

Software Engineering Design

Introduction to
Processes, Principles, and Patterns with UML2

Christopher Fox

Introduction to Software Engineering Design

Processes, Principles, and Patterns with UML2



Introduction to Software Engineering Design

Processes, Principles, and Patterns with UML2

Christopher Fox



Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Publisher	Greg Tobin
Executive Editor	Michael Hirsch
Acquisitions Editor	Matt Goldstein
Project Editor	Katherine Harutunian
Associate Managing Editor	Jeffrey Holcomb
Senior Designer	Joyce Cosentino Wells
Digital Assets Manager	Marianne Groth
Media Producer	Bethany Tidd
Marketing Manager	Michelle Brown
Marketing Assistant	Dana Lopreato
Senior Author Support/ Technology Specialist	Joe Vetere
Senior Manufacturing Buyer	Carol Melville
Copyeditor	Kristi Shackelford
Proofreader	Rachel Head
Cover Designer	E. Paquin
Text Designer	Christopher Fox

Cover Image

© 2006 ImageState

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Fox, Christopher John.

Introduction to software engineering design / Christopher Fox.-- 1st ed.

p. cm.

Includes bibliographical references and index.

ISBN 0-321-41013-0 (alk. paper)

1. Software engineering. I. Title.

QA76.758.F685 2006

005.1--dc22

2006008445

Copyright © 2007 Pearson Education, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. For information on obtaining permission for use of material in this work, please submit a written request to Pearson Education, Inc., Rights and Contracts Department, 75 Arlington Street, Suite 300, Boston, MA 02116, fax your request to 617-848-7047, or e-mail at <http://www.pearsoned.com/legal/permissions.htm>.

1 2 3 4 5 6 7 8 9 10—RRDC—10 09 08 07 06

Contents

Preface ix

Part I Introduction 1

Chapter 1 A Discipline of Software Engineering Design	3
1.1 What Is Software Design?	3
1.2 Varieties of Design	12
1.3 Software Design in the Life Cycle	16
1.4 Software Engineering Design Methods*	24
<i>Further Reading, Exercises, Review Quiz Answers</i>	27

Chapter 2 Software Design Processes and Management	33
2.1 Specifying Processes with UML Activity Diagrams	33
2.2 Software Design Processes	47
2.3 Software Design Management*	56
<i>Further Reading, Exercises, Review Quiz Answers</i>	63

Part II Software Product Design 69

Chapter 3 Context of Software Product Design	71
3.1 Products and Markets	71
3.2 Product Planning	74
3.3 Project Mission Statement	79
3.4 Software Requirements Specification	85
<i>Further Reading, Exercises, Review Quiz Answers</i>	92

Chapter 4 Product Design Analysis	98
4.1 Product Design Process Overview	98
4.2 Needs Elicitation	104
4.3 Needs Documentation and Analysis	109
<i>Further Reading, Exercises, Review Quiz Answers</i>	115

Chapter 5 Product Design Resolution	120
5.1 Generating Alternative Requirements	121
5.2 Stating Requirements	126
5.3 Evaluating and Selecting Alternatives	131

5.4 Finalizing a Product Design	136
5.5 Prototyping	142
<i>Further Reading, Exercises, Review Quiz Answers</i>	149
Chapter 6 Designing with Use Cases 157	
6.1 UML Use Case Diagrams	158
6.2 Use Case Descriptions	168
6.3 Use Case Models	178
<i>Further Reading, Exercises, Review Quiz Answers</i>	185
Part III Software Engineering Design 191	
Chapter 7 Engineering Design Analysis 193	
7.1 Introduction to Engineering Design Analysis	194
7.2 UML Class and Object Diagrams	200
7.3 Making Conceptual Models	212
<i>Further Reading, Exercises, Review Quiz Answers</i>	220
Chapter 8 Engineering Design Resolution 226	
8.1 Engineering Design Resolution Activities	226
8.2 Engineering Design Principles	231
8.3 Modularity Principles	233
8.4 Implementability and Aesthetic Principles	244
<i>Further Reading, Exercises, Review Quiz Answers</i>	248
Chapter 9 Architectural Design 253	
9.1 Introduction to Architectural Design	254
9.2 Specifying Software Architectures	259
9.3 UML Package and Component Diagrams	269
9.4 UML Deployment Diagrams*	277
<i>Further Reading, Exercises, Review Quiz Answers</i>	281
Chapter 10 Architectural Design Resolution 287	
10.1 Generating and Improving Software Architectures	288
10.2 Evaluating and Selecting Software Architectures	300
10.3 Finalizing Software Architectures	307
<i>Further Reading, Exercises, Review Quiz Answers</i>	312
Chapter 11 Static Mid-Level Object-Oriented Design: Class Models 318	
11.1 Introduction to Detailed Design	319
11.2 Advanced UML Class Diagrams	324

11.3 Drafting a Class Model	336
11.4 Static Modeling Heuristics	345
<i>Further Reading, Exercises, Review Quiz Answers</i>	352
Chapter 12 Dynamic Mid-Level Object-Oriented Design: Interaction Models	359
12.1 UML Sequence Diagrams	359
12.2 Interaction Design Process	374
12.3 Interaction Modeling Heuristics	381
<i>Further Reading, Exercises, Review Quiz Answers</i>	389
Chapter 13 Dynamic Mid-Level State-Based Design: State Models	395
13.1 UML State Diagrams	395
13.2 Advanced UML State Diagrams*	407
13.3 Designing with State Diagrams	415
<i>Further Reading, Exercises, Review Quiz Answers</i>	423
Chapter 14 Low-Level Design	429
14.1 Visibility, Accessibility, and Information Hiding	430
14.2 Operation Specification	439
14.3 Algorithm and Data Structure Specification*	448
14.4 Design Finalization	452
<i>Further Reading, Exercises, Review Quiz Answers</i>	456
Part IV Patterns in Software Design	461
Chapter 15 Architectural Styles	463
15.1 Patterns in Software Design	463
15.2 Layered Architectures	467
15.3 Other Architectural Styles	473
<i>Further Reading, Exercises, Review Quiz Answers</i>	486
Chapter 16 Mid-Level Object-Oriented Design Patterns	490
16.1 Collection Iteration	490
16.2 The Iterator Pattern	498
16.3 Mid-Level Design Pattern Categories	504
<i>Further Reading, Exercises, Review Quiz Answers</i>	506
Chapter 17 Broker Design Patterns	510
17.1 The Broker Category	510
17.2 The Façade and Mediator Patterns	513

17.3 The Adapter Patterns	522
17.4 The Proxy Pattern*	529
<i>Further Reading, Exercises, Review Quiz Answers</i>	534
Chapter 18 Generator Design Patterns	540
18.1 The Generator Category	540
18.2 The Factory Patterns	544
18.3 The Singleton Pattern	553
18.4 The Prototype Pattern*	557
<i>Further Reading, Exercises, Review Quiz Answers</i>	564
Chapter 19 Reactor Design Patterns	568
19.1 The Reactor Category	568
19.2 The Command Pattern	572
19.3 The Observer Pattern	578
<i>Further Reading, Exercises, Review Quiz Answers</i>	586
Appendices	
Appendix A Glossary	591
Appendix B AquaLush Case Study	609
Appendix C References	689
Index	693

Preface

Introducing Software Engineering Design

This book is an introduction to the technical aspects of software engineering analysis and design for novice software developers.

As an *introduction*, the book surveys all of software engineering design without delving deeply into any one area. Although it contains material from various analysis and design methods, the majority of the material is from object-oriented methods, reflecting a judgment that this approach is the richest and most powerful.

As a book about the *technical* aspects of software engineering analysis and design, it focuses mostly on the technical task of understanding a software problem and specifying a design to solve it. This book does not include complete discussions of managerial issues, such as design project planning, estimating, scheduling, or tracking, or project organization and leadership, though an introductory survey of these topics is provided in the second chapter.

As a book about engineering *analysis and design*, it focuses on the traditional design phase of the software life cycle. It does not consider larger life cycle issues, such as life cycle models or overall software processes, except in passing. The waterfall life cycle model is used pedagogically to provide context for the discussion.

Furthermore, as a book about software *engineering* design, it does not go into as much detail about software requirements specification as it does about technical design. The former topic is covered at a survey level in Part II of the book as a lead-in to the discussion of engineering design.

As a book for *novice* designers, it begins with basic software engineering analysis and design material and uses simple examples for illustrations and exercises. Many deep issues are avoided or glossed over, and few assumptions are made about the reader's familiarity with languages and systems.

Design Perspective

This book takes a *design perspective* throughout. From this perspective, software product development begins with software design and proceeds through implementation, testing, and deployment. The software design activity is divided into *software product design* and *software engineering design*. Software product design occurs in the traditional requirements specification phase of the software life cycle, and includes user interface design. From this perspective, one might call the initial phase of the life cycle the *product design phase* rather than the *requirements specification phase*. During the product design phase, developers specify the features,

capabilities, and interfaces that a software product must have to satisfy the needs and desires of clients. This specification includes determination of how the product must interact with its environment, including human users. From the design perspective, requirements engineering and user interface design are melded into the single activity of software product design. See [Armitage 2003] for a similar point of view.

The goal of the traditional design phase is to figure out a software structure that, when implemented, results in programs that realize the desired software product. The output of this design activity is a specification of software systems and sub-systems, along with their constituent modules, classes, libraries, algorithms, data structures, and so forth. From the design perspective, this phase might be termed the *engineering design phase* rather than simply the *design phase*. Thus, this book's discussion of software engineering design is essentially a discussion of the traditional design phase.

Prerequisites Students with grounding in object-oriented programming and familiarity with fundamental data structures and algorithms should be able to understand this book. More specifically, readers are assumed to have at least nodding familiarity with the following topics:

- Object-oriented programming concepts, including the concepts of objects and classes, attributes and operations, polymorphism and inheritance, encapsulation, abstract classes, and interface types.
- Programming in an object-oriented language, preferably Java.
- Fundamental abstract data types such as stacks, queues, and lists, and how to implement them using linked or contiguous data structures.
- Algorithms for implementing the abstract data types just mentioned, for sorting (such as quicksort, insertion sort, and heapsort), and for searching (such as sequential and binary search).

Learning to Design

There are many things that people must learn to become good designers, including the following items:

The Nature of Design—What design is and why it is important, how design fits into the software life cycle, and the role that design plays in the development process.

A Design Process—A sequence of steps for understanding a design problem, formulating designs, evaluating them, and improving them until a satisfactory design is created and documented.

Design Notations—Symbol systems for expressing designs.

Design Principles or Tenets—Fundamental statements about what makes designs good or bad used in generating and evaluating designs.

Design Heuristics—Rules of thumb, or procedures, that aid in generating good designs and good design documentation.

Design Patterns—Designs, parts of designs, or templates for designs that can be imitated in generating new designs.

This book covers all of these items.

Exercises and Projects	Formal education can be more or less successful in teaching these topics, but only if the learner is asked to actually <i>make</i> designs. Design is mainly a skill rather than a body of declarative knowledge, so design proficiency is gained primarily through the experience of designing. Even the declarative knowledge that must be acquired (such as design principles and heuristics) is useless to most students until they have found occasion to apply their knowledge in generating or evaluating designs. This book emphasizes <i>using</i> the design material introduced. Each chapter includes about 20 exercises of varying difficulty, many of which ask students to actually make designs. There are also team design projects at the ends of most chapters.
Examples	Examples are an important resource for students. An ambitious running example (AquaLush) is discussed in almost every chapter and worked completely in Appendix B. Many other examples are introduced as well. The examples are drawn from a wide range of application areas, with an eye toward the sorts of problems that students understand and are interested in.
Summary Materials	Students find summary materials helpful, so there are many summaries of various kinds in the book: <ul style="list-style-type: none">▪ An overview appears at the beginning of each of the four parts of the book. It describes the contents of each chapter in that part.▪ A chapter overview begins each chapter. It describes the contents of the chapter and lists learning objectives.▪ A section summary appears at the end of each section. It lists the key terms and the main points of the section.▪ A short quiz appears at the end of each section, with answers provided at the end of the chapter.▪ A summary table of heuristics appears at the end of each section that introduces heuristics.▪ A summary table of architectural styles or design patterns appears at the end of each section that introduces styles or patterns.▪ A glossary defines all the emboldened terms in the main text.

Notations I agree with the claim that what people can think is constrained by the notations they know. If this is so, knowing any design notation is a prerequisite for design, and knowing many notations broadens the range of design solutions that can be conceived. Hence, it is good for designers, especially novices, to learn several design notations. On the other hand, attempting to learn too many notations can lead to confusion or indecision between alternatives.

The importance of design notations for novice designers makes them the centerpiece of the book. I have attempted to strike a middle ground between presenting too few notations, resulting in an impoverished design vocabulary, and too many, resulting in lack of fluency and inability to use the right notation for the job. All notations are presented with guidance for when to use them, many examples of their use, and design exercises to help students achieve notational fluency.

The wide acceptance of the Unified Modeling Language (UML), a rich collection of notations that is also a prerequisite for reading most of the patterns literature, makes it an obvious choice. Several other notations are so useful and common that their inclusion is also warranted.

A new version of UML has recently been adopted as a standard. UML 1.5 was already a large and complex collection of notations, and UML 2 is significantly larger and more complex than UML 1.5. Many aspects of UML 2 are not yet clear; nor is it yet clear which notations and notational variations will prove popular. Although this book uses UML 2 notations, it does not include the diagrams introduced in UML 2, and it does not incorporate all the new features of UML 1.5 diagrams introduced in UML 2. Nevertheless, the UML notations covered in the book form a rich and complete collection of notational tools.

Book Contents and Structure

The book has four parts and appendices:

Part I—An introduction to software design. Chapter 1 provides an overview of software design and its context. A survey of software design processes and management comprises Chapter 2.

Part II—A survey of software product design. It begins with a sketch of the organizational context for product design and the product design process in Chapter 3. Chapter 4 considers product design analysis, and Chapter 5 covers product design solution creation. Use case models are discussed in detail as a product design tool in Chapter 6.

Part III—A detailed look at software engineering design. Engineering design analysis is covered first, in Chapter 7, followed by an overview of creating engineering design solutions, in Chapter 8. Chapter 9 discusses architectural modeling, and Chapter 10 covers architectural design resolution. The next three chapters cover mid-level (that is, module- or class-level) design. Chapter 11 discusses class models, Chapter 12 interaction models, and Chapter 13 state models. Chapter 14 surveys low-level design and design phase finalization.

Part IV—An introduction to patterns. Chapter 15 introduces patterns in software design and presents a small collection of architectural styles. Chapter 16 is a detailed case study of iteration and the Iterator pattern. Chapter 17 covers four broker patterns. Chapter 18 discusses four generator patterns, and Chapter 19 covers two reactor patterns.

Appendices—There are three appendices: a glossary, an extensive case study, and references.

Typographical Conventions Model elements are denoted by text in a sans serif font; program code is presented in a constant width font. Important terms are **emboldened** when they are defined or introduced in the main text and when they appear in the summary of the section in which they are introduced. As mentioned above, all emboldened terms appear in the glossary in Appendix A.

Using This Book as a Text There is more than enough material in this book to fill a one-semester course on software engineering design. Most chapters are designed to be covered in one to two weeks. Assuming a typical 15-week semester, instructors can choose about 70% of the material to fill a course. This choice can be made thematically, for example, in the following ways:

- A software design survey course might spend about equal amounts of time on software product design and software engineering design. In this case the instructor might cover most of Chapters 1 through 12 and a few sections from Chapters 13 through 19.
- A course emphasizing engineering design might skip most of the product design chapters. The instructor might choose Chapters 1, 2, 6, and 7 through 14, with chapters from Part IV covered as time permits.

Instructors may also leave out sections, or parts of sections, containing less important material to make more room for other topics. For example, Section 4 of Chapter 1 is a brief historical sketch of analysis and design methods that can easily be omitted, while Section 2 of Chapter 12 discusses state diagram features that are not often used. Such sections are marked with an asterisk in the table of contents.

Supplementary Materials Additional teaching material is available at Addison-Wesley's Web site: <http://www.aw.com/cssupport>. This material includes

- An implementation of the AquaLush case study discussed throughout the book and documented in Appendix B, including Java source code;
- Additional design problems for use in exercises or projects;
- Complete directions for many in-class design activities;
- PowerPoint slides for selected portions of the book.

Appreciations I thank the many people who helped and encouraged me during the long process of writing this book. The following people have been especially helpful.

David Bernstein used drafts of this book to teach software design. He also read the manuscript several times, made many detailed comments, and discussed many aspects of software design and its pedagogy over several years.

Michael Norton's work in his M.S. thesis is the basis for the pattern classification used in Part IV of the book. Mike also helped me understand various aspects of patterns and how to teach them.

Robert Zindle made a diagram of the material in the book upon which the summary diagrams on the insides of the front and back covers are based.

Many students used drafts of this book in my graduate and undergraduate software analysis and design courses. Several made recommendations for improvement, especially Jason Calhoun, David Lenhardt, Stephen Ayers, and Jack Hirsch.

