class diagram. The two products are *ScheduleA* and *ScheduleB*, each created by a method of a factory object. Without specifics about how these schedules are created, the factory is abstract, as are the products. However, high-level modules can still be implemented using these interfaces, leaving decisions on details to be made until the time when a concrete instance of the factory must be created.

Could each of the methods in this pattern be just an instance of the Factory Method? It could, but here are the differences between the two patterns:

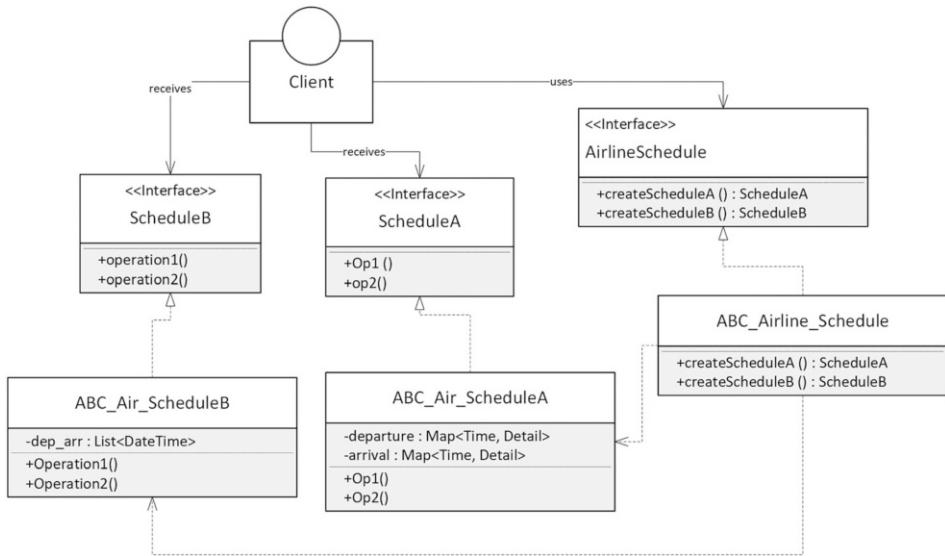


Fig. 8.1 Example of Abstract Factory—airline schedule factory

- The Abstract Factory creates multiple products of different types, whereas the Factory Method creates products of only a single type with varying construction details.
- A factory method must be widely accessible; thus, it is frequently a static entity or a singleton to create polymorphic products, whereas an abstract factory is a runtime factory. By creating different instances of an abstract factory, we create products with varied details. For example, we can create an abstract factory for certain widgets and create a different implementation of the factory when we need a different look and feel of the widgets.
- In a situation that we know all the products' variations (and hence all possible implementations of the products), using a single static factory with multiple factory methods is practically equivalent to using multiple instances of an abstract factory. However, an abstract factory is more beneficial to use when certain products' variations are unknown until runtime. The reason is that code of a static factory must be modified when products' variations become known.

8.2.3 Builder

This pattern provides a small framework to abstract away the construction details of a complex object and divide the construction into parts with construction details left to a client. In contrast to the Abstract Factory, an instance of *Builder* creates a single product but potentially polymorphic ways of building its parts (and then assembling the parts to make a product). The sequence diagram in Fig. 8.2 shows the process of how the pattern works

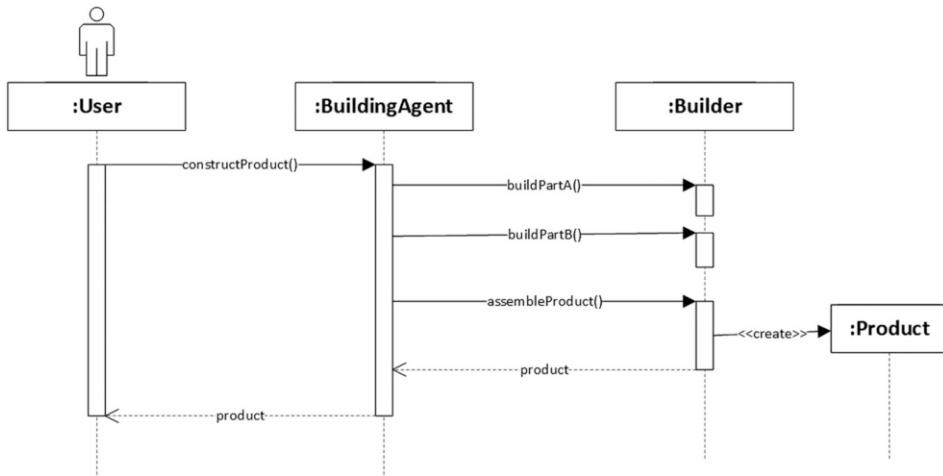


Fig. 8.2 The Builder pattern to represent a construction process of a product

with a simple scenario of a product with two parts: part A and part B. There are two methods to build the parts and one method to assemble the parts and create and return a product. An object of *BuildingAgent* (like an engineer) holds an instance of *Builder* that does the actual building task. User sends message to a building agent (which invokes method *constructProduct*). An agent object delegates the construction work to a *Builder* object. It is a simple framework for a product-building process when an assembly process of a product is stable and can be fully implemented. In such a case, we implement the *Builder* pattern as an abstract class with abstract methods only for building the parts. This is particularly useful to create products in a product line with a stable assembly process.

A useful variation, and more appropriate in some situations, is an external assembly process. In other words, a product is “pre-assembled” with parts open to separate construction. This approach would allow independent part swaps without needing a complete rebuild of a product. The following code demonstrates the process. To modify some parts, we create a subclass (an anonymous class in this example) by overriding some part-building methods (such as the method *buildA()* in the example):

```

Product p = new Product();
Builder base = new ConcreteBuilder();
base.setProduct(p);
base.buildA();
base.buildB();
p = base.getProduct();
Builder modified = new ConcreteBuilder() {
    @Override
    void buildA() { /* altered construction */ }
};
modified.setProduct(p);
modified.buildA();
p = modified.getProduct();
  
```

Consider creation of an airline flight schedule with multiple parts: a header (about passenger information, booking number, flight number, etc.), a body (about the departure and arrival information), and a footnote (about airline policies and airport regulations). Much of the information on an airline schedule is probably standard across airports and airlines around the world. Thus, an abstract schedule builder class is appropriate with a stable assembly process, leaving the customizable parts of a schedule to some abstract methods. In other words, we could design a framework for building airline schedules using a combination of Abstract Factory and Builder. A building process may not be self-contained (such as flight schedule building); thus, appropriate parameters can be arranged to deliver the necessary information or resources needed in a building process.

It is possible that similar parts or even identical parts are rebuilt repeatedly in different products. To alleviate the issue, we can use a shared task handler with a separately executed process for building certain parts. As needed, we can also modify a construction process to make parts exportable. Besides, parts are mutable objects; thus, potential exists that parts; and hence a product could be unintentionally altered, though appropriate remedies can always be found.

Finally, implementations of the pattern may vary. For example, we can embed a building process in a class *Widget* as an inner class for better encapsulation like the code below:

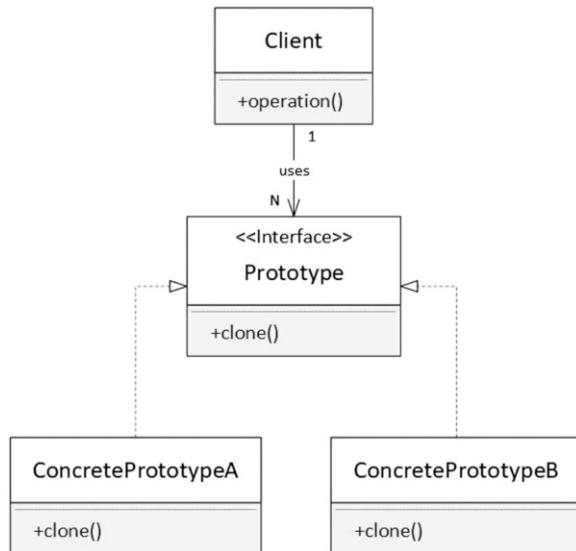
```
Widget.Builder builder = widgetObj.getBuilder();
Widget w = builder.build();
```

8.2.4 Prototype

Object creation can be a complex process such as a user interface frame object or a business object that has certain built-in artificial intelligence models. Thus, repeated creation of such complex objects can be costly. This pattern provides a mechanism of delivering objects by copying existing ones (i.e., prototypes) from a runtime collection, thus avoiding costly repeated object creation at runtime. Instances of sibling types (i.e., subtypes of *Prototype*) are created dynamically and saved, and copies are retrieved when needed. Method *clone* returns an independent copy of an object, not an alias. Figure 8.3 depicts the entities involved and their relations. It is appropriate to load some or all prototyping objects in advance. In general, we also need service methods like “add,” “remove,” etc. to manage a collection of prototypes.

Consider interface *AirlineSchedule* and concrete implementing classes like *AirlineScheduleMKE*, *AirlineScheduleORD*, etc. As discussed earlier, instances of such concrete classes can be created with a pattern like Factory Method, Abstract Factory, Builder, or a combination. Once objects are created, they are saved in a collection so that copies are made when objects are needed again. The following code demonstrates this process, where *schedules* is an instance of *Map<String, AirlineSchedule>*:

Fig. 8.3 Entity relations of the Prototype pattern



```

AirlineSchedule getSchedule(String airportID){
    AirlineSchedule toReturn = schedules.get(airportID).clone();
    If(toReturn == null) {
        toReturn = ScheduleFactory.create(airportID);
        schedules.put(airportID, toReturn)
    }
    return toReturn;
}
  
```

To compare, the Prototype pattern creates products of the same kind (instances of *Prototype*) with polymorphic implementation details, whereas an abstract factory creates different products (with different product types). As appropriate, we may create a product with a single implementation and then make copies when needed. For example, an airline can share one master schedule at all airports it serves. When the state of a copied object is externally updated, the object can be saved back to replace the original instance, if needed.

8.2.5 Section Summary

The design principles behind all creational design patterns are:

- Object creation should be delayed whenever possible and appropriate to allow details to mature.
- Complexity of object creation should be managed and not burden the runtime when objects are created.

The Factory Method and Builder are perhaps the most frequently used creational patterns due to their wide applicability. Often, creational patterns are used in combination.

8.3 Structural Design Pattern

Structural patterns are to solve design problems by establishing structural relationships among the entities in ways often suggested by the patterns' metaphoric names. These structural relations make objects behaviorally richer, more powerful, or easier to use because of object collaboration. Object aggregation is the primary relational mechanism used for all structural patterns, though a contextual relation may be more narrowly characterized as composition, adaptation, delegation, etc. to reflect the way objects collaborate to provide a solution to a particular problem where a pattern is used.

8.3.1 Adapter

The name of the pattern appears accurate in what the pattern does—resolving “mismatch” issues. A mismatch could be as small as a mismatched set of parameters to be passed to a targeted module or as big as communications between systems with mismatched communication protocols, operational environments, or information transfers. Software systems increasingly operate in cloud, yet they may still need to rely on existing systems for functionalities. Therefore, mismatches of all kinds must be addressed in any endeavor of migrating software operations to a cloud environment.

Figure 8.4 illustrates the formation of the pattern. *Target* is an abstraction in which *request* is the method client is to use. *Adapter* implements *Target* and uses an instance of *Adaptee* to invoke, with appropriate adaptation, an operation that fulfills the request. For

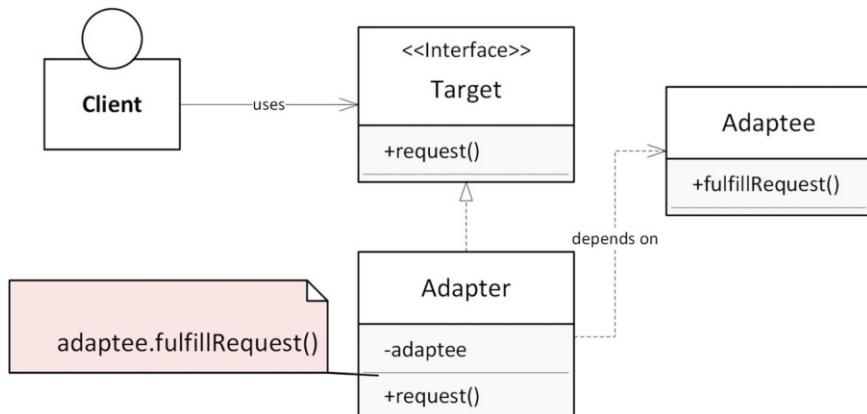


Fig. 8.4 Class diagram for the Adapter pattern

example, old banking systems often do not implement security measures with today's standards. To meet the needs for appropriate security measures, today's online banking systems may rely on adapter components to add a security layer through an adaptee before a client can access banking functionality. If "adaptation" is a process to modify the original process in a nonessential way, then several other structural patterns also have this characteristic as we will see later.

In the following code example, class *B* needs an information string from class *A*. The string must be formatted in certain way before *B* can use it. An adapter object is used to produce a formatted string with an adaptee (an instance of *A*) that delivers the original string:

```
interface Format Adapter{  
    String getFormattedStr();  
}  
B b= new B();  
String formattedStr  
= b.setString(new StrFormatAdapter(new A().getStr()).getFormattedStr());
```

Often, adaptation of runtime tasks may not be determined in advance (like we walk into a room in a foreign country wondering what power adapters we might have to use). To address the issue, we can create an adapter factory to manage a collection of adapters for various adaptation tasks at runtime. In such cases, adapter and adaptee classes must be registered with a factory to provide an association for adapter retrieval. The following code shows the process of retrieving, from an adapter factory, a relevant adapter that does the same work of formatting a string from an instance of *A* to be used in an instance of *B*:

```
FormatAdapter provider =(StrFormatAdapter)  
AdapterFactory.getInstance().getAdapterFromTo(A.class,  
                                         FormatAdapter.class).getAdapterInstance(objA);  
b.setString( provider.getFormatterdStr() );
```

8.3.2 Proxy

The Proxy pattern plays a role of surrogation for operations that often require access logistics such as network connection or security verification. A proxy is a wrapper or agent object called by users to access the real serving object behind the scenes. A user may appear to access an intended object but a proxy object instead. For example, a proxy object would simply verify access protocol, make necessary network connection and communication, deliver the required parameters, retrieve the result from a task handler, and finally convert the result to expected data format and return it to the caller (such as accessing a webservice through a proxy object we explained previously). In fact, any software development platform that supports consumption of remote procedural calls provides a proxy mechanism internally and specifies ways to use it. Cloud computing is delivered through proxies as well.

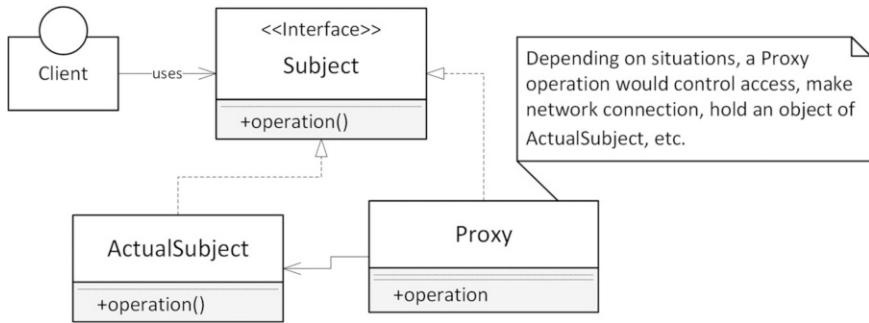


Fig. 8.5 Proxy pattern illustration

Figure 8.5 shows a class diagram of the essential members in the Proxy pattern. In general, a proxy object fetches a nonlocal subject for each request, though it is possible that a proxy may hold a copy of a native object for multiple uses to save connection resources if construction of a local copy is relatively inexpensive.

The design idea of the Proxy pattern is useful in other situations when direct access to certain objects is not available, should be avoided, or must be controlled. In a trivial case, the use of the proxy can simply be forwarding the call to the real object. But typically, a proxy would perform additional work such as executing connection logic, caching when operations of a real object are resource-intensive, or verifying the required data before invoking operations of a real object. For a client, the use of a proxy object is like using a real object, as both are based on implementations of the same interface.

The following code example describes a simple scenario where a proxy process is used to validate the file path before constructing an image object and forwarding the call. In this sense, a proxy can also be viewed as a kind of adapter where an adaptee class (i.e., *ActualSubject*) is a sibling class of *Proxy*:

```

class ProxyImage implements Image {
    private String filename;
    private Image realImgObj;
    public ProxyImage(String filename) {
        this.filename = filename;
    }
    @Override
    public void displayImage() {
        if (!isValidPath( filename.trim() ) ) return;
        if (image == null) {
            image = new RealImage(filename);
        }
        image.displayImage();
    }
    private boolean isValidPath(String path) {
        try {
            Paths.get(path);
        } catch (InvalidPathException | NullPointerException ex) {
            return false;
        }
        return true;
    }
}

```

8.3.3 Façade

The Façade pattern is also about adaptation but on a larger scale to leverage multiple existing components and provide an easy-to-use interface. Organizations often maintain old or legacy systems that still work well. But these systems may not be appropriate for direct consumption from new business applications for a variety of reasons such as mismatched interfaces, outdated security measures, or lack of graphic user interfaces. Applications may also rely on functionalities of multiple complex systems that can be difficult to navigate for users who are not familiar with such systems. These scenarios suggest that adaptations of some kind would be necessary, but they are beyond the scope of the Adapter pattern. A Façade interface provides easy-to-use operations that are implemented by leveraging the use of existing systems. Figure 8.6 illustrates this process, where Façade method *neededOperation()* is an easy-to-use operation implemented by leveraging operations in the three existing systems.

Consider a hypothetic scenario that a bank launches an advanced client program that requires accounts to maintain a high balance that must exceed certain amount. In return, customers have more benefits than they would with regular accounts. To manage such accounts, the bank may need a separate system to open/close an account, monitor the status of an account, manage benefit services, administer rewards or penalties, etc. These operations require an exclusive software interface (i.e., a façade of the new operations) to support this special kind of banking. The implementation, however, requires only a leveraging of existing components and software services. Some new development may still be expected but would be at a much smaller scale than a fresh endeavor.

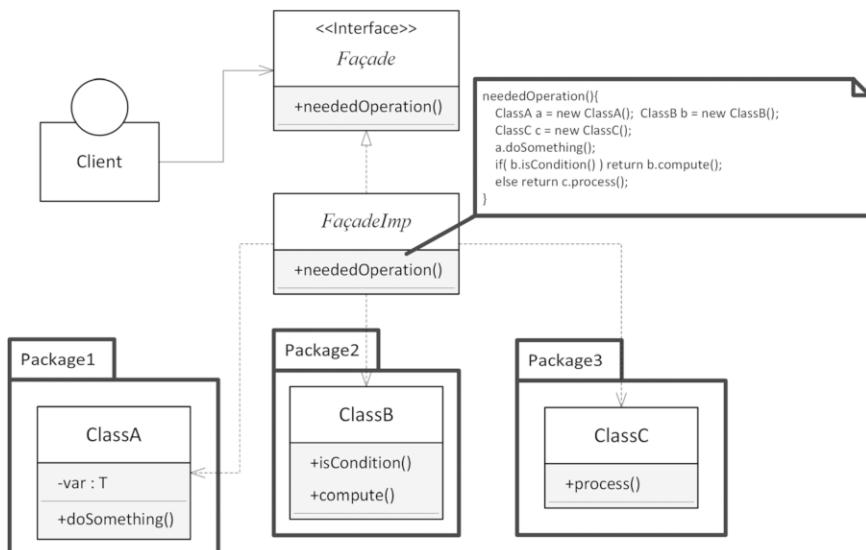


Fig. 8.6 Illustration of the Façade pattern

The Façade pattern is also used in updates of a language library to provide new APIs built on existing library capabilities. The newer Java library packages like *javax.swing* are perhaps among some of the largest façade endeavors known. These newer library APIs provide user more functionalities than did the older APIs, better interoperability of the components, and improved user experience, yet at a relatively nominal development cost.

The Façade pattern is also typically used when a simple interface is required to access a complex system, such as a straightforward interface for data scientists to access data engineering capabilities and machine learning algorithms of an artificial intelligence system. For tightly coupled subsystems, we can design a façade element with new abstractions and implementations to make modules less coupled and more cohesive. However, the pattern is not just a redesign tool. For layered software, we can use a façade module to provide an entry point to each of the layers. If a development heavily depends on microservices, we can also use façade interfaces to package the services for various application modules. By any measure, this pattern is one of the most useful design patterns.

In summary, Adapter, Proxy, and Façade are all about adaptation because of “mismatches” of certain kinds. Topologically, Adapter and Façade are similar, or Adapter is a small scale of Façade. Proxy, on the other hand, is usually more of a process necessity than a design choice. For example, the method *add* in the following class can be called a proxy method for obvious reasons, though not by design:

```
class MyList<T>{
    List<T> lst; //omitted code
    void add(T item){ lst.add(item); }
}
```

8.3.4 Decorator

This pattern provides a way to handle tasks in a hierarchical manner starting with more rudimentary operations. Figure 8.7 is a class diagram that describes all relevant elements and their relations. *BasicComponent* and *Decorator* are both derived types of *Component*, with the latter aggregating an instance of the former. The method *operation* in *Decorator* is “decorated” with that of *BasicComponent*. In other words, a decorator is a way to base an operation on operations of another component. Decorator is structurally like Proxy, but the two patterns are based on very different design intent. For Decorator, the client’s interest is in the aggregating object, whereas for Proxy, the client’s target is the aggregated object.

Reconsider the example of creating airline schedules we had earlier. If we think of a schedule as a component, then a basic component would be a schedule with only departure and arrival information. A decorator would be a complete schedule by adding a header and footer. Thus, we can print a complete schedule in the following fashion:

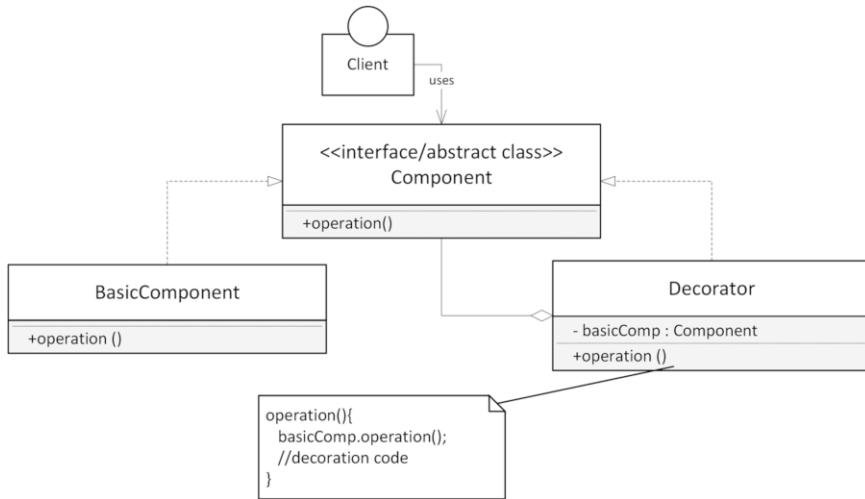


Fig. 8.7 Class diagram for the Decorator pattern

```

AirlineSchedule schedule = new ScheduleWithHeaderFooter(new BasicSchedule());
schedule.print();
    
```

Decorators can be nested (as many layers as needed). For example, if instead we create two decorators, one with a schedule header and the other with a footer, we can print a complete schedule with two layers of decoration (where *ScheduleWithFooter*, *ScheduleWithHeader*, and *BasicSchedule* are all implementing classes of *AirlineSchedule*):

```

AirlineSchedule schedule =
    new ScheduleWithFooter(new ScheduleWithHeader(new BasicSchedule()));
schedule.print();
    
```

As an option, decorators can also provide remedies to inadequate design solutions. However, nested decoration with multiple layers of decorators creates tight coupling, and all nested decorators may be modified if changes happen in the basic component.

A decorator object is analogous to a mathematical composite function $f(g(x))$ where f and g are functions of the same kind, say, functions of a single variable with real number domain and range. Function f is “decorated” with function g , making f more complex. There can be multiple layers of composition like $f(g(h(x)))$. However, when compositions become not only “deeper” but also “wider to form a composition hierarchy like this function, $f(g(h(x, l(x))), j(k(x), m(x)))$, they become analogies of objects created using the Composite pattern.

8.3.5 Composite

Certain software elements are hierarchical such as simulation of an engine, a course assignment, or the management structure of an organization. Such elements are created based on part-whole relations. For example, a course assignment may consist of several parts; each may have its own division of parts (each part is an assignment of a small scale). It is helpful, therefore, to use the same object type for all elements in the same composition so that client code can treat them consistently to execute operations that are polymorphically implemented. Though a decorator has also a part-whole relation as a basic component is part of a decorated whole, it offers only a simple “vertical” composition between the whole and its parts. Figure 8.8 illustrates the Composite pattern, a generalization of the Decorator pattern with an aggregated collection of instances of *Component*. Each instance is either a *Leaf* component or a *Composite* component, much like a root directory with multiple folders, each containing its own folders or just files, forming a hierarchical file system. A folder containing only files is a “leaf” folder, and a folder that has its own hierarchy of folders is a “composite” folder. A *Composite* object delegates part of an operation to its “child” components, and each component would do the same until an operation in a *Leaf* object is executed. Using the analogy of a hierarchy of folders, we see only one folder when the hierarchy is collapsed. If we “expand” a folder, the folder would show all its content (subfolders and files), and if we “expand” a subfolder, the subfolder would “execute” command “expand” in its own way to display its own content. The Composite pattern works the same way.

To use another analogy, a company’s CEO, managers at various levels, and regular employees are all different in terms of their responsibilities, but they can be treated simply

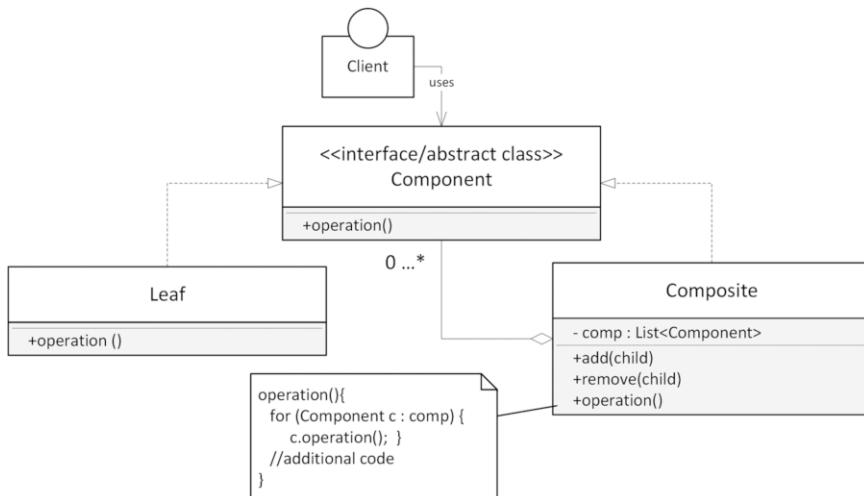


Fig. 8.8 Class diagram for the Composite pattern

as employees of the company when each simply *works* in their own capacities or as members of the society to *vote*, *donate*, or *take social responsibilities* in their own ways.

When a composite object executes an operation, its subordinate objects along the composition hierarchy would execute the operation in their own “terms” (i.e., polymorphically). This pattern supports flexibility of composing behaviorally similar elements and structural expansion of a composition hierarchy.

8.3.6 Bridge

The remaining structural patterns to be covered share a theme of “task delegation.” The most common scenario of task delegation, partially or wholly, is to delegate a task to an aggregated object or a static method, as illustrated in the following code. As discussed earlier, aggregation provides a seamless, safer, more flexible, and less coupled way to reuse existing processes (compared with approaches with inheritance):

```
class A{
    B helperObj; //instantiation omitted
    void doSomething(){
        helperObj.doPartialWork();
        C.helpStatically();
        //other code needed;
    }
}
```

The Bridge pattern is a special case of delegation where client invokes a “shell” operation of an abstraction, which delegates the task to an actual performer of the operation, idiosyncratically called “implementer.” Figure 8.9 describes a design of a remote control abstraction. Functions on a remote control, such as power toggling, setting control, etc., are implemented by a device controller (or an “implementer”) that interacts with the physical device. Similar situations happen often when a customized collection, such as a movie collection, is implemented using a library data structure. Service operations, such as *addMovie*, *removeMovie*, etc., are essentially “shell” operations (or proxies) that delegate the tasks for the “implementer”—the library data structure—to handle.

In many situations, a “shell” method may perform peripheral tasks to facilitate the work by an “implementer” such as necessary environment initialization, data conversion, or releasing the resources before exit of the operation. It is possible that an “implementer” only handles part of a task. For example, a banking operation, such as *deposit* or *withdraw*, may use a logger (as an “implementer”) to handle logging of transaction information. It is also possible that multiple “implementers” are used, such as another “implementer” to handle abnormal situations of an account as the result of a transaction (such as account balance falling below a required level or becoming negative).

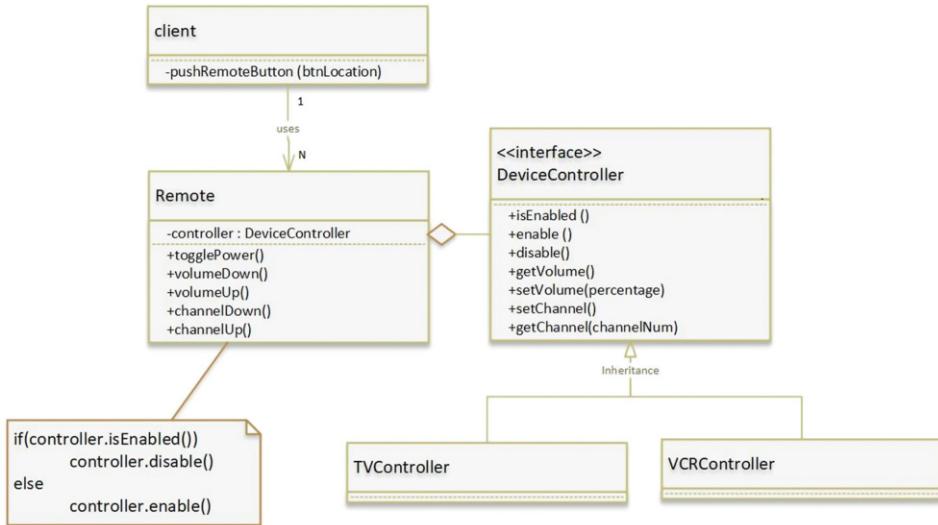


Fig. 8.9 An example of using the Bridge pattern

8.3.7 Flyweight

Flyweight, in literal sense, is a lightweight kind of a boxing competitor. As a metaphor, flyweight refers to some simple object yet to be used in large quantity in an application. For example, before computer graphics became a reality, digital images were made as console outputs, such as drawing a curve by connecting characters of “*.” These characters are flyweights, and a large number of them are needed to make a curve. The Flyweight pattern is a way to create and use lightweight objects in such a way that aliases are used if appropriate, thus avoiding creating a large number of identical, yet independent, objects at runtime that can overburden the memory. Figure 8.10 describes the structure and use of the pattern, where *FlyweightFactory* is a data structure for managing flyweight objects.

It appears that Flyweight, creating copies of lightweight objects, might be a creational pattern with some similarity to the Prototype pattern. They have different design intent. A prototype object is expensive to create and (generally) mutable. Therefore, the issue is how to make a creational process “cheaper” yet as effective. In contrast, a flyweight object is small and often immutable (thus, aliases are safer to use) but needed more frequently. For example, a data conversion tool is needed frequently in a process. Therefore, the issue is to find a more economical process of using a large number of immutable data or services. Thus, *FlyweightFactory* can be considered as a service provider (as opposed to an object creator). In this sense, *Flyweight* is more appropriate to be a structural pattern.

Consider a collection of windows (plain, with scrollbars, with menus, etc.) for runtime use by making copies of them. It would be a prototype collection because such objects are costly to create but cheaper to copy (yet still independent). In comparison, consider a

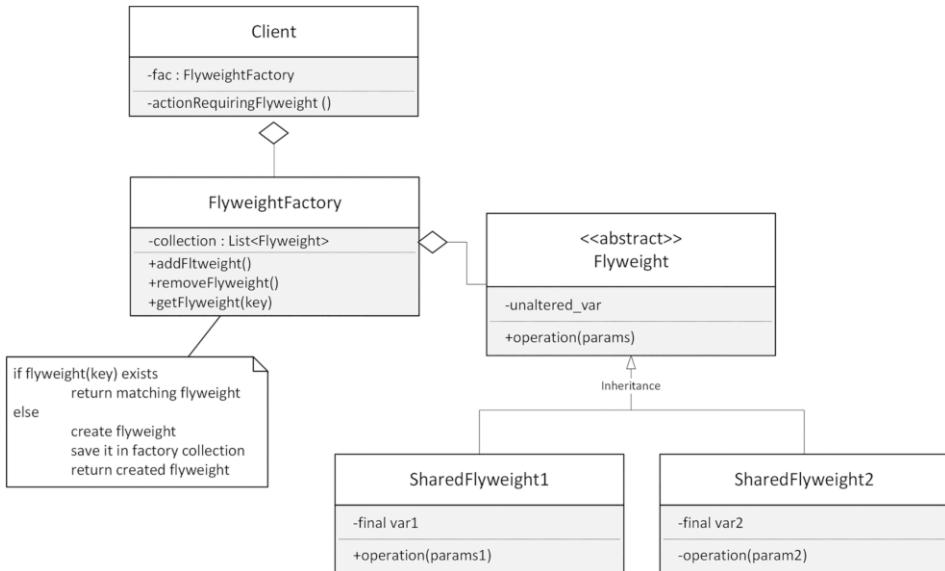


Fig. 8.10 Constructs of the flyweight pattern and their use

collection of plotting routines (for curves, surfaces, histograms, etc.). It would be a flyweight collection as such services use internal memory to store data and are thus expensive on memory usage. It makes sense to create such a service object, store, and reuse (the same object but the same service with a different object).