Several of my colleagues at James Madison University read and commented on drafts of various chapters, including Taz Daughtry, Ralph Grove, and Sam Redwine.

Kristi Shackelford and Rachel Head both did an excellent job copyediting the manuscript; their countless suggestions improved the text enormously. My colleague Alice Philbin also helped me with issues of grammar and style.

Many anonymous reviewers made valuable suggestions that resulted in changes ranging from radical restructuring of the entire manuscript to rewriting a sentence or two.

James Madison University granted me an educational leave in the fall of 2004 during which a large portion of this book was written.

Finally and most importantly, I thank my wife Zsuzsa, who spent many hours taking care of our two daughters by herself while I wrote and revised. This project could not have been completed without her patience and support.

Part I Introduction

The first part of this book introduces the field of software engineering design and places it in the context of software development.

Chapter 1 explains what software design is and why it is important. The chapter also differentiates between product and engineering design, explains the relationship between the two, discusses when they occur in the software life cycle, and describes some popular engineering design methods.

Chapter 2 introduces a process specification notation and uses it to describe the software design process that structures the remainder of the book. Chapter 2 also surveys software design management as a complement to our focus on technical issues in the rest of the book.



1 A Discipline of Software Design

Chapter Objectives This chapter introduces software design by discussing design in general, design in software development, the place of design in the software life cycle, and methods that have been used for software engineering design. Design is presented as a form of problem solving.

After reading this chapter you will be able to explain and illustrate

- What design is, and how different kinds of design deal with various aspects of products, from their external features and capabilities to their internal workings;
- How design is related to problem solving, and how problem solving informs design activities, structures design processes, and provides design techniques;
- The role of abstraction and modeling in design;
- Where design fits into the software life cycle; and
- What software engineering design methods are, and what methods have been popular.

Chapter Contents
1.1 What Is Software Design?
1.2 Varieties of Design
1.3 Software Design in the Life Cycle
1.4 Software Engineering Design Methods

1.1 What Is Software Design?

The Designed World Software design is one of many design activities in contemporary society. It is striking, if one has never thought about it, how much of the world around us is designed. Of course buildings, roads, and landscapes are designed, but so are many foods, materials, services, plants and animals, and much more. We live in an almost completely designed environment, where nearly everything around us is contrived to satisfy our needs and desires.

An obvious consequence of the prevalence of design is that the quality of designs in the world around us profoundly influences the quality of our lives. Well-designed products function smoothly to enable or assist us at work and at play. Poorly designed products frustrate us, decrease our productivity, cost money and time, and may threaten our safety.

In his landmark book *The Design of Everyday Things*, Donald Norman makes a convincing case that the designs of things in our everyday environment strongly affect our feelings and attitudes. Stress levels soar when we are

unable to heat coffee in microwave ovens, open shrink-wrapped objects, remove keys from car locks, or get an outside telephone line. Many people feel stupid and inadequate when unable to figure out how to do simple tasks with their computers, VCRs, or kitchen appliances. In most cases, Norman argues, these problems can be directly attributed to poor design.

Although one often thinks of poor quality in connection with product manufacturing or service delivery, in fact design is often the root cause of poor quality. For example, in mechanical engineering, it is estimated that 85% of problems with new mechanical products are design problems.

Computers are becoming an integral part of our world as they make their way into appliances, homes, vehicles, and our mechanical and social infrastructure, and as more and more people make their livings as knowledge workers. Software drives computers. Inevitably, software design will play an ever-larger role in determining the quality of our lives.

Every design discipline has studies showing the economic importance of design. Poor designs impose costs on suppliers arising from manufacturing and distribution problems, greater service expenses, returns from dissatisfied customers, lost business, and product liability claims. Poor designs cost consumers money due to lower productivity, lost or damaged goods, and increased maintenance costs. Poor design can even lead to injuries or deaths. On the other hand, good design can save suppliers money in manufacturing, distribution, and service, and it can be the factor that makes a product a success. A well-designed product can increase productivity; reduce loss, damage, and injury; and open up new business and leisure possibilities.

To illustrate the economic importance of software design, consider how frequently personal computer programs crash without saving users' work, often resulting in the loss of hours of effort. This fundamental design flaw costs billions of dollars every year, as well as being very frustrating.

Conversely, important software products such as spreadsheets and word processors have increased productivity enormously. As computers become ever more widely used, the economic importance of software design will only increase.

In summary, software is increasingly important as a determinant of our quality of life and because of its economic impact. Software design is crucial, and software designers must be aware of the importance of their work and of their responsibilities for doing it competently and carefully.

Software Products

Now that we have established that software design is important, we ought to be more specific about what it is. Software designers design software products, so we will begin by considering what software products are.

In normal speech, we designate as "software" whatever runs on computer hardware. This observation is the basis for the following definition.

Software is any executable entity, such as a program, or its parts, such as sub-programs.

Software exists in the form of code, but it is not important for our purposes to distinguish between source and object code, or between segments of code written in various languages.

Software needs to be designed, but software designers must worry about more than just program code. Programmers write software to create or modify programs for *clients*. Although software is needed to make computers behave in the ways that clients want, it is not what they are really interested in—clients want things that solve their problems, not just programs. In general, clients are interested in products, not just software.

Many products have software in them, such as cars, houses, refrigerators, and so forth. Software designers help create the software that goes into such products, but they must do more than simply specify programs to run in the computers in such devices. For example, they must define and document the interface between the software and the rest of the product, describing in detail how the software will behave so that the designers of the rest of the product can use it. If the software interacts with consumers, the software designers will have to help determine the product's functions and capabilities, as well as design and document the product's user interface. Software designers will have to provide support to other designers, answering their questions and clearing up confusions about the software.

Software designers also help create software packages that users can load and execute on a computer, such as operating systems, spreadsheets, word processors, and media players. In all cases, software designers must be concerned not just with specifying the program code for these products, but also with designing and documenting the program's functions, capabilities, data storage, and interface with its surroundings (especially its users). Software designers may even specify the services that go along with the program.

Software designers are thus responsible for a larger entity that includes software but is usually more than software alone: a software product.

A **software product** is an entity comprised of one or more programs, data, and supporting materials and services that satisfies client needs and desires either as an independent artifact or as an essential ingredient in some other artifact.

A software product is not the same as a product with software in it. A car is a product with software in it, but it is not itself a software product. Products with software in them contain software products. For example, the software and supporting data, materials, and so on used to control a car's carburetor constitute a software product. On the other hand, many

software products, such as operating systems or Web browsers, are not part of some other product. We might term such products *stand-alone* software products.

A software product will be successful only if it satisfies clients' needs and desires, so software designers must be concerned with creating software products that satisfy clients' needs and desires. In this regard software development does not differ from other creative disciplines—every design discipline aims to satisfy the needs and desires of clients. Typically, there are also constraints on the designer in attaining this goal. For example, a satisfactory design must be feasible, affordable, and safe. Designers, then, strive to satisfy client needs and desires subject to constraints.

**Software
Design
Defined**

Bakers, bankers, and ball bearing makers all satisfy client needs and desires subject to constraints when they bake cakes, serve depositors, or manufacture ball bearings, but they do not design. The designer does not provide services or manufacture artifacts, but specifies the nature and composition of products and services.

Software designers do what designers in other disciplines do, except that they do it for software products. This leads us to the following definition.

Software design is the activity of specifying the nature and composition of software products that satisfy client needs and desires, subject to constraints.

We consider the ramifications of this definition in greater detail in the remainder of this chapter.

**Design as
Problem
Solving**

A client with needs and desires for a product has a problem to solve. A design that specifies a product satisfying these needs and desires, subject to constraints, is a solution to the client's problem. Design is a kind of problem solving.

Thinking about design as problem solving has many advantages. First, it suggests that information may be partitioned between the problem and the solution. One of the greatest sources of error and misunderstanding in design is confusion over what is part of the problem and what is part of a solution. For example, suppose that using a certain network configuration is part of a design problem because the client company mandates it, but the designers believe that it is part of a solution suggested by the client. The designers may change the network configuration to produce a better solution, not realizing that this makes the design unacceptable.

Alternatively, the designers may believe that some aspect of the program is part of the problem when it is not, leading to a poor solution to the real problem. Understanding what is part of the problem and what is part of the solution is one of the first and most important rules of design. We will return to this topic later when we discuss problem analysis.

A second advantage of thinking about design as problem solving is the perspective it gives on design. There are usually many solutions to any problem; some may be better than others, but often, several are equally acceptable. Similarly, there are usually many reasonable designs for any product. Some have drawbacks, but typically several are equally good.

A third advantage of thinking about design as problem solving is that it suggests the use of time-honored general problem solving techniques in design. Some of these techniques are

Changing the Problem—When a problem is too difficult, sometimes it can be changed. Clients may be willing to relax or modify their desires, particularly when offered a solution to a slightly different problem that still meets their ultimate goals.

Trial and Error—Problem solving is an inherently iterative activity, and design should be, too. We discuss the iterative nature of design further when we consider the software design process in Chapter 2.

Brainstorming—The first solution that comes to mind is rarely the best, and different parts of solutions often come from different people. Brainstorming is a valuable problem-solving technique that works well in design.

We will use these and other problem-solving techniques throughout this book.

Abstraction

A general problem-solving technique of particular importance in design is abstraction, which we define as follows.

Abstraction is suppressing or ignoring some properties of objects, events, or situations in favor of others.

When we consider the shape of a thing without regard to its color, texture, weight, or material, or when we consider the electrical properties of a thing without regard to its shape, color, size, and so forth, we are abstracting.

Abstraction is an important problem-solving technique for two reasons:

Problem Simplification—Real problems always have many details irrelevant to their solution. For example, suppose you need to pack your belongings into a moving van. You need not consider the colors, patterns, or styles of your boxes and suitcases, nor the colors, patterns, fabrics, light, or sound in the back of the van. The only information you need to solve this problem is the shape and size of the van's interior, and the shapes, sizes, weights, and load-bearing capacities of your parcels. You can concentrate on the pertinent features of the problem without having to keep other details in mind.

This use of abstraction is embodied in standard design tools and notations in many disciplines. For example, drawing conventions and computerized drawing tools for building design support various levels of

abstraction from dimension, color, texture, shape, size, weight, and materials in various kinds of diagrams and drawings.

Structuring Problem Solving—Many design problems are too large and complex to solve all at once. One way to attack such a problem is to solve an abstract version of the problem, then enhance the solution to account for more and more detail. We call this process **refinement**. This **top-down strategy** is very common in design. Consider, for example, the design of a large building complex, such as a university campus or an office park. Designers begin by abstracting all features of the buildings except their locations, masses, and functions. At this level of abstraction, the focus is on satisfying goals and constraints concerning traffic, parking, sight lines, building access, topography, and the relationships between the development and adjoining land and structures. Once this part of the problem is solved, details are added as the buildings' footprints, facades, and connections are developed. Refinement continues until the entire design problem is solved.

In practice, designers often work partly using a **bottom-up strategy**, solving pieces of a complex problem in detail, then connecting the solved pieces to form a solution to the entire problem. This usually works only when a top-down framework already exists. The top-down problem-solving approach, based on abstraction, is the dominant strategy for large-scale design problem solving.

The use of abstraction for problem simplification is the basis for modeling, which we consider next.

What Is a Model?

A model railroad has parts, such as engines, cars, and tracks, and relationships between parts, such as engines pulling cars, that resemble those in a real railroad. But the model railroad also simplifies a real railroad by not representing every detail of the real thing. The essence of modeling is thus the representation by the model of what is being modeled. This leads to the following definition.

A **model** is an entity used to represent another entity (the *target*) by establishing (a) a correspondence between the parts or elements of the target and the parts or elements of the model, and (b) a correspondence between relationships among the parts or elements of the target and relationships among the parts or elements of the model.

In other words, a model represents a target by having parts and relationships arranged in imitation of the target. Figure 1-1-1 illustrates how modeling works.

In Figure 1-1-1, boxes are parts of the target, and circles are parts of the model. Dashed lines connect model parts corresponding to target parts. The model's circles abstract the target's boxes (details about corners are ignored). In the target, three boxes are arranged in a boxed group. This

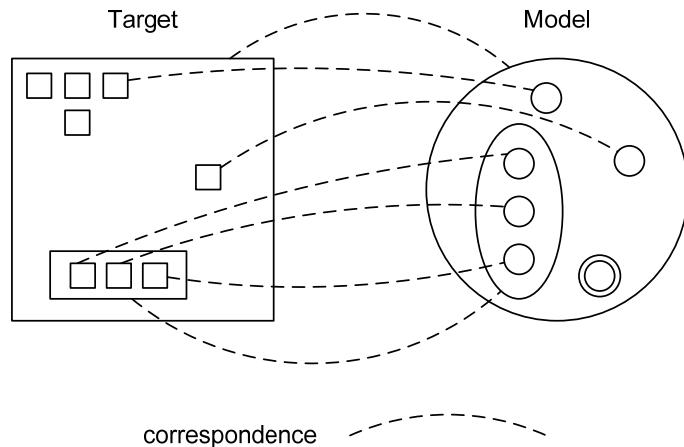


Figure 1-1-1 How Modeling Works

relationship is captured in the model by a corresponding circled group of circles. The model not only has parts corresponding to the parts of the target, but also relationships between model parts corresponding to relationships between target parts. There are also parts and relationships in the target that are *not* captured in the model, and parts and relationships in the model that do *not* correspond to anything in the target. This highlights the fact that models are not perfect renditions of their targets.

Consider a model railroad to further illustrate this definition. As noted, a model railroad has parts that correspond to parts of a real railroad, including engines, cars, tracks, trestles, and so forth. The parts of the model are also related to one another in ways that correspond to relationships in a real railroad: engines pull cars over tracks, while trains can derail, collide, be switched from one track to another, and so on. There are aspects of the target (the real railroad) that are not captured in the model, though. For example, the model has no people in it, does not burn diesel fuel, and does not extend proportionally as far as a real railroad would. The model also has parts and relationships not corresponding to anything in the target. For example, models have electrical connections, fasteners, and materials quite unlike those in real railroads.

Abstraction is essential in modeling. Models represent only some of the parts, properties, and relationships of a target—otherwise they would be copies rather than models. The fact that models are abstractions of their targets is the key to the greatest strengths of modeling. By including only the target details relevant for solving a problem, models allow us to solve problems that would otherwise be too complicated. Models that abstract aspects of a target that make it dangerous, expensive, or hard to study allow us to solve problems safely and cheaply. Models also document the essential aspects of problems and solutions and serve as guides in implementing solutions. These advantages combine to make models a

major tool in science, engineering, and the arts for understanding and investigating problems and creating and documenting solutions.

Abstraction is also the source of the greatest weakness of modeling. If important parts or relationships are missing from a model, or if the behaviors or properties of the model do not closely enough approximate the behaviors and properties of the target, a model can produce misunderstandings leading to incorrect predictions, conclusions, and solutions. Knowing how and what to abstract is a key skill in modeling. So is the ability to use models with an understanding of what they can't do.

Modeling in Design

If design is problem solving, then one might expect modeling to be a central tool in all design disciplines. Models are useful in design in three ways:

Problem Understanding—Designers must understand design problems and constraints before they can create solutions. Models can help represent and explore problems.

Design Creation and Investigation—Floor plans, elevations, schematics, blueprints, and diagrams of all sorts are the mainstays of design creation and investigation in most disciplines. Mock-ups, partial implementations, and physical scale models are also widely used and allow investigation through simulation and testing.

Documentation—Models developed to understand, create, and investigate are also used to document designs for implementation and maintenance. Additional models are sometimes made just for documentation.

Modeling in Software Design

Most software design models are symbolic representations, though programs that implement part of the final result (prototypes) are sometimes used. Software design problems and solutions are often complex, and many aspects of software systems must be modeled. Consequently, many kinds of models are used in software design. Most chapters in this book discuss types of software design models or the notations used to express them.

Software design models may be divided into two broad classes: static and dynamic models.

A **static design model** represents aspects of programs that do not change during program execution.

Generally, static models represent software constituents, their characteristics, and unvarying relationships between them. Static models discussed in this book include object and class models, component and deployment diagrams, and data structure diagrams.

A **dynamic design model** represents what happens during program execution.

Dynamic models discussed in later chapters include use case descriptions, interaction diagrams, and state diagrams.

Both static and dynamic models are important in software design for problem understanding and investigation, solution creation and investigation, and documentation.

Section Summary

- Software design has important and growing consequences in the day-to-day lives of ordinary people, with profound effects on our quality of life and economic well being.
- **Software** is any executable entity, such as a program, or its parts, such as subprograms.
- A **software product** is an entity comprised of one or more programs, data, and supporting materials and services that satisfy client needs and desires either as an independent artifact or as an essential ingredient in some other artifact.
- **Software design** is the activity of specifying the nature and composition of software products that satisfy client needs and desires, subject to constraints.
- Design is a problem-solving activity, and problem-solving techniques are important for design.
- **Abstraction** is ignoring some details of a thing in favor of others.
- A **model** is an entity used to represent another entity (the target), by establishing (a) a correspondence between the parts or elements of the target and the parts or elements of the model, and (b) a correspondence between relationships among the parts or elements of the *target* and relationships among the parts or elements of the model.
- Modeling is the main tool that designers use to understand, create, investigate, and document designs.
- Software designers make models to show the static and dynamic aspects of software systems.