Implementation of a flyweight factory can vary. Common concerns include caching options, concurrent access of cached flyweight objects, and how client may access a service object. Though a flyweight object is conventionally immutable, the use of mutable extrinsic parameters can often provide a customized use of a flyweight object. Construction of flyweight objects can also use external resources to create the components of a flyweight object, but such resource gathering is only triggered when an object is requested that requires the resources to create. Therefore, a flyweight factory may have a Façade interface to facilitate creation of flyweight objects by offering easy utilization of the underlying (potentially complex) resources. Finally, a flyweight factory is commonly created as a singleton to allow global access.

8.3.8 Twin

The *Twin* pattern is another interesting instance of delegation. In simple terms, the pattern is about two objects aggregating each other symmetrically as shown in Fig. 8.11. This is beneficial in some applications. Consider animating a ball object with a separate thread. A common approach is to manipulate the ball in a thread's *run* method. However, we could

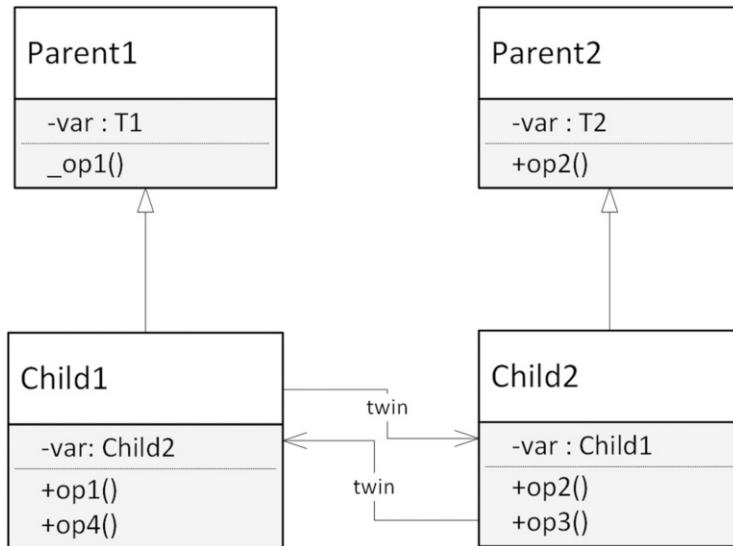


Fig. 8.11 The Twin pattern illustration

create a pair of threads, one controlling pause and resume of ball movement while the other managing ball movement and drawing the ball as needed. By making them twins, one thread can access the operations of the other, making animations more interesting.

Another use of Twin is to gain some benefits of multi-inheritance in single-inheritance semantics. As shown in Fig. 8.11, through an aggregated instance of *Child2*, *Child1* can access data fields of *Parent2*. In the same fashion, *Child2* can access data fields of *Parent1*. If *Child1* needs operations that are considered as overrides of some operations in *Parent2*, *Child2* then can provide the overrides for *Child1* to use. Thus, loss from direct inheritance can be partially recovered by using Twin objects. Type flexibility of multi-inheritance is not completely lost either. For example, if type *Parent1* is expected, we use an instance of *Child1*, yet to access operations of *Parent2* as if it were an instance of *Parent2*. Nonetheless, the Twin pattern provides only a workaround if multi-inheritance is not supported.

8.3.9 Section Summary

When clients want to complete a task using software resources, they may face different obstacles, among which:

- Precondition mismatches
- Unfamiliar operational environments of the resources
- Effectiveness of using the resources due to complexity

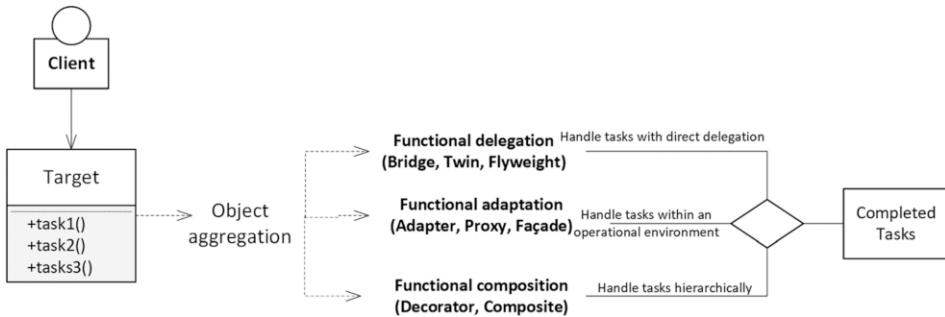


Fig. 8.12 Summary of the structural design patterns

- Constraints of their own operational environments
- Uncertainty about functionalities to use in the resources
- Uncertainty about appropriate ways to use the resources

The structural design patterns provide solutions to overcome some of these obstacles. To complete a task, a client interacts with a design pattern abstraction, which aggregates a resource-handling object that helps complete the task by doing one of the three things—direct delegation, delegation with environment adaptation, or delegation through a composition hierarchy. This mechanism is described in Fig. 8.12.

Structural patterns (and design patterns in general) are solutions to only some of the design scenarios we may encounter. When we are facing a design decision, it is thus more helpful to think about a solution in terms of the ideas behind the design patterns (such as the three task-handling design strategies summarized in Fig. 8.12), as opposed to a particular design pattern that we might use. For example, Java’s *MouseAdaptor* allows user to override only interested methods from the interface *MouseListener* with unused methods hidden. This is also an instance of adaptation to facilitate an overriding process, but not covered by any of the structural patterns.

Functional delegation and adaptation are among the oldest design ideas. In fact, virtually all structural patterns have their procedural counterparts by removing their object-oriented disguise. Aggregated objects can be replaced by direct method invocations (with methods possibly packaged in modules). For example, with the same idea behind the Decorator pattern, we can design a standalone method to construct a “decorated” process as follows:

```

static IProcess getDecoratedProcess(IProcess p){
    /* some necessary processing preparations */
    return new IProcess(){
        void doProcess(){
            /* implement an enhanced process by using
            p.doProcess(), which provides a basis for the process */
        };
    }
}
  
```

The use of the abstraction *IProcess* in the code above is not essential; however, we could use a lambda function instead. In the same vein, the idea behind the Composite pattern is used in the following method to recursively process a composition hierarchy:

```
static void processData(DataType data){
    if(data.getChildren() == null) return;
    else{
        for(DataType d : data.getChildren()){
            /* ... operation(d); .... d.doSomething(); ... */
            processData(d);
        }
    }
}
```

We might think of structural patterns even more broadly as “patterns of bridging.” A client “agent” (an object or a method) bridges a task directly to an implementation source, does some work of adaptation to a source’s environment before bridging, or bridges a task along an object composition hierarchy.

Finally, a side effect of structural patterns is that if task delegation is pervasive by design or by accident, it could lead to tight coupling between “delegators” and delegation “receivers.”

8.4 Behavioral Design Patterns

How an object behaves is controlled by its instance methods, which mutate instance data. The value of object orientation is to enable objects’ behavioral alteration at runtime. If object orientation is only marginally beneficial for structural patterns, it is crucial in behavioral design patterns. Behavioral patterns take full advantage of polymorphism by introducing runtime dynamics of instance data objects or by altering the behavior of an instance method at runtime, thus altering a system’s behavior as needed. The behavioral patterns can be divided into two groups:

- To alter object’s behavior at runtime
- To reduce coupling in object communications

8.4.1 Strategy, Servant

This pattern is perhaps the simplest in the group. It is nothing but an interface with an abstract method that delivers a solution strategy. For example, the following code displays a row of certain character using the pattern:

```
Interface LineStrategy{
    Char getChar();
    int size();
}
static void printRow(LineStrategy obj){
    char c = obj.getChar();
    for (int k = 1; k < obj.size(); k++) { System.out.print(c) + " ";}
}
```

In this simple case, a strategy is about the length of a line and a character to use to tile the line. The behavior can be dynamic if a factory is used to dispatch an instance of an implementation of *LineStrategy* at runtime. Consider sorting an array of names (i.e., strings); we can use a library routine of Java with a parameter object of *Comparator*, which is an abstraction about a “sorting strategy” with an abstract method *compare* to specify an order of sorting (based on the length of a string):

```
Arrays.sort(names, new Cpmparator<String>(){
    public int compare(String str1, String str2){
        return str1.length() - str2.length();
    }
});
```

Recent language updates have often added lambda functions and expressions. If dynamic dispatch of strategy objects is not needed, then we can simply use a functional parameter as a strategy in a higher-order function. Thus, the code above is equivalent to:

```
Arrays.sort(names, (a, b) -> a.length() - b.length());
```

The Servant pattern describes a situation where a (served) strategy is used in multiple (servant) methods of an object, providing a broader context for a strategy to apply (illustrated in Fig. 8.13). As a concrete example, consider a retailer application with many servant methods for determining the prices of the things for sale. Suppose a price is determined by the direct cost for an item and a markup to cover indirect cost and profit. Thus, the retailer needs a markup strategy (which may change at any given time). The following interface creates an abstraction for a markup strategy:

```
interface MarkupStrategy{
    // returns a markup percentage
    double getMarkup(double itemUnitCost);
}
```

A servant method computes the unit price of a sales item, where an instance of *MarkupStrategy* is used based on an implementation of the interface, such as *TelevisionSetMarkupStrategy* or *CellphoneMarkupStrategy*.

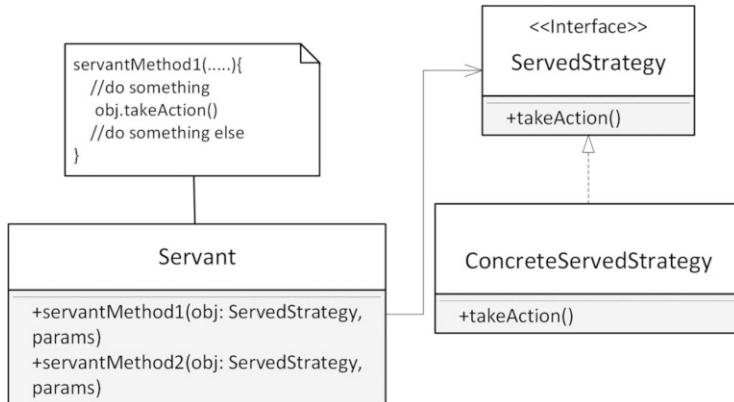


Fig. 8.13 Illustration of the Servant pattern

Often, a strategy object is associated with a context object. For example, a priority queue needs a sorting strategy to determine the order of the elements. In other words, a priority queue is a context that a sorting strategy is associated with. Or a strategy object (or function) is simply a “decision-maker” a context relies on.

8.4.2 Command

This pattern describes a context that an aggregated “command” object is used in. The context delegates a command task to a “receiver” to execute the command (partially or wholly), as described in Fig. 8.14. However, the relational topology of the pattern is nothing but delegation we have seen in structural patterns. Particularly, it might remind us of the Bridge pattern, where “implementer” now would play the role of a “receiver.” The difference is again the design intent. A client of Bridge targets an operation, whereas a client of Command needs to control the behavior of a context through various commands. For example, a stock-trade software application may have an abstract entity *Stock* that aggregates an instance of *Command*, for which entities like *Buy* and *Sell* would be implementing classes. A “receiver” object might represent a stock brokerage firm that executes “buy” or “sell” on behalf of a client. In this case, a stock object provides a context for an instance of *Command*. A “buy” or “sell” command would change the number of shares (a property) of a stock.

An elevator object also provides a context for an aggregated instance of *Command*, which can be implemented with concrete command entities like *Up*, *Down*, *Stop*, *OpenDoor*, *CloseDoor*, *StandStill*, etc. An elevator would behave differently by executing such a command. Each command is executed by a “receiver,” which is likely the control

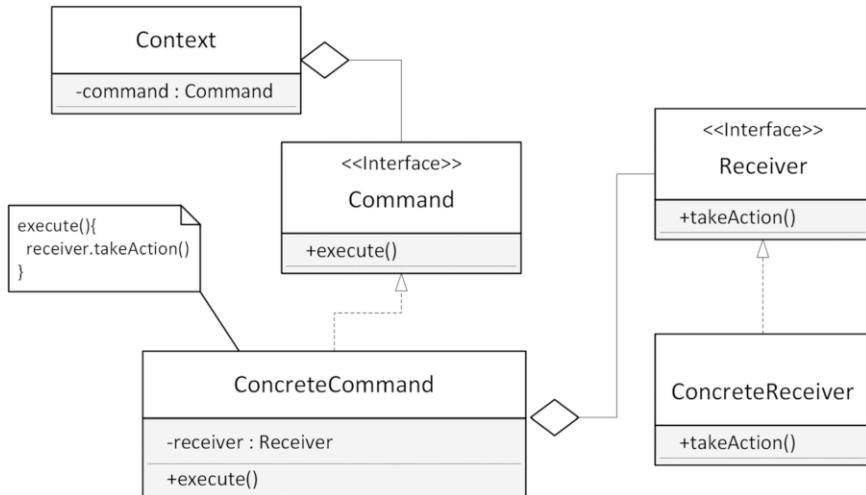


Fig. 8.14 Illustration of the Command pattern

unit of the elevator software to interact with the mechanics of an elevator. An elevator command can be “activated” by a floor or an elevator button. A command would in turn change the state of an elevator. For example, “close door” command may change elevator state from “standstill with door open” to “standstill with door closed,” which may subsequently be changed to “moving up” state by an “up” command. Thus, a command changes elevator behavior.

8.4.3 State

An instance variable may also be called a state variable because a state of an object is a view of the current values of the instance variables. Some objects may have explicitly observable states, such as an elevator or a car, with explicitly defined state variables to be associated with the observable states. This pattern offers an object-oriented approach to simulating states of any contextual object using a state object. Practically, if a state machine diagram works well to describe system behavior, the State pattern can be used to provide an “executable” model of the system operations. Figure 8.15 illustrates the object types involved in this pattern. Note that instance methods of a state object must be able to access the contextual information through a context object (i.e., method *takeAction* has a parametric object of *Context*). This is important because operations of *State* then can use the contextual information to appropriately implement transition protocols and set a new instance of *State* when protocols are met.

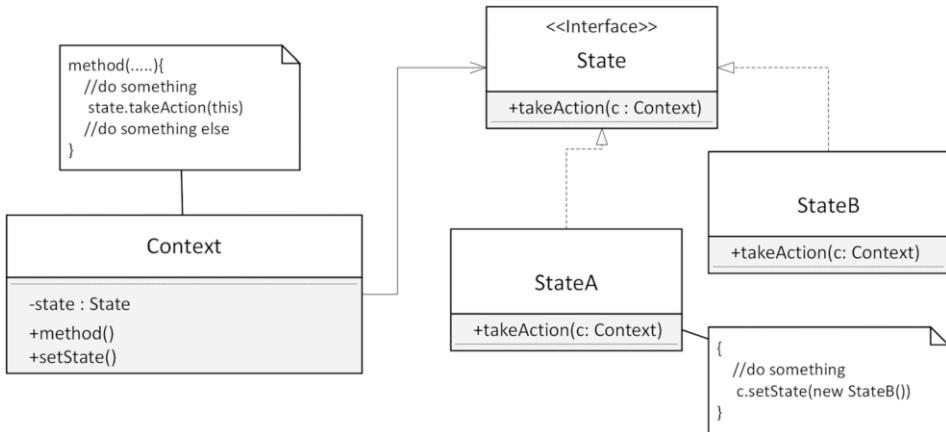


Fig. 8.15 Illustration of the State pattern

Consider elevator operations again, and suppose elevator is the context for using the State pattern:

```

class Elevator{
    State elevatorState;
    ...
}
  
```

The abstraction *State* may have implementing classes like *MovingUp*, *ArrivedAtFloor*, *DoorClosed*, *DoorOpen*, etc. As an example, assume elevator is currently at state *DoorClosed*. With availability of an elevator's context, an implementation of *DoorClosed* might look like:

```

class DoorClosed implements State{
    takeAction(Elevator e){
        int x = e.requestFromFloor(), y = e.requestFromElevator();
        if(e.isDoorClosed()){
            if(x < y && x > currentFloor){
                e.goUpToFloor(x);
                e.setNextState(new MovingUp());
            }
            else{ .... }
        }
    }
}
  
```

An implementation of *MovingUp* might look like this:

```
class MovingUp implements State{
    takeAction(Elevator e){
        e.showElevatorStatus();
        int fl = e.nextFloorToArrive()
        if(e.isFloorArrived(fl)){
            e.turnOffLitButtons();
            e.setNextState(new ArrivedAtFloor(fl));
        }
    }
}
```

Figure 8.16 is a state diagram that describes a cycle of elevator state transitions. The code above illustrates how the state boxes can be implemented.

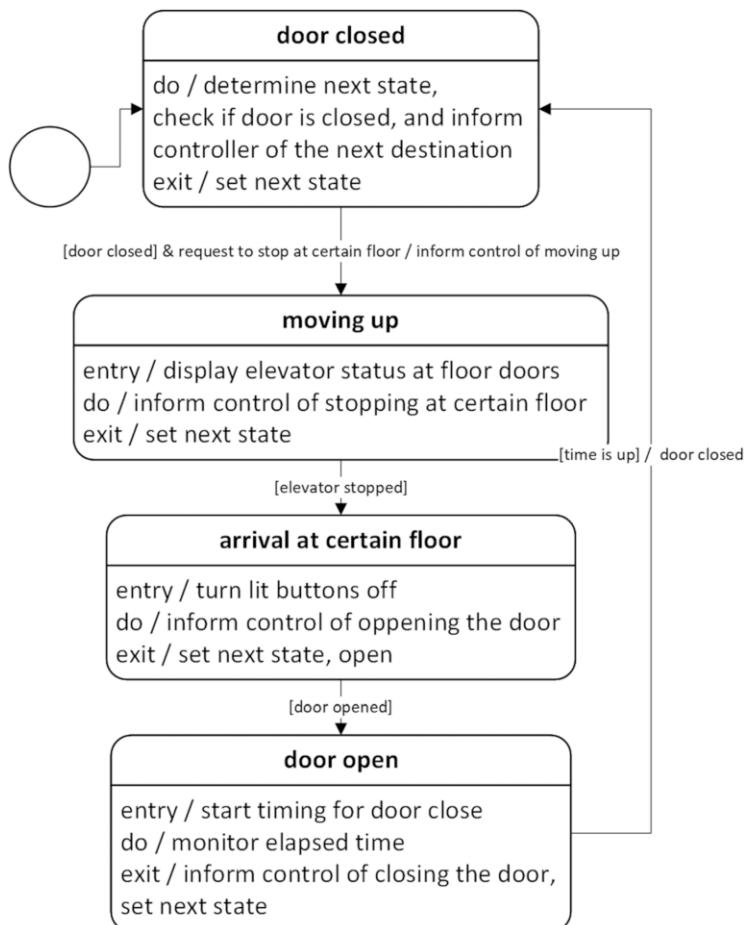


Fig. 8.16 Elevator operations using the State pattern

The elevator example has shown that the State pattern can be used as a simple and clean architectural design style for applications that can be effectively modeled with a state machine diagram. This pattern can also work well in applications when “states” can be effectively defined:

- For a self-checkout process at a grocery store, we might organize transition conditions, user events, and system actions around a set of states like “scanning items,” “processing payment,” and “finishing transaction.”
- A postal-package tracing system may also be implemented by structuring software operations around the states like “mail initiated,” “mail in transit,” “mail arrived in destination city,” and “mail delivered.”
- In certain applications, we may use a state object to render different user interfaces or allocate different resources for users of different kinds with transition protocols based on interactions with users.

The State pattern supports the design notion that state-specific behavior should be independent and adding new states should not affect the behavior of existing states. However, adding or removing states may invalidate state transitions. To address the issue, we can use an abstraction such as the following to allow transition details to be specified in an object dispatched at runtime:

```
interface Transition{  
    boolean transitionCondition(Context c);  
};
```

The three behavioral patterns discussed above are structurally similar using a delegation object. We could view the patterns from a “strategy” viewpoint:

- An operation can be supported by an “operational strategy.”
- An object’s behavior can be controlled by a “command strategy.”
- A system can operate on a “state strategy.”

In this way, these patterns could be understood (and remembered) as “strategy-oriented” patterns. A context object would determine which pattern is appropriate to use. The Strategy pattern is not usually architecturally significant. The Command pattern, however, may impact an architecture given a right scenario such as a GUI-oriented system to be modeled around a set of commands. In comparison, the State pattern is often applied architecturally to a subsystem or even to the entire system.

8.4.4 Visitor

This pattern can help with the situation when an object may have resources but not processing capabilities to offer needed operations. For example, a used car entity may have all the information about a car but not an ability to assess its value. Thus, car value assessors, as visitors, can be invited to perform used car value assessment. These visitors are car value assessment experts with specialties in different car models.

By design, the primary abstraction *Visitor* has a set of overloaded operations (or “expert” modules), each representing a “specialty” and working only with a subset of objects (e.g., cars of certain kind) in a contextual space. Correspondingly, a contextual object only accepts a visit by an “expert” with the right “specialty.” To ensure adequate information about an object, a visiting “specialist” must be “granted” an access to a hosting object’s properties and operations. Figure 8.17 illustrates the pattern as it is applied to car value assessment scenario. Values of used cars may be different in different regions of a country; therefore, multiple implementations of *Visitor* are possible.

For another example, consider a student object that may have the information such as semester and cumulative grade point averages but not be able to assess whether a student is in good academic standing. The following interface creates two visitor methods to perform assessment on a student’s academic status. Note that a “right” visitor method is always selected given a contextual object:

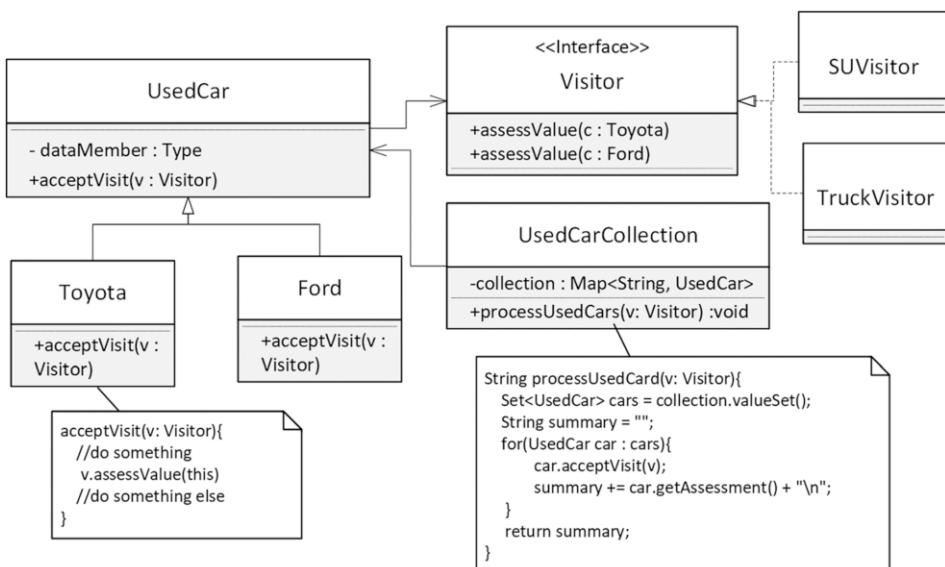


Fig. 8.17 Illustration of the Visitor pattern

```

interface AcademicStatusVisitor{
    String assessAcademicStatus(GraduateStudent s);
    String assessAcademicStatus (UndergraduateStudent s);
}
class GraduateStudent implements Student{
    //data fields and other methods omitted
    void sendAcademicStatusLetter (AcademicStatusVisitor visitor){
        String status = visitor.assessAcademicStatus(this);
        if( status.equals("probation") ) sendProbationLetter();
        else if(status.equals("suspension") ) sendSuspensionLetter();
        else { ... }
    }
}

```

One of the exercises in Chap. 5 introduced “double (operation) dispatch” technique for object interaction. It addresses a general problem that message dispatches may depend not only on the receiver but also on the arguments. The effect of a “double dispatch” operation call depends on the runtime types of two objects involved in the call. This technique is also demonstrated in the Visitor pattern. The following statement is the first operation dispatch:

```
student.sendAcademicStatusLetter(academicStatusVisitor);
```

The second dispatch would be:

```
academicStatusVisitor.assessAcademicStatus(the student object);
```

The runtime effect is that object *student* would determine which *visitor* object to interact with. In general, when operations appear to be symmetrical among objects, “double dispatch” can be used to avoid complex conditionals using polymorphism.

The Visitor pattern can be useful also in the following situations:

- An object can use a visitor to alter its properties externally and hence change its behavior.
- We can use visitors to add new operations or modify existing ones of an object to avoid more consequential approaches such as subclassing.
- If an algorithm or a process has multiple versions depending on the type of data it uses, a data source context can use an algorithmic visitor for data sorting, searching, etc.

Finally, adding another method of *Visitor* or removing one (due to exclusion of certain contextual objects) results in a modification to the *Visitor* abstraction. Consequently, existing implementing classes of *Visitor* must be modified as well. Therefore, the Visitor pattern can be inefficient to use when an inheritance hierarchy is unstable, especially when subclasses are frequently added to the hierarchy.

8.4.5 Iterator

The iterator pattern is commonly integrated into all library data structures and provides implementation mechanism for a language's newer constructs such as "foreach" loops. The pattern was motivated by the belief that elements stored in a data structure should be accessed and traversed without exposing the data structure's representation and traversal mechanism. Thus, data retrieval (and updates when appropriate) should be defined independently of a data structure itself. Figure 8.18 depicts the two abstractions the pattern is typically associated with. A data structure would implement the interface *Iterable* and provide an implementation of the method *iterator*, which returns an instance of *Iterator*—an abstraction that specifies how an iteration starts, advances from one element to the next, and terminates. As appropriate, modification to a data structure to add or remove elements along an iteration path can also be provided through the interface. The following code describes how the two abstractions are used, with the origin of the data source hidden behind a factory:

```
Iterable iterableObj = IterableFactory.getInstance().getIterable(identifier);
Iterator it = iterableObj.iterator();
while( it.hasNext() ){
    Object item = it.next();
    //do something with item
}
```

To allow easy access to data elements and still provide structural separation, an inner class (relative to a data structure class) can be used to implement interface *Iterator*. External

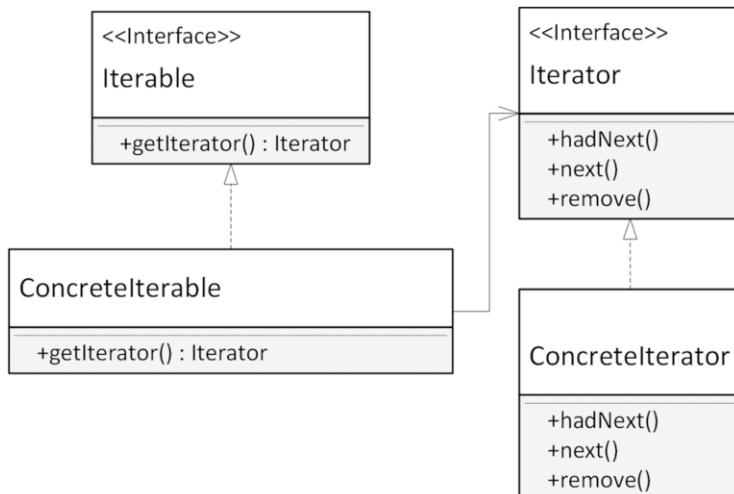


Fig. 8.18 Illustration of the Iterator pattern

implementations of *Iterator* are possible if the data structure also provides native methods for accessing its data elements.

It is possible to skip *Iterable* and make a data structure an implementing class of *Iterator*, shown in the code below:

```
Iterator it = new DataStructure();
```

A side effect is that the operations of a data structure are then exposed to a client of *Iterator*—a hazard for unintended mistakes. Besides, skipping *Iterable* would also invalidate the use of “foreach” loop, as such loops often depend on the *Iterable* interface. For example, *list* in the following code must be an instance of *Iterable*:

```
for(String s : list){
        System.out.println(s);
}
```

Index-based retrieval of data elements appears convenient and more straightforward (especially when underlying data storage is an array), but data retrieval based on the iterator pattern provides a safer, more flexible, and extendible data access without adding much operational overhead. Newer languages, such as Python, may even have eliminated index-based loops entirely in favor of *Iterable*-based (foreach) loops so that an explicit use of the iterator pattern becomes unnecessary. This is an exemplary example of separation of concerns in data management.

The iterator pattern may have been inspired by constructs from block-based or functional languages like APL and Haskell and is behind many data iteration tools in newer languages and introduction of powerful new constructs in older languages. Python uses functions to create iterators for efficient looping and standardize a core set of fast, memory efficient tools that are useful by themselves or in combination with other tools. This tool set forms an “iterator algebra” for constructing specialized, efficient, and often succinct operations. For example, *sum(map(multiply, vector1, vector2))* would compute the inner product of two vectors (where *map* returns an iterable list).

8.4.6 Chain of Responsibility

This pattern defines an iteration over a collection of processing units for a coordinated completion of a task. In fact, the pattern is nothing but a linked list of processing nodes. To see the connection, recall the following object type representing a data node of a linked list:

```
class Node<T>{
        Node next;
        T dataItem;
}
```

The instance field *next* is a self-referential variable, essential to how a linked list works (similar self-referential structures were also seen in the *Decorator* and *Composite* patterns). If *Node* is modified to include operations, it becomes a “processor” entity (as opposed to a “data” entity) capable of processing a request:

```
abstract class Node{  
    Node successor;  
    void handleRequest(Request req);  
    Node getSuccessor(){ return successor; }  
}
```

A request is passed along a chain (i.e., a linked list of processing nodes); each participating node is responsible for processing a request in its own way and then passing the request to the successor. A context object containing a chain is essentially a data structure with methods to add or remove nodes but, most importantly, with methods to instantiate a request and pass it to the chain, as shown in the code below, where *head* refers to the head node of the chain:

```
void processRequest(Request r){  
    Node cursor = head;  
    while( cursor != null ){  
        cursor.handleRequest(r);  
        cursor = cursor.getSuccessor();  
    }  
}
```

Consider a store operation to apply discounts to items in an order. In this case, an order (as a “request”) would be passed to a chain of discount applicators, each applying a discount on items that may or may not exist in the order.

One can use an array-based list of processing nodes to accomplish the same goal, but a linked list is known for its flexibility of adding and removing nodes. Other benefits include:

- Use a context object (containing a chain of processors) to decouple between the sender of a request and the processors.
- Provide flexibility of processing a request when the nature of a request, the kind of processors needed, or an order of handling the request is undetermined before the runtime.

A potential drawback is a situation when there are many processors in a chain with only a few being able to take actions for any given request. In such situations, software overall performance could be inadvertently affected.

8.4.7 Observer

Software operations are often designed to respond to user, system, or external events such as approval or disapproval of a credit card payment, verification status of a user login, a button-clicking by a user, or simply the status of a progress bar. There are system operations that are designed to respond to such events. When events occur, these operations must be able to respond to the events and handle them in their own ways. This is essentially how virtually all event-driven computing models work in software. Event-handling operations would register with an event monitor to become event “observers” and be notified (by the monitor) when events (they are also called “observables”) occur. The Observer pattern, or a variant, is used for implementing an event-response model to provide decoupled communication between events and their handlers. This decoupling is particularly beneficial when data becomes available or events occur randomly during runtime due to varied communication protocols or response delays of the input devices.

Event-driven programming models programming languages provide are frameworks for building event-driven applications. These frameworks may well be implemented based on the Observer pattern. The pattern manages a loosely coupled one-to-many relation. When an event object changes its state, an open-ended number of dependent objects are informed to take (synchronous or asynchronous) actions as appropriate. Figure 8.19 illustrates the structure of the abstractions used in this pattern.

A context object, responsible for managing the mechanics of the Observer pattern, can be an event monitor or simply an event object itself. This context entity is essentially a data structure with methods to add (i.e., to register) and remove observers and manage communication with registered processes when an event occurs. As illustrated in Fig. 8.19, when

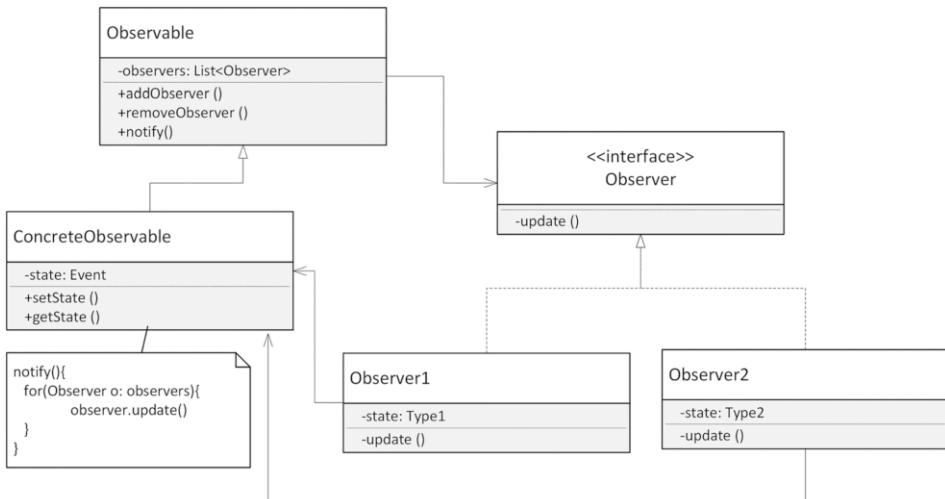


Fig. 8.19 Relations of the abstractions in the Observer pattern

an event's state is changed, it triggers operation *notify*, which subsequently sends messages to registered observers to update their own states.

The pattern can be used to implement certain social networking features. For example, when a user has just uploaded a photo, written a post, or made a comment at a post, all user's connections are "informed," and their home sites are "updated." A user's personal site is an observable entity, and all connections of the user are observers.

In event-driven programming, users interact with a system through graphic widgets like buttons or textboxes. These are event-enabling elements that observers are registered with. Thus, widgets are event monitors too. Observers are typically called "listeners" (to listen to broadcast of event occurrences). The following code describes the process of registering a listener with a button:

```
button.registerListener(new ActionListener(){  
    public void actionPerformed(ActionEvent e){  
        //code to handle event  
    });
```

The listener (or observer) is an instance of interface *ActionListener*. This interface has one method *actionPerformed*, which we use to implement a response to a button-clicking event. The following code describes, in principle, the underlying object communication when a button is pushed. *EventSource* is a monitor, and operation *execConsoleInputEvt* triggers an input event and notifies observers to update what is needed (in implementations of the interface *Observer*):

```
interface Observer {  
    void update(String event);  
}  
class EventSource {  
    private List<Observer> observers = new ArrayList<>();  
    private void notifyObservers(String event) {  
        observers.forEach(observer -> observer.update(event));  
    }  
    public void addObserver(Observer observer) {  
        observers.add(observer);  
    }  
    public void execConsoleInputEvt() {  
        Scanner scanner = new Scanner(System.in);  
        while (scanner.hasNextLine()) {  
            String line = scanner.nextLine();  
            notifyObservers(line);  
        }  
    }  
}
```

Java also provided some native support for customized use of the Observer pattern with two abstractions: *Observer* and *Observable*. *Observer* is the same interface above, and *Observable* is a concrete class, like an event monitor, with several methods to manage observer objects including the method *notifyObservers*. Although this framework is

convenient for applying the pattern, it didn't provide a rich enough event model to suit a variety of applications and has been deprecated since Java 9.

A monitor in the *Observer* pattern can be viewed as a “broker” to provide loose coupling between a subject and those that are interested in the subject. We will discuss more broker-based communication structures later.

8.4.8 Mediator

In comparison with the Observer pattern, the Mediator pattern provides a more conventional “broker” structure to mediate communications among clients. An instance of *Mediator* aggregates a collection of clients, and all client objects share a reference to *Mediator*. When a client issues a message, the mediator object notifies all clients, but only one who is the receiver of the message responds. Figure 8.20 illustrates the abstractions involved and the communication process. The design idea behind the Mediate pattern is that all client objects should be decoupled from each other. It might be interesting to note that when one sends a message, a client that the message was intended provides a real-time response (by setting a response with sender object). In comparison, communications in the Observer pattern are always one-directional from a monitor to observers. When helpful, a mediator object can also maintain a communication log.

This pattern can be used in a chat room application to broker communications among the users, where a messaging board could serve as a mediator. Each user receives a message every time one user sends a message, but a response from a receiver to the sender is not

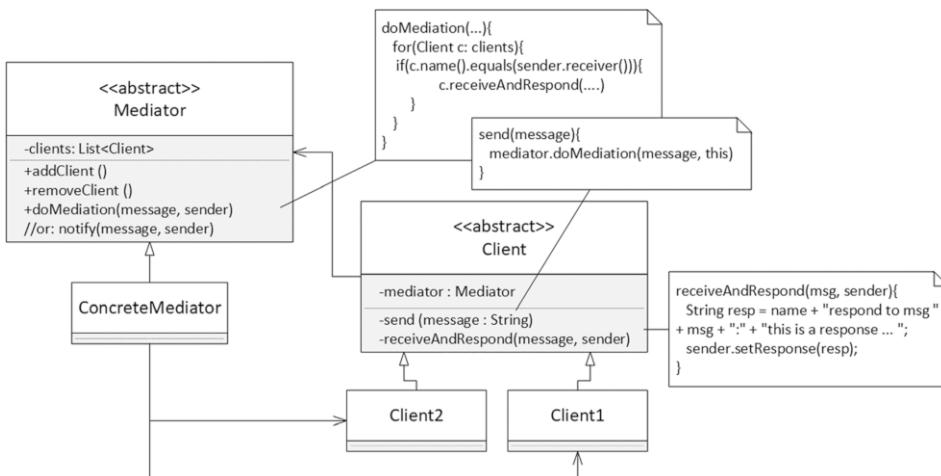


Fig. 8.20 Illustration of the Mediator pattern

required. Therefore, a chat room application is more appropriate for remote communications. The social networking scenario mentioned earlier could also be implemented using this pattern where a mediator maintains information needed to create users' contents. When a user creates a new piece of content through the mediator, the mediator checks validity and security of the piece and, as appropriate, updates the contents of the user's social connections.

8.4.9 Section Summary

All behavioral patterns can be summarized in a single but general code statement:

```
context.contextualMethod();
```

Object *context* provides a context to which the pattern is applied. For half of the behavioral patterns discussed above, the context is an integral part of the pattern. But for the other half, a context is not specified; these patterns, therefore, can be applied to a variety of contexts. A contextual operation can be either directly related to the pattern or a method that depends on the effect of the pattern. Table 8.1 elaborates a context object and a contextual method for each of the behavioral patterns covered:

Table 8.1 Contextual object behavior for each of the behavioral patterns

	Contextual Method	Method Detail
Strategy	<i>context.opWhereStrategyUsed()</i> Or: <i>do(s :Strategy)</i>	... <i>s.applyStrategy(); ...</i>
Command	<i>context.opWhereCommandExecuted()</i>	... <i>command.execute(); ...</i> <i>where:</i> <i>execute(){ ... receiver.act(); ... }</i>
State	<i>context.opLeadingToStateChange()</i>	... <i>state.act(context); ...</i> <i>where:</i> <i>act(c: context){ ... c.setState(new</i> <i>NewState()); ... }</i>
Visitor	<i>context.acceptVisit(v : Visitor)</i>	... <i>v.visit("this" context)...</i>
Iterator	<i>iterableContext.getIterator()</i>	<i>while(it.hasNext()){</i> ... <i>it.next() ... }</i>
Chain of Responsibility	<i>contextualChain.processRequest(Request r)</i>	<i>while(p != null){</i> ... <i>p.act(r) ...</i> <i>p = p.nextProcessor(); ... }</i>
Observer	<i>ObservableEvenContext.notify(event)</i>	<i>for(o : observer_list){</i> ... <i>o.update(event); ... }</i>
Mediator	<i>contextualMediator.doMediation(msg, sender)</i>	<i>for(c : client_list){</i> ... <i>c.respond(msg, sender); ... }</i>

8.5 More About Object Communication

The last two behavioral design patterns are about ways to provide loose coupling of object communication in one-to-many or many-to-many situations. The Observer pattern uses an implicit communication “broker” (an event source) because communication is one way, hence simpler. The Mediator pattern uses an explicit mediation object to manage bi-directional communications between clients so that the two client ends do not depend on each other for communication details. Nevertheless, there is still some communication “tightness” in both patterns. In the case of the Observer pattern, frequent updates (in response to, for example, a stream of messages or a series of *repaint* calls) may cause actions to become unresponsive. Similarly, brokering with a mediator object on one-to-one (real-time) communications can be costly when there may be too many communicating clients present. It is even more so when a mediator must be accessed in a multi-threading environment where access must be controlled and synchronized. To reduce the frequency of communication, a time interval with a fixed duration can be used, in which communication would pause. For communications with a mediator, a suspension of communication may require a temporary storage to save messages and senders and then retrieve the information when communication resumes. Besides, it is not always easy to determine the amount of communication delay that would not impact the application in any essential way.

If real-time communication is not required, then asynchronous messaging is an effective way to decouple senders from receivers and avoid blocking a sender to wait for a response. Each receiver has a dedicated message queue and retrieves messages asynchronously. Using dedicated message queues, however, does not scale applications. Besides, with messages to be delivered to right message queues, a sender is also aware of a message receiver (though indirectly).

A classical communication structural pattern proposed in the late 1980s, known as the Publish-Subscribe pattern (Birman and Joseph, 1987), can also be used to decouple senders (called publishers) and receivers (called subscribers). However, messages are categorized, and subscribers subscribe to messages only in certain topics or certain contents (much like people subscribe to YouTube channels or follow others’ Twitter posts). Subscribers can only subscribe to one or more categories of messages through the broker (without the knowledge of who would publish the messages) and retrieve messages asynchronously. Figure 8.21 describes this communication model (though in a simplified scenario). The abstraction *MsgBroker* maintains a collection of subscribers identifiable with a message topic key (or with a set of filtering criteria). To start a communication, a publisher sends a message (in a category) through a bus to the broker. The broker does some filtering and routes the message to all category subscribers through a channel (broker may prioritize messages in a queue before routing). Finally, the subscribers receive the message through that channel. This communication looks like one with the Mediator pattern, but the difference is that subscribers have no knowledge who the sender is. In comparison, an

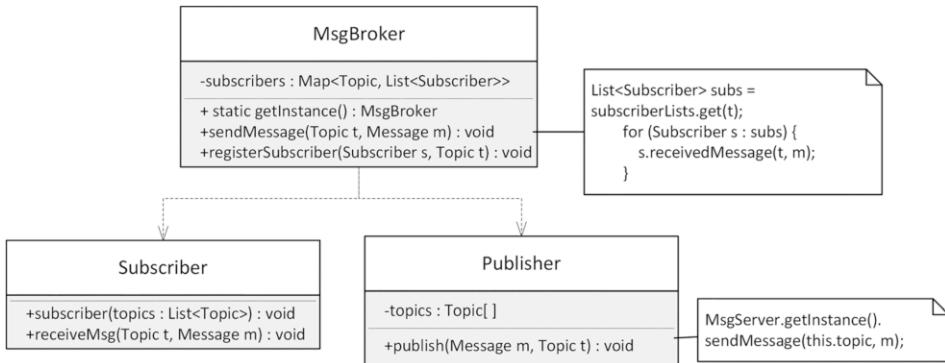


Fig. 8.21 Illustration of Publish-Subscribe communication pattern

instance of *Mediator* would forward not only the message but also the sender (thus, a “real-time” response to the sender is possible).

To apply this model to the scenario of real estate sales, buyers and sellers are both subscribers to messages categorized by, say, an MLS (Multiple Listing Services) identification number. However, a sender would not be able to receive “real-time” responses and receive a response asynchronously whenever the receivers choose to respond. This model would work for scenarios appropriate for the observer pattern too, in which case, subscribers are grouped based on events and an event broker is also responsible for monitoring event occurrences.

The primary benefit of the Publish-Subscribe model is loose coupling between publishers and subscribers that supports scalability. Publishers do not need to know even the existence of their subscribers. With the topics (or contents) being the focus, publishers and subscribers are allowed to remain ignorant about system topology. Each can continue to operate independently of each other. It improves responsiveness of the sender and hence the scalability of the communication. However, making sender and subscriber completely unaware of each other may also be responsible for some of the issues with the model such as reliability of message delivery, among others. Such a messaging system would put a significant pressure on messaging infrastructure building for secure, responsive, and reliable delivery of messages to interested subscribers. Nonetheless, this Publish-Subscribe model can often be considered when the Observer and Mediator patterns are in question.

An object *a* communicates with an aggregated object *b* for functional delegation. However, aggregation may introduce unnecessary coupling if *a* merely needs certain information that *b* collects at runtime. Passing *b* as a parameter to retrieve information would be not good for couple either when most of what *b* does is not needed in *a*. Instead, consider the following interfaces:

```

interface IDeposit<T>{
    void depositInfo(T data);
}
interface IRetrieval<T>{
    T retrieveInfo();
}

```

We then create the following implementation of both interfaces (illustrated in Fig. 8.22) and pass an object to both *a* and *b*:

```

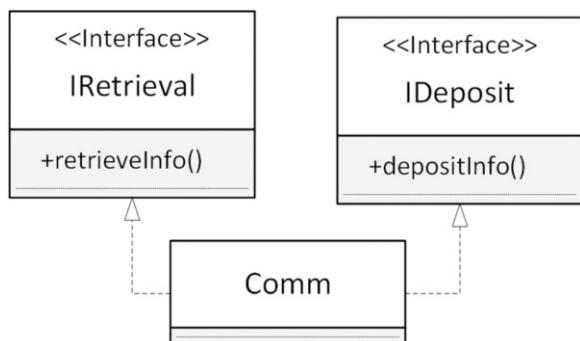
Class Comm<T> implements IDeposit<T>, IRetrieval<T>{
    //...
}
Comm c = new Comm();
a.setDataRetrieval(c); //a aggregates an instance of IRetrieval
b.setDataDeposit(c); //b aggregates an instance of IDeposit

```

Both *a* and *b* receive the same object created in a context; thus, each has an alias. The coupling between *a* and *b* is kept to a minimum as each is limited to know no more than it needs to. Decisions on how information is deposited and retrieved can be delayed. For a simple application of this (lightweight) communication mechanism, consider a canvas object *b* collects the location information of a mouse when clicked on the canvas, and display the information on another widget.

Another behavioral pattern known as memento is also an example of lightweight communication between objects. In this case, we need to save the state of an aggregated object *b* (in a context) and restore it later. An object that provides a temporary store is an instance of the abstraction *Memento*. A caretaker object is generally needed to manage all memento objects. Applications include “undo” or “redo” operations of various kinds such as saving and restoring a windows’ look-and-feel attributes or software configuration data.

Fig. 8.22 A simple mechanism to provide loosely coupled object communications



8.6 Summary

A design pattern is a repeatable design solution to a commonly occurring problem. A design pattern is often represented by a set of collaborating interfaces to describe a solution without concerning contextual details and thus can be used in many different situations. In many ways, design patterns have become a design language to facilitate communications in software modeling and construction activities. Inversion of control is often the design principle used in design patterns. Thus, the use of design patterns supports implementation flexibility and software extendibility. Some design patterns may be of architectural significance and are often used to design structural or behavioral objects to support a chosen architecture. We have discussed most of the Gang-of-Four design patterns in three categories. As useful as they are, design patterns are not design “hammers” to look for problem “nails.” In practice, we are more likely to use the design ideas behind the patterns than actual patterns themselves.

Broadly, the SOLID design principles are behind all design patterns. But another set of design practices, known as GRASP, are probably of more direct applicability to not only pattern design but also our routine design work. GRASP stands for General Responsibility Assignment Software Patterns, referring to design practices: *controller*, *creator*, *indirection*, *information expert*, *low coupling*, *high cohesion*, *polymorphism*, *protected variations*, and *pure fabrication* (Larman, 2001). These practices are useful when we design object collaborators. We have explained these design practices throughout this book, though not always using the same idiosyncratic names. A few might be worth elaborating in the context of design patterns:

- The *creator* practice suggests that object type *B* is responsible for creating instances of *A* (possibly with a factory) if *B* aggregates an instance of *A* or has the data to initialize objects of *A* (though exceptions always exist).
- The *controller* practice is central to an object-oriented design as discussed earlier. Design patterns often provide controls for object creation, functional delegation, or object’s behavioral alteration.
- The *indirection* practice suggests the use of an intermediate object responsible for mediation between communicating elements to support loose coupling. We have provided some in-depth discussion about *indirection* based on the last two behavioral design patterns we covered.
- The *pure fabrication* practice is to suggest creation of an object type that does not represent a concept in the problem domain to achieve low coupling, high cohesion, or module reuse. Abstractions in design patterns are mostly fabrications guided by creational, structural, or behavioral needs of a software model, not by problem domain concepts.

Object collaboration is central to an object-oriented design and manifested in design patterns. Collaboration may mean to bridge a task through delegation, adaptation, or

composition or to manage collaborations in a contextual method resulting in behavioral alteration of a contextual object. Collaboration between modules appears much simpler in functional programming through higher-order functions (i.e., functions that accept functions as parameters and return functions as “values”). We have commented earlier that structural design patterns may have much simpler counterparts in forms of functional collaboration. Here is another example that may involve some functional counterparts of the design patterns (which makes functional collaboration more succinct and attractively simple):

```
words.stream()
    .filter(word -> word.length() <= 5)
    .map(String::toUpperCase)
    .limit(10)
    .forEach(y -> System.out.println(y));
```

Despite the fact that some design patterns can be represented in terms of functional collaborations with much simpler formulations, stateful computing with mutable data will remain appropriate and preferred for many business applications and so will design patterns in their original formulations.

Exercises

These exercises were designed to practice on the use of the design patterns and hence gain better understanding about applications of design patterns. As such, details are not as important as the program structures based on the patterns being used. Some of the exercises can be used as projects with full implementations. If used as an exercise however, complex details of a problem can be simply replaced by message displays. Also, hints are given for some of the problems, but should not limit the thinking along different solution paths.

1. Use Abstract Factory pattern to create a factory with two button products: one that changes button’s background color when mouse over and the other that toggles two background colors when clicked.
2. Figure 8.1 described a process of creating two airline flight schedules using the abstract factory pattern. Write an implementation based on this scenario with trivial details. For example, one schedule uses solid lines to separate flights, and the other uses dotted lines instead. Also, a method may use two parameter lists for departure and arrival times.
3. Write a program, using appropriate creational design patterns, to create college academic transcripts (through console output). Each transcript consists of three parts:
 - A header that contains student’s personal data and information about a major (and minor).
 - A list of courses taken (including courses taken at different institutions), each with a course number, name, number of credit hours, and grade, organized by year in a chronological order and then by semester with Autumn term first and then Winter,

Spring, and Summer terms as appropriate. (In an actual application, the course information needed would be from a database.)

- A footer that contains summary of total number of credit hours taken, as well as the honors and degrees earned.

Create transcripts for two different kinds of students: regular student and nondegree-seeking student with differences only in the footer section.

4. Airlines may create flight itineraries differently depending on whether a flight is domestic or international. Each itinerary consists of three parts:

- A header that contains traveler's name, ticket number, frequent flyer number, seat info, and date of issue
- A flight departure and arrival schedule including all connection flights
- A footer that contains general policy statements such as meal information, luggage dimension and weight requirements, and a courtesy reminder about the custom regulations at the port of entry for international flights
 - (a) Write a program to create an itinerary and display it as console output. (For the sake of this exercise, appropriate parameters can be used for the information needed to construct a schedule.)
 - (b) Write a program to create two itineraries, one for domestic flights and the other for international flight, which may need additional information about regulations at a foreign airport.

5. An outlet mall management maintains two lists: one for new items at each store and the other for on-sale items. The management makes the lists available to the public once a week. To grab shoppers' attention, these lists are formatted differently. Information needed includes the store name and its identification number and items that just arrived or items on sale. Each item includes the brand name, brief description, and regular or on-sale price. The lists should be uniquely available.

Write a program to create the lists specified. You may use console input or a text file for the information needed. (Hint: If you view each list as a product, you might use an abstract factory to create the lists and then create a Singleton object containing the lists. But other approaches are also possible. For example, you can create a list with default objects representing stores. Each week, only updates are needed for each store with information about the latest new or on-sale items. In other words, a list is a product, and stores are its parts.)

6. Write a program to build a resume based on a set of files you have written representing the resume sections such as Heading, Objective, Education, Experience, Skills, and References. Besides, an implementation can also vary on:

- Input source that could be text files, JSON strings, or a database
- Number of sections needed
- Output formats

(Hint: A few patterns can be appropriate for this question, more noticeably Builder, Chain of Responsibility for constructing the sections, and Decorator for formatting the output based on an existing basic format. Use an interface to capture input variations.)

7. Write a program, with console output or a graphic user interface, to simulate fulfillment of a pizza order at an eatery. There are four kinds of crust to choose from, stuffed, cracker, flat bread, or thin crust, and six toppings to select, pepperoni, sausage, mushroom, onion, black olives, or green pepper. A crust of any kind costs \$8.50. Meat-based toppings are \$1.25 each, and others are \$1.10 each. A customer can select preferences through console input. The output should include ingredients listed individually with the price and the total cost. (Hint: Using a builder might be more straightforward, but applying multiple decorators of different ingredients can be interesting too.)
8. A word generator is an object that can select a random word from its collection. Write a console program to construct a sentence in the form of “subject verb object adverb” by selecting words from four words generators (i.e., they are “subject” generator, “verb” generator, etc.). The program can generate sentences like “a dog chases a cat amusingly,” “Dave closes a box gradually,” etc., though some of the sentences might make little sense. Use the Chain of Responsibility pattern in your implementation with the following abstraction:

```
abstract class WordGenerator{
    private WordGenerator next;
    abstract Sentence processRequest(Sentence s);
}
```

9. Implement the same scenario described in the previous question using the visitor pattern instead. (Hint: Each word class (like subject, verb, etc.) is a context for a visitor operation.)
10. Simulate a stopwatch illustrated in Fig. 8.23. The watch label displays “00:00:00” when the application starts. Clicking the START/STOP button would toggle between starting the timer and stopping it. RESET button, when pushed, returns the timer to its initial state. Clicking the button “HOLD” would toggle between time suspension and resuming from where the time was suspended. The “HOLD” button can be pushed as many times as needed with the same behavior.
- (Hint: The State pattern appears to be appropriate as buttons may correspond to watch’s states, though implementation without patterns might be more straightforward. You might also use library methods to track elapses of real time.)
11. Write a console program to simulate tracking of a priority mail’s delivery status using these four states of delivery: *initiated*, *in transit*, *arrived* (in the destination city), and *delivered*. To simplify the implementation, the program may simply display what state a mail is in. (Hint: Either State or Chain of Responsibility can be used. It can be

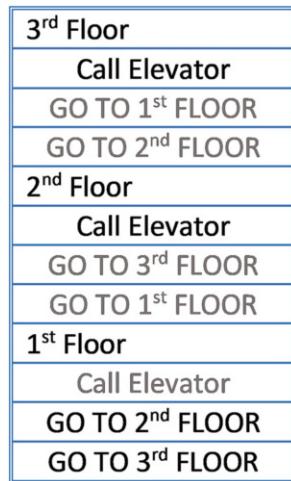
Fig. 8.23 Illustration of a stopwatch



convenient to use a dummy state in implementations when the State pattern is used. This “dummy object” is known as the *Null Object* pattern—an implementation of the abstraction with null data and empty methods.)

12. Write a console-based program to simulate the operation of a garage-door remote opener. (Hint: A remote control can be a context for applying the Command pattern.)
13. Write a program to simulate an electric switch that controls a light with a graphic interface.
 - (a) Implement the program with the Command pattern.
 - (b) Implement the program without any pattern, and compare it with the implementation in part (a) in terms of pros and cons.
14. Write a console-based program to simulate exchange between a seller and a buyer in real estate sales negotiations using the Mediator pattern. The implementation can be limited to only price negotiations between two parties until a consensus is reached. The mediator (i.e., a real estate agent) also keeps a log of the negotiation, which should be displayed before exit. A buyer or a seller enters an offer or a counteroffer through console input.
15. Academic events such as students adding a class, withdrawing from a class, or changing to “audit” status of a class may trigger reviews by various campus offices for potential issues. For example, a withdrawal that causes the number of credit hours to drop under certain level can affect a student’s financial aid eligibility. Write a program to simulate handling of such events using the observe pattern with two observes: registrar office and financial aid office. The program can implement only trivial responses to events by simply displaying an acknowledgment of the event.
16. Suppose the IT department of a company has employees of four kinds: developers, business analysts, team leads, and testers; they receive messages sent by the department manager. Each message from the manager has a receiver identifier, *dev*, *ba*, *tl*, or *tt*, to refer to intended receivers. Based on the receiver identifier, an employee either reads or ignores a message. However, if the identifier is “*all*,” then everyone needs to read it. Write a console-based program, using the Observer pattern, to simulate the process of manager sending multiple messages with console inputs. Each message consists of a receiver identifier and a message body. Each receiver responds to a message with an appropriate console output.
17. Write a program to simulate an elevator operation of a three-floor building illustrated in Fig. 8.24. The program models elevator behavior as shown in the image below. Each panel consists of three buttons and one textbox. The “Call Elevator” represents a floor button. There are two elevator buttons for going to respective floors and a textbox for displaying the door status of the elevator. A deactivated button is dimmed when the service is currently unavailable. When elevator remains at a floor, the door should remain open until a service is called. The image below shows that the elevator currently is at the first floor (with door open). Suppose elevator is called at the second floor, the elevator will then close the door (with a display of “Door Closed”) and deactivate the elevator buttons. At the same time, the second-floor door textbox will display “open,”

Fig. 8.24 Illustration of elevator operations of three-floor building



and elevator buttons (GO UP and GO DOWN) will be activated. The state of the third floor will remain unchanged.

- (a) Implement the program using the State pattern. (Hint: There are only three states of all floors—elevator being at first, second, or third floor, respectively. Either a floor button or an elevator button triggers a transition.)
 - (b) Implement the program without using a pattern and compare the implementation with that in part (a). (Hint: To do so, each floor can be simulated with a panel with three buttons and one textbox. Whether buttons are dimmed and what to display in the textbox constitute a state of the panel. An event handler determines which floor the elevator moves to and reset the panels' states.)
18. A mail-based retailer needs a prototype for its shipping operation. This retailer currently accepts orders for some sought-after music albums. It is the policy of this retailer to charge shipping cost depending on the total amount a customer has ordered from the retailer:
- Orders are fulfilled free of shipping charge if the customer has ordered \$500 or more.
 - Orders are fulfilled with 50% shipping charge discount if the customer has ordered \$250 or more but less than \$500.
 - Orders are fulfilled with 20% shipping charge discount if the customer has ordered \$125 or more but less than \$250.
 - Otherwise, orders are fulfilled with full shipping charge.

- (a) Write a console-based program, using the Chain of Responsibility pattern to simulate this business process. An order is made through console input, and an output indicates the shipping charge to be applied.
- (b) Implement the program again using the Observer pattern and compare it with the implementation in part (a).
19. Template method is another behavioral design pattern not mentioned in this chapter, yet useful especially in framework design. It is a method in an abstract class to implement a stable (thus invariant) control process with certain elements abstracted. In other words, the method implements a high-level control with calls to abstract methods defined in the class. The low-level details are provided by a subclass when implementations of the abstract methods are provided. These abstract methods are called “hook operations.” For example, consider dinning in a restaurant. The “control” flow would be always to order, then eat, and finally pay—a stable control flow with varying details for each dinning instance.
- (a) Write an abstract class that has a template method to print a line with the alternating segment pattern abstracted out. For example, the following three lines are the results of calling the template method in three subclasses:
- ```
***** ----- ***** ----- ***** ----- ***** ----- *****
***** +++++ ****+ +++++ ****+ +++++ ****+ +++++ ****+
***** +-+ + ****+-+ -+ + ****+-+ -+ + ****+-+ -+ + ****+
```
- Reproduce two of the lines by writing two subclasses.
- (b) Describe how a card game can be implemented as a template method.
20. As mentioned earlier, structural patterns are essentially procedural and useful in a design using mixed paradigms. Describe how Adapter, Bridge, and Façade patterns might look like if only standalone methods and functions are used.

---

## References

- Birman, K., & Joseph, T. (1987). Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (pp. 123–138).
- Gamma, E., et al. (1994, November). *Design patterns: Elements of reusable object-oriented software*. Addison Wesley Professional.
- Larman, C. (2001). *Applying UML and patterns: An introduction to object-oriented analysis and design and the unified process* (2nd ed.). Prentice Hall.



# Software Architecture and Architectural Views

9

---

## 9.1 Overview

In software development, many things we embrace and promote do overlap and intersect. We have discussed about design of modules, developing software families based on larger software elements, use of design patterns, and modeling with mix of design paradigms. These are all different aspects of software design, and each aspect has its focus, specific design principles, and distinctive characteristics and practices. However, trying to separate these areas by drawing boundaries is not productive. We work on different aspects of a development under the same *software architecture*, guided by the same *architectural design* of the software. Though we have introduced these terms superficially earlier, we will explore, in this chapter, what they mean and entail in relation to what we have discussed elsewhere. Luckily, we have all had some intuition about what an architecture is: a high-level structural design of a product (like a structure of a house). It is a kind of design that provides guidance to what we can or must do in executing a wide range of design activities. But at the same time, an architecture imposes design constraints too. Software analysis has the closest relation to software architecture because analysis tries to answer questions, based on software requirements, about behavioral and structural characteristics of the software architecture, and modeling then leads to a software model supported by the architecture.

Software architecture, as a subject, is relatively young and became an area of study only in the late 1980s—more than 20 years after software engineering became a discipline. Part of the reason might be that people needed some necessary experience to realize that a software system was much more about producing expected outcomes, and other quality attributes were important too and could be achieved by carefully structuring the software. Besides, software systems must be large enough for people to realize the importance of capturing and preserving designers' intention about system structure and behavior to

provide some defense against system decay. Since then, significant progress has been made in development of software architecture such as n-tier client-server, agent-based, and service-oriented architectures. Viable and popular architectural patterns or styles like Model-View-Controller have become integrated into commonly used software development platforms. Despite the progress that has been made in the last 30 years, we are still far from having even a consensus on what appears to be a basic question: *What is software architecture?* Despite many attempts to provide an answer, we still lack a good, short, and widely accepted description today. Meanwhile, we are still facing challenges in many practical aspects of software architecture such as finding right languages to represent a software architecture, ways to maintain architectural consistency in architectural representations, and methods to assure conformance between an architecture and the code. In this chapter, we will first try to explain what software architecture may be. Then, we will make an argument on why a comprehensive representation of software architecture might not be practical, if not impossible. Thus, the best effort we can make is perhaps to produce helpful architectural views for different software stakeholders including developers and software engineers. The rest of the chapter will then focus on two architectural view models, one is more traditional, and the other is more recent.

---

## 9.2 What Is Software Architecture?

Software architecture can often be considered as “big design upfront” with a negative connotation that it will likely become useless and abandoned in a development process full of uncertainties. People may be distrustful of architecture practices and avoiding conscious architecture-focused activities in favor of an evolutionary architecture that is believed to emerge from development teams that are agile and self-adapting. However, there are good reasons why architecture, which is an upfront and structure-first design, is necessary for any complex software project, among which:

- Software development must meet required software quality attributes with larger-scale architectural initiatives that can be associated with project planning.
- A development cannot afford excessive redesign and associated delays that can lead to decline of system qualities, low reuse of common components, and redundant solution elements.
- Force architectural decisions to be made early and explicitly to avoid making implicit architectural decisions often based on local needs and less informed practices.
- Avoid excessive design refactoring and componentization that can result in a solution being fragmented, hence difficult to comprehend.

To understand the reasoning, we need a reasonable understanding about software architecture in terms of its nature, its styles in structuring software elements, its

composition, and its scope in a development process. This understanding is largely independent of many perspective views about the content software architecture may or may not entail.

### 9.2.1 A Characterization of Software Architecture

Architectural creativity might start with thinking concurrently about a system from three different perspectives:

- A static perspective in terms of a set of code units and their relations
- A dynamic perspective in terms of how software elements behave and interact at runtime
- An operational perspective in terms of a collaborative operational environment involving not only the software and its hosting agent but also physical devices and external software systems that represent different processes running in concert with the software, satisfying a set of performance and other quality attributes

These common architectural concerns may suggest that a software architecture can be a collection of architecturally significant software elements in some chosen forms to satisfy major functional and performance requirements of the software system, as well as non-functional requirements in areas such as reliability, scalability, portability, and availability. As architectures in many other engineering fields might suggest, software architecture should deal with the design and implementation of a product at higher structural levels, not with implementation details. Despite numerous other definitions, Perry and Wolf defined software architecture succinctly with four entities (Perry & Wolf, 1992):

$$\text{Software architecture} = \{\text{Elements}, \text{Forms}, \text{Rationale}, \text{Constraints}\}.$$

When we think about the architecture of a house, we think about not only various kinds of rooms, stairways, and open spaces—the elements of a house—but also how these elements are connected and laid out based on a chosen architectural style. The builder would also tell the client what constraints the house might have, and why, because of the architecture. Software architecture works in much the same way. As software analysis and modeling evolve, larger software elements are further divided into domain elements, processing elements, design elements, data elements, and connection elements. Architectural forms specify the expected properties of the elements, their relationships, organization of the elements, and ways elements interact within architectural constraints. Today, the use of UML diagrams is a standard way to express architectural forms. Meanwhile, software systems vary in application domains they serve and are constrained in ways they need to be. These are often the reason for choosing a particular architectural style over other

choices. Therefore, an architecture must also explicate the rationale behind the architecture to defend the design and facilitate more open-minded discussions.

### 9.2.2 Architectural Styles

A software architectural style is a general, reusable solution to a commonly occurring problem in software architecture within a given context. It provides a guidance of how software elements are connected and organized into layers, packages, or specialized functional units. It is a specific way of structural organization applicable to a variety of systems. An architectural style is characterized by the features that make it distinguishable from other architectural styles. Thus, a chosen architectural style provides guidance for us to make architectural decisions. Like one we choose to build a house, an architectural style (like Cape Cod, Colonial, Contemporary, etc. we often hear) is a reusable “package” of design decisions and constraints that are applied to an architecture to induce the chosen architectural qualities. Like architectural styles of a house, software architectural styles also have their vocabulary of components and connectors with constraints on how they can be composed. While we will not elaborate, Table 9.1 gives some of the commonly used architectures, architectural styles appropriate for each architecture, and design patterns that are most appropriate for implementations of each architecture with appropriate styles. Names of the architectural styles are simple metaphors that may suggest how the styles may work like. For example, the Blackboard style is like a group of specialists sitting in a room that has a large blackboard. They work as a team to brainstorm a solution to a problem, using the blackboard as the workplace for developing a solution cooperatively.