Review Quiz 1.1

1. Why is software design important?
2. Give an example, different from the one in the text, of a product with software in it that is not itself a software product.
3. Name three advantages of thinking about design as problem solving.
4. Give three reasons why abstraction is an important problem-solving technique.
5. List the benefits and dangers of modeling.

1.2 Varieties of Design

Product Design

The academic and professional field of design began in the mid-19th century during the later stages of the Industrial Revolution. Industrial and mass production require precise specification of finished goods as a prerequisite for setting up manufacturing processes. Designers emerged to meet this need. Early designers were primarily concerned with consumer products such as furniture, china, glassware, cutlery, household appliances, and automobiles. Before long, designers also began working in communications media, producing posters, books, flyers, and advertisements.

In the late 20th century designers turned to planning and specifying more abstract things, such as services, systems, corporate identities, and even lifestyles. Design has grown into a specialized field with sub-fields such as communications, graphic design, and industrial design. It is closely allied with several older disciplines, principally architecture and studio art. Designers are educated in schools of design, art, or architecture.

Although commonly known simply as “design,” we will hereafter refer to this profession as **product design** and its practitioners as *product designers*. We adopt this terminology to help distinguish product design from the other design disciplines discussed next.

Engineering Design

Product designers by inclination and training are mainly concerned with styling and aesthetics, function and usability, manufacturability and manageability, and social and psychological roles and effects of artifacts and services. They are less concerned with, and largely unqualified to specify, the inner structure and workings of technically complex products. Such matters are usually delegated to other professionals, mainly engineers. Product designers handle the “externals” of product design while engineers take care of the “internal” technical details.

For example, a product designer may specify a hand-held vacuum cleaner’s form, colors, surface materials, and controls, but mechanical and electrical engineers specify the motor, the dust filter, the fan blade and housing, and the exhaust port capacity.

These latter, technical matters are the province of **engineering design**, the activity of specifying the technical mechanisms and workings of a product. *Engineers* receive specialized training in applying scientific and mathematical techniques to the specification of efficient, reliable, safe, and cost-effective mechanisms, systems, and processes to realize a product.

Design Teams

Some products, such as rugs, pottery, clothing, or pet-care services, require little engineering but a great deal of aesthetic and functional appeal. Product designers often specify such things without engineering help. Other products, including device components, manufacturing equipment,

and various technical consulting services, demand great technical expertise but little else, and engineers usually specify such products without help from product designers. But many products, such as buildings, cars, stereos, telephones, computers, home appliances, and home security systems, are both technically complex and required to meet stringent demands for function, ease of use, and style. The talents and skills of both product designers and engineers are needed to design such things.

Table 1-2-1 illustrates the complementary responsibilities of product and engineering designers for several products.

Product	Product Designers	Engineering Designers
Recliner	Size, styling, fabrics, and controls	Reclining mechanism and frame
Clothes Drier	Capacity, features (timed dry, permanent press cycle, etc.), dimensions, controls and how they work, styling, and colors	Frame, cylinder, drive and fan motors, heating elements, control hardware and software, electrical and mechanical connections, and materials
Clock Radio	Features (number of alarms, snooze alarm, etc.), displays, controls and how they work, case and control styling and colors	Clock, radio, display, digital and mechanical control and interface hardware, control software, electrical and mechanical connections, and materials
Refrigerator	Capacity, features (ice maker, ice-water spigot, etc.), dimensions, number and arrangement of compartments, shelves, doors, controls and how they work, colors, lighting, and styling	Refrigeration mechanisms, insulation, water storage, pumps, plumbing, electrical wiring and connections, mechanical frame and connectors, and materials

Table 1-2-1 Product and Engineering Designers' Responsibilities

Some especially talented and learned individuals can design all aspects of a sophisticated product, from its form and function to its most complex technical details. But usually product designers and engineers must work together to design complex and sophisticated products on a **design team**. Furthermore, sophisticated products are often too large for a single person to design. Design teams may enlist many product designers and engineers to get the job done better and faster.

Besides product designers and engineers, design teams may have marketing specialists, manufacturing engineers, quality assurance specialists, and product managers, depending on the nature of the product and the needs and desires of clients and the developers.

Software Product Design Software design is such as the design of other artifacts when it comes to the roles of product designers and engineers. Software products often need to include a careful selection of features and capabilities, be easy to use, be visually appealing, structure and present complex information, fit into existing business processes, and so forth. Specifying software products to meet these sorts of needs is a specialized kind of product design that we call software product design.

Software product design is the activity of specifying software product features, capabilities, and interfaces to satisfy client needs and desires.

Depending on the application, software product design may demand talent and expertise in user interface or interaction design, communications design, industrial design, and marketing. These disciplines are large, sophisticated, and highly specialized. Software product designers may be trained in one or more of these areas, and specialists in several areas may be needed on a design team.

Software Engineering Design

Besides product specifications to meet client needs in the ways described above, a software design must of course include specification of the structure and workings of the code. This sort of knowledge is equivalent to the technical knowledge of engineers in other disciplines. Thus, software design includes software engineering design.

Software engineering design is the activity of specifying programs and sub-systems, and their constituent parts and workings, to meet software product specifications.

Software engineering design demands knowledge at least about programming and the capabilities and organization of programming languages and systems. Often, more advanced knowledge is required in areas such as data structures and algorithms; design principles, practices, processes, and techniques; software architectures; and software design patterns. Sometimes designers also need in-depth knowledge of particular application areas, such as operating systems, database systems, or networking. Software engineering designers may be trained in one or more of these areas, and often specialize in some of them. Today, many products are so large and complex that specialists in all these areas need to be part of the design team.

In summary, the field of software design can be divided into two sub-fields that each demand considerable skill and expertise: software product design and software engineering design. Figure 1-2-2 depicts this division.

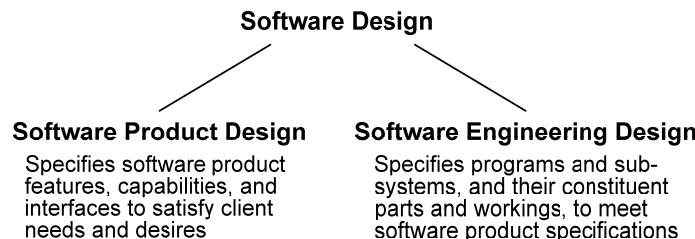


Figure 1-2-2 Software Design Sub-Fields

Software Design Teams

Some software products need to be functionally and aesthetically appealing but are not very technically sophisticated. Many Web sites have this characteristic, for example. Specialists in interaction design are often able to design such products without help from software engineers; much of the engineering is so routine that software tools can do it.

On the other hand, some software products may be technically sophisticated but well-understood products with simple or no user interfaces and only technical documentation needs. Many components of operating systems, database systems, Web servers, networks, and other large software systems are like this. Similarly, many development tools, such as compilers, development environments, and configuration management tools, have these characteristics. Software engineers without much expertise in software product design can specify such products without help from product designers.

But many modern commercial software products are technically sophisticated programs that must compete in the marketplace based on their functions, ease of use, supporting materials, and visual appeal. Such products require both highly polished product designs and complex and sophisticated engineering designs. Design teams whose members have both product design and engineering design expertise are clearly needed.

Focus on Software Engineering Design

Both software product design and software engineering design are essential parts of software design. Both activities require well-developed skills and knowledge in specialized areas. This book provides a survey of software product design, but it focuses mainly on developing skills and imparting knowledge in software engineering design.

Section Summary

- **Product design** is a discipline that arose during the Industrial Revolution and is now an established field whose practitioners specify products.
- The major issues in product design are aesthetics, product features and capabilities, usability, manageability, manufacturability, and operability.
- **Engineering design** is the activity of specifying the technical mechanisms and workings of a product. Engineers apply mathematical and scientific principles and techniques to work out the technical details of complex products.
- Product designers and engineers often work together in design teams to specify large and complex products.
- **Software product design** is a kind of product design: It is the activity of specifying software product features, capabilities, and interfaces to satisfy client needs and desires.
- **Software engineering design** is a kind of engineering design: It is the activity of specifying programs and sub-systems, and their constituent parts and workings, to meet software product specifications.

**Review
Quiz 1.2**

1. Name two sub-fields of design.
 2. Compare and contrast the roles of product designers and engineers in product design.
 3. Make a list of some specific concerns of software product and software engineering designers.
-

1.3 Software Design in the Life Cycle

The Software Life Cycle

The **software life cycle** is the sequence of activities through which a software product passes from initial conception through retirement from service. Every product must be subject to certain activities, whether the development team members are aware of it or not: Every product is specified, designed, coded, tested, deployed, and, if successful, maintained. Team members may be more or less conscious of moving their products through these activities. For example, there may be no conscious attention paid to setting requirements, but they still exist and they still drive the development of the product.

Software design is an essential life cycle activity, so we conclude that every software product is designed. Some products are designed almost unconsciously, while writing code in the midst of a completely undisciplined development process. Some are designed purposefully and carefully, using accepted and proven principles, techniques, heuristics, and patterns, and the design is documented using accepted design notations. Naturally, products designed in the latter way tend to make clients happier and are more maintainable than products designed in the former way. If a product is designed anyway, why not design it right?

The logical sequence of activities through which products pass is represented in the traditional waterfall life cycle model displayed in Figure 1-3-1.

Although this sequence can be followed temporally, in practice it rarely is. Most current products are developed incrementally, with portions being designed, coded, tested, and sometimes even deployed before other parts are developed. For example, in the Rational Unified Process, a widely used software development process, requirements specification, design, implementation, and testing are called *workflows*. These workflows all occur, to varying degrees, in every one of the many iterations that take a product gradually from initial conception through user testing.

Nevertheless, when a portion of a product is developed, requirements specification for that portion must come first, followed by design, coding, and testing activities. Thus, the waterfall model still captures the logical relationship among life cycle activities, even if their temporal sequence is more complex.

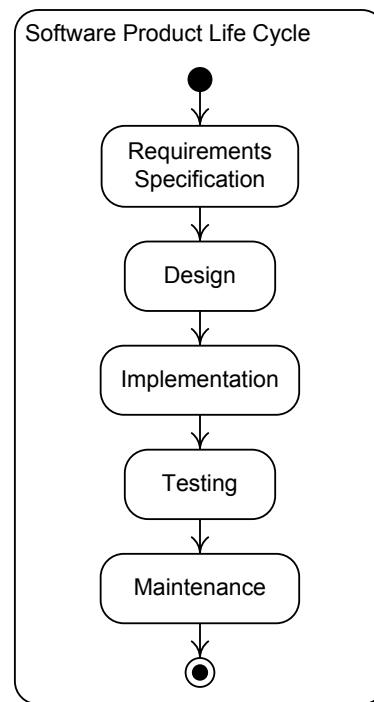


Figure 1-3-1 The Waterfall Life Cycle Model

Requirements Specification Activity

A **software (product) requirement** is a statement that a software product must have a certain feature, function, capability, or property. Requirements are captured in **specifications**, which are simply statements that must be true of a product. The requirements specification activity is the period during product development when product requirements are determined and documented.

This activity is aimed at finding out from the product's intended clients, and other interested parties, what they need and want from a software product. These needs and desires are translated into specifications of the functions, capabilities, appearance, behavior, and other characteristics of the software product. These specifications constitute the software product requirements, and they are recorded in a **software requirements specification (SRS)** document.

Factors that limit the range of design solutions, such as cost, time, size, user capability, and required technology, are called design **constraints**. Design constraints are usually given as part of the problem specification (see Chapter 3), though they may show up in the SRS, contracts for work, or various management documents.

The SRS can be regarded as a detailed statement of the problem that the programmers must solve—their task is to build a product that meets the requirements recorded in the SRS.

Design Activity	<p>During the design activity, developers figure out how to build the product specified in the SRS. This includes selecting an overall program structure, specifying major parts and sub-systems and their interactions, then determining how each part or sub-system will be built. The constituents of sub-systems and their interactions are specified, and the internals of these constituents are determined, sometimes including the data structures and algorithms they will use.</p> <p>The result of the design activity is a design document recording the entire design specification. The design document solves the (engineering) design problem posed in the SRS. Design documents are discussed in Chapter 11.</p>
Implementation Activity	<p>Code is written in accord with the specifications in the design document. The product of the implementation activity is a more or less finished, working program satisfying the SRS.</p> <p>Programming essentially includes some engineering design work. Even a very explicit design specification leaves many decisions to programmers. For example, most designs do not wholly specify functional decomposition, variable data types, data structures, and algorithms. These details constitute small design problems typically left for programmers to solve. Thus, although implementation is mainly a product realization activity, some engineering design work occurs as well.</p>
Testing Activity	<p>Unfortunately, no non-trivial program is fault free. Testing is needed to root out bugs before a product is given to clients. Programs produced during implementation are executed to find faults during the testing activity. Testing is usually done bottom up, with small parts or program units tested alone, and then integrated collections of program units tested as separate sub-systems, and finally the entire program tested as a whole. The output of the testing activity is a finished, working program whose worst faults have been found and removed. The program is now ready for distribution to clients.</p> <p>No design occurs during this activity. Specifications may change when bugs are fixed, but we can regard this as an extension of the requirements specification or design activities.</p>
Maintenance Activity	<p>Maintenance activity occurs after a product has been deployed to clients. The main maintenance tasks are enhancing the program with new functions, capabilities, user interfaces, and so forth; adapting the program for new environments; fixing bugs, and improving the design and implementation to make it easier to maintain in the future.</p> <p>Maintenance activity recapitulates development; requirements specification is redone, followed by redesign, recoding, and retesting activities. The design tasks done in initial development recur during maintenance.</p>

Design Across the Life Cycle

It should be obvious from the previous discussion that the major task of the requirements specification activity is software product design. User interaction design, which is sometimes not considered part of requirements specification, occurs early in the life cycle as well and is also a software product design activity. We consider software product design to be part of requirements specification, so we hereafter assume that it includes user interaction design.

The design activity is where the bulk of software engineering design occurs. Some minor engineering design decisions are left for the implementation activity. The maintenance activity includes both product redesign and engineering redesign.

Figure 1-3-2 illustrates how software design activities are spread across the life cycle.

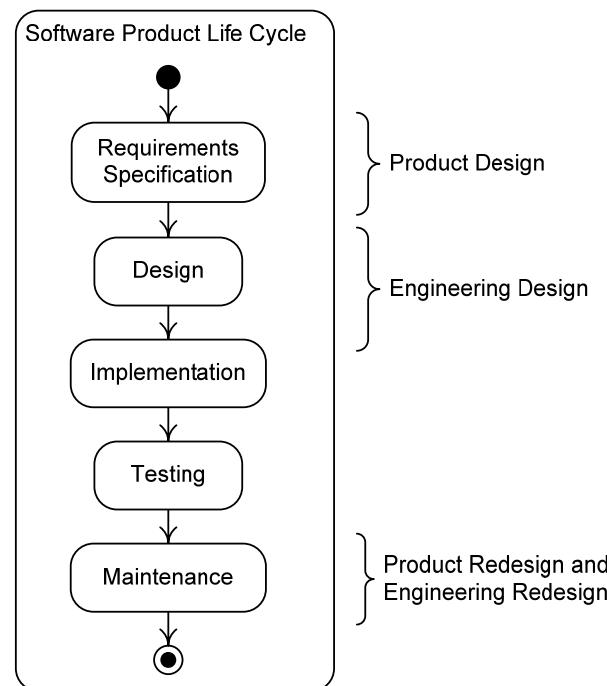


Figure 1-3-2 Design Across the Life Cycle

This book covers the main non-managerial tasks of the first two activities of the waterfall life cycle, as well as tasks relevant to the implementation and maintenance activities. It therefore covers much of the non-managerial work of the life cycle.

The "What" Versus "How" Distinction

The traditional way to distinguish requirements specification and design activities is to say that during the requirements specification developers specify *what* a product is supposed to do, while during design developers

specify *how* the product is supposed to do it. Although this is very simple, making the distinction in this way does not hold up under careful scrutiny. This criterion breaks down because plenty of “what” and “how” decisions are made within both the requirements specification and design activities. For example, a requirement may state that a program must store certain data (*what* the product must do), while another requirement states that the product must store its data in a relational database (*how* the product must do it). Likewise, during the design activity, engineers may decide that a product implemented over a network should encrypt communications to achieve adequate security (*what* the product should do) and also what encryption mechanisms should be used (*how* the product should do it).

No matter what we come up with as an example of something that is about *how* a program is supposed to work, it is easy to state a scenario in which clients want the product built that way, making this a specification of a product requirement. The “what” versus “how” distinction is thus not helpful in distinguishing the requirements specification and design activities.

The right way to distinguish the requirements specification and design activities is in terms of problems and solutions. The requirements specification activity formulates the software engineering design problem whose solution is specified in the design activity.