**Table 9.1** Commonly used software architectures and styles

| Architecture<br>(module<br>organization) | Style(ways elements are related)                                                                                                                                                    | Design pattern<br>(localized design)      |
|------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| Data flow                                | Pipes and filters, batch and sequential processing                                                                                                                                  | Chain of responsibility, visitor          |
| Data-centric                             | Blackboard, repository, domain-centric                                                                                                                                              | Façade, command                           |
| Hierarchical                             | Multilayered architecture, microservices architecture, component-based, layered, master/slaves                                                                                      | Façade, command                           |
| Interaction                              | Peer-to-peer (P2P), representational state transfer (REST), model-view-controller (MVC), presentation-abstraction-controller (PAC), reactive architecture, event-driven, page-based | Mediator, Observer, State, Façade,        |
| Distribution                             | Client-server (N-tier), service-oriented architecture (SOA), broker, shared nothing                                                                                                 | Adaptor, proxy, façade, command, mediator |
| Virtual machine                          | Rule-based, interpreters                                                                                                                                                            | Façade, command                           |

When a session starts, the specialists all watch the blackboard as the problem is written onto it, looking for an opportunity to apply their expertise to the solution development. When they can, specialists apply their knowledge to move the solution on the board forward. This process of applying the collective knowledge to a solution would continue until the problem is solved.

An architectural style has “components” and “connectors”—the software modules that work together to implement the style. For example, the Blackboard style has a component (playing a role of a “connector”) called “control shell,” which controls coherently the flow of problem-solving activities by the specialist modules (or the knowledge sources), much like a moderator would do to coordinate a discussion. The blackboard also has a component used as a shared repository of problems, partial solutions, heuristics, and contributed information.

A design pattern often plays a role as an architectural connector to facilitate application of an architectural style. For example, if a “style” is collaborative problem-solving with a “task” being a software solution. Each “expert” processor can be a processing “node” if the chain of responsibility pattern is used. A task is passed along a chain of experts, each working on the task in its own way when environmental conditions are favorable for them to be able to meaningfully contribute. A task can also be an enduring process in which a chain of processors is activated periodically, such as a sequence of security checks in a mission critical product. When data to be processed are of different types, the visitor pattern might be more appropriate to use. For example, in a software process handling insurance claims, each “visitor processor” is an “expert” of processing claims by insurers of a particular insurance policy. Design patterns may help implement an architectural style locally, but, as mentioned earlier, some could also be primary architectural structurers.

Styles are documented in a style guide that defines the element and relationship types indigenous to a style, along with any semantic restrictions on their use. A style guide also lists what design problems a style is and is not intended to address. It also discusses any notations or analytical approaches available using that style and any related styles. Many architectural styles listed in Table 9.1, however, were proposed in the mid-1990s when remote networked components were still rare and software framework technologies were still few and basic. As technologies have vastly improved and advanced, some of the architectural styles have largely been behind times. With fast Internet connections, cloud software has been replacing traditional desktop applications, and Enterprise-Resource-Planning (ERP) software is a commonplace that enables organizations to manage operational processes with integrated applications and multiple business fronts. As a result, architectural styles such as Multi-tier Client-Server, MVC, SOA, and Microservices are much more commonly used today than others. Framework technologies, such as .NET, PHP, and React, are increasingly sophisticated to provide “templates” for implementing software in certain styles, most noticeably Client-Server, MVC, and REST (Representational State Transfer) styles.

“Architectural pattern” is also a term used to mostly mean “style.” Architectural patterns are prepackaged sets of design decisions, each with its own vocabulary and known effects

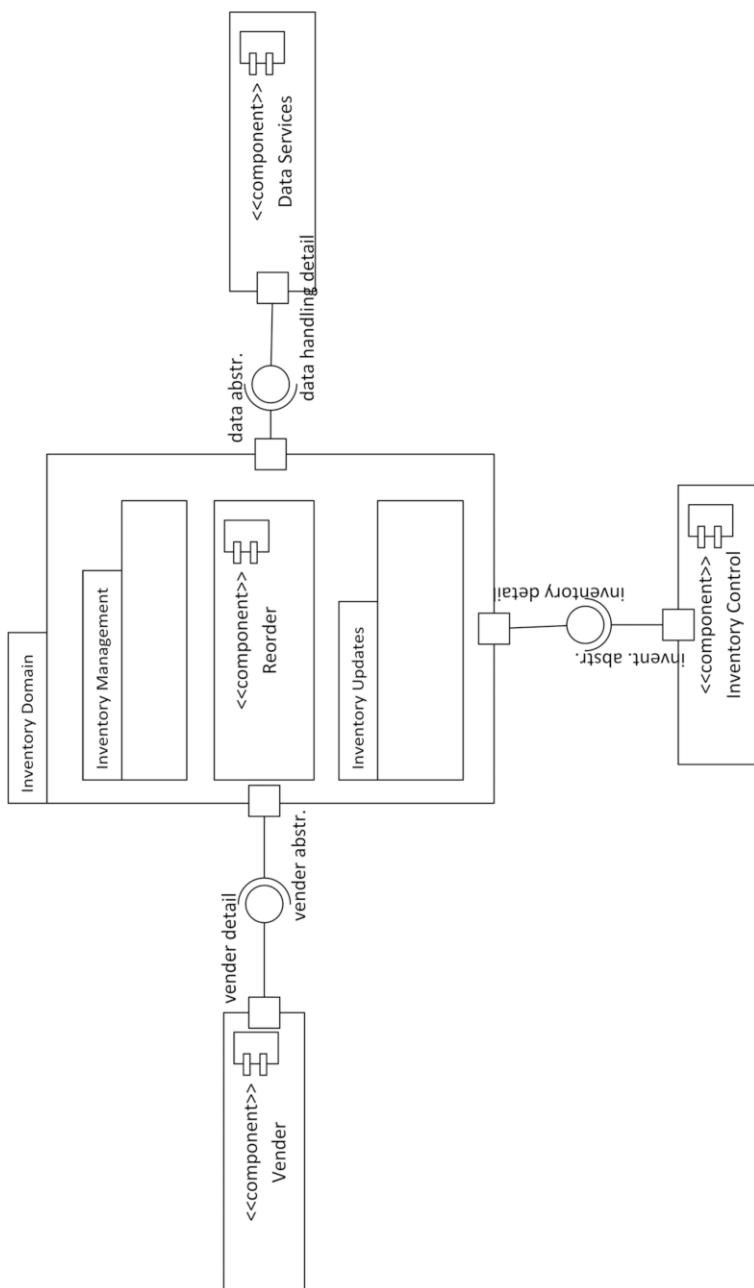
on software quality attributes. Perhaps the best-known catalog of architectural patterns is the two-volume set of books “Pattern-Oriented Software Architecture” (Buschmann et al., 1996, 2000). Strictly speaking, architectural patterns focus on the context of a problem, restricting the attention to elements and their relation types and imposing topological and behavioral constraints on how the elements are structured. An architecture style, in contrast, focuses more on solution guidance about when a particular style may or may not be useful; they are generally not contextual, thus more widely applicable.

As an example, Fig. 9.1 illustrates how a domain-centric architectural style may look like. If we want to make a layer for domain elements of various kinds, we then need to understand how other aspects of the system (like “data,” “vendor,” and “control” in the example) would communicate with domain elements through either “requires” or “provides” interfaces at appropriate ports. In contrast, when elements were organized in a layered style, not only would elements be packaged differently, but the communication mechanisms might also follow different protocols. For example, for the same inventory management scenario, a three-layer approach would involve a management layer (as a controller) communicating with a service layer for management functionality, which communicates with a data layer. Each layer can also independently communicate with other components to support their internal operations.

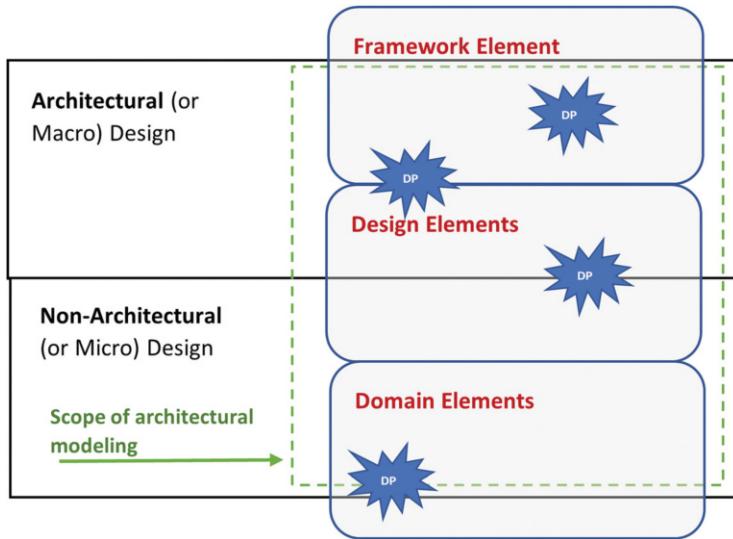
Choosing an appropriate architectural style is a result of software analysis. We would test the feasibilities of potential style candidates through conceptual and structural simulations before narrowing down the choices.

### 9.2.3 Structural Layers of Software Elements

Software architecture may sound elusive. That’s because the formation of an architecture may be gradual and relations among the elements and their architectural constraints can be evolving. We may not have a full understanding about the architecture we have been working on until we do at some point of the development. Nonetheless, a rudimentary understanding about how software elements are distributed across layers of abstractions of a software system might help to understand the ingredients of an architecture even if we might never see a software architecture like we see a structural blueprint of a house printed on paper. Figure 9.2 shows three layers of elements in terms of their architectural significance (Braude, 2004). This layered view of element organization is independent of any particular architecture or architectural style. Domain elements are a “workhorse” of the software, directly responsible for delivering the functionality. They are generally not structuring elements, thus of little architectural significance. On the other end of the spectrum of an architectural organization of the elements, framework elements mostly control the flow of software operations to provide a generic software solution constructed using abstractions. They implement the stable parts of a system (often forming an overall control of software operations) by abstracting out the variations of the system. Thus, these elements are entirely of architectural significance. The rest of the elements are design



**Fig. 9.1** Domain-centric architectural style



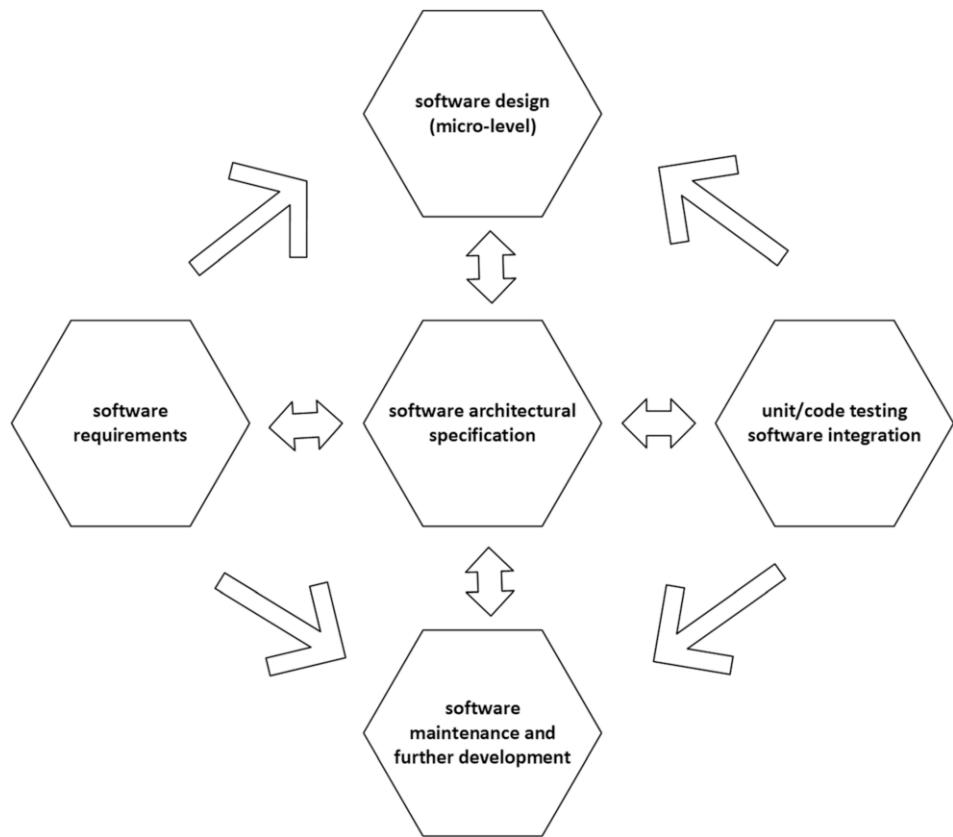
**Fig. 9.2** Visualization of structural distribution of software elements

elements across both architectural and nonarchitectural design realms, scattered between two dichotomies of an architectural organization of the elements. These design elements are neither pure domain elements nor straight top-level structural elements. They play some architectural roles yet are functional in some other ways. These are elements that resolve design issues, function as connectors, or provide structural or communicational means with varying architectural significance. The “blast” shapes in Fig. 9.2 represent design patterns potentially applicable to any of the layers. For example, the Template Method pattern can often be used in a framework design. To reuse existing software resources, the Façade pattern can be used for design of domain elements, whereas the Memento pattern is more likely a design element. For the sake of illustration, boundaries are depicted in Fig. 9.2 for the three layers of elements, but such boundaries can only be defined in practical terms even if they are helpful.

Finally, architectural modeling (contrasting to the term “software modeling”) focuses on uncertain parts of the architecture to examine the feasibility, potential issues, possible modifications, or even considerations of a redesign. Architectural modeling often has a defined scope but can be invoked at different times in a design process as appropriate and needed.

#### 9.2.4 The Impact of Software Architecture

Software architecture plays a central role in software development. Research studies (McBride, 2007) suggested that for every 25% increase in complexity of a system, there



**Fig. 9.3** Impact of architecture on software life cycle

is 100% increase in complexity of a system solution and explicit requirements explode by a factor of 50 or more into implicit (design) requirements as a software solution proceeds. Perhaps more than any other task, managing complexity is an essential challenge of any architectural endeavor. To manage complexity and functional requirements, architectural design must start early in a process of requirement elicitation and validation to derive a high-level design solution and establish important architectural baselines. These baselines would include initial understanding of the problem domain, high-level requirements, and unstated design expectations that must be identified and validated. Figure 9.3 illustrates how software architecture supports every aspect of a software life cycle. As discussed earlier, requirement analysis leads to software architecture. In return, software architecture provides confirmation and validation of the requirements and direct guidance for further design. We need an architecture to design test cases and component integration setups. Testing, in return, provides feedback on testability of architectural modules. Software maintenance and evolution must be supported by the software architecture to be

meaningful and feasible. In other words, an architecture provides guidance to what we do in a software process whether we manage the process with agility or well-defined plans.

One of the primary risks in software development is a gradual structural erosion with implementation details not guided by the architecture once code construction becomes a primary development activity. Thus, a sound software development process must ensure architectural sustainability with practices including the following:

- Create or select an architecture based on requirements analysis.
- Maintain an alignment among business goals, software requirements, and high-level software elements.
- Communicate the architecture to stakeholders.
- Validate and refactor the architecture when changes (of any kind) occur.
- Ensure the conformance of software construction to the architecture.

To create a sustainable architecture, the workflow for software architecture usually spans the entire spectrum of the development. As things change, we update the architecture as necessary yet remain focused on the essential, architectural perspectives to maintain an alignment with business goals and high-level software requirements. Being conservative is perhaps a useful attribute in developing a sustainable architecture because false architectural assumptions can be costly when they are carried too far before we realize. Thus, architectural decisions must be fully informed with potential impact. The following practices may help with an architectural decision-making process:

- Clearly articulate assumptions and test them against the reality.
- Use some critical parts of an application to build prototypes to test assumptions at a minimal cost.
- Treat certain (implicitly) expected architectural attributes as hypotheses (about potential values of the architecture) so that they must be explicitly stated and tested.
- Develop some mechanism, based on evidence and data, to avoid making implicit architectural decisions.
- Some healthy doses of skepticism may not be a bad thing.

Finally, architectural refactoring should address not only the changes in requirements but also structural improvements. Architectural modification should always be accompanied with an impact assessment on low-level implementation details.

### 9.2.5 Architectural Views

Because software architecture is an intellectual product of enduring importance, communicating an architecture to others is as important as creating it in the first place. In traditional engineering disciplines, an architectural design is understandably comprehensive, meaning

little is left to be architecturally specified. It is, however, difficult for us to do the same for software architecture primarily because of the following reasons:

- It may not be possible to completely understand a viable architecture and all its constraints before construction work takes place, as formation of a software architecture can be a gradual process.
- Architecture tends to evolve as requirements are clarified, understood, updated, and further solicited and developed.
- Early construction of software may provide valuable insights into architectural feasibility and can take place before the architecture is fully understood.

Therefore, producing comprehensive architectural artifacts in a traditional sense is practically difficult, if not impossible. There is (almost always) a cognitive gap between an architecture and its implementation, even though this gap must be traveled over time as we gain more architectural insights. However, before this gap closes (if ever), we must communicate to stakeholders about the current version of the architecture we have already understood in ways they understand and appreciate.

Modern software architecture practice embraces the concept of architectural views. A view is a representation of a set of system entities and their relations. Views are representations of the system structures that are present simultaneously in a software system. One view might show the hierarchical decomposition of the system's functionality into modules or how the system is structured into layers. Another view might reveal how the system accomplishes its tasks through communicating processes showing the dynamic behaviors of the interacting objects or processes. Still another might capture how software elements are organized into files and packages or how the final product should be configured at the time of deployment. Architectural views, by design, are not comprehensive. Each view has a focus. Common viewing focuses are elements and their relations, the system's dynamic behavior due to interactions of the elements, the physical organization of code artifacts, and system's operational environments. To facilitate construction of views, we use "view packets"—the smallest meaningful view units—to compose larger architectural views for different viewing interests.

Since the mid-1990s, traditional architectural forms had become obsolete because of the fundamental shift of software development to object-oriented methodologies. They were replaced by UML artifacts, manifested in architectural views. The first influential view model is the (object-oriented) 4+1 View Model (Kruchten, 1995), which is discussed in the next section. Later, another view model known as the Siemens Four Views was proposed (Hofmeister et al., 1999), suggesting views in conceptual, module, execution, and code architectures. These two view models were semantically similar, and they were both integrated later into the Unified (Software) Process Model with IBM-supported computer-aided software engineering (CASE) tools including those for modeling with the UML. However, these view models are intended to visualize only the "kernel" of the architecture—the most critical part of an architecture design. Therefore, we should consider

how other views may also be helpful not only for the sake of keeping stakeholders better informed but also for architectural documentation to support software evolution.

To construct views, it's helpful to think about a view in terms of the kind of information it conveys. For example, a view about the system modules would describe how the system or a subsystem is to be structured as a set of code units or as a set of communicating software components. A view about system's operational environment would depict how the system communicates with other internal or external systems and devices. Therefore, we can then define view categories based on the information a view conveys. A particular view of a system may fall precisely into one of these view categories or an intersection of some.

The other architectural view model we will also discuss later, known as the C4 Model, is semantically different from the 4+1 Model with its view categories based on levels of module abstractions. It is not necessarily a competing view model, though the model does reflect the creator's philosophical viewpoint about what would be more important to an architecture. They can be complimentary, however, when we choose views from both models to form combined views tailored to individual interests.

In summary, there isn't a simple analogy between software architecture and architectures in traditional engineering disciplines. Engineering architectures are concrete and comprehensive. An architectural view of a traditional architecture is an enlargement of an aspect of the entire architecture and must be understood in the context of the global architectural structure of a product, for example, how the power in automobiles is transmitted from the engine to wheels or how the plumbing system is laid out within the spatial arrangement of a building. These views are high-level representations that can be developed before the product is constructed. They are views as if we could zoom in on the architecture. In contrast, a software architecture may never be explicitly specified in its entirety. Nonetheless, architectural views can be developed in any aspect of an architecture and in any specificity of the aspect, as we deem useful. Thus, software architectural views represent the architecture, but not necessarily "zoom-in" artifacts. They must be timely updated to remain relevant. Inconsistencies can happen when:

- Views overlap in certain areas.
- There are different "perspective" views about the same process.
- Different aspect views show conflicting information.
- Related views are not update.
- Views are overloaded with low-level details.
- Views are created about essentially the same thing.

Despite the challenges in producing and maintaining useful architectural artifacts, architectural views are what we communicate about an architecture, and they are strategically important in the entire software life cycle and beyond.

### 9.3 The 4+1 Architectural View Model

Philippe Kruchten (of Rational Software Corporation at the time) proposed what has been known as 4+1 Architectural View Model (Kruchten, 1995). The prevailing consensus in software architecture research is that these views are adequate representations of a system architecture, where each view provides a different abstraction of the underlying implementation details. The views in this architectural view model are built around use cases. Use cases provide ingredients for the views and basis for validating the views, and software analysis is behind the formations of the views.

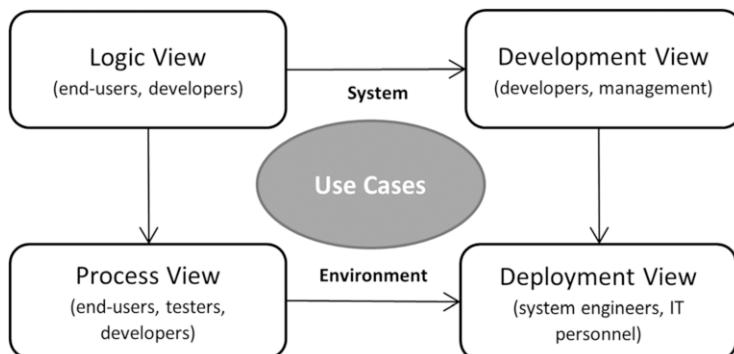
#### 9.3.1 An Overview of the View Model

Figure 9.4 is a visual description of the model. Each box represents a view aspect (or category) with possible consumers. An arrow points to a view aspect for which the connected view aspect provides the basis for the information needed to construct views.

According to the creator of this view model:

- The logical view is the object model of the design when an object-oriented design method is used.
- The process view captures the concurrency and synchronization aspects of the design.
- The physical view describes the mapping(s) of the software onto the hardware and reflects its distributed aspect.
- The development view describes the static organization of the software in its development environment.

Table 9.2 gives more details about the content of each view, its potential stakeholders, what to consider when constructing a view, and appropriate UML diagrams to use. The



**Fig. 9.4** The 4+1 Architectural View Model

**Table 9.2** Detailed description of the 4+1 View Model

|                        | Logical                                                         | Process                                                                                                                             | Development                                                                       | Physical                                                                          | Scenario                                                               |
|------------------------|-----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|------------------------------------------------------------------------|
| Description            | Shows the components of the system as well as their interaction | Shows the processes and workflow rules of the system and how those processes communicate with a focus on dynamic view of the system | Shows building blocks of the system and static organization of the system modules | Shows the installation, configuration, and deployment of the software application | Shows the design is complete by performing validation and illustration |
| Viewer/<br>stakeholder | End users, analysts, and designers                              | Integrators and developers                                                                                                          | Programmers and software project managers                                         | System engineers, operators, system administrators, and system installers         | All viewers and evaluators                                             |
| Consider               | Functional requirements                                         | Nonfunctional requirements and operational environment                                                                              | Software module organization (management of reuse, constraints of the tools)      | Nonfunctional requirement regarding underlying hardware                           | System consistency and validity                                        |
| UML<br>diagram         | Class, state, object, sequence, communication                   | Activity (primarily), state, sequence                                                                                               | Component, package                                                                | Deployment                                                                        | Use case                                                               |

logic view describes what the system is. It provides an essential reference for other views. In general, we provide views from conceptual, static, and dynamic perspectives to describe how the system logically works, focusing on the functional aspects of the system. The process view goes beyond the functional aspects of the system to show how various processes work in concert to interact with the software. Some can be the system's own processes that are distributed, concurrent, or asynchronous in nature. Some are interoperational processes with external software elements or hardware devices.

UML diagrams are typically used for presenting perspective views even for systems that are not entirely object-oriented. Diagrams of any other kind can also be appropriate. For example, traditional dataflow diagrams can be used to show the flows in data exchange and transformation. Even simple flowcharts may be used to construct process-rich, multimode communication views.

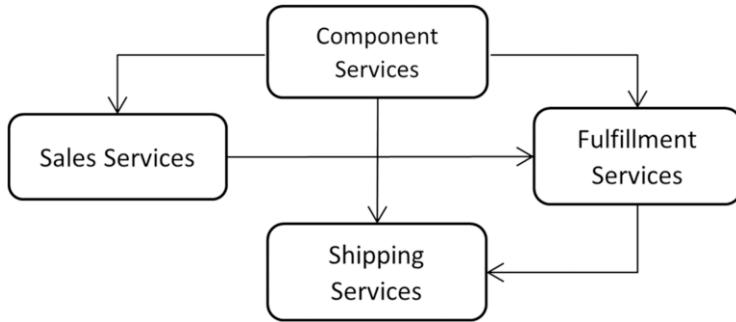
Because of many online or printed references, the discussions provided below about the views are relatively brief, focusing on understanding of the roles of the views in visualizing the architecture.

### 9.3.2 The Logic View

A logical view is to show the functionality that a system provides. It typically has a focus such as a concept decomposition, a visual description of entities and their relations, or an understanding of dynamic element interactions. Thus, a relatively comprehensive logic view generally consists of multiple perspective views that are of architectural significance. UML class diagrams are used for describing elements' relations, supplemented by other appropriate diagrams such as entity-relation diagrams or dataflow diagrams to describe data models. Sequence diagrams are typically used to describe the dynamic behavior of the system by showing interactions of the elements. When states can be meaningfully and adequately defined, state machine diagrams are often effective in describing what the system or a subsystem is.

When appropriate, a logic view may also be about an aspect, like a data aspect, that contributes to the system's functionality. It can also be helpful to include a view about the business rules to provide a context for other views if the rules contributed to the complexity of the system. Figure 9.5 is an interaction diagram that describes the software services an online commercial system provides. It is not a UML diagram yet appropriate given the service orientation of the design. It is a high-level container view about the software services, with each box representing a category of services. Each box, when helpful, can be expanded (or "zoomed" in) to show its composition such as the layers of sub-domain elements, their contextual boundaries, and event flows.

The 4+1 View Model was proposed decades ago when user interface design was still minimal (much less design of user experience). Today, it is hard to imagine that software architecture can exist entirely independent of user interfaces. Therefore, a logic view may also include how user interface elements are related to and interact with system operations.

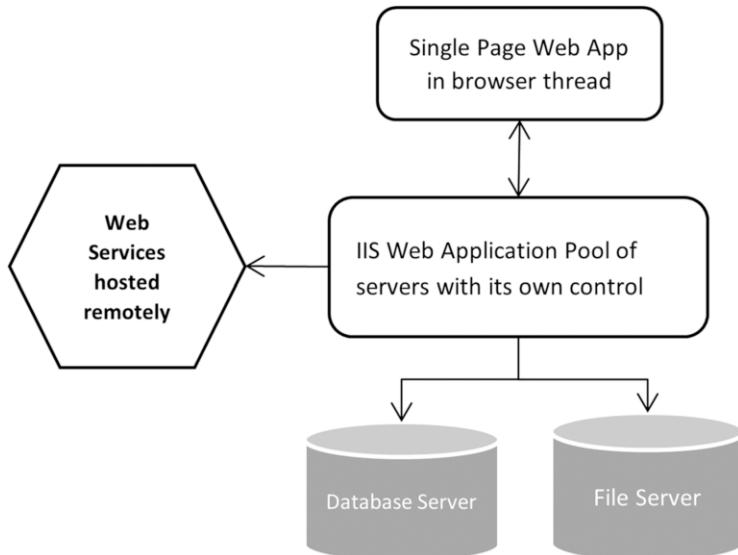


**Fig. 9.5** Interaction between software services

### 9.3.3 The Process View

The process view mostly considers nonfunctional aspects of the system. More accurately, consider the primary process being the operation of the software system. The system process may have its own subprocesses such as concurrent, distributed, or asynchronous processes that must be understood. However, there could be many other processes the system interacts with such as data running on its own process, communication processes it uses, external (software or hardware) processes it interacts with, and resource it relies on. Besides, expectations on system performance and integrity can also be included as appropriate, as can design decisions on process selection and maintenance. This view category can be further elaborated as follows:

- A process view may capture flows of inter-process information exchange such as exchange through REST API calls, as well as the sequence and timing of such inter-process communications. As appropriate, design decisions and potential issues such as reliability may also be included.
- A process view can be described at several levels of abstraction, each addressing a different concern. For example, at the highest level, the process architecture can be viewed as a set of independently executing networks of communicating programs distributed across a set of hardware resources or a collection of networks used to support separation between online and offline versions of the software system. Lower-level views would provide details or specifics.
- A process view may also describe processes that can be replicated for increased distribution of the processing load or for improved availability.
- When appropriate, a state machine diagram can be used to describe the control of a process with tasks in the process to be associated with a set of process states like start, recovery, reconfigured, scaled out, or shut down.



**Fig. 9.6** A process view of a web application

As an example, consider a web system that contains the following processes:

- IIS Web Application Pool
- Single-page application running in a browser
- Database engine running stored procedures or SQL scripts
- Web services used

Figure 9.6 is a high-level process view of this web application with again a non-UML diagram. Arrows indicate the direction of communication—one way or both ways. The diagram can be refined, however. A lower-level process view would further partition each process, when necessary, into a set of independent tasks or smaller processes (e.g., for buffering, time-outs, conversions, etc.), handled by separate threads of control.

A process view may also organize tasks into architectural-level tasks and local-level tasks to help better understand the nature of the tasks in the process. This separation can be useful for implementation reasons too. For example, local tasks may communicate via shared memory. Tasks at architectural-level communicate via a set of inter-task communication mechanisms such as synchronous or asynchronous communications. Such tasks can also be concurrent communication service tasks, remote procedure calls, or event broadcasting tasks.

Finally, a process view is also a view to show how the elements and abstractions fit within the process architecture. However, when a program is a single-threaded, all included process, the process view would have no architectural significance because the only other process is the hosting process. Thus, a process view may simply a depiction of dynamic

workflows of the system with only the main thread running, which can be already captured by logic views. In such a case, the view model may not include a process view, though rationale should always be provided.

### 9.3.4 The Development View

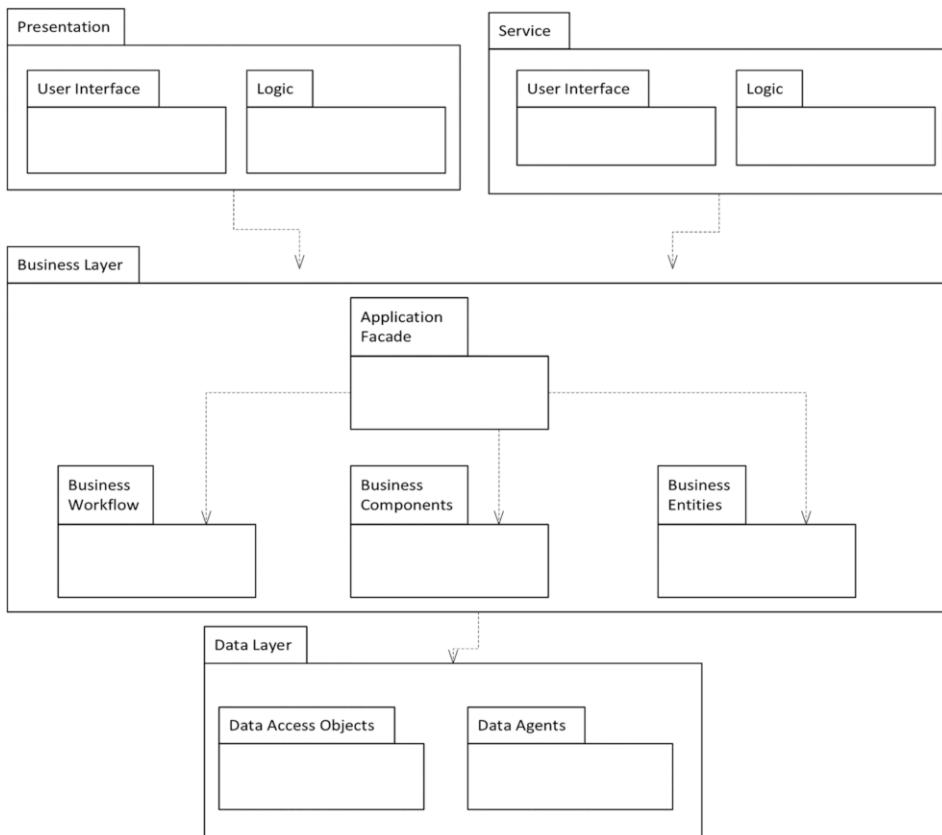
The development view is a visual description of the development architecture of the software focusing on organization of software elements in the development environment. A software application is packaged in chunks with program libraries, layers of subsystems, resources being used, etc. The subsystems may be further organized in terms of their own hierarchies of layers. In other words, a development view is about how code and resources are logically related with respect to their presence in storage media.

However, a complete development architecture can only be described when all the elements of the software have been identified. Thus, it is a common practice to impose a list of rules that governs the development architecture about partitioning, grouping, dependency relationships, visibility, and reuse of existing software elements. Documentation of such rules should also be part of the development view.