Software Design Problems and Solutions

One might make the same observation about problems and solutions as a way of distinguishing requirements specification and design as was made about the “what” versus “how” distinction: Problems and solutions appear during both activities. This shows that there are smaller design activities within overarching design efforts. This point has consequences for the design process and will come up again.

Let’s illustrate design problems and solutions with an example, the AquaLush Irrigation System. This example is one we will use throughout the book, and the entire example is presented in Appendix B.

Cumulative Case Study: AquaLush

MacDougal Electronic Sensor Corporation (MESC), an electronic sensor manufacturer, has decided to start a company to exploit a newly perfected soil moisture sensor. The new company, Verdant Irrigation Systems (VIS), will develop and market lawn and garden irrigation products. Typical irrigation products use timers to regulate irrigation, releasing water for a fixed period on a regular basis. This over-waters if the soil is already wet and under-waters if the soil is very dry. VIS products will use the new soil moisture sensors to control irrigation. The first VIS product is the AquaLush Irrigation System. It is a moisture-controlled irrigation system targeted at residential and small commercial clients. A small team with expertise in irrigation products, hardware, and software will develop AquaLush.

Problems and Solutions in Developing AquaLush

The first tasks for the development team are to understand this problem and specify a product to solve it. In other words, the first job is product design, an important part of which is software product design. An obvious starting point is a detailed list of required software product features and capabilities. Such specifications might state, for example, that the AquaLush software must keep track of irrigation zones with particular moisture sensors and irrigation valves, that it must read moisture levels from moisture sensors in each zone, that it must read a clock indicating the time of day, that at certain times of the day it must turn sprinkler valves on and off in each zone based on moisture levels in that zone, and that it must detect and react to sensor and valve failures.

Once feature and capability specifications are set, one software product design sub-problem is solved, and it now becomes part of another software product design sub-problem. Interactions between the program and its environment, and in particular between the program and its users, must be designed. User interactions must satisfy various needs and desires.

AquaLush user interactions, for example, must allow consumers to set all parameters controlling irrigation, but be simple enough for them to do so infrequently and with no training. When this second product design sub-problem is solved, the result is a complete solution to the software product design problem in the form of a software requirements specification.

Figure 1-3-3 illustrates how these problems and solutions fit together in software product design.

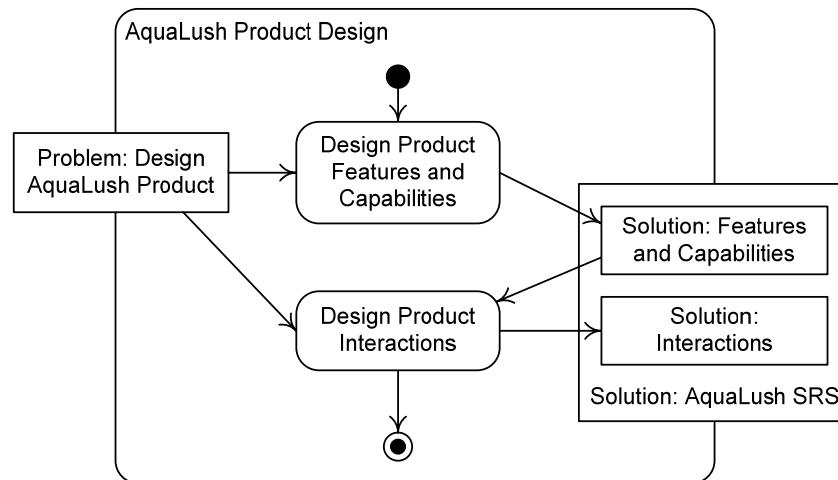


Figure 1-3-3 AquaLush Product Design Problems and Solutions

The product design in the SRS yields an engineering design problem, which is to specify software to realize the product. Taking a top-down approach, the first engineering sub-problem is to determine the major constituents of this program, and their properties and interactions. A three-part structure seems appropriate for AquaLush: one part interfaces with the user, one

part interfaces with the hardware (sensors, valves, clock, display, keyboard, and so forth), and one part controls irrigation. Specifying this high-level design solves the initial engineering design sub-problem.

The next engineering design sub-problem is given by the SRS plus the high-level design: It is to create a low-level design that specifies the internal details of the parts given in the high-level design. Solving this problem might involve finding out whether some reusable parts are available, dividing each program into parts, and specifying each part's functions and interfaces. For example, some AquaLush hardware interfaces might be provided by the operating system or by device drivers that come with the hardware, so these can be reused. The design may specify details down to the level of data structures and algorithms. For example, it might specify the details of the irrigation control algorithm, which is the heart of the product.

When all this design work is done, the low-level engineering design sub-problem is solved, and the result is a solution of the entire engineering design problem in the form of a complete design document. Figure 1-3-4 shows the problems and solutions in the engineering design portion of this process.

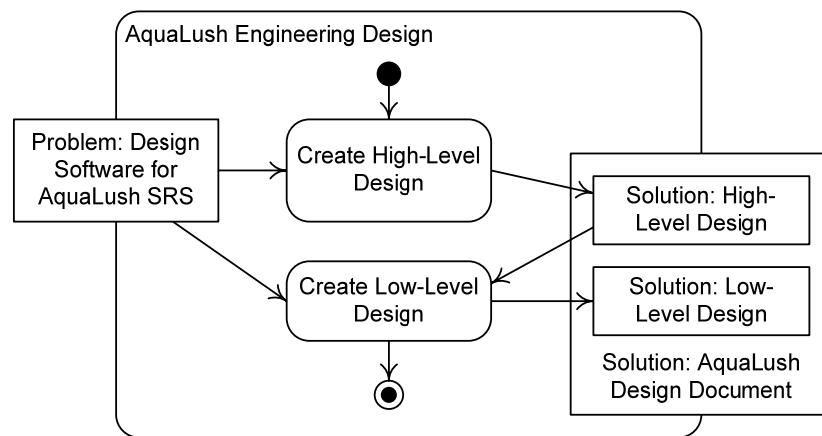


Figure 1-3-4 AquaLush Engineering Design Problems and Solutions

Note that the starting problem for engineering design is the solution produced from product design. Similarly, the design document produced as a solution from engineering design poses the problem solved in the implementation activity, which is to code the design. For example, programmers must figure out how to code the AquaLush user interface, device interfaces, and irrigation control algorithm. In doing this they will likely have to make design decisions about how to decompose functions, declare variables, and so forth. These solutions are embodied in the code for the program.

Figure 1-3-5 puts all this together in the general case, illustrating how design solutions form new design problems and how a larger design activity is divided into sub-activities over the course of software design during the first several activities of the life cycle.

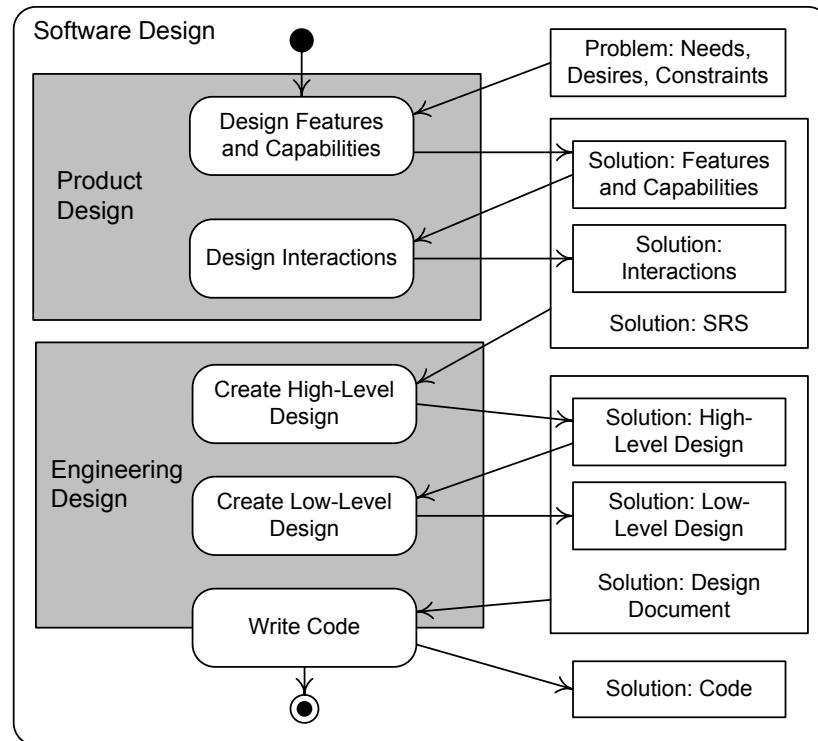


Figure 1-3-5 Software Design Problems and Solutions

"Design" as a Verb and a Noun

In the first section of this chapter we characterized design as an activity. This activity is what we refer to when we use the word “design” as a verb, as in the sentence “Engineers design programs meeting requirements specifications.” But we have also used “design” as a noun, as in the sentence “Engineers develop a design meeting requirements specifications.” Obviously, the word “design” is both a verb and a noun and refers to both an activity and a thing. A design specification is the output of the design activity and should meet the goals of the design activity—it should specify a program satisfying client needs and desires, subject to constraints.

Remember that a design is a specification of a program rather than the program itself. In general, software designs are not executable themselves, but are descriptions of executable things. Also note that software design specifications are ultimately (though not exclusively) for programmers—a programmer must be able to tell from a design what code to write, how to interface it with other code or devices, and so forth. Programmers must

also be able to tell what design decisions are *not* specified in the design, so they can make these decisions for themselves.

Section Summary

- The **software life cycle** is the sequence of activities through which a software product passes from initial conception through retirement from service.
- The goal of the *requirements specification* activity is to specify a product satisfying the needs and desires of clients and other interested parties. Product **specifications** are recorded in a **software requirements specification (SRS)**.
- In the remainder of the book, we assume that every SRS includes a user interface design.
- During the design activity developers determine how to build the product specified in the SRS and record their results in a **design document**.
- In the implementation activity programs are written in accord with the specifications in the design document.
- Programs are run during the testing activity to find bugs.
- After deployment to clients, products are corrected, ported, and enhanced during maintenance activities.
- Product design occurs during the requirements specification and maintenance activities, and engineering design occurs during the design, implementation, and maintenance activities.
- The “what” versus “how” distinction is the claim that requirements specify what a product is supposed to do and design specifies how a product is supposed to do it. It does not suffice to distinguish the requirements specification and design activities.
- Requirements specify an engineering design problem that is solved during the design activity. This is the essential distinction between these activities.
- Problems and solutions demarcate various software design activities. Product design tackles a client problem and produces a product specification as a solution. This solution presents the problem to engineering designers, who produce a design document as their solution.

Review Quiz 1.3

1. Characterize design tasks in each activity of the software life cycle.
2. How would you characterize the differences between the requirements specification and design activities of the software life cycle?
3. Comment on the sentence “They must design a design.”

1.4 Software Engineering Design Methods

What Is a Design Method?

A **software design method** is an orderly procedure for generating a precise and complete software design solution that meets client needs and constraints. A method typically specifies the following items:

Design Process—A **process** is a collection of related tasks that transforms a set of inputs into a set of outputs. Most methods specify quite detailed instructions about each step of a design process, what inputs are

required, and what the intermediate and final outputs should be. For example, a design method might specify a process in which classes and their attributes are identified, then interactions between classes are studied to derive class operations, then the class inheritance hierarchy is modified in light of the class operations, and so on.

Design Notations—A **notation** is a symbolic representational system. Most design methods specify particular design notations and how and where they should be used in the design process. The Unified Modeling Language (UML) is a collection of object-oriented design notations that can be used with almost any object-oriented design method. We will begin studying UML, along with several other popular design notations, in the next chapter.

Design Heuristics—A **heuristic** is a rule providing guidance, but no guarantee, for achieving some end. A design method's heuristics generally provide advice about following its process and using its notations. An example of a process heuristic is, “Do static modeling before dynamic modeling.” An example of a notation heuristic is, “Make sure that every state diagram has exactly one initial pseudo-state.”

Design methods generally adhere to universal **design principles** or **tenets**, which are fundamental statements about the characteristics of designs that make them better or worse. For example, one basic design principle is that designs with small modules are better than designs with larger modules. We discuss design principles in detail in Chapter 8. Although design principles are universal, methods may emphasize certain principles or provide heuristics for resolving conflicts among principles.

A design method provides detailed guidance for part or all of a design effort. Methods can be thought of as recipes for design—ideally, a developer can follow a method to produce a design as easily as a baker can follow a cookie recipe to make a batch of cookies. Unfortunately, design is usually somewhat more difficult than baking cookies, so using a method is often not as easy as one might hope.

History of Software Engineering Design Methods

Niklaus Wirth described the first software engineering design method in 1971. Wirth’s method, called **stepwise refinement**, is a top-down approach that repeatedly decomposes procedures into smaller procedures until programming-level operations are reached. Stepwise refinement is an elementary method consisting of a simple process, no notation, and few heuristics. Although quickly superseded by more powerful methods, it persists to this day as a fundamental technique for low-level procedural design.

Stevens, Myers, and Constantine introduced **structured design** in 1974. This work introduced the notations and heuristics at the foundation of the structured methods that dominated software engineering design literature and practice for two decades. A series of books in the 1970s standardized and popularized this approach. Dozens of slightly different methods

refining the basic structured design approach appeared in the late 1970s through the 1980s.

Structured design, like stepwise refinement, emphasizes procedural decomposition. The central problem specification model is the data flow diagram, which represents procedures by bubbles and the data flowing into and out of them by arrows. Bubbles can be decomposed into lower-level data flow diagrams, allowing procedural decomposition. The main solution specification model is the structure chart, which shows the procedure-calling hierarchy and the flow of data into and out of procedures via parameters and return values.

Later versions of structured design methods include more detailed and flexible processes, many sophisticated and specialized notations, many heuristics, proven effectiveness in actual use, and wide support by Computer Aided Software Engineering (CASE) tools. Nevertheless, dissatisfaction with structured design grew by the late 1980s. Certain difficulties with structured design processes were never overcome. In particular, the transition from problem models to solution models was an issue. Furthermore, these methods did not seem able to handle larger and more complex products. Most importantly, structured design is completely unsuited for designing object-oriented systems, which were becoming popular in the late 1980s.

Object-oriented design methods began to appear in the late 1980s, and an onslaught of these methods appeared in the 1990s. More than 50 object-oriented design methods had been proposed by 1994.

Object-oriented design methods reject traditional procedural decomposition in favor of class and object decomposition. In the object-oriented approach, programs are thought of as collections of objects that work together by providing services to one another and communicate by exchanging messages. The central models in object-oriented design are class diagrams that show the classes comprising a program and their relationships to one another. Many of the modeling tools developed for structured methods have been incorporated into object-oriented methods, and the best features of the old methods have been retained in the new methods.

Today, structured design is still used but is being displaced rapidly by object-oriented methods. No object-oriented design method dominates, but the software engineering community has reached consensus on UML as the standard object-oriented design notation. It may be that this agreement is the first step toward consensus on a common object-oriented design method.

Method Neutrality

There are hundreds of design methods, but only one generic design process, about a dozen basic design principles, and perhaps a score of widely used design notations exist. A good grounding in the software design process, design principles, and common design notations is preparation for whatever design method an engineer may encounter. On

the other hand, it is clear that the structured design methods are not nearly as good as object-oriented design methods. Consequently, this book is method neutral, but strongly emphasizes object-oriented notations, heuristics, and models.

Most of the notations used in this book are UML notations, but some other important notations are included as well.

The heuristics discussed in this book are either notation heuristics or design task heuristics. A **design task** is a small job done in the design process, such as choosing classes or operations, or checking whether a model is complete. Notation and task heuristics are discussed throughout the book when notations and design tasks are introduced.

Section Summary

- A **software design method** is an orderly procedure for generating a precise and complete software design solution that meets client needs and constraints.
- A design method typically specifies a design **process**, design **notations**, and design **heuristics**.
- The first design method was **stepwise refinement**, a top-down technique for decomposing procedures into simpler procedures until programming-level operations are reached.
- The dominant design methods from the mid-1970s through the early 1990s were various versions of **structured design**.
- Structured design methods focus on procedural composition but include other sorts of models as well.
- Object-oriented design methods emerged in the 1990s in response to shortcomings of structured design methods.
- Object-oriented methods promote thinking about programs as collections of collaborating objects rather than in terms of procedural decomposition.

Review Quiz 1.4

1. Distinguish design methods, design processes, design notations, design heuristics, and design techniques.
2. For each decade from the 1960s through the 1990s, indicate the dominant design method of the time.
3. What is the essential difference between structured design methods and object-oriented design methods?

Chapter 1 Further Reading

Section 1.1 Other introductions to software design include books by Budgen [1994], Braude [2004], Larman [2005], and Stevens [1991]. An interesting book about problem solving is [Polya 1971]. The data about mechanical engineering design problems is from [Ullman 1992].

Section 1.2 [Hauffe 1996] is a short history and [Heskett 2002] a short introduction to the professional field of design. [Ertas and Jones 1996] provide an in-depth study of the engineering design process.

Readers interested in more detailed discussions of software product design are advised to consult the rich literature in this area. Armitage [2003] discusses product design for digital products and shows how design sub-disciplines fit. Extended discussions of material relevant to product design are found in [Nielsen 1994], [Norman 1990], [Preece et al. 1994], [Preece et al. 2002], [Schneiderman 1997], [Tufte 1983], [Williams 1994], [Williams 2000], and [Winograd 1996].

Section 1.3 The software life cycle is covered in depth in most software engineering textbooks, including [Pressman 2001] and [Sommerville 2000]. The Rational Unified Process is presented in [Jacobson et al. 1999]. Davis [1993] gives a more detailed example to illustrate the inadequacy of the “what” vs. “how” distinction.

Section 1.4 Budgen [1994] discusses methods as specifications of processes, notations, and heuristics. Stepwise refinement is introduced in [Wirth 1971]. Structured design is introduced in [Stevens et al. 1974] and elaborated in [Meyers 1978], [Orr 1977], and [Yourdon and Constantine 1979]. The best recent reference for structured analysis is [Yourdon 1989].

Seminal object-oriented design methods are documented in [Meyer 1988], [Shlaer and Mellor 1988], [Booch 1991], [Coad and Yourdon 1991], [Rumbaugh et al. 1991], and [Coleman et al. 1994]. UML is presented in [Booch et al. 2005] and [Rumbaugh et al. 2004].

Chapter 1 Exercises

The following problems are used in the exercises below.

Chicken Coop You have 60 feet of chain link fence that you want to use to enclose a patch of your five-acre yard around a chicken coop housing your prize chickens. You want to enclose as much area as possible around the coop to give the chickens lots of room. Design an enclosure for the chickens.

Sheep and Wolves Sheila the shepherd is conveying two sheep and a wolf across a river. Her boat will hold only herself and one animal. The wolf cannot be left alone with a sheep. How can Sheila get herself and the animals over the river?

Section 1.1

1. Go inside and face away from any windows. Can you name something within sight (besides yourself) that has *not* been designed by a human being?
2. Designs embody solutions to problems. Pick two competing brands of some consumer product and state the design problem that these products solve.
3. Give an example of how a good or bad product design directly affected your life for better or worse.
4. Give an example of how a good or bad software product design directly affected your life for better or worse.

5. In solving the Chicken Coop and Sheep and Wolves problems above, which details of the problem did you ignore and which did you not ignore?
6. What main solution alternatives did you consider in solving the Chicken Coop and Sheep and Wolves problems?
7. What models did you use in solving the Chicken Coop and Sheep and Wolves problems?
8. What are your solutions to the Chicken Coop and Sheep and Wolves problems?
9. [Booch et al. 1999] defines a model as “a simplification of reality”. Analyze this definition in light of the discussion of models in this chapter.

- Section 1.2**
10. Classify each of the following tasks as a software product design activity, a software engineering design activity, or possibly both, depending on the circumstances:
 - (a) Determining the layout of buttons, labels, text boxes, and so forth in a window
 - (b) Brainstorming the classes in an application
 - (c) Choosing colors for a window
 - (d) Wording error messages
 - (e) Deciding whether a product should have a client-server architecture
 - (f) Deciding whether a program should be a stand-alone application or a Web service
 - (g) Choosing the data structure for a list
 - (h) Determining the reliability that a product should have
 - (i) Determining the sequence of processing in a program
 - (j) Specifying the states that a program may be in, and how it changes state in response to inputs
 11. Give a specific example of a program where the major design challenge is product design rather than engineering design.
 12. Give a specific example of a program where the major design challenge is engineering design rather than product design.
 13. Give a specific example of a program where both the product design and engineering design are challenging.
- Section 1.3**
14. Consider the following statement of the “what” versus “how” distinction: A requirements specification states *what* the clients need and desire, while a design states *how* the program will satisfy these needs and desires. Does this statement of the “what” versus “how” distinction escape the criticisms made in the text? Explain why or why not.
 15. Can you tell by looking at a single specification statement in isolation whether it is a requirement or (engineering) design specification? How, or why not?

16. Make a diagram showing the inputs and outputs of each activity of the software life cycle.

- Section 1.4**
17. Give an example of a heuristic.
 18. Given an example of a notation.
 19. Apply stepwise refinement to create a program to read a text file and print a report of the number of characters, words, and lines in the file. State your steps in English.
 20. *Find the error:* What is wrong with the following statement? Structured design is a top-down method for decomposing complex functions into simpler functions that was developed to help design and implement operations in classes.

- Team Project**
21. Form a team of at least three. Consider the things you have to think about and the decisions you have to make when writing a program, and come up with notations that can help you do this job. In other words, think up your own software design notations. Develop at least one notation for making static models and one for dynamic models.

- Research Projects**
22. Consult works about problem solving and make a list of problem-solving techniques applicable in software design.
 23. Consulting whatever resources you need, define and write a short description of each of the following terms used in the object-oriented programming paradigm. The remainder of the book assumes that you are familiar with these terms:
 - (a) Class and object
 - (b) Attribute, operation, and method
 - (c) Constructor and finalizer
 - (d) Single and multiple inheritance
 - (e) Polymorphism
 - (f) Overriding methods
 - (g) Abstract and concrete classes and methods

Chapter 1 Review Quiz Answers

Review Quiz 1.1

1. Software design is important because software is important. Software is becoming increasingly ubiquitous and essential in the day-to-day functioning of our society. The crucial importance of software means that its design is likewise crucially important.
2. Common examples of products with software in them that are not themselves software products are consumer electronic devices, such as cell phones, music players, DVD players, televisions, digital cameras, and so forth.
3. Thinking of design as problem solving has the following advantages:
 - (a) It focuses attention on the distinction between the problem and the solution, helping ensure that the design problem is understood before an attempt is made to solve it.
 - (b) It reminds us that there are many solutions to most problems, and likewise many good designs satisfying client needs.

- (c) It suggests that we may use well-known problem-solving methods, such as changing the problem, problem decomposition, trial and error, and brainstorming, in design.
4. Abstraction is an important problem-solving technique because it (a) allows focus on only those problem details relevant to a solution, and (b) helps structure the problem-solving process in a top-down fashion.
 5. A model has the benefit that it is a simplification of the target that it represents, making it easier to understand, document, and investigate than the target. A model may also be much cheaper and safer to investigate than the target. However, a model never exactly replicates its target, so conclusions drawn from studying a model may not be correct.

Review Quiz 1.2

1. Product design includes as sub-fields communications, graphic design, and industrial design. Particular application areas, such as furniture design, ceramic design, or fashion design, are also sometimes regarded as distinct fields.
2. Product designers and engineers are both concerned with satisfying client needs and desires. Product designers focus on the “external” features of a product, such as its visual form, capabilities, functions, interface with users, operability, manageability, and role as a symbol. Engineers focus on the “internal” mechanisms that enable a complex product to work. Ultimately, perhaps the biggest differences between product designers and engineers are their preferences and training. Product designers tend to be interested in aesthetics and people’s interactions with their environment, and design training emphasizes these things. Engineers tend to be interested in science and technology, and engineering education certainly emphasizes these things. Thus, although product designers and engineers often face the same problems, with the same criteria for success, their approach to design problem solving, and the designs that result, are very different.
3. The following paragraphs list specific concerns of software product and software engineering designers.

Some specific software product design concerns are screen layout, including the placement of user interface components, their size, font, color, and look and feel; the sequence of interaction between users and the program; the wording of messages; the use and appearance of images; the functions that the program will perform; the inputs that the program will require and the outputs that it will produce; the quality and reliability of the program; the ease with which users can accomplish their goals using the program; and the way that the program fits into and changes the way users do their work.

Some specific software engineering design concerns are the major parts or subsystems comprising the program, including their functions, interfaces, interactions, speed, reliability, size, and so forth; the modules in a program and their interfaces; the interactions between modules in a program; the internal details of modules, including their data and behavior; the data structure and algorithms used in a program; the exception- and error-handling mechanisms in a program; the processes and threads in a program; and the way program components are deployed in a network and how they communicate.

Review Quiz 1.3

1. The type of design done during the requirements specification activity of the software life cycle is product design. Developers must understand client needs and desires and come up with a specification of a product whose features and

functions, interactions with users and other systems and devices, and so forth meet these needs. Engineering design during the design activity of the software life cycle results in a specification for a program to realize the target product. Various low-level engineering design details left unspecified in the design document are determined during the implementation activity as programs are written. No design occurs during the testing activity. Both varieties of design occur during maintenance, as the effort to correct, port, and enhance a product inevitably requires redesign of the product and its implementation.

2. The goal of the requirements specification activity is to solve the problem of determining a product to meet client needs and desires. The goal of the design activity is to solve the problem of specifying a program to realize the product specified by the requirements. Though both encompass problem-solving activities, the requirements specification and design activities solve different problems. Furthermore, the requirements specification activity states a problem solved in the design activity.
3. In the sentence “They must design a design,” the word “design” is used as both a verb and a noun. As a verb, “design” refers to the activity of stipulating the characteristics of a product meeting client needs and desires. As a noun, it refers to the resulting specification. Thus, this sentence could be paraphrased as “They must produce a specification of a product meeting client needs and desires.”

**Review
Quiz 1.4**

1. A design method is an orderly procedure for generating a precise and complete design solution. A method must have some sort of design process, which is a sequence of steps for understanding a design problem, solving the problem, and documenting the solution. Generally this process will involve modeling and the use of special design notations, which are symbol systems for expressing designs. Most methods also provide advice, or rules of thumb, called design heuristics, for following the process and using the notation. A design technique is a way of doing small activities in a design process.
2. No software design method had yet been proposed in the 1960s. Various versions of structured design were the preferred design methods in the 1970s and 1980s. Object-oriented design methods began to be popular in the 1990s and were the dominant method by the turn of the century.
3. Structured design methods are based on procedural decomposition. Problems and solutions are framed in terms of the transformation of input values to output values, and the method essentially proceeds by figuring out how to break a complicated procedure into smaller and simpler steps. In contrast, object-oriented design methods frame problems and solutions in terms of the objects that comprise them. Objects hold data and exhibit behavior that can be called on by other objects as all objects cooperate to achieve some goal.

2 Software Design Processes and Management

Chapter Objectives	This chapter continues an introduction to software design. The first section introduces activity diagrams, a UML process specification notation. The second section describes generic processes for both software product design and software engineering design. The third section surveys design project management. This chapter continues the theme that design is a form of problem solving.
	By the end of this chapter you will be able to
	<ul style="list-style-type: none">▪ Read and write UML activity diagrams;▪ Explain how design consists of analysis and resolution activities;▪ Illustrate and explain generic processes for software product design and software engineering design;▪ List and explain the five main tasks of project management;▪ Show how iterative planning and tracking can be applied in software design; and▪ Explain how design can drive a software development project.

Chapter Contents	2.1 Specifying Processes with UML Activity Diagrams 2.2 Software Design Processes 2.3 Software Design Management
-------------------------	--

2.1 Specifying Processes with UML Activity Diagrams

Processes and Process Descriptions	A process is a collection of related tasks that transforms a set of inputs into a set of outputs. A design process is the core of any design endeavor, so it is essential that designers adopt an efficient and effective process. Processes are also specified by software engineering designers as part of their work.
---	---

We need a process description notation for our discussion of the design process and for specifying processes during design. This notation will document dynamic models of processes. Many such process description notations have been developed. Programming languages, for instance, are partly notations for describing computational processes. Although we will consider other algorithmic specification notations later, here we will introduce a UML process description notation called an activity diagram.

Activity Diagrams

An **activity diagram** describes an activity by showing the actions into which it is decomposed and the flow of control and data between them. An **activity** in UML is a non-atomic task or procedure decomposable into actions, where an **action** is a task or procedure that cannot be broken into parts, and hence is *atomic*. Ultimately, all activities must be decomposable into actions, so actions are the basic units of process activity. Actions and activities can be anything that humans, machines, organizations, or other entities do.

Actions and Execution

Activity diagrams model processes as an *activity graph*. An activity graph has *activity nodes* representing actions or objects and *activity edges* representing control or data flows. An *action node* is represented by an *action node symbol*, which is a rounded rectangle containing arbitrary text naming or describing some action. Activity edges are represented by solid arrows with unfilled arrow heads. Figure 2-1-1 shows several action nodes connected by activity edges.

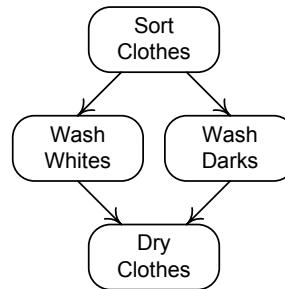


Figure 2-1-1 Action Nodes and Activity Edges

These actions are part of a laundry activity. This model indicates that after the clothes are sorted, they are washed, and then they are dried.

Although this example is very simple, it already raises questions of interpretation. For example, one might ask

- Can the washing of whites and darks occur at the same time?
- Can clothes drying begin before all the washing is done?

Activity diagrams employ a very precise mechanism for representing behavior so that questions like this can be answered. Activity diagrams can be regarded as specifications of virtual machines whose basic computations are given by the diagrams' action nodes. These virtual machines compute using *tokens*, which are markers (not represented on the diagram) passed around during execution. Tokens are produced and consumed by nodes and flow instantaneously along edges. An action node begins executing when tokens are available on all its incoming edges. When an action node begins execution, it consumes these tokens. When it completes execution, it produces new tokens and makes them available on all its outgoing edges.

The nodes and edges in Figure 2-1-1 partially describe a virtual machine. Suppose that the action node labeled **Sort Clothes** is executing. When it is done, it produces tokens and makes them available on the two edges leaving it. Both the action nodes **Wash Whites** and **Wash Darks** now have tokens available on all the edges entering them, so they consume the nodes flowing over the edges from **Sort Clothes** and begin executing. Eventually each finishes executing, produces a token, and makes the token available on the edge leaving it. When both tokens are available, **Dry Clothes** can consume the tokens and begin executing.

We see now that the answers to our questions are

- Washing whites and darks can occur at the same time.
- Drying clothes can occur only after both whites and darks are washed.

Two more things are needed to produce a complete activity graph: A way to start execution and a way to stop it. A special *initial node*, represented by a small filled circle, can have only a single outgoing edge. It produces a token and makes it available when an activity begins. Execution stops when tokens can no longer flow through the diagram, but the end of an activity can also be indicated explicitly (or forced) using an *activity final node*, represented by a small filled circle within another circle (a bull's eye). Edges can only enter an activity final node. When a token is available on one of the nodes entering it, an activity final node consumes the token and terminates the activity (by removing all tokens from the virtual machine).

Activities	Activities are represented by rounded rectangles called <i>activity symbols</i> . An activity symbol contains an activity graph along with the name of the activity at the upper left. Figure 2-1-2 shows a complete activity diagram.
------------	--

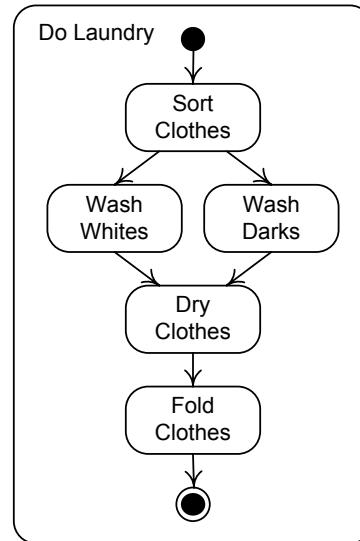


Figure 2-1-2 A Laundry Process

The diagram in Figure 2-1-2 shows an activity symbol containing an activity graph with initial and activity final nodes. When the **Do Laundry** activity is begun, a token is produced at the initial node and made available on the edge to **Sort Clothes**. The **Sort Clothes** action node consumes the token and begins executing. Execution continues as discussed before, until **Dry Clothes** finishes executing. It then produces a token and makes it available on its outgoing edge. **Fold Clothes** consumes this token, executes, and produces a token that it makes available on the edge into the activity final node. The activity final node consumes this token and terminates the activity.

Branching

Few interesting processes have no alternative or repetitive flows, so activity diagrams provide for flow branching. Branching is based on the value of Boolean (true or false) expressions. A Boolean expression placed inside square brackets and used to label a control flow is called a **guard**. UML does not specify the notation used between the square brackets of guards, except to say that the guard **[else]** may be used to label one of the alternative control flows at a branch point.

Flow branching is shown on an activity diagram using a *decision node* represented by a diamond. A decision node can have only a single edge entering it but several edges leaving it, each labeled with a guard. Branched flows can rejoin at a *merge node*, also represented by a diamond, but with several edges entering it and only a single edge leaving it. Decision and merge nodes are shown in Figure 2-1-3 realizing a loop.

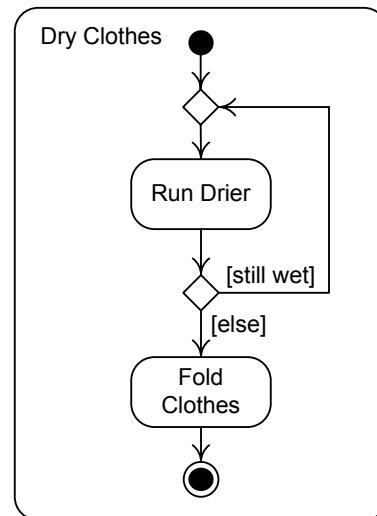


Figure 2-1-3 Drying Clothes in a Loop

In this example the drier runs for a while and stops, and then the clothes are checked. If they are still wet, the drier runs some more. If the clothes are dry when checked, they are folded and the process ends.