Development views can serve as the basis for teamwork allocation, cost evaluation, project planning, and monitoring project progress. The view can also provide some basis to reason about software reuse, portability, and security. Often, a development view is primarily a view about code organization with, if helpful, both physical and logical perspectives. A view of the physical perspective is comprised of views about resource repositories, file systems, and solution structure of an integrated development environment (used as the primary development platform). This physical perspective helps individuals and teams to manage version control and collaboration. The logical perspective is comprised of views about modules, their dependencies, and layering of modules. It helps developers understand how the code modules work together in terms of their static dependencies and the execution flow through the actual source code.

Figure 9.7 gives a development view in a logical perspective. The system is a generic business software program depicted using a UML package diagram. The system has three layers with business layer in the middle and data access layer at the bottom. When helpful, a mapping mechanism can be provided showing how logical perspective views are mapped to views of physical perspectives.

Commonly, we use UML package diagrams to represent a development view. However, the notion of “package” only exists in a few programming languages such as Java. Similar notions may bear different names such as “module” or “namespace” in different languages. To avoid confusion, a development view should also provide clarification between the terms used in the diagrams and similar terms supported by the language. Usually, there is a one-to-one correspondence between a development view of the system and code organization in the storage media, i.e., what we see in the view is what happens in code file organization. However, this is not always the case. Packages showing in the development



**Fig. 9.7** Development view of certain business software

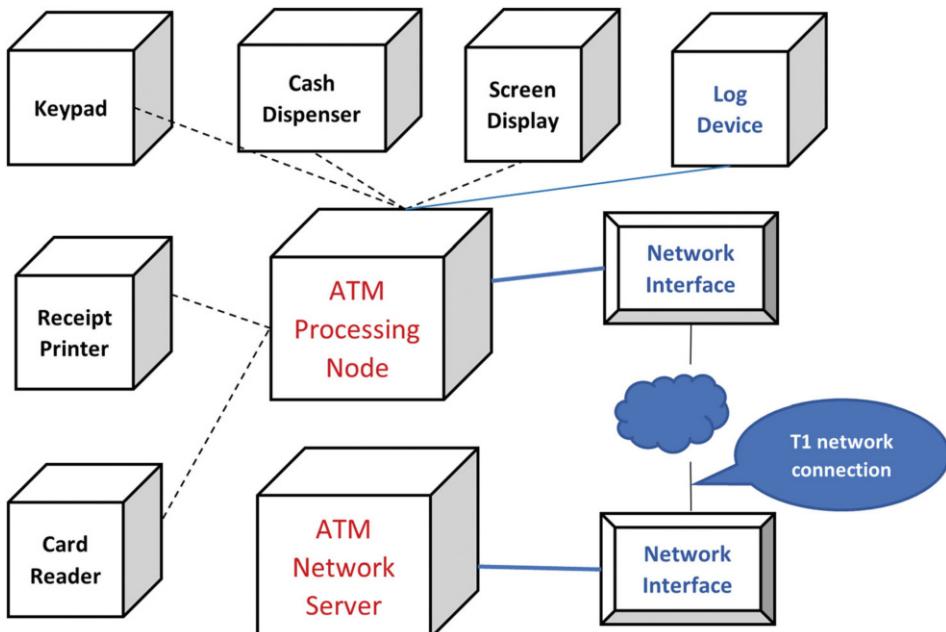
view may be implemented across several communicating code layers. It may also be possible that packages appearing in several conceptual layers in the development view require implementation in only one code layer. A development view should document such “discrepancies” to avoid unnecessary confusions, and appropriate mappings, as mentioned earlier, should help with such “mismatches.”

### 9.3.5 The Physical View

The physical view (also known as the deployment view) depicts the system, from a system engineer’s point of view, about the topology of software components of the physical layers, and the physical connections between these components. The following are a few scenarios about what a physical view may typically consist of:

- Software often executes on a network of computers or processing nodes. These are physical elements like servers, virtual machines, Docker containers, configuration files, etc. They can also be networking channels such as routers, firewalls, proxies, load balancers, etc. A physical view would show how such physical elements and various software elements of networking, processing, or tasking are mapped onto various processing nodes. (The mapping of the software to the processing nodes, therefore, needs to be highly flexible and have a minimal impact on the source code.)
- There could be several different physical configurations used in a software operation. Some are for development and testing, most for deployment of the system at various client sites for different customers. In such cases, there can be multiple parallel physical views or a primary physical view with multiple secondary views.
- Physical views can be critical in understanding system performance and capacity. For example, views may illustrate the association between hosting processing nodes and the system processes in the process view. This allows developers to start considering computing environment capacity, as well as latency and other performance implications.

Figure 9.8 is a physical view of an automated teller machine's operational environment. ATM processing node is the software that controls network devices and different physical devices for ATM operations.



**Fig. 9.8** Physical view of an ATM operational environment

Though UML deployment diagrams are typically used to represent physical views, many other diagrams of convenience can also be appropriate with notational legends as appropriate. With computing infrastructures like cloud computing and virtualized hardware, the physical architecture of a software system and its views might not be as critical as they were in the past.

Finally, the four views of this view model are not necessarily fully orthogonal or independent. Elements of one view can be connected to those in other views to provide additional viewing perspectives of the architecture. For example, a proxy object may appear in a logic view, but it can also appear in a process view as part of a remote service process. Logical views are always beneficial. Depending on the complexity of the software, other views can be optional. Software projects supporting only local business operations often lack process and physical complexity due to small scale, limited scope, well-understood operational environment, and a mature business process the software is based on. For such projects, physical and process views can be combined or simply skipped. In general, the larger and more complex the project is, the greater distinction there is between the views.

### 9.3.6 The “+1” View

Use cases provide original ingredients for the four architectural views. Use-case scenarios may convey the most important functional requirements and provide a context for initial identification of software elements and ways they interact. The process of software analysis and modeling produces the artifacts that we use to construct the four views. Thus, this “+1” view appears redundant. Nonetheless, the reason for the use-case view to be an integral part of this view model may include the following:

- It is used as a driver for discovering additional architectural elements as use cases evolve.
- As a structured form of software requirements, it is a direct input for designing test cases to validate the architecture.
- It provides a summary of the system’s functions, through use-case diagrams, to contextualize the four views.
- Use-case scenarios help tie the elements together architecturally, as different use cases help illustrate how logical, process, physical, and development components are working together to produce the software model.

This view could be spared when software does not have significant narratives that contextualize software requirements to support a meaningful architectural design. Often, another form of software requirements can be useful instead such as a scientific process, a complex mathematical formula, or an economic model. But in general, software

requirements are always used for architectural identification and design. Section 10.6 provides a case study using the 4+1 View Model.

---

## 9.4 The C4 View Model

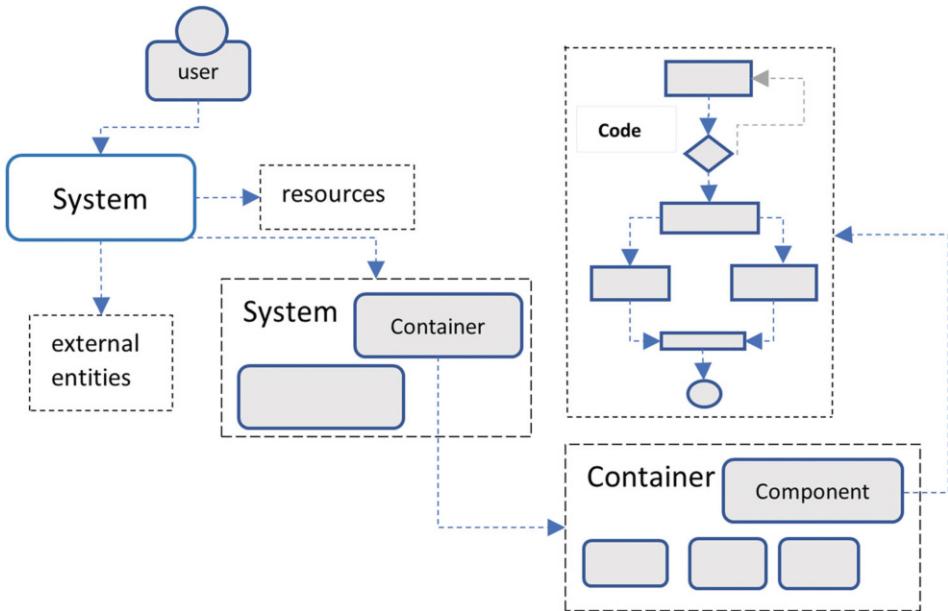
The C4 View Model (<http://c4model.com>) defines the context, container, component, and code views and is an abstraction-first approach to presenting software architecture with a layered decomposition. It uses five basic diagramming constructs, persons, software systems, containers, components, and relationships, with flexibility about their visual representations. It is encouraged to use simple diagrams to allow nesting of diagrams to facilitate interactive collaborative drawing. The model also promotes good modeling practices such as providing a title and legend on every diagram and clear labeling to facilitate understanding. This model was created with an intent to facilitate architectural collaboration and evolution in the context of agile teams where more formal documentation and upfront architectural design are not desired. The model lacks top-level structural diagrams in favor of more structurally flexible and open-ended notion of container.

A small set of constructs with flexible details makes this model easy to learn and use. Without more rigid UML diagrams used at top layers, the views are more likely to be consistent with the code. However, without an explicit top-level structural view of the architecture, one can argue that the eventual architecture may become a “moving target” with existed yet unspecified code structures. There could be structural erosion over time due to potential ad hoc practices, though remedies exist as will be discussed later.

### 9.4.1 The Layers of the C4 View Model

The *Person* construct of the C4 Model may represent a system actor, an external actor, or a source that triggers software interactions. The *Software System* construct is the highest level of abstraction and delivers the value of contextualizing the system to the audience. In many ways, software development teams “own” the software system as it is being developed; thus, teams must have a good understanding about the system’s contextual, operational environment.

The *Container* construct represents a data store or an aspect of the software application where modules related to that aspect are gathered. For example, a web application may have aspects of development in its front-end, back-end, data, local API or services, remote web services, and mobile interfaces. These aspects could be represented as containers. A container sets a boundary inside which modules are implemented, some data is stored, or some resources are gathered. Each container can be separately deployable, runnable with code units that can be independently developed. A container can also be a runtime environment running in its own process space locally or at remote. As such, communication between containers can take the form of an inter-process communication.



**Fig. 9.9** Zoom-in views of the C4 View Model

The *Component* construct represents a code unit or a grouping of code behind a set of well-defined interfaces to implement a functional aspect of the system that can be independently compiled. Therefore, it generally refers to a software component, consistent with the discussions we had earlier. However, because of the flexibility of the view model, “component” may also be interpreted broadly. For example, a dynamic link library can be a component too (which is compiled source file containing outprocess software elements used by in-process code).

Finally, *Core Diagrams* construct constitutes the code view, the bottom layer of the C4 Model to describe some critical components. There is no restriction about the kinds of code diagrams that can be included. Therefore, the code view is open to all appropriate diagrams that help better understand about the code logic, element relations, or interconnections of the critical modules. In particular, diagrams in logic views of the 4+1 Model can be appropriate core diagrams.

As Fig. 9.9 shows, each view level down is a “zoom-in” of the view immediately above. C4 layers appear to focus on the core software, but as commented earlier, appropriate views or view packets of the 4+1 Model can be adopted when helpful. Today, microservices have become an effective alternative to more traditional components with elements that are highly maintainable, testable, individually deployable, and runnable in their own processes. Thus, it is appropriate to add a microservice layer or replace the component layer with a service layer in a C4 view structure to provide a view on microservice distribution and their support structures.

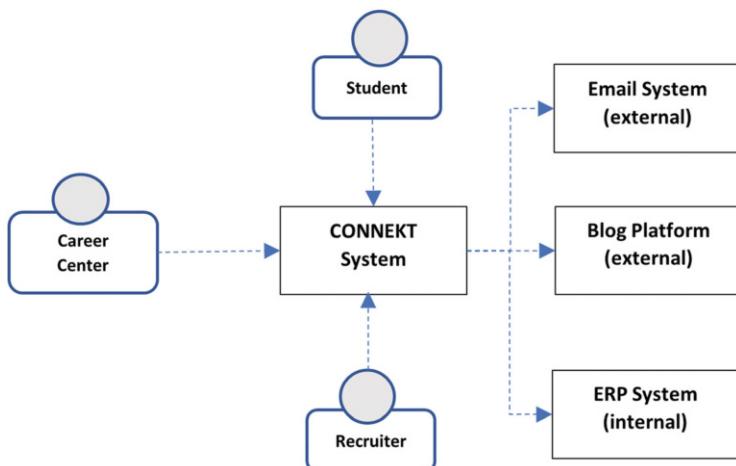
### 9.4.2 An Example

To demonstrate the use of the C4 Model, consider a hypothetical software application named CONNEKT, used by colleges' career centers for job recruitment. Users are current college students, recruiters, and college career centers' employees. The system uses three external systems:

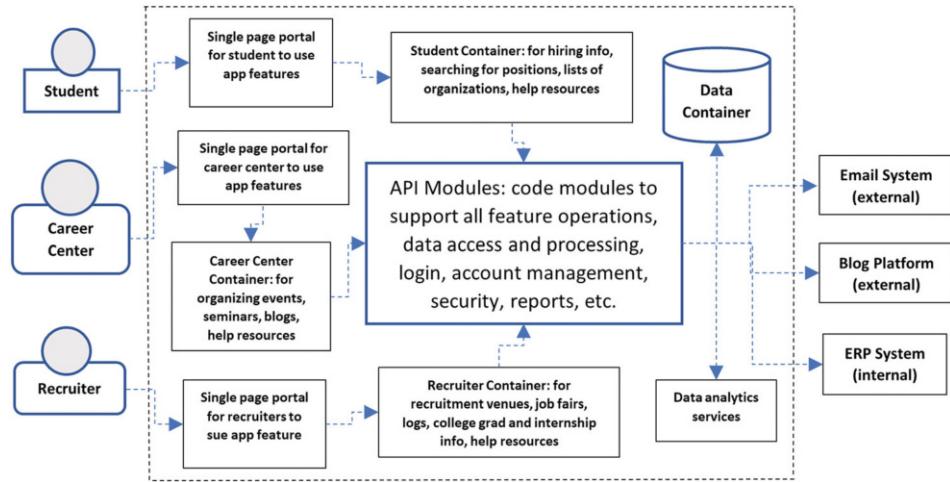
- The system needs to access an internal ERP system for student information verification and external payment processing (e.g., if recruiters want to rent school's facilities for recruitment events).
- The system uses an external blog platform for communication among the users.
- The system also uses an external mailing system through a proxy email server. Though there isn't much implication on the system, this external email system is viewable as part of the operational context.

These external systems are part of a context view. However, external code resources such as remote components or language's library modules may not be visible at the context level because they are not independently executable systems with their own service capabilities (though they may be visible at component layer or code layer if useful). Figure 9.10 is a context diagram of the system CONNEKT that uses essentially three constructs:

- *User*—a circle attached to a box
- *System*—a rectangular box representing the system to be built or existing systems the primary system depends on
- *Relation*—a dotted arrow representing a dependency relation



**Fig. 9.10** Context diagram of the system CONNEKT

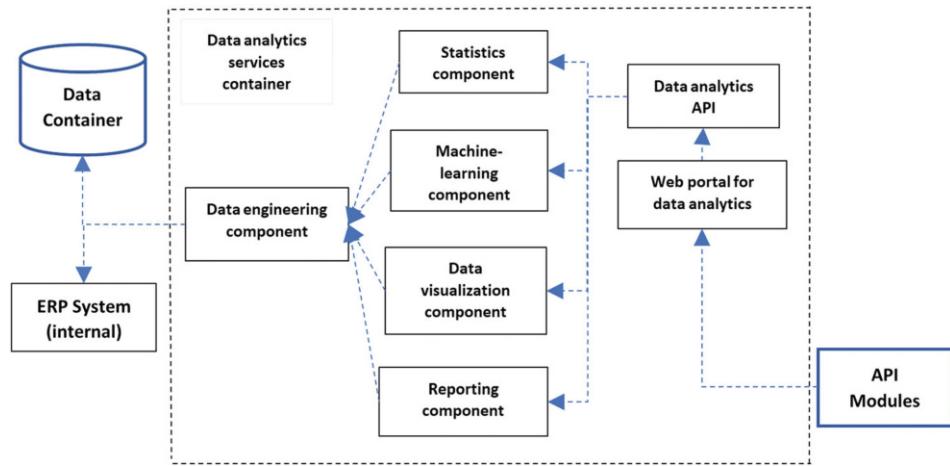


**Fig. 9.11** Container view of the system CONNEKT

Figure 9.11 is a container view, an elaboration of the system showing a functional division of the system; each container could potentially be deployed separately. We determine what each container may entail based on the role it plays and the software requirements, though we don't necessarily know the extent of the containment. As development advances, a container view will most likely evolve first as we gain more structural insights of the software. The arrows in Fig. 9.11 are not labeled, and there are no additional side notes either (for better readability). But even with annotated arrows and side notes, associated documentation is still generally needed to elaborate rationale of the design behind the containers. For example, the stakeholders of this view model might be interested in:

- How containers may communicate with one another, and with APIs in terms of communication mechanisms, protocols, and styles.
- What technology the implementation of a container may rely on (including front-end, back-end, and database technologies).
- How services of the API container can be accessed (directly or through proxies).
- Whether a mobile-compatible interface is also needed.

Some containers communicate with external processes, some are independently executable processes (such as single-page portals), and some (such as service containers) are service providers that may have their façade interfaces. Thus, an associated process view may be useful to further elaborate the communications among the containers and with other



**Fig. 9.12** Component view of the container data analytics services

processes. This process view may also help identify appropriate “structuring” of the containers and can be beneficial in terms of architectural stability of the views.

As the container view evolves, so should the relevant documentation. To make views always relevant, we may need a process to manage the views and their updates. For example, we might use or build a cloud service for container version control and maintenance of the current container view with appropriate linkage to the documentation. As container design is likely a collective effort, appropriate management of container view evolution is important for making informed design decisions and effective technical communications among development teams.

To zoom in on a container, we can create multiple component views to show how modules inside a container work together with their observable relations in delivering the functionality. Component views help developers understand nuances and interdependencies of the components. These views can be technical showing not only the inner workings of the components but also means of communication between the components.

Figure 9.12 is a component view of the container “data analytics services.” The container includes service operations of data analytics, which may rely on an ecosystem of the language (Python, R, etc.) for its implementation. An ecosystem may include code packages for statistical and numerical computation, data manipulation, data visualization, and machine learning. These packages should generally not be included in a component view because they do not require development effort. Again, a component view alone may not be adequate for all technical details about the components; it needs to be also accompanied with appropriate documentation. For example, with the contents of data analytics mostly produced using a convenient language like Python or R (with support of

an ecosystem), the audience may need to know an appropriate web platform that would facilitate the construction of a web portal.

Finally, a view about the core diagrams consists of diagrams that are helpful for code construction. As software construction becomes a primary focus, such diagrams can be fragile if they depict low-level code details and require frequent updates to remain relevant. This can be a challenge. To avoid the pitfalls, this view may include only diagrams that have architectural significance. For example, module-relation diagrams, component diagrams, or diagrams about frameworks' workflows are appropriate, as are business process diagrams and high-level dataflow diagrams.

To briefly summarize, the C4 View Model is a layered approach to architectural views with a benefit of focusing on high-level abstractions first. If we think of the 4+1 Model as a “horizontal” model with parallel categorical and perspective views, the C4 Model would be a “vertical” sequence of views with increasingly refined architectural details. Arguably, the container view is perhaps the most consequential or impactful view of the model. This view is based on a top-level functional decomposition with “fluid” contents of each container that can be swapped or regrouped. The connectors that hold containers together are not explicit in this view. Relations between containers are described with annotated arrows, but the information can often be inadequate. For example, dynamic interactions may happen across the containers such as one container communicating dynamically with a component in another container. Thus, the structure that holds all the containers together can be instable and become a moving target as the contents of the containers continue to shuffle around (and some risk management measures should be used as discussed earlier).

Finally, architectural views are constructed to last for a variety of reasons. Even though architectural views are of high level, they can still be vulnerable to changes in requirements, project scope, technology adoption, or even in management logistics. Therefore, an effective view model management process may be useful or even crucial to make view models sustainable. This management process would consist of not only cloud-based documentation tools, version control, and defined notations (as appropriate) to streamline diagram constructions but also risk control mechanisms.

---

## 9.5 Effective Development of Architectural Views

How can we effectively create architectural views that software stakeholders would appreciate? Professional standards (such as those created by IEEE or by the American National Standards Institute) may use the term *architecture framework* to mean capturing the conventions, principles, and practices when describing an architecture that has been established within a specific domain of application or a community of stakeholders. For example, colleges and universities use learning management software systems. An

**Table 9.3** Common architectural view categories

| View type               | Primary concerns                                                                                                                                           | Styles                                                          |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| Module                  | Modules and elements responsible for implementing functional requirements and their relations (association, dependency, specialization, or generalization) | Composition, generalization, aggregation or association, layers |
| Component and connector | How components interact at runtime, shared data store, element replication, data evolution, parallelism, structural change in execution                    | Pipe and filter, peer to peer, client-server                    |
| Allocation              | Processor allocation, code file identification during test, code builds, and assembling, assignments to development teams                                  | Any style for work assignment, deployment, or implementation    |

architecture framework of such a system captures terms and assumptions in course management that are common across colleges, a virtually shared set of educational principles in education delivery, and very similar practices in managing classes of an academic term. Developers must understand such commonalities, principles, and practices and build them into an architectural framework. However, colleges do have their own ways of managing their course offerings, maintaining school-specific educational policies, and designing term schedules for classes, school recess, and exams. An architectural framework must address the commonality and, more importantly, the different practices in various aspects of learning management. Such an architectural framework would, in turn, subordinate architectural frameworks in different areas of the system such as “course management,” “student advising,” “class registration,” etc. to better capture the commonalities and different practices across college campuses. Consequently, architectural views may also need to capture the conventions, principles, and practices that are applicable in those areas where views are constructed.

The views in the 4+1 View Model can be slightly reorganized around system’s code units, runtime interactions of the software with other processes, and development and physical organizations of the software. Table 9.3 explains these three essential view types, their primary concerns, and potential architectural styles to be used. According to the literature, these three view types are essential to understanding an architecture and provide basic guidance for constructing architectural views of any kind. For example, a stakeholder of a learning management system might be interested in viewing the architecture of the feature “course registration.” The view to provide would contain necessary information in the three view types about the feature. Any additional items can be added. For example, the person might also be interested in how school’s ERP system might affect the design of the modules related to “course registration.”

Effective development of architectural views is critical to effective communications with stakeholders. Practitioners have suggested a three-step process based on the structures that are inherently present in the software and on interests and concerns of the stakeholders:

**Table 9.4** An example of views constructed for different stakeholders

| Views                          | Stakeholders                 |                                   |                              |
|--------------------------------|------------------------------|-----------------------------------|------------------------------|
|                                | Db designer/<br>system engr. | Software engr./<br>programmer     | Operators/admin/<br>managers |
| Functionality view             | Data entity view             | Software engineering view         | Hardware view                |
| Software decomposition view    |                              |                                   |                              |
| Software/business process view |                              |                                   |                              |
| Software/business logic view   | Data logic view              | Application interoperability view | Software communication view  |
| Software organization view     |                              |                                   | Software processing view     |
| Software workflow view         |                              |                                   |                              |
| View of business rules         |                              |                                   | Software cost view           |
| Software reusability view      | Dataflow view                | Software distribution view        | Software standard view       |
| Work assignment view           |                              |                                   |                              |
| Service view                   |                              |                                   |                              |
| More applicable views, etc.    | More customized views, etc.  | More customized views, etc.       | More customized views, etc.  |

1. Produce a list of candidate views with a stakeholder view table, illustrated in Table 9.4. The first column contains the “standard views” we can construct based on the three view types above. The rest of the columns are candidate views various stakeholders might be interested. Each cell represents a candidate view certain stakeholders might be interested and an association with some of the “standard views.”
2. Combine the candidate views from the first step to yield a practical number of views. Views can be combined if they share significant similarities, lack depth, or serve very few stakeholders. When necessary, stakeholders should be consulted to confirm if they could be equally well served by other views that have stronger constituencies. Combined views are then designed.
3. Prioritize delivery of views based on the urgency and other priority measures. For example, the project management of a company often desires attention and information early and often, and their needs can be catered first when appropriate and feasible.

There are other aspects to consider when constructing views for stakeholders. For example, users of a system might be interested in views from the perspective of how they interact with the system, such as system availability, response time, access to required information, or workflow overview. But they might not be interested in any operation details of the system. Developers of the system, in contrast, need views from very different perspectives, such as ways software is connected to hardware distributed over a network. However, different types of developers (for database, security, etc.) may not necessarily appreciate the same views derivable from the model. Regardless of the views they may

**Table 9.5** The concepts appropriate for a system engineering architectural view

|                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| View concept        | Elaboration                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Viewing perspective | To understand how to assemble software and hardware components into a working system                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| The stakeholder     | The system engineers                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Concerns            | Ways the system can be implemented and assembled from the perspective of hardware, software, and networking and, specifically, the location, modifiability, reusability, and availability of all components of the system, including a technology overview about what is available currently or in the future and its impact on the architecture                                                                                                                                                                                                                                                                                                  |
| View effect         | This view may help the stakeholder better understand the system requirements to ensure that the appropriate components are developed and deployed in an optimal manner. This view may also be beneficial in the selection of best configurations for the system                                                                                                                                                                                                                                                                                                                                                                                   |
| Key issues          | For an information system, the issues may include: <ul style="list-style-type: none"> <li>• Computing models appropriate for a distributed computing environment</li> <li>• Whether to support the migration of legacy systems</li> <li>• Elaboration on the model being used (with technology advancement, terminologies literature has used often need clarification, e.g., models like host-based, master/slave, multitiered, peer-to-peer, and client/server might overlap in some respects but very different in others)</li> <li>• System functionality in terms of modules for presentation, application, data management, etc.</li> </ul> |

desire, stakeholders can be limited by the views to understand how the system works in general. Therefore, a brief overview of the system can be attached to any customized view to provide a viewing context. There are other shared interests such as a view of the process set up for users to communicate problems directly to the development team.

IEEE standards for software architecture also recommended development of key concepts and terminology that can help us better understand the concerns of key stakeholders. By understanding their concerns, we would put their concerns in perspectives, document the concerns, and, more importantly, effectively deal with those concerns. For example, to develop an effective system engineering view of the software, we might focus on those concepts listed in Table 9.5.

Last but not the least, documenting software architecture has progressive importance as the product matures, evolves, or transforms. As an essential element of an architecture, architectural forms must be documented. A traditional way to do so is to use a tabulated list of the properties of the elements and the relationships between the elements for easy lookup, including a description about the interaction mechanisms of the connectors and components. With object-oriented methodologies becoming standard, traditional expressions of architectural forms are replaced by UML artifacts to describe elements' structural compositions and relational connections. Architectural views constructed for

stakeholders may be of only temporary value for different reasons, whereas those constructed with lasting value must be documented. To streamline the process of constructing views, we document important view packets that can be used to construct a variety of views for different purposes.

Finally, the rationale for the architecture and its constraints must also be documented, including justifications for the constraints and technological assumptions behind the architecture. The following is an abbreviated list of the contents of an architectural document:

1. An overview of the architecture
  2. Element catalog (elements and their properties, relations, interfaces, and behaviors)
  3. Context diagram
  4. Variability guide
  5. Background of the architecture (rationale, analysis results, constraint, and assumptions)
  6. Related view packets
- 

## 9.6 Summary

A system's architecture is a critical aspect of a software product for its sustainability. It represents the design decisions related to overall system structure and behavior. An architecture structures software elements but may also provide a guide, through an architectural style, about how software elements are designed. There are essentially three kinds of software elements. Framework elements define software structures, domain elements address the software's functionality, and design elements connect functional elements to the architecture. Software architecture plays a central role in an entire process of software development and evolution. It serves as the conceptual glue that holds every phase of a development process together and as technical center for software estimates, risk management, process adoption, and continuous system evolution. The architecture of a system helps stakeholders understand how the system will satisfy the software requirements with specified quality attributes. Architectural iterations may trigger detailed design iterations, assessments on the deployed system for maintenance needs, and progressive evaluations on architectural fitness with respect to an organization's business goals.

Not like an architecture in a traditional sense, a software architecture is not a "static" product that exists before construction takes place, and it can evolve with changes that take place in a development process. It is thus often not realistic to ask for the entirety of an architecture. While we try to protect an architecture's overall structural stability and integrity, we do make structural adjustments to address emerging problems, issues, and concerns.

Architectural views are a primary tool we use for stakeholders to understand how the software system may work and possess the expected quality attributes. Architectural views

may also be valuable for providing guidance for nonarchitectural design activities. The 4+1 View Model is a perspective view model to address:

- What the elements are and how they are related and interact.
- How the system functions as a result of different processes working together.
- How code artifacts are organized, structured, and distributed.
- How the system is present physically in its operational environment.

In contrast, the C4 View Model is a layered view model organized around a context, containers, components, and core visual artifacts. These view models can be used separately, concurrently, or jointly in some fashion as appropriate. They provide conceptual foundations for constructing many perspective views with varying architectural focuses to meet a variety of needs of stakeholders.

## Exercises

1. Describe what software architecture is in your own words.
2. What is an architectural view?
3. What are the aspects of a software system that an architecture must address?
4. Is a primary control of a software system to determine the architecture of a software system? Why or why not?
5. Why would it be unrealistic to provide a comprehensive set of architectural artifacts for stakeholders to choose from?
6. Describe how software architecture impacts every aspect of software development from requirement elicitation to product maintenance and evolution.
7. For the 4+1 View Model, the logic view is about how the software works as designed elements work together, and process view is about how the software works as processes involved work together. Is this understanding correct? Why or why not?
8. Assume you want to write a Java program that animates a ball moving freely and getting bounced when hitting an edge (and then moving in the reflected direction). The ball is manipulated with a separate thread. Give a logic view of the 4+1 View Model focusing on how the program might work at a high level and a process view focusing on how the ball thread interacts with the main thread.
9. For the animation scenario in Question 8, use the C4 View Model to provide the context, container, and component views of the same design.
10. Consider the views that you produced in Questions 8 and 9.
  - (a) Would the combination of the views provide a better coverage of the program's architecture?
  - (b) Are there any redundancies in the combined views?
  - (c) How would you reorganize, modify, or reproduce some of the views in the combined collection to produce better architectural artifacts for the program?

11. Suppose you are assigned to work on a sign-up/log-in web element, which would be used in all user interfaces for the services the company provides. Provide a logic view of the 4+1 View Model about how the code modules work in principle at a more abstract level and a process view to describe how the modules may interact with other processes.
12. Notepad is a simple document editor of Windows for years.
  - (a) Use the 4+1 View Model to provide logic, process, and physical views of the software from your perspective.
  - (b) Use the C4 Model to provide context, container, and component views of the software.
  - (c) Which model is a better fit for showing the architecture of Notepad?
13. Investors of a software product are a stakeholder too.
  - (a) What views listed in Table 9.4 might they be interested?
  - (b) Develop some view concepts related to their interests in the software and provide an elaboration of each using a table like Table 9.5.
14. Assume you are working on a software project for a course in software engineering, for which your instructor is a stakeholder. You are required to document your work to adequately show your effort in the learning process.
  - (a) What views listed in the Table 9.4 might your instructor be interested?
  - (b) The instructor would be interested in your understanding in learning software life cycle principles and practices. Develop some view concepts (and elaborate each) about the learning outcomes that would guide you in constructing the views (the instructor would appreciate).

---

## References

- Braude, E. (2004). *Software design: From programming to architecture* (1st ed.). Wiley.
- Buschmann, F., et al. (1996). *Pattern-oriented software architecture. Vol. 1: A system of patterns*. Wiley.
- Buschmann, F., et al. (2000). *Pattern-oriented software architecture. Vol. 2: Patterns for concurrent and networked objects*. Wiley.
- C4model.com. *The C4 model for visualizing software architecture*. <https://c4model.com/>
- Hofmeister, C., et al. (1999, November 14). *Applied software architecture* (1st ed.). Addison-Wesley Professional.
- Kruchten, P. (1995). Architectural blueprints-The “4+1” View Model of software architecture. *IEEE Software*, 12(6), 42–50.
- McBride, M. (2007, May). The software architecture. *ACM Communications*, 50(5), 75–81.
- Perry, D. E., & Wolf, A. L. (1992, October). Foundations for the study of software architecture. *Software Engineering Notes, ACM SIGSOFT*, 17(4), 40–52.

## Further Reading

- Baragry, J., & Reed, K. (2002, August 07). Why we need a different view of software architecture. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*.
- Clements, P., et al. (2011). *Documenting software architecture – View and beyond* (2nd ed.). Addison Wesley.
- Taylor, R., et al. (2010). *Software architecture, foundation, theory, and practice*. Wiley.



# Design Case Studies

10

---

## 10.1 Overview

This chapter is to present some design case studies to reinforce what has been discussed in early chapters. A successful design solves a class of problems, not a single instance. A successful design attempt has a focus yet does not restrict possibilities. A sustainable design always starts with an analysis on commonality and variability of the system, its parts, and the potential software elements. The case studies were not intended to be complex; rather, they were designed to demonstrate the use of design principles, practices, and techniques introduced in early chapters in a relatively straightforward way. They also have different focuses. Some focus on code organization and modularity while others on the appropriate use of data abstraction and inheritance hierarchies. For most of the case studies, code that may be of interest is also partially or fully included. The use of the architectural view models is also demonstrated in two of the cases. The examples are all desktop applications (though data could be accessed in a network environment). There are a few reasons why desktop applications were chosen:

- They are more likely self-contained compared with web applications.
- A web framework often imposes an architectural style such as the MVC style, which can be too technical to introduce and restrictive in terms of other design opportunities.
- With web development front-end technologies, such as JavaScript, React, and Angular, increasingly more powerful, one might feel like developing desktop application in a web runtime environment.