Decision and merge node behavior is explained in terms of token flow. If a token becomes available on any of the edges entering a merge node, it immediately becomes available to the edge leaving the node. If a token becomes available on the edge entering a decision node, its guards are evaluated, and the token is made available on the edge whose guard is true. If no guard or more than one guard is true, the activity diagram is ill-formed and its behavior is undefined. Decision and merge nodes pass tokens along without consuming or producing them.

In the diagram in Figure 2-1-3, when a token is produced and made available by the initial node at the start of the activity, the merge node passes it along to the **Run Drier** action node, which consumes the token and begins execution. When this node completes execution it produces a token and makes it available on the edge leading to the decision node. At this point the guards are evaluated. If the clothes are still wet, the decision node makes the token available on the edge leading back to the merge node, which makes it available to **Run Drier**, which consumes it and executes again. On the other hand, if the clothes are dry when checked, the decision node makes the token from **Run Drier** available on the edge leading to **Fold Clothes**, which consumes it and begins execution.

Note that the merge node is needed to make this activity work. Consider the alternative diagram in Figure 2-1-4.

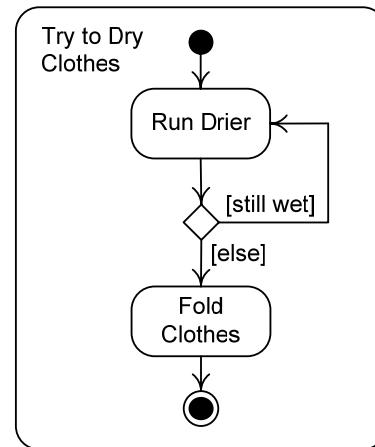


Figure 2-1-4 A Deadlocked Activity

When **Try to Dry Clothes** begins, a token is produced by the initial node and made available on the edge leading to **Run Drier**. But **Run Drier** cannot execute until a token is made available on the other edge leading into it as well, the one coming from the decision node. Such a token will not be produced until **Run Drier** completes execution. Hence, **Run Drier** cannot begin execution until it completes execution. This activity is deadlocked and cannot continue.

Forking and Joining

The activity diagram execution model already provides for concurrent execution and implicit synchronization at action nodes by virtue of the token-based execution model. Two additional nodes provide more control over concurrent execution and synchronization. A *fork node*, represented by a thick line, has one edge entering it and multiple edges leaving it. A token available on the entering edge is reproduced and made available on all the outgoing edges. This effectively starts several concurrent threads of execution. A *join node*, also represented by a thick line, has multiple edges entering it, but only one edge leaving it. Tokens must be available on all entering edges before a token is made available on the outgoing edge. A join node synchronizes concurrent threads of execution.

A simple example is the laundry process from Figure 2-1-2 that is revised with fork and join nodes in Figure 2-1-5.

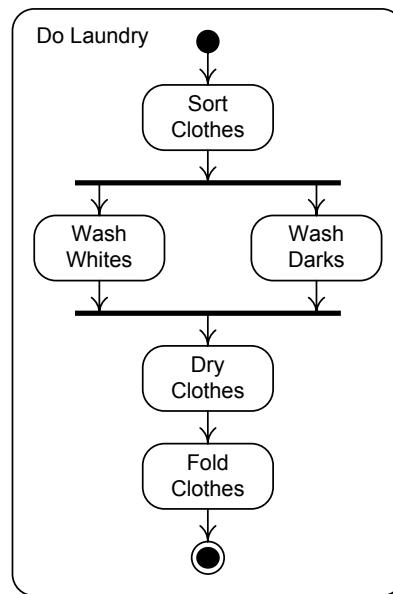


Figure 2-1-5 Laundry Process with Forks and Joins

When the **Sort Clothes** process finishes, it produces a token and makes it available on the edge leading to the fork node. The fork node reproduces this token and makes it available on the edges leading to **Wash Whites** and **Wash Darks**, which consume these tokens and begin execution. When they are done, they each produce tokens and make them available on the edges leading to the join node. When tokens are available on both edges entering it, the join node consolidates them and makes a token available on the edge leading to the **Dry Clothes** node, which then consumes it and begins execution.

Another node used to manage concurrent execution is the flow final node. Recall that an activity final node halts all activity execution. A flow final

node halts only the activity in an execution path. A *flow final node*, represented by a circled X, can have one or more edges entering it but no edges leaving it. When a token is available on an entering edge, it consumes the token. To illustrate, consider the activity diagram in Figure 2-1-6.

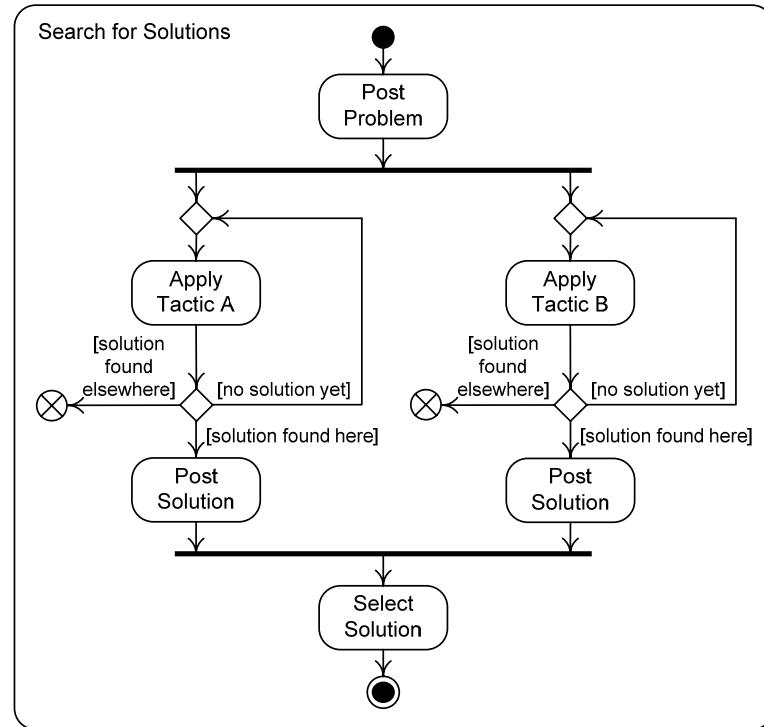


Figure 2-1-6 Concurrent Problem Solution

This diagram illustrates searching for problem solutions using two tactics concurrently. The problem is posted in a central repository where solutions appear as well. Problem-solving tactics A and B are applied concurrently. Every so often, or when a solution is found, problem solving leaves off and the repository is checked. If no solution is posted and application of a tactic has not found one, then problem solving goes on. If application of the tactic has yielded a solution, then it is posted. Several solutions may be posted because of concurrent problem solving. Execution is synchronized by the join node before the solutions are evaluated and the best one selected.

Flow final nodes come into play when a concurrent problem-solving activity fails to find a solution, but one has been posted. In this case the application of a tactic should simply be abandoned. This is done by sending a token into a flow final node, which halts that flow of concurrent execution.

Data Flows Control is not the only thing that flows through a process: Data does too. Data and objects are shown in activity diagrams as *object nodes* represented by rectangles containing data or object type names. An object node rectangle may also specify the state of the data or object in square brackets following the name. The state description can be arbitrary text. Figure 2-1-7 illustrates object nodes from a war game process.

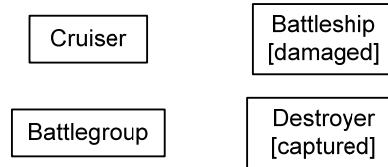


Figure 2-1-7 Object Nodes

Activity diagrams model data flow using tokens the same way they are used to model control flow. Tokens may contain data or objects, leading to a distinction between kinds of tokens: *Control tokens* do not contain data or objects, although *data tokens* do.

These two kinds of tokens are associated with two kinds of edges:

Control Flow—This kind of activity edge is a conduit for control tokens. Control flows begin at action or initial nodes and end at an action, activity final, flow final, decision, merge, fork, or join node. An edge leaving a decision, merge, fork, or join node whose incoming edges are all control flows is also a control flow. All activity edges in the examples presented so far are control flows.

Data Flow—This kind of activity edge is a conduit for data tokens. Any flow that begins or ends at an object node is a data flow. Furthermore, any edge leaving a decision, merge, fork, or join node at least one of whose incoming edges is a data flow is also a data flow.

The previously discussed rules for token-based execution apply just as well to data flows as to control flows, with the addition of a mechanism for adding and removing data from tokens. When a data token is consumed by a node, the data or object it contains is extracted. When a node produces a data token, it places data or objects in the token. Consider the diagram in Figure 2-1-8.

This activity begins when the initial node creates a control token and makes it available to the **Wash Clothes** action, which consumes the token and begins execution. When **Wash Clothes** completes, it produces a data token holding an instance of **Clothes [wet]**, as indicated by the **Clothes [wet]** object node. This token is accepted by the **Clothes [wet]** object node and immediately made available to the merge node, which in turn makes it available to the **Run Drier** action. **Run Drier** consumes the data token, removing the **Clothes [wet]** object for processing. When it has completed execution, **Run Drier** produces a data token containing the **Clothes** object

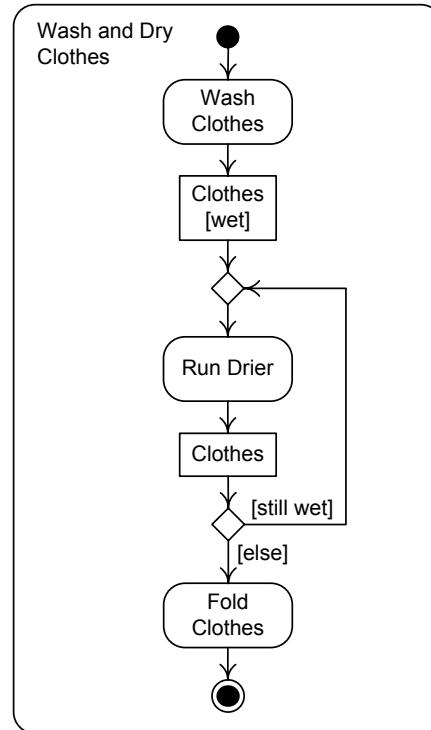


Figure 2-1-8 Data and Control Flows

and passes it to the **Clothes** object node. The **Clothes** object node makes the data token available on the edge leading to the decision node. The guards are evaluated. If the **Clothes** are still wet, the data token is directed back through the merge node to **Run Drier**, which consumes the token and executes as before. Otherwise, the data token is made available to the **Fold Clothes** action, which consumes the token and extracts the **Clothes** object. When it has completed execution, **Fold Clothes** produces a control token made available to the activity final node. The activity final node consumes the control token, terminating the activity.

In this example, the only control flows are those from the initial node to **Wash Clothes** and from **Fold Clothes** to the activity final node; all other flows are data flows. Nevertheless, the token-based execution mechanism works just as in previous examples, even though data tokens rather than control tokens pass through the decision and merge nodes.

A *pin* is a square attached to an action node that serves as a terminator for data flows into or out of the action node. Each pin must either have a single edge entering it (an *input pin*) or leaving it (an *output pin*). The name of the data or object type appears beside the pin; a description of the state of the data or object can appear after the name in square brackets, just as with an object node. Pins are simply an alternative way to show data flows.

indicated by object nodes. Figure 2-1-9 illustrates pins by using them in place of the object nodes in Figure 2-1-8.

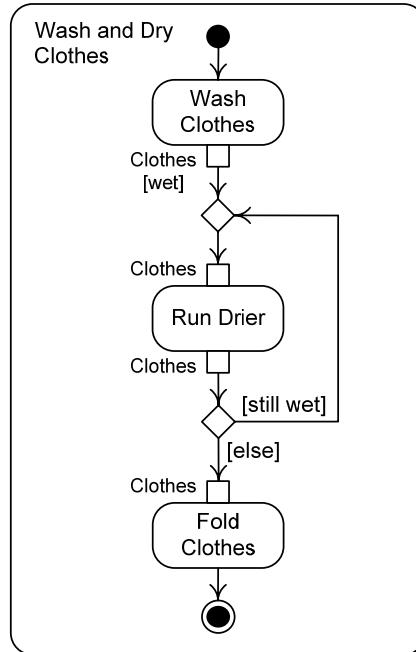


Figure 2-1-9 Pins and Data Flows

This diagram works just like the previous diagram. As before, the initial node makes a control token available to the **Wash Clothes** action, which consumes it and begins executing. When **Wash Clothes** finishes it produces a data token holding a **Clothes [wet]** object, as indicated by its output pin with that name. The token is made available on the edge leading to the merge node, which makes it available to the **Run Drier** activity at its **Clothes** input pin. **Run Drier** consumes the data token, extracting the **Clothes** object. When it is done, it produces a new data token holding **Clothes** and makes it available on the edge leaving its output pin leading to the decision node. The decision node evaluates its guards and routes the data token back through the merge node to **Run Drier** if the **Clothes** are still wet, or to **Fold Clothes** if they are dry. **Fold Clothes** consumes the data token at its **Clothes** pin, extracts the object, and begins execution. When execution completes, it produces a control token and makes it available on the edge leading to the activity final node. The activity final node consumes the control token and halts the activity.

Activity Parameters

Processes consume input and produce output, so activity diagrams need a way to represent this. An *activity parameter* is an object node placed on the boundaries of an activity symbol to represent data or object inputs or outputs. Input activity parameters have only outgoing arrows, and output

activity parameters have only incoming arrows. It is an error to have both incoming and outgoing edges on an activity parameter.

Activity parameters differ from other object nodes in that they contain the name of a particular data element or object rather than the name of its type. Parameter types are indicated by listing each parameter's name and type just below the activity name at the top left-hand corner of the activity symbol. The format for these specifications is

parameter-name : parameter-type

The activity parameter rectangle can still contain the state of the object or data item written below its name in square brackets, just as in other object nodes. Figure 2-1-10 illustrates activity parameters.

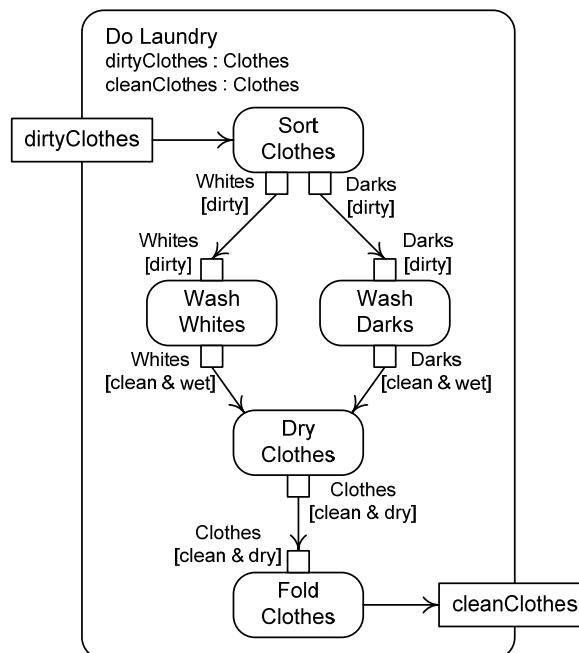


Figure 2-1-10 Activity Parameters

Input parameter nodes make data tokens containing the input data or object available as soon as the activity is begun. Output parameter nodes consume data tokens, extracting the data or object and making it available as an activity output. In the Figure 2-1-10 example, the **dirtyClothes** object node produces a data token containing the **dirtyClothes** object as soon as the activity begins. The **Sort Clothes** action consumes this token and begins execution. Thus, there is no need, in this case, for an initial node to get execution started. Similarly, when the **Fold Clothes** action completes, it produces a data token holding the **cleanClothes** object and makes it available to the **cleanClothes** output parameter node. This node consumes the token, extracting the object as an activity output. At this point no more

tokens flow through the diagram, so execution halts. No activity or flow final node is needed to halt execution in this case.

Activity Diagram Heuristics

With a little practice, activity diagrams are easy to write. Some simple heuristics make them easier to read:

Flow control and objects down the page and from left to right. English speakers are used to reading from left to right and down the page, so activity diagrams are easier to read if drawn this way.

Name activities and action nodes with verb phrases. Activities and actions are things that are performed or done, and hence they are best described by verbs or verb phrases.

Name object nodes and pins with noun phrases. Object nodes and pins are labeled with the names of object or data types, which are things; hence, these names should be noun phrases.

Don't use both control and data flows when a data flow alone can do the job. There is a tendency (encouraged by other notations, including older versions of UML activity diagrams) to document control flows and object flows separately. This often leads to both control and data flows between the same action nodes. Although this is not incorrect, the control flows are not needed, and they only clutter the diagram.