It is hoped that these case studies also provide contexts where design is not only about high-level abstractions but also low-level details. The case studies should be accessible completely after Chap. 6 or much earlier with optional UML diagrams.

## 10.2 Two Simple Case Studies

Design does matter even for simple software applications. The reason is that simple applications are more likely to be modified and extended, which requires an appropriate design to do so. The two case studies presented in this section are very different, one is about a public software service, and the other is a software tool. Regardless of their simplicity, evaluation of variability of the requirements is still among the very first things we do.

### 10.2.1 A Simple Service Application

Local governments often have service offices (such as offices for vehicle registration and driver's licenses), and consumer service companies often have service stores (for a cell phone, Internet, or cable TV services). If we go to such an office or store, we often use a touch screen to enter a few lines of information. By doing so, we are placed in a virtual wait line, which is used by store's service staffers. This case study is to simulate this process in a consumer service store. To be specific, a customer is required to enter the following information:

- First name and last initial
- Whether you have an appointment
- The purpose of the visit to be selected from a list such as:
  - TV service
  - Internet service
  - Cell phone service
  - Steaming service
  - Other

For staffers, the only thing they do with the application is to retrieve information about the front customer of a virtual queue to render the requested service.

Analysis is straightforward with two obvious entities to simulate—customer and staffer. However, no matter how simple this use-case scenario is, data must be handled, and the software process must be controlled. What might the requirements evolve going forward? Intuitively, information a customer enters may vary in the future, as might the way a staff member handles a service request. Thus, we use the following four data abstractions:

```
interface Customer{ String enterInfo(); }
interface Staffer{ void renderService(String custInfo); }
interface InfoHandler{
 void saveInfo(String info, String filename);
 String retrieveInfo(String filename);
}
interface AppController{
 void appInit();
 void runApp();
}
```

The following is a controller class implemented against the abstractions:

```
class ServiceController implements AppController{
 Customer attCustomer;
 Staffer attStaffer;
 InfoHandler handler;
 @Override
 public void appInit() {
 attCustomer = new ATTCustomer();
 attStaffer = new ATTStaffer();
 handler = new StoreVisitInfo();
 }
 @Override
 public void runApp() {
 String input = JOptionPane.showInputDialog("User or Service Staffer? (1 or 2)");
 if(input.equals("1")){
 String info = attCustomer.enterInfo();
 handler.saveInfo(info, "customer.txt");
 }
 else{
 String custInfo = handler.retrieveInfo("customer.txt");
 attStaffer.renderService(custInfo);
 }
 }
}
```

Despite the simplicity, the method *runApp()* runs the business process () and hence is a higher-level code module. Its implementation, therefore, should not contain low-level details such as information software collects from customer and how this information is used by staffers. This code is also independent of how data is handled, as data-handling details are also hidden. This is a simple example of programming to an interface, not to an implementation. A few implementation details might be useful in other situations:

- There are different ways to writing texts into a file and retrieve them later. The way implemented follows first-in-first-out order. Thus, an in-memory queue is not necessary.
- Removing the first line (i.e., information about the front customer) does require retrieval of all content from the file; saving the lines, except the first, into a collection; and writing them back into the file, though the code is still straightforward (see method *retrieveInfo* below).

The rest of the code is given below:

```

class ATTCustomer implements Customer{
 @Override
 public String enterInfo() {
 String first = JOptionPane.showInputDialog(null, "first name");
 String lastInit = JOptionPane.showInputDialog(null, "last initial");
 String hasAptm = JOptionPane.showInputDialog(null, "has appointment? (1/2)");
 String whyHere = JOptionPane.showInputDialog(null,
 "why here:\n1. TV Service\n2. Internet\n3. Phone\n4. Streaming Service");
 return first + " " + lastInit + "," + hasAptm + "," + whyHere;
 }
}

class ATTStaffer implements Staffer{
 @Override
 public void renderService(String custInfo) {
 String[] services = new String[]{ "Cable", "Internet", "Phone", "Streaming"};
 String[] parts = custInfo.split(", ", 3);
 String custName = parts[0];
 String aptm = parts[1];
 String service = parts[2];
 JOptionPane.showMessageDialog(null, custName + " needs: " +
 services[Integer.parseInt(service)] + "\nHi, I am taking care of it now");
 }
}

class StoreVisitInfo implements InfoHandler{
 @Override
 public void saveInfo(String info, String file) {
 try{
 FileWriter writer = new FileWriter(file, true);
 PrintWriter out = new PrintWriter(writer);
 out.println(info);
 writer.close();
 out.close();
 }
 catch(IOException e){
 System.out.println(e.getMessage());
 }
 }
 @Override
 public String retrieveInfo(String file){
 File fpath = new File(file);
 String firstLine = null;
 try{
 Scanner scanner = new Scanner(fpath);
 ArrayList<String> lines = new ArrayList<String>();
 firstLine = scanner.nextLine();
 while (scanner.hasNextLine()) {
 String line = scanner.nextLine();
 lines.add(line);
 }
 scanner.close();
 FileWriter writer = new FileWriter(fpath);
 for (String line : lines) {
 writer.write(line + "\n");
 }
 writer.close();
 }
 catch(IOException e){
 e.printStackTrace();
 }
 Return firstLine;
 }
}

```

Simple variations of the application include:

- Customers with appointments may be given a higher priority to receive a service than those walk-in customers.
- Staffers may use a log file to make notes for certain services that require follow-ups.

Only a minor code modification is needed to implement each of the variations. For example, we can perform a simple search while retrieving all text lines from the file to find the first customer who had an appointment. These variations can be good exercises.

### 10.2.2 A Generic Data Sorting Framework

Java library collections *Arrays* and *Collections* provide overloaded sorting operations for data of many different types. These methods also accept a *Comparator* object, which defines a sorting order. They are mostly adequate for meeting various sorting needs. This case study is just a different approach to implementing sorting tasks with a simple framework. Our goal is to meet the following requirements:

- It sorts data of any type.
- It allows choice of an algorithm.
- It accepts a *Comparable* instance.
- It provides a way for user to extract data from a data source or define a particular order in raw data, which can be useful for algorithm comparison.

To elaborate the last requirement, users may choose to use data preloaded into an array or a list. They might also choose to build an input interface to manually enter data before sorting it or allow data to come from a data source. This sorting process could be part of a “pipeline” where data flows from one process, gets sorted, and then flows into another. The following interface provides an abstraction to allow variations of a data source with data of any type:

```
interface IData<T>{
 <T> T[] getData();
}
```

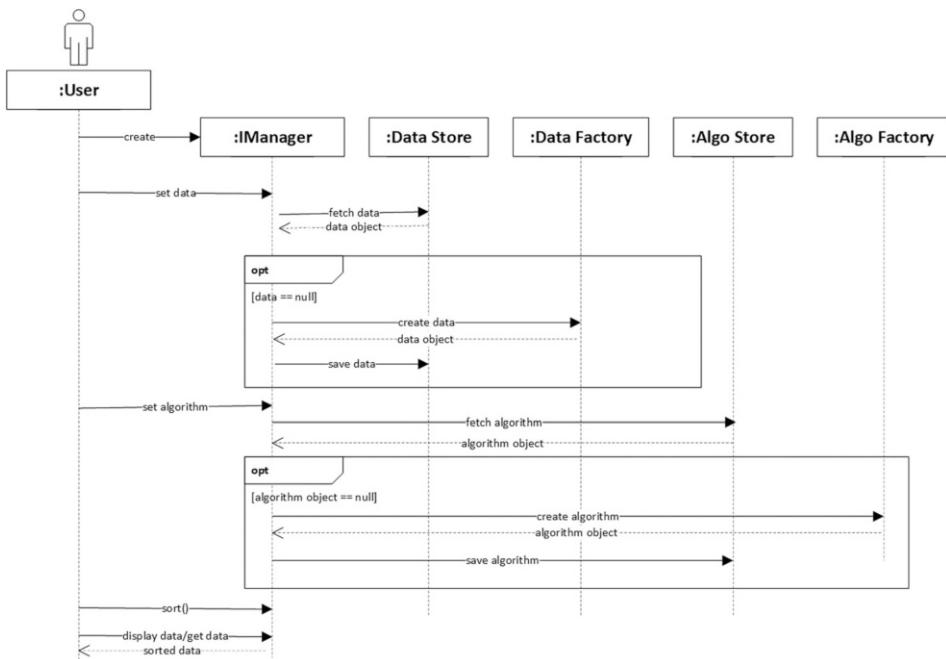
The abstraction for accommodating algorithmic variants is straightforward:

```
interface ISort<T>{
 <T> T[] sort(T[] data, Comparator<T> c);
}
```

The following (static) generic method is implemented against the above data abstractions through two factory methods:

```
static <T> T[] processSorting(Comparator<T> comp, String dataKind, String algoSelection){
 T[] data = DataFactory.getDataObject(dataKind).getData();
 ISort algoritm = AlgoFactory.getSortingAlgorithm(algoSelection);
 return algoritm.sort (data, comp);
}
```

There are interesting variations of the framework. For example, with the interfaces above, we can use a context object to keep a collection of existing algorithms and raw datasets as “flyweights” for future use. When desired raw data or algorithm exists in the collection, it would be reused. Otherwise, a new data or algorithm object is created, used, and then saved into the collection. Figure 10.1 is a sequence diagram showing the object communications based on this variation behind the idea of the Flyweight pattern. To keep the diagram simple, the process of object creation in a factory method is simplified. The



**Fig. 10.1** Sequence diagram for a simple data sorting management framework

two entity stores (one for data and the other for algorithms) can be implemented in any conventional way. For example, maps are appropriate. The following code is a high-level data sorting process (to be executed in the *main* method). It hides all implementation details (which are exposed in the sequence diagram Fig. 10.1):

```
IManager<Integer> mgr = new SortManager<Integer>();
mgr.setData(DATA_KIND.int_random);
mgr.setComparator(comp);
mgr.setAlgorithm(ALGO.heap);
mgr.sort();
mgr.displaySortedData();
}
```

---

## 10.3 A Card Game

Object-oriented design generally works well on simulations as software entities designed often match well with real-world objects. Simulation of a card game makes an interesting, yet still relatively, simple case for applying object-oriented design. This case study is to create a game structure that can be applied to many card games with multiple players, though the game known as Old Maid is used to demonstrate the design. Despite possible variations of the play rules for Old Maid, the following rules are assumed:

- The standard 52-card pack is used with one of the four queens being removed, leaving a total of 51 cards.
- The goal is to form and discard pairs of cards by the players with one card left at the end (a queen), and the player having the card is the *Old Maid* and loses the games.
- Before game start, the card pack is shuffled and dealt around, one at a time, to each player, until all the cards have been handed out. Players do not need to have an equal number of cards.
- Each player removes all pairs (i.e., two cards having the same value) from the hand. To start the game, a randomly picked player offers the hand (cards spread out face down in a face-to-face play) to a neighboring player to draw one card from (and the same order is followed throughout the game). This player discards any pair that may have been formed by the drawn card. The player then offers the hand to the player following the same order. Play proceeds in this way until all cards have been paired except the queen that cannot be paired.

### 10.3.1 Analysis

Although model-view-controller is still a relevant style, a card game model and control appear not separable. In this case, it is appropriate to design software elements based on their real-world counterparts like card, card deck, and player. Further analysis may reveal whether appropriate modifications to these domain objects would be needed. We look at player entity. The responsibilities of a player are to play based on the game rules and manage player's hand of cards. For different games, a player may need to organize cards in very different ways. Therefore, how a player plays in a particular game is generally not reusable. However, ways a hand of cards can be organized may be common to all games. For example, a hand of card can be organized by rank or gathered by suit. Therefore, it can be helpful if we create an entity to model a hand of cards to offer operations on commonly used ways cards are organized or gathered. In other words, this is a service entity to facilitate a play move independent of any specific game rules. An operation to find two cards that have the same rank regardless of the suits is specifically applicable to the game of Old Maid, but it can be useful in other games too. Thus, object type Hand was created with a dependency relation—a player depends on its hand (operations) to play.

Next, we are interested in a game-play control structure that can also be applied to other games. We look at stable parts of a game play and the parts that can vary in different games. We observed that to abstract the following are common to all card games:

- A game must be initialized appropriately (such as cards being shuffled, etc.).
- A game may have a particular way to start (e.g., for the game of Old Maid, Jokers and one queen are removed).
- Players take turns to play in certain order. The concept of play round, i.e., every player has played once, is also a common process across different games.
- After each play round, we can query whether game is finished with an abstraction on a completion criterion.
- There may be also a process to end game.

If above processes are abstracted with abstract methods, a game control can be implemented against these abstract methods. This generic control can thus be used across different card games and is a primary operation of a main entity of the framework—an abstract class.

Certain processes are shared across different games such as shuffling cards, dealing cards, etc. Therefore, entities to simulate cards and deck of cards can be implemented with

**Table 10.1** Entity, responsibility, and collaboration of a card game framework

| Type                          | Responsibility                                                                                                                                                                                                                                                         | Collaborator                              |
|-------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| Card                          | Representation of values and suits including Jokers, as well as string representation (like <i>Jack of Diamond</i> ) for each card                                                                                                                                     | None                                      |
| Deck                          | Representation of a deck of cards with operations to shuffle a card deck, deal the cards, etc.                                                                                                                                                                         | Card                                      |
| Hand                          | Representation of a hand of cards <ul style="list-style-type: none"> <li>• Add, remove, or return a card</li> <li>• Clear hand</li> <li>• Sort hand by suit or by value</li> <li>• Remove a matching card by suit or by value</li> <li>• Remove pairs, etc.</li> </ul> | List of cards                             |
| Player (abstract class)       | Representation of a player <ul style="list-style-type: none"> <li>• Operations needed to implement specific game rules</li> </ul>                                                                                                                                      | Hand                                      |
| Game control (abstract class) | Representation of a general card game with concrete and abstract operations discussed earlier with the code given in the next section                                                                                                                                  | View, deck, a list of player objects      |
| View                          | Representation of a user interface <ul style="list-style-type: none"> <li>• Operations to process user input, display game status information, etc.</li> </ul>                                                                                                         | <i>System.out</i> ,<br><i>JOptionPane</i> |

concrete classes (which are less likely to be changed later). Table 10.1 summarizes the object types we have discussed.

As mentioned earlier, object type *Hand* is a service entity. However, it is not possible to pre-implement all useful ways of how we might organize or gather a hand of cards. Therefore, subclasses of *Hand* are useful with potentially a deep hierarchy. Though deep hierarchies are generally discouraged, but *Hand* makes a rare exception that a deep hierarchy can be safe because all operations are stable. An implementer of a card game using the framework can subclass *Hand* and provide additional operations for manipulations of cards specific for the game but potentially useful in other games. As appropriate, a *Hand* flyweight factory can also be created to be part of the framework with a “catalog” of various versions of *Hand* and provide an enumerated name for each. Might a *Hand* library be warranted?

Abstract class *Player* encapsulates an instance of *Hand*, which is used in abstract method *play* to implement game-specific play moves. Game control should not access *Hand* object, which contains the details of a game, so they both should depend on an abstraction (which is *Player*). A *Player* object should not make a *Hand* object available externally either, as this would be a violation of Law of Demeter. In other words, game

control should not “play” for players, and players play for themselves (via *play* method). Because a player may draw cards from another player or from a card pile and remove cards according to game rules, the method *play* accepts a list of cards as a parameter and returns a list of discarded cards.

Finally, this case study assumes that players are all “pre-implemented,” meaning they all apply the same play strategies as implemented. A scenario of involving a human player (to play a game in real time) can be interesting. The abstraction *Player* can still work with a separate subclass for a human player if cards the player must draw are drawn in advance (meaning not in real time). A real-time draw of cards needs an access to a card pile or a player’s hand, or both may complicate a design in certain ways. One design option would be to create another abstract method in the game control class that returns an instance of *Player*, which is outlined below:

```
public Player getHumanPlayer(){
 //local declarations if needed
 return new Player(){
 ...
 public List<Card> play(List<Card> cards){ }
 };
}
```

A local inner subclass can access data in the game control class to allow real-time data manipulations. This can be an interesting exercise to explore the implementation details.

### 10.3.2 Two Abstract Classes

Various implementations for card, deck, and hand can be easily found in the public domain (though they also warrant reimplementations as interesting exercises). Therefore, only two abstract classes primarily analyzed in the last section are given below. Also listed below is an interface *IGameView* that handles input and output using console or simple dialog boxes with *JOptionPane*:

```
public abstract class GameControl {
 protected IGameView view = new IOHandler();
 protected Deck deck;
 protected List<Player> players;
 public GameControl(){
 deck = new Deck();
 deck.shuffle();
 players = new ArrayList<Player>();
 init();
 }
 public void runGame(){
 int numRounds = 1;
 Character input = view.getInput("Play? (t/f) " + "?");
 if(input != 't') return;
 do {
 startGame();
 do{
 view.display("Round " + numRounds + ":");
 playRound();
 view.display(playersHands());
 numRounds++;
 }while(moreRounds());
 endGame();
 } while (((char) view.getInput("Play again (t/f) " + "?")) == 't');
 }
 abstract void init();
 abstract void startGame();
 abstract void playRound();
 abstract void endGame();
 abstract boolean moreRounds();
 private String playersHands(){
 String hands = "";
 for(Player p : players){
 hands += "Player " + p.getName() + "\n" + "hand :" + p.handString() + "\n";
 }
 return hands;
 }
}

public interface IGameView{
 void display(String message);
 <T> T getInput(String msg);
}

public abstract class Player {
 protected Hand pHand;
 protected String playerName;
```

```
public Player(String name){
 pHand = new Hand();
 playerName = name;
}

public String getName(){ return playerName; }
public int numCards(){
 return pHand.getCardCount();
}
public void getCard(Card c){
 pHand.addCard(c);
}
public Card giveCard(){
 if(pHand.getCardCount() == 0) return null;
 Random rd = new Random();
 int i = rd.nextInt(pHand.getCardCount());
 Card toReturn = pHand.getCard(i);
 pHand.removeCard(i);
 return toReturn;
}
public Card giveCard(int position){
 int count = pHand.getCardCount();
 if(count == 0 || (position < 0 || position >= count)) return null;
 Card toReturn = pHand.getCard(position);
 pHand.removeCard(position);
 return toReturn;
}
public boolean isEmpty(){
 return pHand.getCardCount() == 0;
}
public String handString(){
 String str = "";
 for(int i = 0; i < pHand.getCardCount(); i++){
 str += pHand.getCard(i).toString() + ", ";
 }
 return str;
}
public void removePairs(){
 pHand.removePairs();
}
public abstract List<Card> play(List<Card> cards);
}
```

## 10.4 A Framework for Animation

Design of an animation program can be interesting because of different programming aspects involved—event driven, threads, user interface design, and managing animation scenarios. Animation works because of the optical illusion. By presenting a sequence of still images in quick enough succession, the viewer interprets them as a continuous moving image. This case study is to design a framework for animations that use the same presentation interface. A panel, embedded in a frame, is used to host an animation, managed by a separate thread.

### 10.4.1 Analysis

Because of the infrastructural assumptions we made, we can focus on the design of a structural presentation of an animation. Every animation task has a “story” to tell—a presentation of visual objects and ways they change locations and appearances. A “story” can be abstracted as a sequence of steps:

1. Create the image objects and draw them at their initial locations.
2. Update the locations.
3. Redraw the image objects.
4. Repeat steps 2 and 3 as many times as necessary with a microtime pause in between.
5. End animation with necessary cleanups.

Based on these steps, we can design an abstraction with abstract methods corresponding to the steps, which captures the variability in animation stories:

```
interface AnimationStory{
 void init();
 void updateStory();
 void paintStory(Graphics g);
 boolean storyEnds();
}
```

The responsibilities of these methods are summarized in Table 10.2.

An abstract class that plays a primary role in the framework is listed below with all stable parts implemented. Here are some of details of the class:

**Table 10.2** Responsibilities of the methods in interface *AnimationStory*

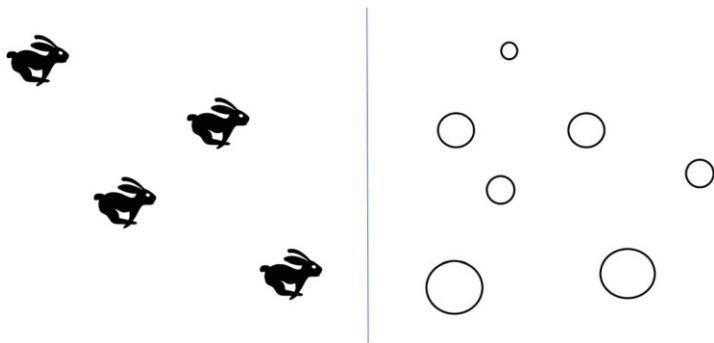
| Method             | Responsibility                                                                            |
|--------------------|-------------------------------------------------------------------------------------------|
| <i>Init</i>        | To initialize objects and do other preparations as needed                                 |
| <i>updateStory</i> | To update objects' locations and appearances based on an animation design                 |
| <i>paintStory</i>  | To draw each of the objects on a “canvas”                                                 |
| <i>storyEnds</i>   | To end the animation when a defined goal is achieved and clean up the resources as needed |

- The canvas panel *AnimationPanel* is implemented as a local class for consistent modularity (though it can also be a regular inner class). The method *getAnimationPanel* can be abstract too if we want to customize the appearance of the canvas panel.
- The implementation of the canvas panel is sustainable due to the abstract method *paintStory*, which is invoked to update objects' locations and appearances before method *repaint* is invoked.
- The abstract method *initiate* is responsible for creating an instance of *AnimationStory*, creating objects (which can be delayed), and starting the animation thread.
- Java interface *Runnable* is used to abstract away an animation task through its only method *run* to be independently threaded. It is possible to implement this *run* method sustainably across different animations with the help of the abstract method *updateStory*. However, thread execution may need to be controlled in some way in applications. For example, we might want to pause a thread execution and resume it later (e.g., with a key press) to capture a moment of the animation. Thus, the method *getTask* is best left as an abstract method. *Runnable* is a “requires” interface:

```
abstract class AnimationFrame extends JFrame{
 protected Thread animator;
 protected JPanel pnl;
 protected AnimationStory story;
 public AnimationFrame(String title){
 super(title);
 this.setSize(600, 600);
 this.setLocation(300, 100);
 pnl = getAnimationPanel();
 this.add(pnl);
 this.addWindowListener(new WindowAdapter(){
 public void windowClosing(WindowEvent e){
 animator.interrupt();
 e.getWindow().dispose();
 System.exit(0);
 }
 });
 animator = new Thread(getTask());
 initiate();
 }
 public void repaint(){
 super.repaint();
 pnl.repaint();
 }
 public JPanel getAnimationPanel() {
 class AnimationPanel extends JPanel{
 public AnimationPanel(){
 this.setBackground(Color.white);
 }
 public void paintComponent(Graphics g){
 super.paintComponent(g);
 story.paintStory(g);
 }
 }
 return new AnimationPanel();
 }
 abstract void initiate();
 abstract Runnable getTask();
}
```

### 10.4.2 Two Examples of Using the Framework

Figure 10.2 shows static images of two animation stories. The racing story features racers “running” by picking one of the three rules for position change calculations; some may make the racers go backward. The other story is a simulation of raindrops on a glass area such as a car windshield. A raindrop is created at a random location; it then ripples gradually until it reaches certain size when a new drop is created at another random location. The process of using the framework is straightforward. The primary work is an implementation of the interface *AnimationStory*. The following code for raindrop simulation demonstrates this implementation:



**Fig. 10.2** Two animation stories

```
public class RaindropStory implements AnimationStory{
 private Raindrop[] drops;
 private Dimension d;
 long startTime;
 long elapsedTime;

 public RaindropStory(Dimension d){
 this.d = d;
 startTime = System.currentTimeMillis();
 elapsedTime = 0;
 }
 @Override
 public void init() {
 drops = new Raindrop[10];
 for(int i = 0; i < drops.length; i++){
 drops[i] = new Raindrop()
 drops[i].setPosition((int) (Math.random() * d.getWidth()), (int) (Math.random() *
d.getHeight()));
 }
 }
 @Override
 public void updateStory() {
 for(int i = 0; i < drops.length; i++){
 Raindrop drop = drops[i];
 drop.ripple();
 if(!drop.isVisible()){
 drop.setPosition((int) (Math.random() * d.getWidth()), (int)
(Math.random() * d.getHeight()));
 }
 }
 elapsedTime = (new Date()).getTime() - startTime;
 }
 @Override
 public void paintStory(Graphics g) {
 Graphics2D g2 = (Graphics2D) g;
 for(int i = 0; i < drops.length; i++){ g2.draw(drops[i].getDrop()); }
 }
 @Override
 public boolean storyEnds() {
 if(elapsedTime < 60*100){ return false; }
 return true;
 }
}
public class Raindrop {

 private int x, y, currentSize, visibleSize;
 private final int MAX_RIPPLE = 50;
 private final int RIPPLE_STEP = 2;
 public Raindrop(){ currentSize = 0; visibleSize = 1; x = y = 1; }
 public void setPosition(int xPos, int yPos){
 x = xPos; y = yPos;
 visibleSize = Math.abs((r.nextInt() % MAX_RIPPLE)) + 1;
 currentSize = 1;
 }
 public void ripple(){
 x -= RIPPLE_STEP / 2;
 y -= RIPPLE_STEP / 2;
 currentSize += RIPPLE_STEP;
 }
 public boolean isVisible(){ return currentSize < visibleSize; }
 public Shape getDrop(){
 return new Ellipse2D.Double(x, y, currentSize, currentSize);
 }
}
```

This case study and the last have shown the power of abstract classes in developing software frameworks. Framework development is among the most difficult software design tasks. These two relatively simple case studies do not necessarily show the kinds of complexity we might encounter in framework design for larger systems. Nevertheless, the essential design ideas behind frameworks are the same regardless of the complexity of a framework—analyzing the software requirements to understand what's changeable and what's stable of the potential software elements, especially the control modules.

---

## 10.5 An Analyzer for Frequencies of Words in a Text

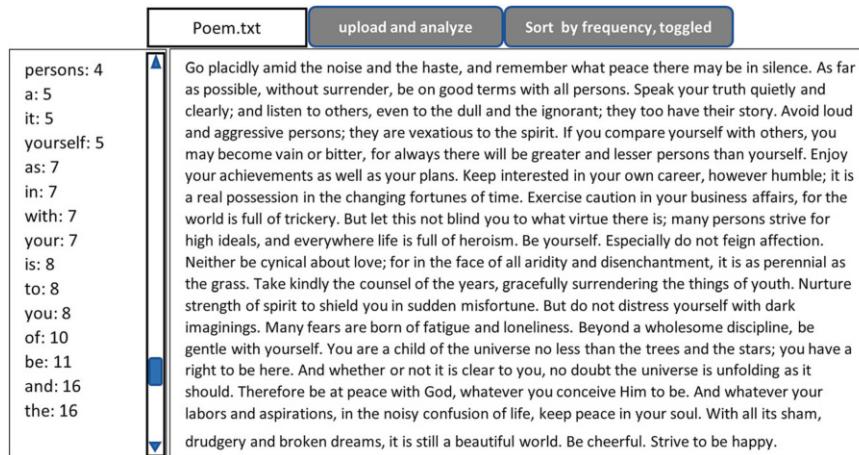
Document processing is common in software systems with various editing features and support for search and sorting capabilities. This case study is to design and implement a program that analyzes the frequencies of the words in a text file. Although it has only a narrow focus, the program can be extended in multiple ways.

### 10.5.1 Design Story and Use Cases

The program provides services for analyzing the frequencies of the words in a text file. The primary graphic user interface consists of two side-by-side panels: one displays the text, and the other shows the frequencies of the words in the text. There is a control panel (on the top or at the bottom). The control panel has one textbox for input of a file name and two buttons. One button, when clicked, triggers reading the file and displaying the context in the right panel and the analysis of the word frequencies in the left while the other button for sorting the frequencies with an effect of toggling the order of sorting when clicked. The textbox and panels are empty at initiation of the graphic user interface. Figure 10.3 is an illustration of the user interface after both buttons were activated.

The primary use case has the precondition that the application has been initialized with the following scenario:

1. User enters a file name into the textbox and clicks a button to upload the content of the file and compute the frequencies of the words in the text.
2. The application uploads the file content into a text panel and the result of a frequency analysis into the other panel (frequency data is listed without a particular order).
3. User clicks the other button to sort the frequencies.
4. The application redisplays the frequency analysis data (replacing existing content), sorted by frequency.
5. User clicks the button to sort the frequencies again.



**Fig. 10.3** Graphic user interface of the word frequency analyzer application

6. The application redisplays the analysis result (replacing existing content), sorted by frequency in reverse order.

*Exception flow:*

- 2a. The file name entered is incorrect.
  - 1) The application throws an exception.
  - 2) User corrects the error and clicks upload button again. Return to step 2 of the main.

## 10.5.2 Analysis

Data is loaded from a text file upon a button activation and stored in an in-memory data structure. This data structure also provides data store-related services. Thus, this data structure can be considered as the “model” of the application considering the MVC style. The user interface frame aggregates the data structure and an inner class for even handling, which is, therefore, the controller. The use of an inner class makes data access easier. We used a view interface to abstract away the details of view operations to make the application easily adapted to a console runtime environment. Handling of input data is separated from the model and view to make the application flexible for an alternative means of data handling. The following CRC cards reflect the analysis above:

| (Model) Word Frequency Data Service                                                                                                                               |                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <b>Responsibilities:</b><br><i>Update data when a new word is added</i><br><i>Search a word for frequency count</i><br><i>Sort words based on frequency, etc.</i> | <b>Collaborators:</b><br><i>List, WordFreq (a composite type to group a word and its frequency), data handler</i> |

| (View) Graphic User Interface of the Application                                                         |                                                                                  |
|----------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------|
| <b>Responsibilities:</b><br><i>Initialize frame</i><br><i>Set layout</i><br><i>Add visual components</i> | <b>Collaborators:</b><br><i>Java Swing components</i><br><i>Button listeners</i> |

| (Controller) Button Listener<br>(implementing interface <i>ActionListener</i> )  |                                                                                             |
|----------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| <b>Responsibilities:</b><br><i>abstract methods implementing event responses</i> | <b>Collaborators:</b><br><i>Word frequency data service</i><br><i>Java Swing components</i> |

| Data Handler                                                                 |                                                                                                        |
|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <b>Responsibilities:</b><br><i>Load data</i><br><i>Save frequencies data</i> | <b>Collaborators:</b><br><i>Library input and output objects</i><br><i>word frequency data service</i> |

The data service object type aggregates a library data structure for in-memory storage. Though, technically, an array may suffice, the use of a data structure would facilitate the implementation of the services. A *List* can be used to store objects of type *WordFreq*, but a map is equally convenient, with word as key and its frequency as value.

### 10.5.3 More Details About Implementation

The initial analysis led to the following three abstractions. The abstraction *ITextHandler* is a collaborator of the model (i.e., the data service) to deliver data from a storage medium. A user interface frame is an instance of *IView* but also a context object for event handlers (which play the role of a controller):

```
interface ITextHandler{
 String getText(String connStr);
}

interface IView{
 void displayText();
 void displayFrequencies();
 void initView();
 void setModel(WordFrequencyModel m);
}

interface IWordFrequencyModel{
 void fillMap(String connStr);
 Map<String, Integer> sortWordFrequencies(Comparator<Map.Entry<String, Integer>>
 comp);
 String getText();
 int queryFrequency(String word);
}
```

The implementation was mostly straightforward with the following elaboration on a few details:

- The model's method *fillMap* uses a string parameter, which could be a file name of a local text file, a database connection string, or the URL of a data source. The model doesn't (and should not) handle data access directly. Instead, it delegates the task to an instance of *ITextHandler*, which uses string pattern matching with regular expression "*\r\n{Punct}\r\ns+*" to split the returned string into words, count, and save the words and their frequencies into a map.
- To display the text in a text area, we can set an attribute of the component so that lines are wrapped appropriately and automatically. However, for console display, manually breaking the text into a desired number of lines would be necessary, though it's easy to do.
- The *IView* method *initView* is where appropriate view is initialized (and called in a constructor of an implementing class). In the case of a graphic user interface, the builder pattern could be used in the construction of a frame.
- The following code launches the application. *ViewFactory* dispatches a view instance (graphic or a console dialog user interface), which is appropriately initialized by calling method *initView*, as mentioned above:

```
IView viewInstance = ViewFactory.getViewInstance("gui");
WordFrequencyModel modelInstance = new WordFrequencyModel();
viewInstance.setModel(modelInstance);
```

If another view is desired, we write another implementing class of *IView* and add another conditional branch in *ViewFactory* for creating an instance, which would be the only modification to the code. An abstract factory can be used if dynamic switches between views are desired.

#### 10.5.4 Details of the Application Control

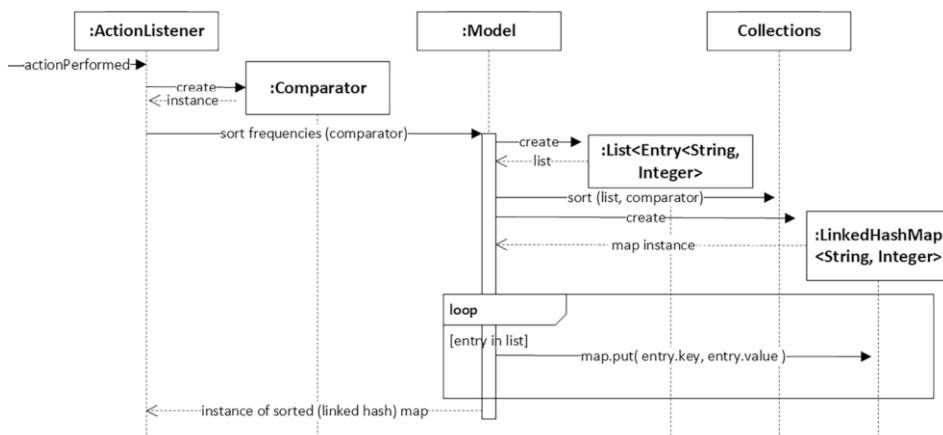
The two button event handlers control the operation of the application. These processes are depicted in two sequence diagrams. Figure 10.4 is a sequence of object communications for file upload and frequency analysis. After reading of a file name, objects send each other messages for the following operations in that sequence:

- Read the text from the file and compute the frequencies of the words.
- Display the text in one text area and frequencies of the words in the other.

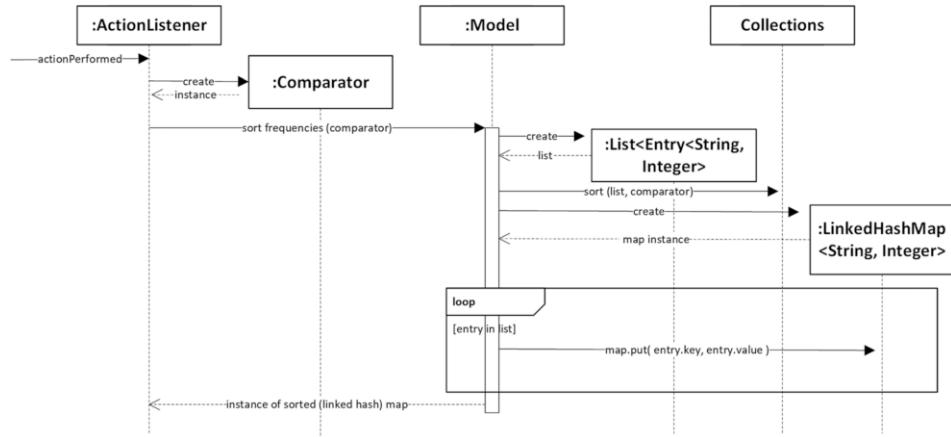
Entries of Java's *TreeMap* are typically sorted with respect to the keys in an order specified with a *Comparator* instance if desired. However, we need to sort the entries by ordering the values (i.e., words' frequencies). We can do so in two steps (though there are other approaches):

- Extract map's entry set, make it a list, and then use a library sorting routine *Collections.sort* (for lists) with two *Comparator* instances for ascending or descending order. A Boolean variable can be convenient for toggling between the two sorting orders.
- Use an instance of *LinkedHashMap* to reconstruct the map with the sorted entries. This map structure maintains the same order as the keys when entries are added.

Figure 10.5 is a sequence diagram depicting this process, though the process of toggling the sorting order is not represented. To display the map content in a more readable format, we may include another abstract method in the model abstraction to return a formatted string of key value pairs.



**Fig. 10.4** Sequence diagram of actionPerformed for file upload and frequency analysis



**Fig. 10.5** Sequence diagram of actionPerformed for toggled sorting

Finally, the application can be extended in multiple ways:

- More text analyses could be added such as grouping words based on their frequencies
- More features can be added like searching a word for its frequency, counting the number of words in the text, saving a frequency analysis to a text file, etc.
- Provide manual entry of a text.
- Add a *File* menu and use pop-up frames to collect user inputs and display results.

## 10.6 Getting People to Sleep

This case study was adopted from a research article in computer science education, which used the design performances by the students who participated in the study to argue that graduating students in computer science cannot design software (Eckerdal et al., 2006). The same design case was reused in a subsequent study at a different college, and the authors made the same conclusion (Loftus et al., 2011). As a result, the authors of both articles made recommendations about how design education should be strengthened in their respective institutions. The articles, however, didn't discuss about possible solutions. The software scenario is probably best suited for a mobile application. The design presented in this section was for a desktop prototype with artificial passage of time so that features can be tested without constraints due to using a real-time system clock.

### 10.6.1 Application Requirements

This application was nicknamed “super alarm clock,” intended to help college students manage their own sleep patterns. Sleep deprivation among college students can be a serious health hazard, and the cumulative long-term effects of sleep loss have been associated with a wide range of health issues. Potentially, such software applications can also provide data to support research inquiries into the extent of the problem in this community.

According to the articles, the application shall:

- Allow users to set an alarm to wake themselves up or remind themselves to go to sleep.
- Record the time when informed of the user going to sleep or having woken up (whether it was because of an alarm).
- Remind users when they need to go to sleep based on the alarms set by the user. This shall include “yellow alerts” when they will need sleep soon and “red alerts” when they need to sleep immediately.
- Store collected data in a database for later retrieval and analysis.
- Create reports about users who are becoming dangerously sleep-deprived to inform those who care (such as the parents). A user with three “red alerts” in a row shall trigger such a report.
- Provide reports to the user showing the sleep patterns over time, allowing review of the ignored alarms and clusters of unhelpful and unbeneficial sleep behavior.

In the following, we provide only architectural discussions using the 4+1 Architectural View Model. However, code fragments are also included when helpful.

### 10.6.2 Use Cases

The following use case provides a context for most of the requirements. It assumes that the system is activated and the application’s internal clock is ticking.

*Main scenario* (in a 24-h period):

1. User, at some point in time, informs the system that she/he woke up or is going to sleep.
2. System saves the time. Use case ends.

*Extension scenario 1:*

- 1a. When system detects there are 30 min left before the set time for user to sleep and is not informed of user going to sleep:
  - 1) System invokes a “yellow alert.”
  - 2) User informs the system of going to sleep.
  - 3) System saves the time. Use case ends.

*Extension Scenario 2:*

- 1b. When system detects there are 15 min left before the set time for user to sleep (after it invoked a “yellow alert”) and is not informed of user going to sleep:
- 1) System invokes another “yellow alert.”
  - 2) User informs the system of going to sleep.
  - 3) System saves the time. Use case ends.

*Extension Scenario 3:*

- 1c. When system detects it is 10 min past the set time for user to sleep and is not informed of user going to sleep:
- 1) System invokes a “red alarm” and saves the time and an indicator of a red alert. Use case ends.

This use case helps clarify the timeline for invoking yellow or red alerts (but such requirement clarifications would be typically provided by a client).

### 10.6.3 Logic View of the Design

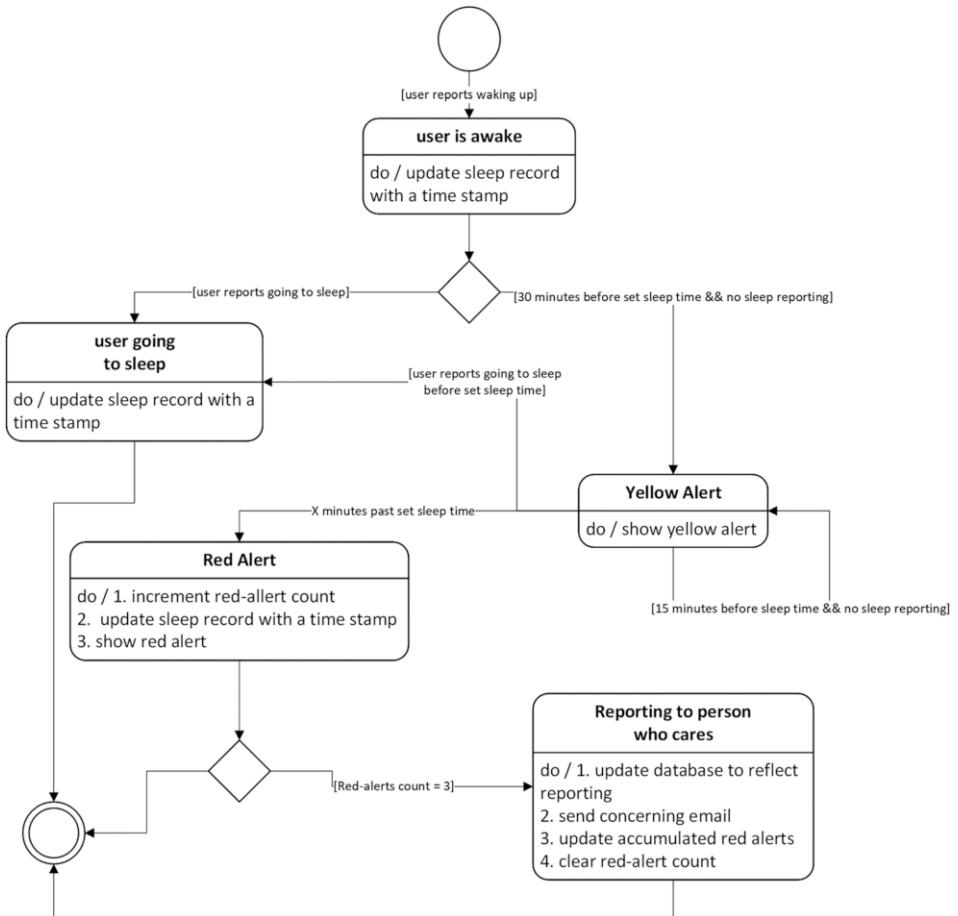
The application is event-rich as alerts, alarms, and user interaction with the application can all be interpreted as events. The application is essentially a sequence of system responses to those events. These events may define state transitions, which means a state-machine diagram can be appropriate for modeling the dynamics of the system functions. Figure 10.6 is a state diagram based on the use case and provides an aspect of the logic view of the 4+1 View Model.

When we think about the details of a state transition and system actions when a state endures, some ambiguities of the requirements became apparent. For example:

- How many “yellow alerts” shall be raised before a “red alert”?
- If multiple “yellow alerts” were possible, how soon shall it be to issue the next “yellow alert”?
- Shall “yellow alerts” be saved? If so, in what form?

These questions can be easily answered with the presence of a client. For the prototype, we made assumptions and integrated them into the use case presented earlier.

Languages may offer real-time clock operations, such as `javax.realtime.Clock` in Java, to support applications that need to simulate passing of time. Not only can real time be queried (like `Clock.getTime()`), but events can also be queued on the clock to be fired when an appointed time is reached. Thus, a possible design of the application would be to use

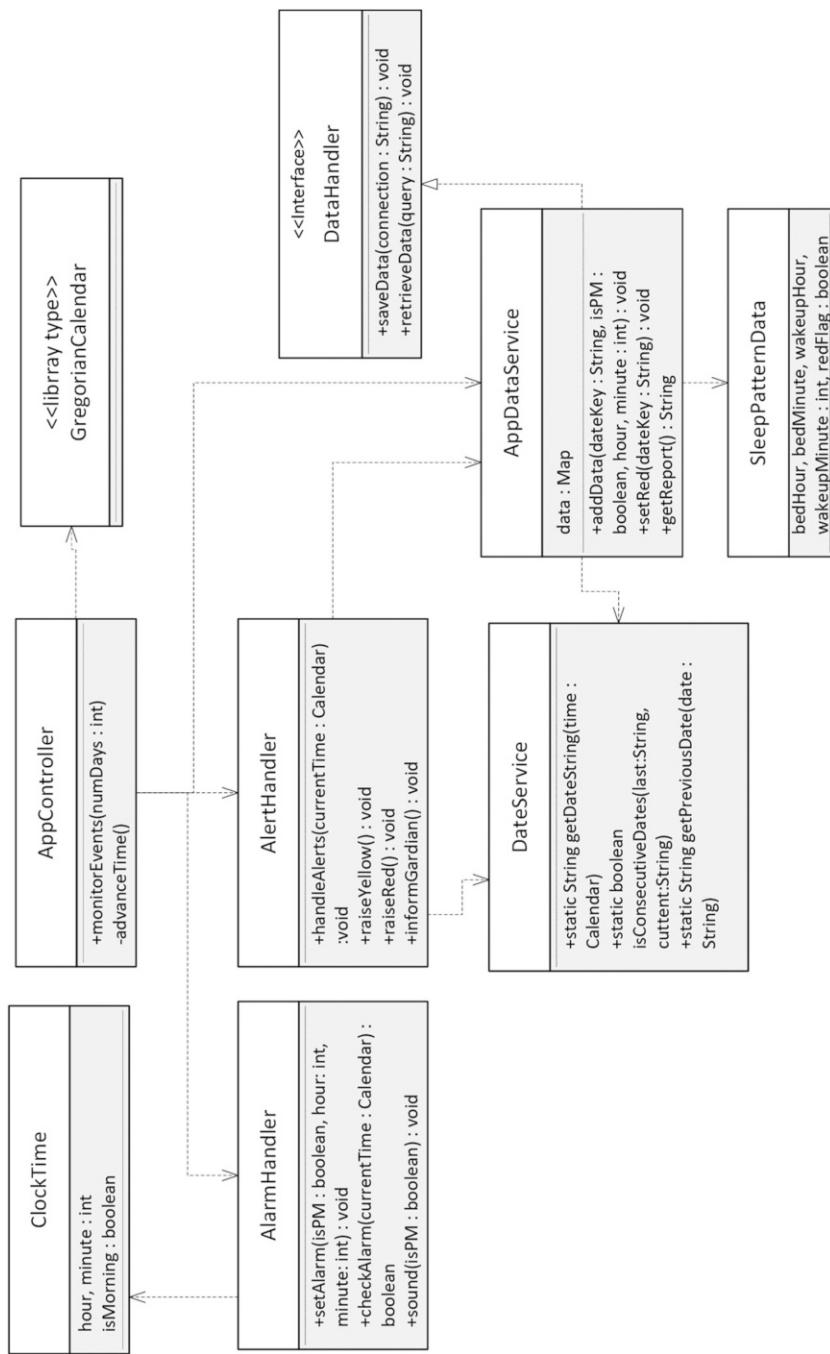


**Fig. 10.6** State diagram for Super-clock software application

such a clock function. However, we chose to implement the prototype using a controller in a more “regular” way to monitor events, raise the events when they occur, and process the events by delegating to event handlers. The relations of the controller with data and event handlers are summarized in Fig. 10.7.

Here are the design ideas behind a few methods in Fig. 10.7:

- To run the program as a console application that allows completion of an execution momentarily while still going through all the events, we need to be able to advance the time (in minutes) manually. It appears that a Java class *GregorianCalendar* offers a variety of methods for time and date manipulations that we can take advantage of.
- An in-memory data storage of map type was used with date strings as keys and instances of *SleepPatternData* as values. The use of a map instance is convenient since there is a



**Fig. 10.7** Class diagram of the control structure of the application

one-to-one correspondence between a map entry and a database record. *SleepPatternData* was designed to facilitate the data-handling process.

- An object of *AppController* is a monitor that executes a perpetuate time loop to control event firing and event handling through collaborators.

A less direct but potentially more flexible approach is to use the Observer pattern with observers of three different kinds to handle alerts, alarms, or reports. The approach might be structurally more complex, but it makes extensions of the application easier. For example, adding an effect of taking a nap may affect the timelines of issuing alerts. Changes are only made in an alert observer object independent of other observers. Figure 10.8 is a possible set of classes and their relations with this approach, in addition to all the object types, except the control class, in Fig. 10.7.

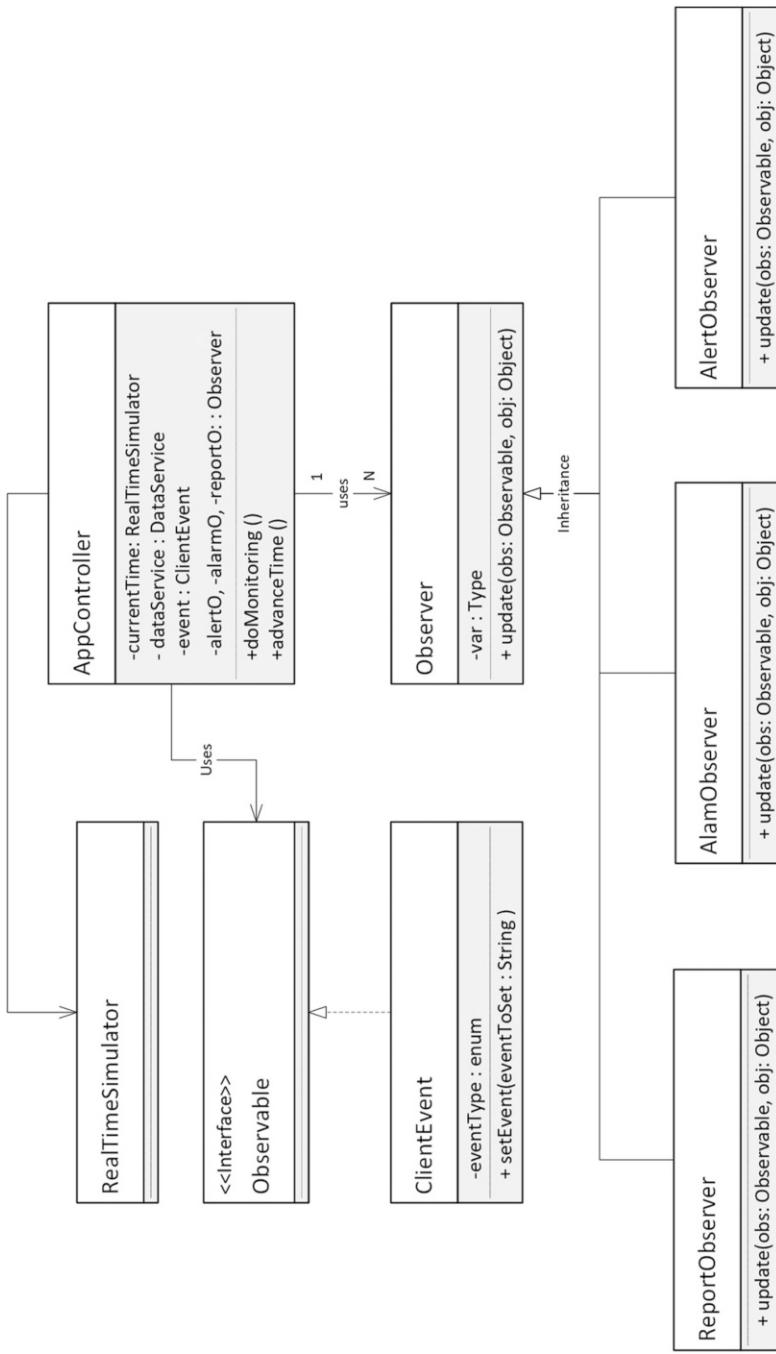
#### 10.6.4 Process View

The process view describes how the application works when it runs as a process, while multiple processes may exist in the software's operational environment. There are no other processes that the primary application interacts with in this case. However, if we view software tracking of real time as a separate process, it is helpful and architecturally significant to view how the primary software operations are executed around this time-tracking process or how the control process would raise events as time advances with a desired increment. The following code demonstrates how we can advance the time "manually" by adding a minute to the current time in each loop iteration:

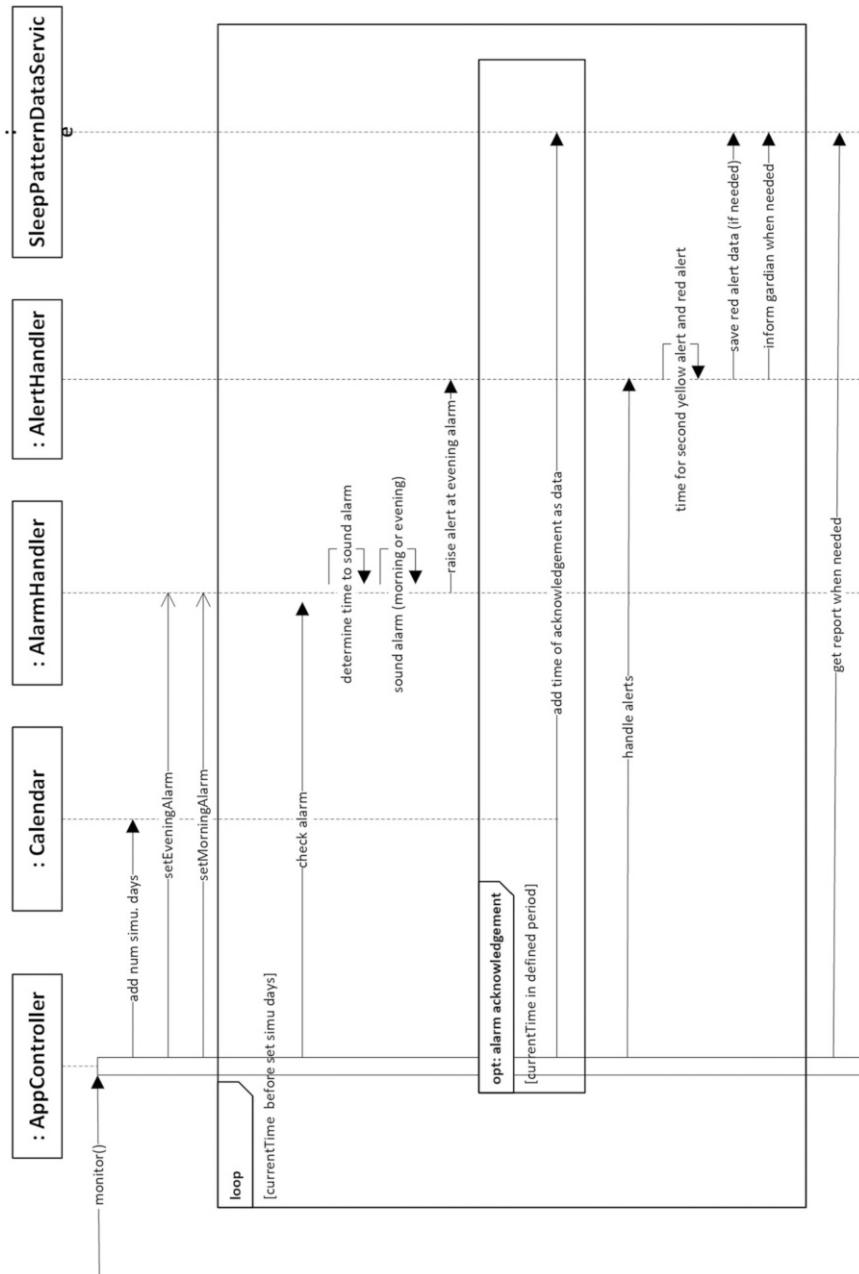
```
Calendar currentTime = new GregorianCalendar(), daysLater = new
GregorianCalendar();
daysLater.add(GregorianCalendar.DATE, numDays);
while (currentTime.before(daysLater)){
 ...
 currentTime.add(GregorianCalendar.MINUTE, 1);
}
```

This simulation of time passage can be useful in many other application scenarios, where time advances must be controlled in a simulation.

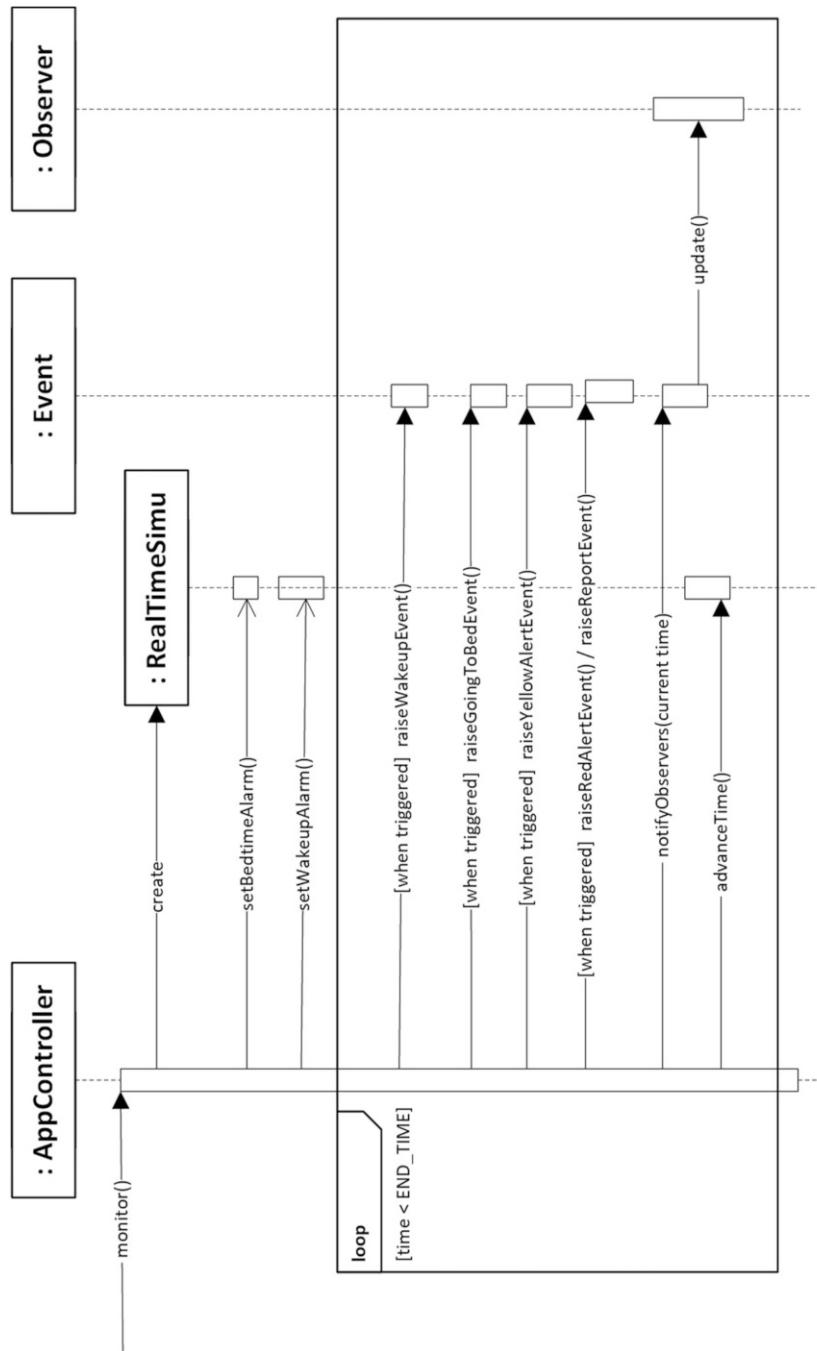
Figure 10.9 is a sequence diagram based on the first design approach with a centralized control. Open arrows represent asynchronous operations—those that can be executed at different time and circumstances. To keep the diagram focused on events, the time control mechanism was not drawn. Figures 10.6 and 10.9 are about the same software process but complementary to each other with different view angles. Figure 10.10 describes the object dynamics in the second design approach using the observer pattern.



**Fig. 10.8** Class diagram of the approach using the observer pattern



**Fig. 10.9** A view of sequence of events in a time loop



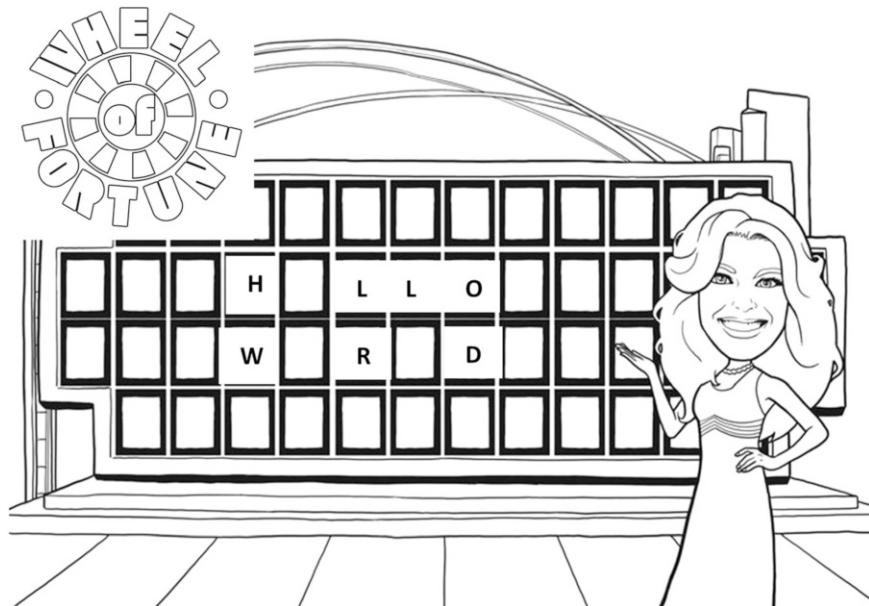
**Fig. 10.10** Dynamic behavior of “super clock” operations based on the Observer pattern

Since the design is about a prototype of the application, development and physical views are unimportant.

---

## 10.7 Wheel of Fortune Game

This case study is about the design and implementation of the television game show *Wheel of Fortune*. The show started in 1975 and is one of the longest-running syndicated game shows in American television. As well known, contestants take turn to play. To play, a player spins the wheel, which would stop at a wedge with a dollar amount for prizes if the player guessed some of the words in the puzzle correctly. A few wedges also indicate some unfortunate events, which could wipe out the player's accumulated prizes. A game completes when the puzzle is revealed by a contestant. Figure 10.11 is only a symbolic illustration of the game. Identifiable domain entities include contestant, game host, the wheel, the puzzle board, and a puzzle. How these entities interact is an interesting design question. To focus on object design, not GUI, we will use console input and output, though adding a graphic interface can be an exercise. As a demonstration, we will use the C4 View Model for construction of architectural artifacts.



**Fig. 10.11** Wheel of Fortune game show

### 10.7.1 The Requirements

Since the game rules are well understood by the public, the requirements can be briefly summarized as follows:

- The host of the game initializes the game with a hidden sentence and an empty textboard visible to contestants. The textboard reveals the characters in the hidden sentence as contestants take turns to play and make guesses for consonants or the whole puzzle. Letter cases play no role in the game, however.
- The current player turns the wheel with wedges labeled with dollar amounts (i.e., the prizes a contestant would win) and with some unfortunate cases such “bankrupt” (forfeiting the total prizes the player has earned) or “lose a turn.” (There are typically two “bankrupt” wedges and one “lose a game” wedge.)
- When the wheel stops turning, the current player guesses a (hidden) character of the puzzle. If the character player guessed matches a character in the unseen sentence, all instances of the character will be displayed on the textboard at the matched position(s) in the sentence, and the player earns a prize that is the multiplication of the dollar amount on the wheel wedge and the number of matched instances. The contestant continues to make another play and accumulates the prizes until the player fails to guess correctly. Then, the next player (in a defined order) starts to play.
- A player may guess an entire sentence and, if correctly, earns prizes on all characters that were still hidden at the time. The game is finished when the entire sentence is revealed. The player with the highest total prize wins the game.
- In the TV show, the player does not spin the wheel for guessing vowels. Instead, a player may wish to and can “buy a vowel.” A set amount, say \$250, is deducted from player’s accumulated total (thus the total must be more than the cost). If the vowel is in the puzzle, all instances are displayed with no money amount awarded. Regardless of consonant or vowel, if the letter the player guessed is in the puzzle, the player’s turn continues; otherwise, the player’s turn ends.

To focus on object design, we simplified a few things. It was assumed that a player always earns a score if the player guessed correctly on a vowel or a consonant. Multiple contestants with possibility of losing a turn or going bankrupt were also not considered. A puzzle was hard coded as opposed to pulling from a data source such as a text file. Puzzles could be categorized in a data source, and the category of a puzzle could be displayed before the game starts (which is the case in the TV show). These omissions can be worthy but simple exercises. Nonetheless, they are less intrigued given the focus of the case study.

### 10.7.2 A Use Case

Even with well-understood game rules, a use case can still be useful to understand the responsibilities of the abstractions to be designed and their relations. The use case assumes that all objects have been instantiated and appropriately initialized.

#### *Normal Scenario*

1. Player turns the wheel.
2. System displays a positive integer number (dollar amount) corresponding to a wedge pointed when the wheel stopped.
3. Player inputs a character.
4. System searches for and finds matches of the character in the hidden puzzle string.
5. System displays the matched characters in the puzzle, computes the score (i.e., prize), and updates the player's total score.
6. Player continues to play (back to step 1). Use case ends.

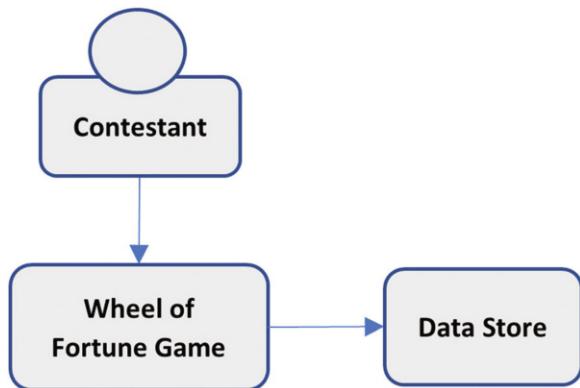
#### *Alternate Scenarios*

- 2a. The wheel spin results in “bankrupt”:
  - 1) System displays “bankrupt.”
  - 2) System sets total score of the player to zero and assigns the turn to the next player (and normal scenario follows).
- 3a. Player guesses the entire hidden sentence:
  - 1) Player inputs a sentence.
  - 2) System verifies if it is the correct sentence and updates the total score (or does nothing if system finds that the sentence is incorrect). Use case ends.
- 4a. System finds no matches:
  - 1) System assigns the turn to the next player (and normal scenario follows).
- 5a. System finds the puzzle has been fully displayed:
  - 1) System declares the winner. Use case ends.