Make sure that all flows entering an action node can provide tokens concurrently. As the example in Figure 2-1-4 illustrates, a process can deadlock if two or more alternative (rather than concurrent) flows meet at a node. This problem is solved by using merge nodes to consolidate non-concurrent flows.

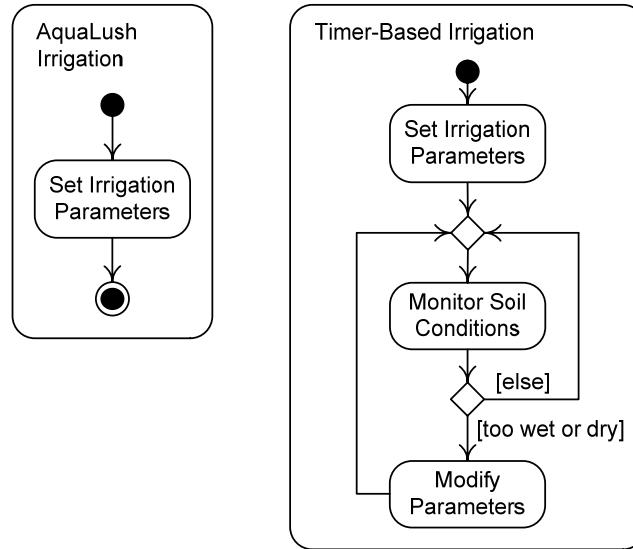
Use the [else] guard at every branch. When control reaches a branch, it must be clear which path should be followed. The [else] guard ensures that there will be a path for control to follow when all other guards are false. Therefore, it is safest to use [else] for one of the alternatives in a branch.

When to Use Activity Diagrams

Activity diagrams can document dynamic models of any process. Processes occur at all levels of abstraction in software design, and process modeling is useful for analysis as well as resolution. Thus, activity diagrams are potentially useful throughout software design. Two examples illustrate this wide range of applicability: Activity diagrams can describe client processes in product design analysis and algorithms during low-level design.

Process Description: AquaLush

Suppose the AquaLush marketing team wants to illustrate how much more convenient AquaLush is than a conventional irrigation system when conditions are too wet or too dry. The activity diagrams in Figure 2-1-11 contrast the processes that customers must use with AquaLush and a conventional timer-based irrigation system.

**Figure 2-1-11 AquaLush Versus Conventional Irrigation**

Algorithm Description

At the other end of the spectrum of abstraction, activity diagrams can be used in detailed design in place of flowcharts, an old notation for diagramming algorithms. Consider the activity diagram in Figure 2-1-12.

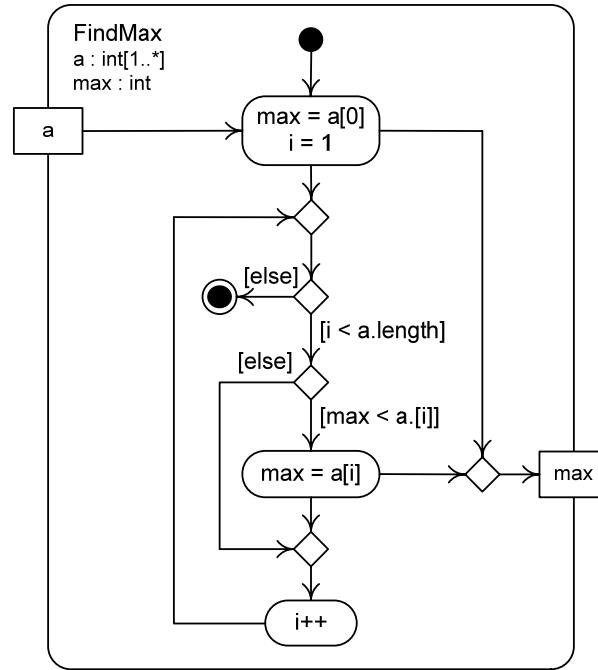
**Figure 2-1-12 Maximum Finding Algorithm**

Figure 2-1-12 illustrates an algorithm for finding the maximum value in an int array `a` with at least one element. Java is used in the action nodes and guards.

Heuristics Summary

Figure 2-1-13 summarizes the activity diagram heuristics discussed in this section.

- Flow control and objects down the page and from left to right.
- Name activities and action nodes with verb phrases.
- Name object nodes and pins with noun phrases.
- Don't use both control and data flows when a data flow alone can do the job.
- Make sure that all flows entering an action node can provide tokens concurrently.
- Use the [else] guard at every branch.

Figure 2-1-13 Activity Diagram Heuristics

Section Summary

- A **process** is a collection of related tasks that transforms a set of inputs into a set of outputs.
- An **activity diagram** is a UML dynamic process description notation that depicts actions and the flow of control and data between them.
- An **activity** is a non-atomic task or procedure that can be broken down into **actions**, which are atomic tasks or procedures.
- Activities are represented by *activity symbols* containing graphs showing various kinds of *activity nodes* connected by *activity edges*.
- The model represented in an activity diagram is specified in terms of a virtual machine that processes tokens.
- Tokens are produced and consumed by *action nodes* and flow over activity edges.
- Activity diagrams have symbols to depict process initiation and termination, control flow, data flow, input and output parameters, concurrency, and synchronization.
- Activity diagrams can document dynamic models of any process, and they are useful throughout software design.

Review Quiz 2.1

1. Define “process” and explain how activity diagram elements correspond to the things mentioned in this definition.
2. What are tokens and why are they important?
3. What is the difference between a control flow and a data flow in an activity diagram?
4. List three ways that the flow of control through an activity diagram might be improperly specified.

2.2 Software Design Processes

Software Design Activities

We have discussed how software design consists of two rather different design activities: software product design and software engineering design. We can thus say that the software design process consists of two actions: software product design and software engineering design, as depicted in Figure 2-2-1.

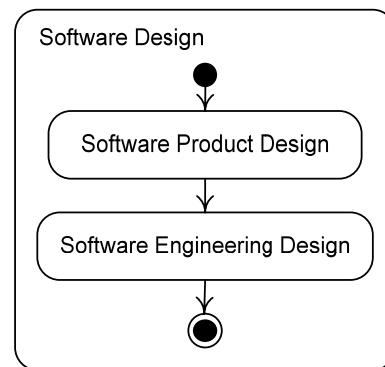


Figure 2-2-1 Software Design Process

Let's refine this very abstract description of the software design process.

Analysis and Resolution

The first step of problem solving must always be to understand the problem. If design is problem solving, then this activity must be the first step in design. *Analysis* is the activity of breaking down a design problem for the purpose of understanding it.

Once understood, a problem can be solved. Unfortunately the activity of solving a design problem does not have a good, widely accepted name. Traditionally this activity has been called *design*, but this is very confusing. In the traditional way of speaking, design consists of the following steps:

Analysis—Understanding the problem.

Design—Solving the problem.

In this book, we refer to the activity of solving a design problem as *resolution*.

Now that you are aware of it, you will probably notice that the term “design” is often used ambiguously. You must be careful when studying other materials to distinguish from context between uses of “design” that refer to the entire activity of specifying a product and the sub-activity of producing and stating a solution to the design problem.

The following definitions summarize our terminology about the sub-activities of design.

Analysis is breaking down a design problem to understand it.

Resolution is solving a design problem.

Analysis occurs at the start of product design with a product idea and again at the start of engineering design with the SRS. Product design resolution produces the SRS, and engineering design resolution produces the design document. The activity diagram in Figure 2-2-2 summarizes this overall decomposition of the software design process in terms of analysis and resolution activities.

In accord with our focus on engineering design, most of the analysis techniques and notations discussed in this book are presented in the context of analyzing an SRS document. Most analysis tools are useful for product design analysis as well.

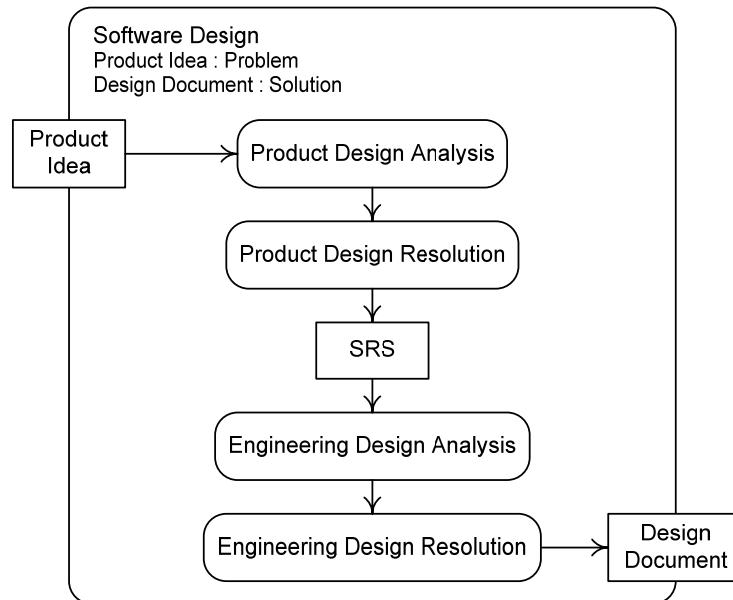


Figure 2-2-2 Analysis and Resolution in Software Design

Further refinement of this process depends on closer examination of problem solving, to which we now turn.

A Generic Problem-Solving Strategy

Suppose you are faced with putting together a class schedule for next semester. You might follow these steps:

1. You must know what courses to schedule, so you need to figure out the courses you need or want to take. You might need required courses or be interested in certain electives. Some courses may not be candidates because of prerequisites or because they are not offered next semester. You must also understand all constraints on the solution. For example,

some times of the day may be out of the question because you have other activities then, and you may like or dislike certain professors or classrooms. In short, you must first *understand the problem*.

2. You might next generate several candidate class schedules with the help of a list of class offerings and perhaps a grid of class times. In this step you *generate candidate solutions*.
3. You will have to check the candidate class schedules to make sure they really work. This may include making sure that class times do not conflict and that all constraints are satisfied. You may consider other things, too. For example, you might check that you can get from each class to the next in the allotted time, and that you have blocks of free time for studying and homework. Finally, you may attempt to register or to check somehow that the classes you chose are open. In summary, during this step you *evaluate solutions*.
4. If you have one or more workable class schedules, you can choose the one you like best. This may involve some sort of ranking—for example, you might rank schedules based on how early classes begin each day, or by whether classes are clustered on certain days or spread out across the week. In this step, you *select the best solution(s)*.
5. If none of your candidate class schedules pass muster, you may need to try again, going back to step 2 to generate more alternatives. Also, you may discover when checking over your candidate solutions, when selecting one of them, or even when generating them that you did not completely understand the problem. For example, you might discover when checking schedules that some sections of a class have an additional recitation section, and you must go back to step 1 to find out about these special sections before going further. In short, you may need to *iterate if no solution is adequate*.
6. If you have one or more good schedules and have selected the best one, you have a solution, and you can go ahead and register for classes. You will need to keep a copy of your schedule. If you are wise, you will keep the other materials you used as well—one of your classes may be cancelled, forcing you to redo your schedule. Having good documentation of your work will make this easier. The final step, then, is to *ensure that the solution is complete and well documented, and deliver it*.

The problem-solving strategy outlined and illustrated in this example is simple and complete, and we will use it to further refine our description of the software design process.

A Generic Design Process

We will now use this general problem-solving strategy as a model for a general design process. The input to this process is a design problem; the output is a well-documented design solution. The steps of this process are

1. *Analyze the Problem*—Obtain information about the problem and then break it down to understand it. This task may begin with an existing problem statement, or one may have to be written. This step should

produce a clear design problem statement to guide the remainder of the process.

2. *Generate/Improve Candidate Solutions*—Generate candidate solutions, or improve inadequate candidate solutions generated and evaluated before.
- 3 *Evaluate Candidate Solutions*—Evaluate the newly generated or improved candidate solutions, especially against the problem statement.
4. *Select Solutions*—Rank the solutions, and if one or more of them are adequate, select the best one as the final solution; otherwise, select several of the best solutions for further improvement.
5. *Iterate*—If a misunderstanding is discovered during solution generation, improvement, or evaluation, return to step 1 to correct the misunderstanding. If none of the solutions is adequate, return to step 2 to improve the best solutions or generate new solutions.
6. *Finalize the Design*—Make sure that the design process and the final solution are well documented and deliver the finished design.

We can best represent this process using two UML activity diagrams: Figure 2-2-3 shows iteration between analysis and resolution, and Figure 2-2-4 elaborates the resolution activity.

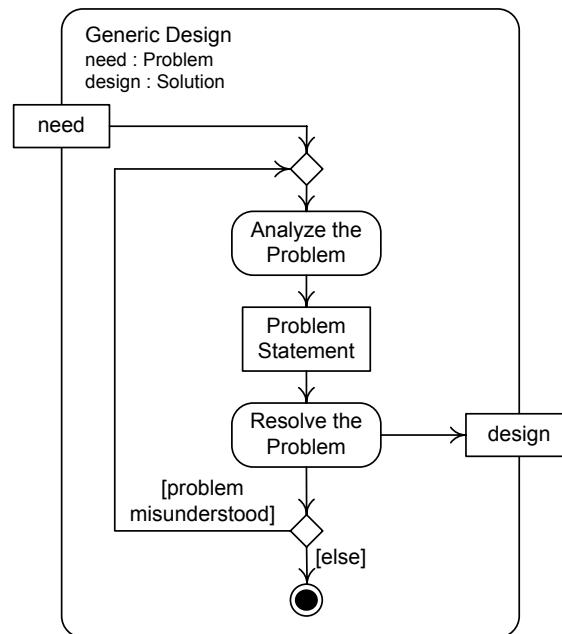


Figure 2-2-Generic Design Process

This diagram shows that any misunderstanding of the problem discovered during problem resolution results in iteration back to analysis. Figure 2-2-4 shows the activities that take place during the **Resolve the Problem** activity of Figure 2-2-3.

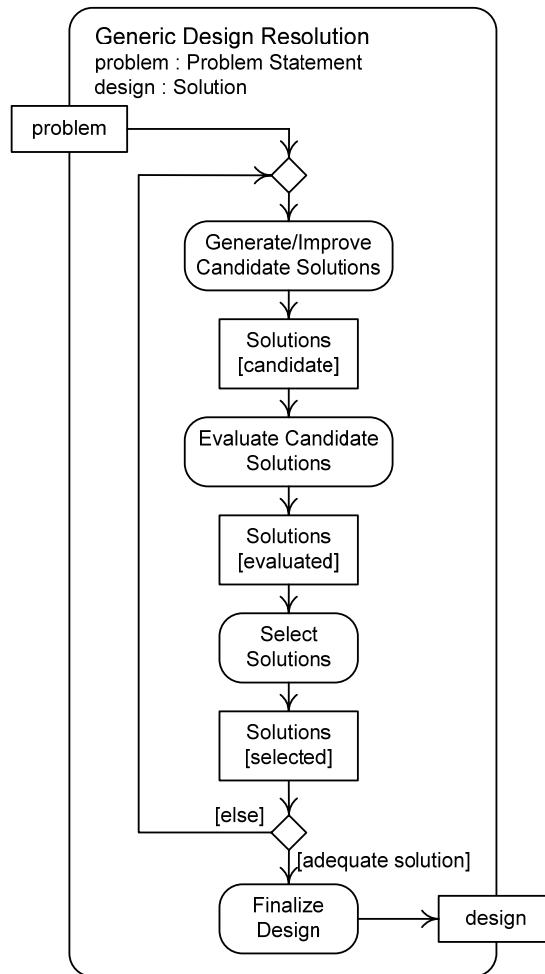


Figure 2-2-4 Generic Design Resolution Process

Process Characteristics There are two points to note about the generic design resolution process of Figure 2-2-4. The first is that the resolution generation step should produce *several* candidate solutions or improvements. A common mistake is to jump at the first solution that comes to mind. The first idea is rarely the best. We summarize this point in the following recommendation.

Designers should generate many candidate solutions during the design process.

Designers with a tendency to adopt the first solution that comes to mind can help correct it by brainstorming—writing down several solutions to every design problem, big or small, before evaluating any of them. Teams can combat this problem by having each team member generate solutions alone before discussing the problem as a group.

The second point to notice is that design is highly *iterative*. Complex problems are usually not completely understood until after a prolonged effort to solve them, and good designs are never generated on the first try, even for quite simple problems. Usually tens or even hundreds of iterations are needed to thoroughly analyze and resolve the design problem for a non-trivial product.

The design process is highly iterative; designers must frequently reanalyze the problem and must generate and improve solutions many times.

Designers who don't expect much iteration may worry that they are not making progress, try to bite off too much in each iteration, or end the design process prematurely. Experienced designers know that iteration is the way to understand and solve hard design problems, and that the bigger the problem is, the more iterations should be expected. Design, like most problem solving, is essentially a trial-and-error process.

A Generic Software Product Design Process

Software product design begins with a product design problem and ends with delivery of an SRS. The first step is to understand the product design problem, including its scope, its clients and other interested parties, and the enterprise goals the new product must achieve. This starting point is provided by a document we call a *project mission statement* (discussed in Chapter 3). Analysis of more and more detailed needs is interspersed with product design resolution. Abstract product requirements are iteratively refined by eliciting detailed client needs and desires about abstractly specified parts of the product, and then making more detailed specifications of product features and functions to meet them. This process eventually results in requirements that are sufficiently detailed to realize in software. The activity diagram in Figure 2-2-5 describes a generic software product design process.