### 10.7.3 First Three Views in the C4 View Model

The context view of the C4 View Model is about a contextual view of the software operations interacting with other systems that may be present in the operational environment. For this simulation, there are two possible operational environments. One is a local game environment where multiple players are sitting at the same computer. The other scenario is a networked game environment where players are represented by their screen characters sharing the same game instance with their own networked devices, such as a mobile phone, to play. Distributed multiplayer games generally require responsive

**Fig. 10.12** Context view of Wheel of Fortune architecture



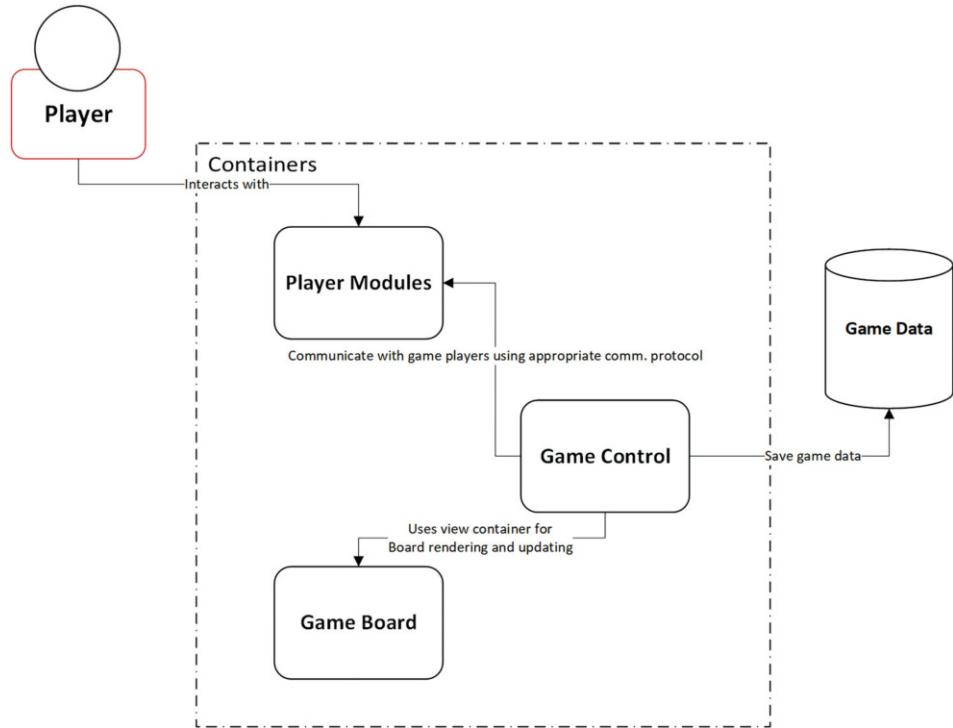
networks and a good middleware support for scalability and error-free updates to a shared game instance. Given the focus of the case study, a local game environment is assumed, which leads to a trivial game context, involving only the game application and a data store, as illustrated in Fig. 10.12.

A container view of the C4 Model categorizes system modules and software resources. Often, an architectural style may define associated containers. For example, if the MVC were used, we would likely see containers like data container, model container, and view container, among others. A service-oriented architecture may define different service containers, tool containers, etc. Software resources could also be grouped into containers, as could modules that communicate with external systems or devices. Networking containers may also be expected if games are played in networked game environments.

Figure 10.13 depicts the containers in the game architecture based on the roles the elements play. “Game data” process is not generally part of game application. It can be an existing system that can be used to save play data if one wants to use the data to make game-play stories or study about effective gaming strategies. This external system might also be able to provide a platform for creating puzzles, categorizing them, and managing a puzzle repository. It can be an interesting exercise on its own.

Here are a few considerations behind the design of the containers:

- In terms of the MVC style, a controller is often more clearly separable than the other two entities. The gameboard container is part of the view aspect of the game. A view may also include interfaces with user and input. Console input and output in this case are simple processes and can be included in game control as local methods for easy implementation. The design would be very different if we wanted an easy switch between a graphic user interface and a console interface. Since the game is about how a player would spin the wheel and make guesses, the player container may be thought of as “game model.”
- Gameboard is where puzzles are displayed (partially or fully). We might wonder if it would be plausible for gameboard to assume verification responsibility of player’s

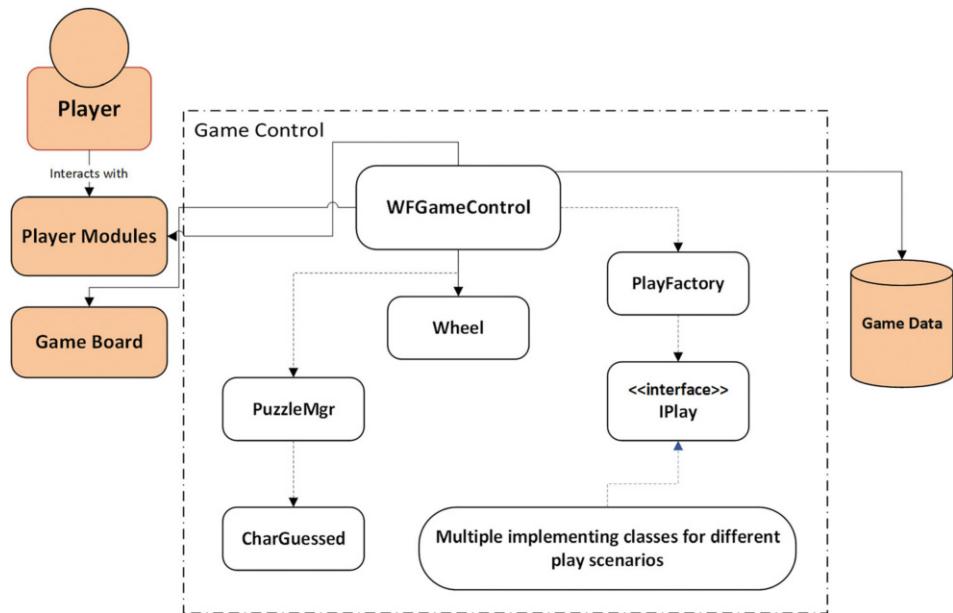


**Fig. 10.13** Container view of Wheel of Fortune architecture

guesses. But as elaborated later in the component view, a mental simulation about the feasibility would quickly conclude that we would better leave gameboard as a pure display entity, especially when a graphic user interface is used.

- The player container includes modules that are commensurate with a contestant's responsibilities such as spinning the (simulated) wheel and making guesses. However, a simulated player is a fabricated entity with responsibilities not necessarily consistent with those of its real-world counterpart. For example, "turning the wheel" is about picking a random number that game control can also do "for a player" with the same outcome. When options are equally reasonable, a design choice might simply be an approach that has a better conceptual alignment. Thus, we made wheel spin a responsibility of a contestant with an operation below (where wheel is a singleton object):

```
public String turnsWheel(){
 Random rand = new Random();
 Wheel w = Wheel.getInstance();
 return w.stopAt(rand.nextInt(w.length()));
}
```



**Fig. 10.14** Component view of Wheel of Fortune architecture

- The game control container is the richest and offers most design opportunities. The controller is responsible for verifying player's guesses, managing players' turns, determining the current portion of the puzzle to be displayed, and perhaps turning the wheel (for players as discussed above). This container also includes helper modules, related interfaces, and object factories.

Figure 10.14 is a component view of the “richest” container “*Game Control*.“

As mentioned earlier, gameboard may assume the responsibility of verifying a contestant’s guesses. Here is an analysis about one of the possibilities (assuming the puzzle is stored in an array):

1. The player makes a guess.
2. The control gets a list of indices of the puzzle array where the guessed character matches instances in the puzzle.
3. If this list is empty (i.e., no matches were found), the next player gets to play; otherwise, the control asks gameboard to confirm whether the guessed character is unrevealed (as the gameboard maintains a partially displayed puzzle).
4. The gameboard confirms whether the newly guessed character is already in the partially displayed puzzle and updates, as appropriate, the displayed puzzle (by adding the instances of the correctly guessed character).

5. As appropriate, the control updates the player's total score, and the current player continues to play.

This appears to be an awkward communication between the controller and gameboard with logic redundancy (checking for matched indices and then with displayed portion of the puzzle) and similar puzzle data in both the controller and gameboard. Besides, a nested loop would be needed with outer loop managing players' turns and inner loop managing plays by the current player.

One of the elements in Fig. 10.14 is a simple object type *CharGuessed* that pairs a character with a Boolean value so we can “turn a character off or on”:

```
class CharGuessed{
 char ch;
 boolean guessed;
 public CharGuessed(char c){
 ch = c;
 guessed = false;
 }
 public int isMatch(char c){
 if(ch == c && !guessed){
 guessed = true;
 return 1;
 }
 else return 0;
 }
}
```

The abstraction *PuzzleMgr* maintains a list of *CharGuessed* instances and offers services methods needed for executing the game logic. Both classes are design classes. As a result, checking for matches is performed only once in a single loop with cleaner and more readable code (see code below).

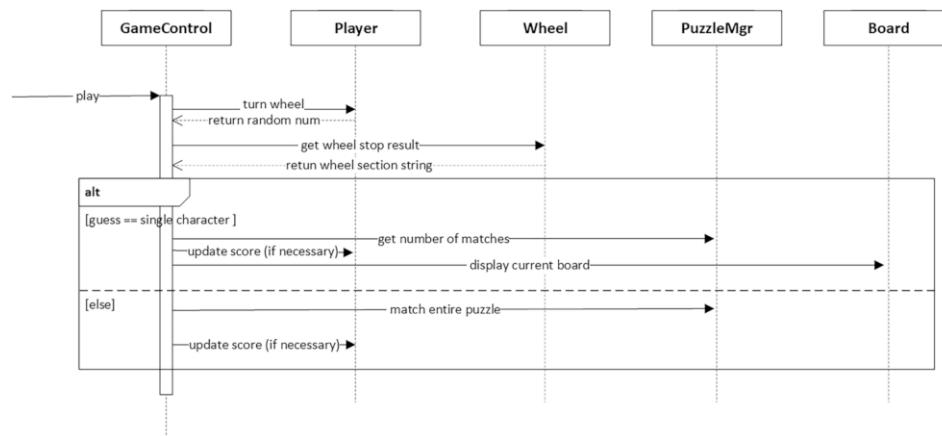
#### 10.7.4 The Code View

The code view of the C4 Model is only loosely defined to include any diagrams that are deemed helpful. In particular, class diagrams are appropriate in this view to show more details of the components. Table 10.3 summarizes the responsibilities and collaborators of the primary elements (equivalent to a class diagram).

Figure 10.15 is a sequence diagram that describes the game logic in terms of interactions between the game control and other elements. It is equivalent to a contestant playing once. The primary method of the game control is short and presented below in its entirety. The Strategy pattern was used to implement the alternate scenarios of the use case, and a play “strategy” is implemented as an instance of *IPlay* and created by a play factory:

**Table 10.3** Primary abstractions, their responsibilities, and collaborators

| Type                | Responsibility                                                                                                                                                                                                                                                                                                                | Collaborator                                                                                                        |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| <i>Player</i>       | <ul style="list-style-type: none"> <li>• “Turns” the wheel (as discussed earlier)</li> <li>• Makes guesses</li> <li>• Sets score total</li> </ul>                                                                                                                                                                             | <i>None</i>                                                                                                         |
| <i>Gameboard</i>    | <ul style="list-style-type: none"> <li>• Displays a partially solved puzzle (with the help of <i>toString</i> method)</li> </ul>                                                                                                                                                                                              | <i>None</i>                                                                                                         |
| <i>Wheel</i>        | <ul style="list-style-type: none"> <li>• Initiates the wheel</li> <li>• Gets label at a given position</li> <li>• Implements a string representation of the wheel (using <i>toString</i> method)</li> </ul>                                                                                                                   | <i>None</i>                                                                                                         |
| <i>PuzzleMgr</i>    | <ul style="list-style-type: none"> <li>• Gets number of matches</li> <li>• Checks for matches with characters in the puzzle</li> <li>• Determines number of unrevealed characters in the puzzle to control the completion of a game</li> <li>• Constructs partially revealed puzzle through <i>toString</i> method</li> </ul> | <i>List, CharGuessed</i>                                                                                            |
| <i>Game control</i> | <ul style="list-style-type: none"> <li>• Initialize a game</li> <li>• Executes the game logic</li> </ul>                                                                                                                                                                                                                      | <i>Player, wheel, board, PuzzleMgr, IPlay and its implementing classes, PlayFactory (to dispatch IPlay objects)</i> |

**Fig. 10.15** Sequence diagram of executing game logic

```

public void playGame(){
 int turn = 0;
 Player currentPlayer = players[0];
 do{
 System.out.println("player " + currentPlayer.name + " playing ..");
 String result = wheel.stopAT(currentPlayer.turnWheel());
 System.out.println("wheeled score: " + result);
 IPlay play = new PlayFactory().getPlay(result);
 boolean good = play.isGoodPlay(currentPlayer);
 System.out.println(currentPlayer.name + " total score: " +
 currentPlayer.getTotalScore());
 if(good){ continue; }
 turn = ++turn % players.length;
 currentPlayer = players[turn];
 }while(mgr.moreToGo());
}

```

Finally, the scenario of buying a vowel can be easily implemented with the following code (which could be included as the first task of the game loop):

```

String buyVowel = JOptionPane.showInputDialog(null, "Buy a vowel?");
if(buyVowel.charAt(0) == 'y'){
 if(currentPlayer.getTotalScore() > VOWEL_COST){
 currentPlayer.setTotalScore(currentPlayer.getTotalScore() - VOWEL_COST);
 char vowel = JOptionPane.showInputDialog(null, "enter a vowel..").charAt(0);
 int matches = mgr.numMatches(vowel);
 if(matches != 0){
 bd.setChars(mgr.toString());
 System.out.println(bd);
 }
 }
}

```

### 10.7.5 Console Output

Table 10.4 is the output of an execution of the application with puzzle “hello, world.” As discussed earlier, console output and gameboard are independently managed. This separation allows easy switch between a console and a graphic interface with appropriate abstractions to capture the variations.

**Table 10.4** The output of a game play with puzzle “hello, world”

|                                                                                                                                                                                                                                                                |                                                                                                                                               |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|
| <p><i>player p1 playing ..</i><br/> <i>wheeled score: 200</i></p> <hr/> <pre>                               </pre> <hr/>                                                                                                                                       | <p><i>p2's total score: 310</i><br/> <i>player p2 playing ..</i><br/> <i>wheeled score: lose a turn</i><br/> <i>p2's total score: 310</i></p> |
| <p><i>p1's total score: 400</i><br/> <i>player p1 playing ..</i><br/> <i>wheeled score: 600</i></p> <hr/> <pre>                               </pre> <hr/>                                                                                                     | <p><i>player p3 playing ..</i><br/> <i>wheeled score: 110</i><br/> <i>p3's total score: 0</i></p>                                             |
| <p><i>player p1 playing ..</i><br/> <i>wheeled score: 170</i></p> <hr/>                                                                                                                                                                                        | <p><i>player p1 playing ..</i><br/> <i>wheeled score: 2200</i><br/> <i>player p2 playing ..</i><br/> <i>wheeled score: 300</i></p> <hr/>      |
| <p><i>p1's total score: 2200</i><br/> <i>player p1 playing ..</i><br/> <i>wheeled score: 190</i><br/> <i>p1's total score: 2200</i><br/> <i>player p2 playing ..</i><br/> <i>wheeled score: 110</i></p> <hr/> <pre>                               </pre> <hr/> | <pre>                               </pre> <hr/> <p><i>h     1   1   o     w   o     1   d  </i></p> <hr/>                                    |
| <p><i>p2's total score: 110</i><br/> <i>player p2 playing ..</i><br/> <i>wheeled score: 200</i></p> <hr/> <pre>                               </pre> <hr/>                                                                                                     | <p><i>p2's total score: 610</i><br/> <i>player p2 playing ..</i><br/> <i>wheeled score: 300</i></p> <hr/>                                     |
| <p><i>player p2 playing ..</i><br/> <i>wheeled score: bankrupt</i></p> <hr/> <pre>                               </pre> <hr/>                                                                                                                                  | <pre>                               </pre> <hr/> <p><i>h     1   1   o     w   o   r   1   d  </i></p> <hr/>                                  |
| <p><i>p2's total score: 910</i><br/> <i>player p2 playing ..</i><br/> <i>wheeled score: 190</i><br/> <i>p2's total score: 910</i><br/> <i>player p3 playing ..</i><br/> <i>wheeled score: bankrupt</i></p> <hr/>                                               | <p><i>p3's total score: 0</i></p>                                                                                                             |
| <p><i>player p1 playing ..</i><br/> <i>wheeled score: 170</i></p> <hr/> <pre>                               </pre> <hr/>                                                                                                                                       | <p><i>player p1 playing ..</i><br/> <i>wheeled score: 2370</i></p>                                                                            |
| <p><i>game ends:</i></p>                                                                                                                                                                                                                                       | <p><i>p1 total score: 2370</i><br/> <i>p2 total score: 910</i><br/> <i>p3 total score: 0</i></p>                                              |

## Exercises

1. Design and implement a console application to manage a to-do list. A to-do list item has two parts: a date string and a description of a task. The application shall allow a user to:
  - (a) Enter a to-do item.
  - (b) Search for an item given a date.
  - (c) Remove an item given a date.

The application shall use a text file to save the items and retrieve the items for editing.

When the items are displayed, they shall be displayed in a chronological order. The application shall also allow easy switch (using a name string) between two different input modes: one with output stream *System.out* and the other with dialog boxes of *JOptionPane*.

2. Design and implement a console application for managing a vehicle inventory (though the design shall make it easy to switch to a graphic user interface). The application shall use a text file to save vehicle data. Here are other requirements:
  - (a) An auto product has a serial number and a purchase price. Other data attributes may include vehicle type (Sedan, SUV, truck, etc.), two-wheel or four-wheel drive, maker, model, color, miles per gallon, and year manufactured. Data related to a vehicle type:
    - Sedan: number of doors
    - SUV: number of passengers
    - Truck: load capacity (like 3 tons or 2 tons)
    - Van: size (like full-size van or minivan)
  - (b) The auto inventory needs to provide at least the following service operations:
    - To add vehicle data
    - To retrieve vehicle information given a serial number
    - To search for vehicles based on a given specification
  - (c) The search method shall allow user to input a specification of vehicle properties, and the program then displays all matching vehicles. The user may use any combination of properties. For example, a buyer may only care about the type of a vehicle, maker, model, and color but nothing else.

This is a relatively simple application. However, it is the data search operation that makes the application interesting. With large number of attributes of a vehicle, we need to design an effective filtering of vehicles based on user-specified criteria. A straightforward strategy is to compare vehicle data with a search criterion with a multibranch if-else statement, which can be difficult to read and maintain. Another approach is to use the Inversion of Control principle with the following abstraction:

```
interface VehicleSpec{ boolean matches(AutoSpec spec); }
```

The abstract method is designed to capture the variability of a vehicle search. To implement the abstract method, only the attributes that a user is interested need to be considered with a much more efficient if-else statement. For example, if a user only cares about the color of a vehicle, we may use the following object in search:

```
new VehicleSpec(){
 boolean matches(AutoSpec spec){
 return spec.color.equals("white");
 }
};
```

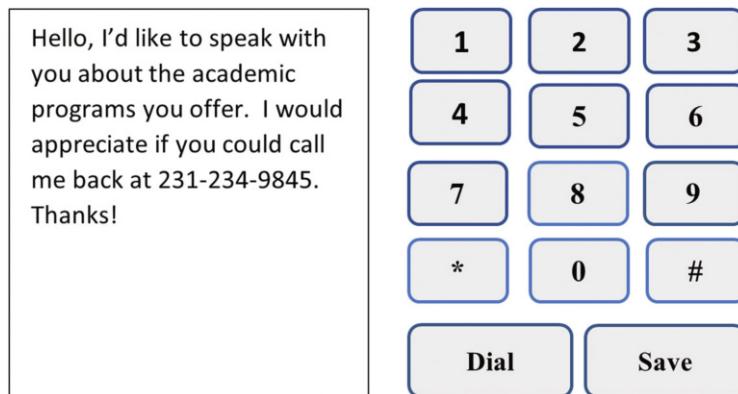
3. Design and implement an application for airplane seating assignment (the scenario can be extended to seating assignment in a movie theatre or sport arena). The following is a specification of the application. A possible console output is illustrated in Fig. 10.16:

- There are only two seating types: business and economy. Each row has a variable number of seats, and there are a variable number of rows divided into two sections: business section at the front of the plane followed by the economy section. There is only one aisle in the middle of the plane with seats evenly allocated on each side.
- A customer can request for a seat based on these preferences: business or economy class and an aisle or a window seat. A customer can also request for a seat with no preference. Multiple consecutive seats can also be requested within the limit of the row capacity (e.g., if each side of the aisle has only three seats, then the requested number of consecutive seats cannot be more than three). No seating assignments are made if the request cannot be satisfied.
- A seat is marked with “A” for “available” or “O” for “occupied.”

**Fig. 10.16** Illustration of airline seating assignment

| Flight Seating Info: |     |     |     |     |     |     |
|----------------------|-----|-----|-----|-----|-----|-----|
| 1:                   | [A] | [A] | [A] | [A] |     |     |
| 2:                   | [O] | [O] | [A] | [A] |     |     |
| 3:                   | [A] | [A] | [A] | [A] |     |     |
| 4:                   | [A] | [A] | [A] | [A] |     |     |
| 5:                   | [A] | [A] | [A] | [A] |     |     |
| 6:                   | [A] | [A] | [A] | [A] | [A] | [A] |
| 7:                   | [A] | [A] | [A] | [A] | [A] | [A] |
| 8:                   | [A] | [A] | [A] | [A] | [A] | [A] |
| 9:                   | [A] | [A] | [A] | [A] | [A] | [A] |
| 10:                  | [A] | [A] | [A] | [A] | [A] | [A] |
| 11:                  | [A] | [A] | [A] | [A] | [A] | [A] |
| 12:                  | [A] | [A] | [A] | [A] | [A] | [A] |
| 13:                  | [A] | [A] | [A] | [A] | [A] | [A] |
| 14:                  | [A] | [A] | [A] | [A] | [A] | [A] |

- A customer can also request to cancel a seating assignment by giving the seat number (seats shall be numbered according to the airline convention).
  - The design shall support easy extension of the application to use of a graphic interface (so a user can click a seat button to select the seat).
4. Design and implement a simulation of ATM machine operations as specified in an exercise of Chap. 5. The simulation shall:
- Have a graphical user interface.
  - Interact with a printer and a banking system with features trivially implemented but adequate to support the simulation.
  - Offer two services: checking balance and cash withdrawal (though the design should be ready for additional ATM services). Upon completion of a transaction, a user can select another transaction or exit. Here are the steps for cash withdrawal:
    - (a) Insert a bank card.
    - (b) Select a language.
    - (c) Enter the personal identification number (pin).
    - (d) Select a transaction type and an account type.
    - (e) Enter a cash amount.
5. Design and implement a card game Crazy Eights as a console application. There may be variations of the game rules, but the following rules are assumed:
- The game uses a standard 52-card pack with no Jokers. The dealer deals seven cards to each player. The undealt stock is placed face down on the table, and the top card of the stock is turned face up and placed beside the stock to start this discard pile.
  - Starting with any one player, and continuing clockwise, each player in turn must either play a card face up on top of the discard pile or draw a card from the undealt stock, based on the following rules:
    - 1) If the top card of the discard pile is not an eight, one may play any card that matches the rank or suit of the top card (e.g., if the top card was the king of hearts, one could play any king or any heart). If the card to be played matches the rank, one can play (i.e., discard) multiple cards if they have the same rank. (For example, if the top card was the king of hearts, one could play multiple cards if they were all kings.)
    - 2) An eight may be played on any card, and the player of the eight must nominate a suit. If an eight is on top of the pile, one must play either another eight or any card of the suit nominated by the person who played the eight.
    - 3) A player, when there is no card to play, must draw cards, one at a time, from the stockpile until one that plays.
    - 4) The first player with an empty hand wins.
    - 5) If the stockpile runs out of cards, the discard pile, leaving the top card in the pile, can be used as the new stockpile. Alternatively, the game can be declared inconclusive.



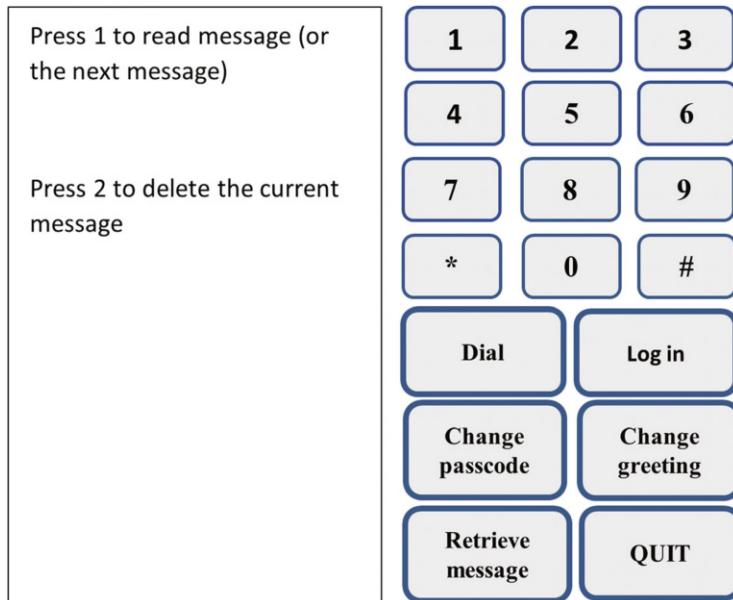
**Fig. 10.17** Interface for a user to leave a message

- There might be good winning strategies, but only a straightforward strategy for all players is needed such as checking the rank for a match first and then the suit. The card game case study in Sect. 10.2 provides a framework for implementing similar games and is applicable to this exercise too. It can be interesting to implement a human player playing against a computer. The program displays the virtual hand of a human player. The human player enters the indices of the cards to play, and the program then removes the corresponding cards from the virtual hand. When playing an eight, the human player enters a desired suit (a character) to be applied to the card.
6. Design and implement a program to simulate a simple “voice” mail system. There are two different kinds of users—those who leave messages and those who retrieve messages (among other things). Specifically, to leave a “voice” mail (illustrated in Fig. 10.17):

- User dials a four-digit number (with a key panel) and then pushes button “Dial.”
- System displays a greeting message in the text area: You have reached mailbox XXXX. Please leave a message; when done, push a button to save the message.
- User enters a message in the same text area and clicks “Save” button.
- System saves the message and displays “your message has been saved.”
- If the number user dialed doesn’t exist, system displays “you dialed a wrong number.”

User of a “voice” mailbox must log in to use its functions with the following steps (Fig. 10.18):

- User enters a four-digit mailbox number through the key panel and then clicks the “Dial” button.
- System displays a prompt: “enter a passcode.”
- User enters a passcode (consisting of characters available on the keypad) and pushes “Log in” button.



**Fig. 10.18** Interface for a mailbox user

- System displays a message: “you may retrieve messages, change the greeting message, or change the passcode.”
- User may repeat the steps above when the text area displays “invalid mailbox number or passcode.”

User can select a task or quit upon a successful login. Available tasks are:

(a) Retrieve message:

1. User clicks button “retrieve message.”
2. System displays the following: “press 1 to read message (or the next message), press 2 to delete the current message.”
3. User presses 1 or 2 as appropriate.
4. System displays current (or the next) message or removes the current message as appropriate.
5. Steps 2–4 may repeat until the system displays “No more messages” or until user selects a different action.

(b) Change greeting message:

1. User clicks “change greeting message” button.
2. System displays prompt: “enter new greeting message, and press ‘star’ to save.”
3. User types the new greeting in the text area, and presses “star” button
4. The system sets the new greeting message, replacing the old.

## (c) Change passcode:

1. User clicks “change passcode” button.
2. System displays prompt: “enter new passcode and press ‘#’ to save.”
3. User types the new greeting in the text area and presses “#” button.
4. System sets the new passcode.

At start of an execution, a user can use a (*JOptionPane*) dialog box to choose between leaving a message or accessing mailbox operations, and an appropriate frame is then activated.

How to handle button events is an interesting design question. A straightforward option is to use a separate event handler for each function button, but it is inelegant. If we think of the functions as “strategies,” one event handler may suffice to handle all function button events with a “strategy” object dispatched from a factory. Similarly, creating keyboard buttons and their listeners with “least” code possible can also be interesting to explore.

7. Design and implement an animation using the animation framework in the case study 10.3. You can implement one of the following scenarios (or your own):

- Pong is one of the oldest digital games people have implemented. It’s a two-dimensional sports game that simulates table tennis. The player controls a screen object called paddle (a short vertical stick) by moving it vertically across the left or right side of the screen to catch a ball. You can compete against another person who controls the paddle on the opposing side. Players use the paddles to catch (i.e., touch) the ball, which would bounce back until one fails to catch the ball.
- Simulation of falling objects can also be interesting. For example, one moves a basket along the bottom edge (with left or right arrow key) to catch falling apples until one apple escapes. Instead, one can also “fire” an object from the screen bottom with the “up” arrow key to hit an object moving across the top. One can use left and right arrow key to control the location of firing to avoid “misfires.”
- A simulated racing among several tortoises and hares. They all start from the same line. A tortoise runs a slow but steady race, whereas a hare runs in spurts with the rest in between. Each tortoise runs at a constant speed, but this speed can individually differ slightly. Each hare spurts at its own constant speed but rests a number of times with a different duration each time. The simulation concludes when a tortoise or hare reaches the finish line.

8. Design and implement a program to animate a ceiling fan operation with four control buttons—Start/Stop, High, Medium, and Low—as they are conventionally defined. The following table summarizes how Start/Stop button would work:

| Start/Stop button | Fan current state    | Fan state to enter                           |
|-------------------|----------------------|----------------------------------------------|
| Push              | No motion            | Running at high speed                        |
| Push again        | Running at any speed | Running at the next lower speed or no motion |

9. Design and implement the game of Paper-Rock-Scissors to play with the computer. At the start of an execution, user chooses between a console and a graphic user interface. People have studied the game scientifically (<https://arstechnica.com/science/2014/05/win-at-rock-paper-scissors-by-knowing-thy-opponent/>), and simple winning strategies do exist (and easy to implement too). To apply the game logic, we typically use an if-else statement (though it can be messy in this case). However, an object-oriented technique known as “double dispatch” can be used instead. The code below explains the way the technique works. Classes *Rock* and *Scissors* can be implemented similarly:

```
enum MatchOutcome{ WIN, LOSE, TIE }
interface PRSPlayer{
 MatchOutcome fights(PRSPlayer other);
 MatchOutcome fightsRock();
 MatchOutcome fightsRock();
 MatchOutcome fightsRock();
}
class Paper implements PRSPlayer{
 MatchOutcome fights (PRSPlayer other){ return other. fightsPaper(); }
 MatchOutcome fightsRock(){ return MatchOutcome.WIN; }
 MatchOutcome fightsPaper(){ return MatchOutcome.TIE; }
 MatchOutcome fightsScissors(){ return MatchOutcome.LOSE; }
}
```

With polymorphism, a match can be executed uniformly with code like *p1.fights(p2)*, where *p1* is called first dispatcher and *p2* second dispatcher (*p1* or *p2* can be a paper, rock, or scissors object). For example, code *rock.fights(paper)* invokes *paper.fightsRock()* resulting *WIN*.

This technique applies when both method calls *obj1.op(obj2)* and *obj2.op(obj1)* are meaningful. Temperature conversion method like *temp1.convert(temp2)* is also symmetrical and therefore can also be implemented using double-dispatch technique with the following interface:

```
interface TempConv{
 double convert(TempConv otherT);
 double toCelsius();
 double toFahrenheit();
}
```

The idea of double dispatch is also used in the visitor design pattern.

10. Write a console-based program to simulate a simple auction system. The system maintains a list of items for auction and provides an interface for users to submit auction items. The following abstraction is to manage auction operations:

```
interface IAuctionServer {
 void placeItemForBid(String ownerId, String itemId, String itemDesc, double startBid);
 void bidOnItem(String bidderName, String itemId, double bid);
 void finishBiddingOnItem();
 ArrayList getItems();
 Item getItem(String itemId);
 void registerClient(AuctionClient c, String itemId);
}
```

where *AuctionClient* is the following abstraction:

```
interface AuctionClient {
 void update(Item item);
}
```

An execution of the application follows these steps:

- On start, the first auction item is displayed along with the current bidding price.
- Bidders submit their bidding amounts during a bidding round, and the system updates the current highest bidding amount after a bidding round completes. Then, the next bidding round starts. The process stops when no biddings are received during a bidding round.
- The auction item goes to the highest bidder, and the next item is displayed for auction until all items are sold.

A bidder may need to apply a bidding strategy. For example, if *high increment* is defined to be the difference between the current and the previous highest bidding amounts, then a bidder might keep a running average of *high increments* and make a bid no more than this average to bid for items with similar starting bidding prices.

11. Design and implement the address book application specified in an exercise of Chap. 5.

---

## References

- Eckerdal, A., et al. (2006, March). Can graduating students design software systems? In *Proceedings of the 37th SIGCSE technical symposium on Computer science education* (pp. 403–407).
- Loftus, C., et al. (2011, March). Can graduating students design: Revisited. In *Proceedings of the 42nd SIGCSE technical symposium on Computer science education* (pp. 105–111).