The control flow branch back to **Elicit/Analyze Detailed Needs** reflects the iteration done as requirements are specified in greater and greater detail. The control flow branch back above **Generate/Improve Candidate Requirements** realizes the main iteration from the generic design resolution process in Figure 2-2-4. Note that iteration back to either of the analysis steps of this process can occur any time a misunderstanding of the problem is discovered, though the branches showing this possibility have been left out of the diagram to simplify it. Iteration back to analysis frequently occurs as designers discover gaps or errors in their understanding of the problem.

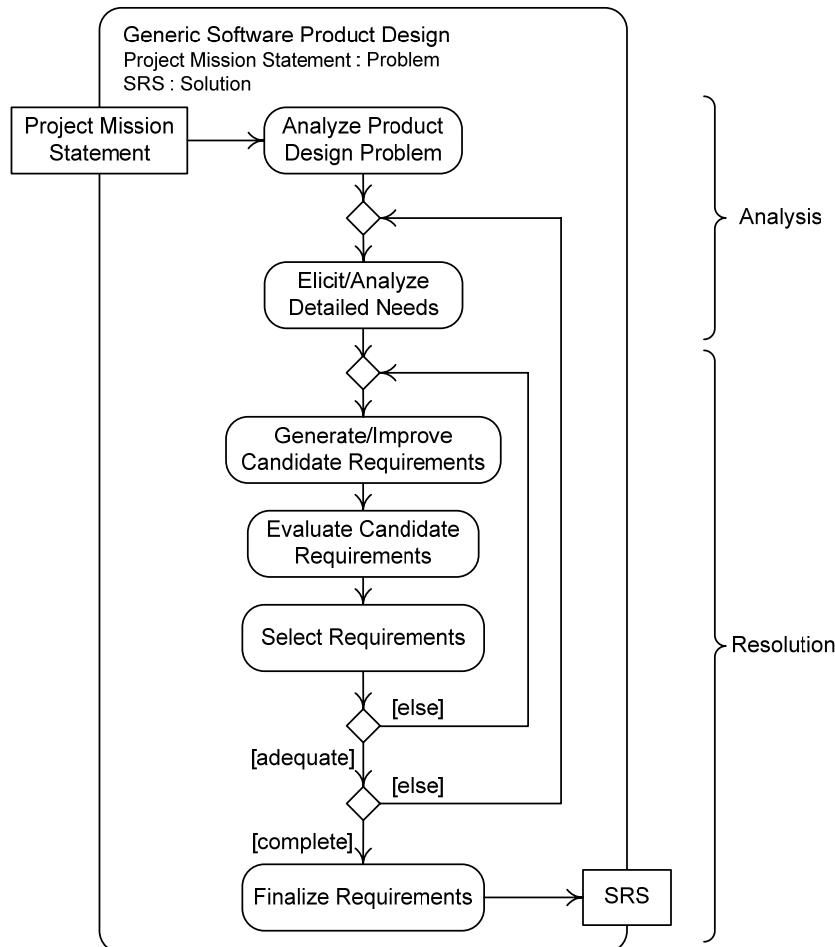


Figure 2-2-5 Generic Software Product Design Process

A Generic Software Engineering Design Process

The generic software design process can also be refined to include details relevant to software engineering design. In particular, software engineering design resolution is traditionally broken into two major *phases*, or *levels*, as pictured in Figure 2-2-6.

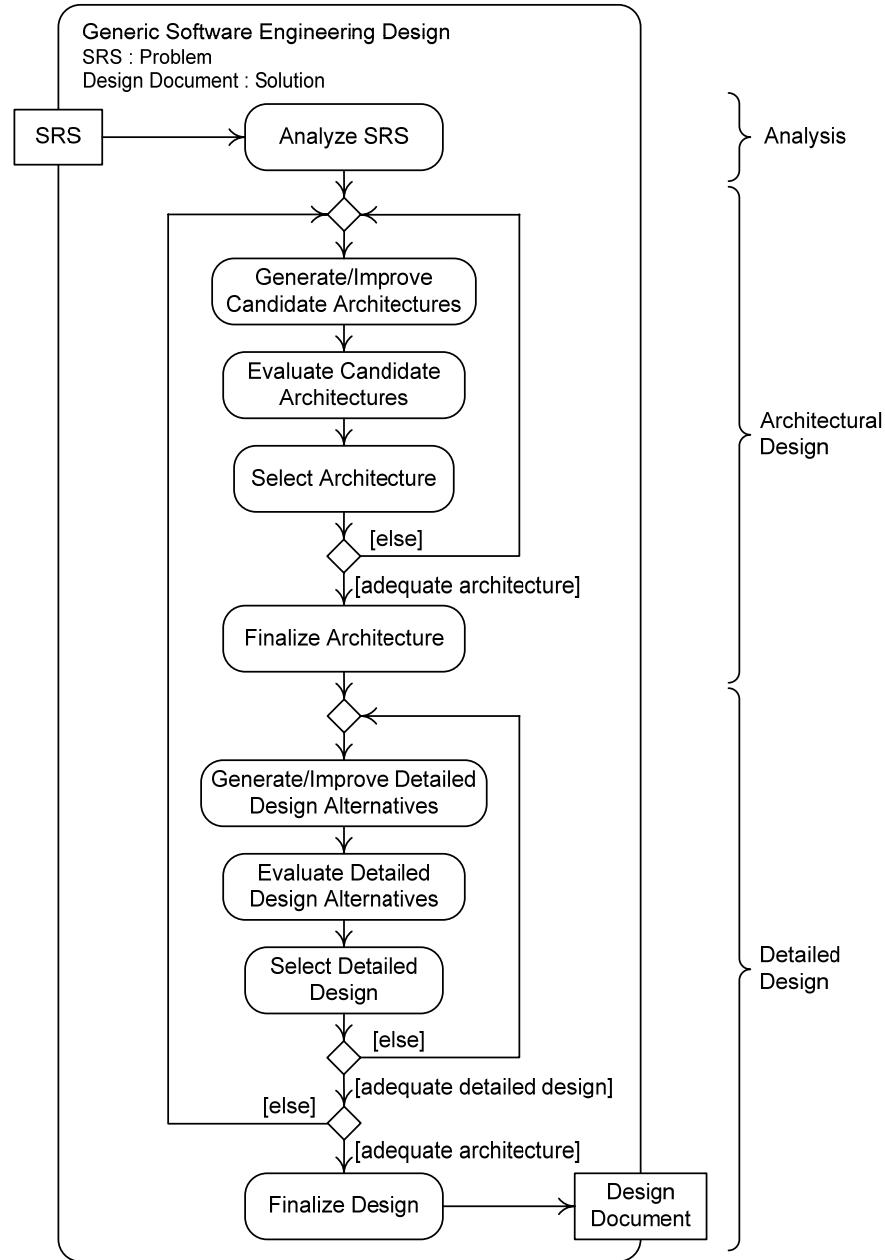


Figure 2-2-6 Generic Software Engineering Design Process

This diagram shows that the first big job after problem analysis is **architectural design**, which is high-level software engineering design resolution. This phase is comprised of iterative design generation,

evaluation, and selection. Attention then turns to **detailed design**, which is low-level software engineering design resolution. Detailed design likewise consists of iterative generation, evaluation, and selection activities. High-level resolution comes first, followed by low-level resolution, making this design process a top-down approach.

The diagram shows a branch back to architectural design after detailed design. This extra iteration is a token indication that at any point designers may need to return to an earlier activity and make further revisions. For example, analysis problems may surface during architectural or detailed design, forcing reconsideration of the problem, or architectural shortcomings may surface during detailed design or when the design is being finalized, forcing adjustments in the architecture. Backtracking occurs often—the software engineering design process is highly iterative throughout.

This generic software engineering design process structures the arrangement of Part III of this book. Software engineering design analysis is covered first, followed by chapters on architectural design, then detailed design. The latter is a large topic; it is further divided for discussion into mid-level detailed design and low-level detailed design.

Section Summary

- At the highest level of abstraction, the software design process consists of a software product design activity followed by a software engineering design activity.
- **Analysis** is breaking down a design problem to understand it. **Resolution** is solving a design problem. Design begins with analysis and ends with resolution.
- A generic design process begins with analysis and has an iterative resolution phase emphasizing repeated candidate solution generation, improvement, and evaluation.
- The generic software product design process begins with analysis and has a highly iterative resolution activity for stating requirements in greater and greater detail.
- The generic software engineering design process begins with analysis and has two highly iterative resolution phases: architectural design and detailed design.
- Designers should generate many candidate solutions during the design process.
- The design process is highly iterative; designers must frequently reanalyze the problem and must generate and improve solutions many times.

Review Quiz 2.2

1. Distinguish design analysis from design resolution.
2. Explain the steps in a generic design process.
3. Explain the steps in the generic software product design process.
4. Explain the steps in the generic software engineering design process.

2.3 Software Design Management

Project Management

Software development is complex, expensive, time consuming, and usually done by groups of people with varying skills and abilities. If it is simply allowed to “happen,” the result is chaos. Chaos is avoided when software development is *managed*. Software development must be planned, organized, and controlled, and the people involved must be led.

There are at least two sorts of business activities that must be managed.

Operations are standardized activities that occur continuously or at regular intervals. Examples of operations include personnel operations, such as hiring and performance review; payroll operations, such as collecting data about work done and paying people for it; and shipping and receiving operations, such as receiving raw materials and sending out finished products. This book does not cover operations activities.

In contrast, a **project** is a one-time effort to achieve a particular, current goal of an organization, usually subject to specific time or cost constraints. Examples of projects are efforts to introduce new products, to redesign tools and processes to save money, or to restructure an organization in response to business needs.

Although both operations and projects need to be managed, the things that need to be done in managing them are very different. For example, not much planning is needed during ongoing operations, but much planning is needed to complete a project.

Software development clearly fits the description of a project, so software development management is a kind of project management. There are five main project management activities:

Planning—Formulating a scheme for doing a project.

Organizing—Structuring the organizational entities involved in a project and assigning them responsibilities and authority.

Staffing—Filling the positions in an organizational structure and keeping them filled.

Tracking—Observing the progress of work and adjusting work and plans accordingly.

Leading—Directing and helping people doing project work.

We will examine these software project management activities in more detail and discuss how they can be done for the design portion of a software project next.

Project Planning

The first step in working out a project plan is to determine how much work must be done and the resources needed to do it. **Estimation** is calculation of the approximate cost, effort, time, or resources required to achieve some end. Thus project planning begins with estimation of the effort needed to do the work, how long it will take, and how many people

and other resources (such as computers, software tools, and reference materials) will be required to do it.

Estimation is a difficult task that has been much studied. Several estimation techniques are available. Most begin by estimating the size of work products such as source code, documentation, and so forth, and then deriving estimates of effort, time, cost, and other resources.

Once estimates are available, a schedule can be devised. A **schedule** specifies the start and duration of work tasks, and often the dates of milestones. A **milestone** is any significant event in a project. Schedules are based on effort and time estimates, and they must take account of the fact that some tasks must be completed before others can begin. For example, a sub-program must be coded before it can be tested, so sub-program coding must be scheduled before sub-program testing.

After a schedule is drafted, tasks are allocated human and other resources. These allocations are based on effort and resource estimates, and they must be coordinated with the schedule so that the same resources are not allocated to two or more tasks at once. Schedules must often be adjusted to avoid such conflicts.

Things can always go wrong, and it helps to be prepared. A **risk** is any occurrence with negative consequences. **Risk analysis** is an orderly process of identifying, understanding, and assessing risks. The project plan should include a risk analysis along with plans for preventing, mitigating, and responding to risks.

The final portion of the project plan is a specification of various rules governing work. Such rules fall into the following categories:

Policies and Procedures—A project plan might specify that pairs of programmers must write all software or that only software that has passed a code inspection may be added to the completed code base.

Tools and Techniques—A plan might specify that only a certain Integrated Development Environment (IDE) be used or that a certain analysis and design technique be used.

Often these rules are part of an organizational standard, or at least an organization's culture, and so they may be in force without appearing explicitly in the project plan.

In summary, a project plan consists of estimates, a schedule, resource allocations, risk analysis, and rules governing the project. The project plan guides the project, and it is the basis for project tracking.

Project Organization

There are many ways to organize people into groups and assign them responsibilities and authority, called *organizational structures*. There are also many ways for people in groups to interact, make decisions, and work together, called *team structures*. For example, groups might be responsible for carrying projects from their inception through completion, which is called a *project organization*, or they might be responsible for just part of the project, such as design or coding or testing, which is called a *functional*

organization. A team might have a leader who makes decisions, assigns work, and resolves conflicts, (a *hierarchical team*), or it might attempt to make decisions, assign work, and resolve conflicts through discussion, consensus, and voting (a *democratic team*).

Projects are organized in light of the tasks that need to be done, the culture in the organization, and the skills and abilities of available and affordable staff. For example, a small company with a highly skilled staff that does small- to medium-sized software development work may use a project organizational structure with democratic teams.

A project must be organized, and the organization documented, early in a project. Often, a specification of a project's organization is included in the project plan.

Project Staffing

An organizational structure has groups with roles that must be filled. For example, an organizational structure with a testing group certainly has roles for testers and possibly also for test group supervisors, test planners, and test support personnel.

Project staffing is the activity of filling the roles designated in an organizational structure and keeping them filled with appropriate individuals. This activity includes hiring and orienting new employees, supporting people with career development guidance and opportunities through training and education, evaluating their performance, and rewarding them with salaries and various kinds of awards and recognitions.

It is often noted that the single most important ingredient in a successful software project is the people doing the project. Thus, project staffing is an essential part of project management.

Project Tracking

Nothing ever goes exactly as planned, of course, so it is essential to observe the progress of a project and adjust the work, respond to risks, and, if necessary, alter the plan. A project can fail to go as planned in any of the following ways:

- The resources needed to accomplish tasks may not be as anticipated. If fewer resources are needed, a task may be finished early or with fewer people, and other tasks may be started ahead of schedule. If more resources are needed, more people or other resources must be found or the task may take longer to complete. In either case the estimates and schedule may have to be adjusted.
- A task may simply take more or less time than expected. In the worst case a task may completely stall—this may occur if a stubborn bug cannot be found or a design flaw cannot be fixed, for example. Such problems force a revision of the schedule, estimates, and risk analysis and response plans.
- Some of the rules governing the project may cause problems. For example, it may prove more expensive or time-consuming than expected to inspect all the source code or use a particular IDE. In this case either

the rules must be adjusted, the estimates, schedule, and risk analysis must be modified, or both.

- Something bad may occur—for example, a key staff member may become ill, or a budget cut may force a reduction in staff or equipment. If this risk was anticipated, then the planned response can be made. Otherwise, a response must be devised on the spot, which may alter the estimates, schedule, resource allocations, and risk analysis.

Tracking is essential so that estimates, schedules, resource allocations, risk analyses, and rules can be revised. But tracking is also important because the current project status and plan are the basis for decisions about the fate of the project. The concept for the product itself may need to be altered if it appears that the project will take too long or be too expensive. If things look bad enough, the project may even be cancelled.

Leading a Project Even the best people with an excellent plan, all the tools and materials they need, and a sound organizational structure cannot succeed without adequate direction and support, a broad category of management responsibility called **leadership**.

Direction is obviously necessary to ensure that everyone is doing work that contributes to the success of the project. Duplicate or wasted effort must be avoided, while someone must do all essential work.

Merely directing people to the right work does not guarantee success. People also need a congenial work environment, an emotionally and socially supportive workplace, and the enthusiasm for their work that comes from feeling they are doing something important. Managers must attend to these intangible but essential factors to make projects succeed.

Iterative Planning and Tracking The ultimate reason that planning is difficult is that it depends on knowing lots of details about what must be done, but many of these details are not known until much of the work is complete. For example, suppose we want to plan a project to develop a product to install software needed for computer science courses. It will be very difficult to make this plan until we know in some detail exactly what this product will do and how it will work. However, we must complete the first phases of the project to discover these details.

The solution to this problem is to do some preliminary work as a basis for planning. This work can be considered a project prelude, or an initial rough plan can be formulated and refined later in light of this preparatory work.

But even plans made later in a project suffer from lack of knowledge of details still to be determined. This suggests that plans should be reformulated on a regular basis. On this model, an initial rough plan is made at the start of a project and refined repeatedly as the project progresses. Tracking between plan revisions consists of observing progress, adjusting the work and the current plan, and collecting information for the next planning phase. This is called **iterative planning and tracking**.

In iterative planning and tracking, the base project plan should include the rules governing the project and the best estimates, schedule, resource allocations, and risk analysis possible, given what is initially known. The base plan schedule must include plan revision tasks and due dates. Revised plans should refine estimates, schedules, resource allocations, and risk analyses as project knowledge grows. The activity diagram in Figure 2-3-1 illustrates the iterative planning and tracking process.

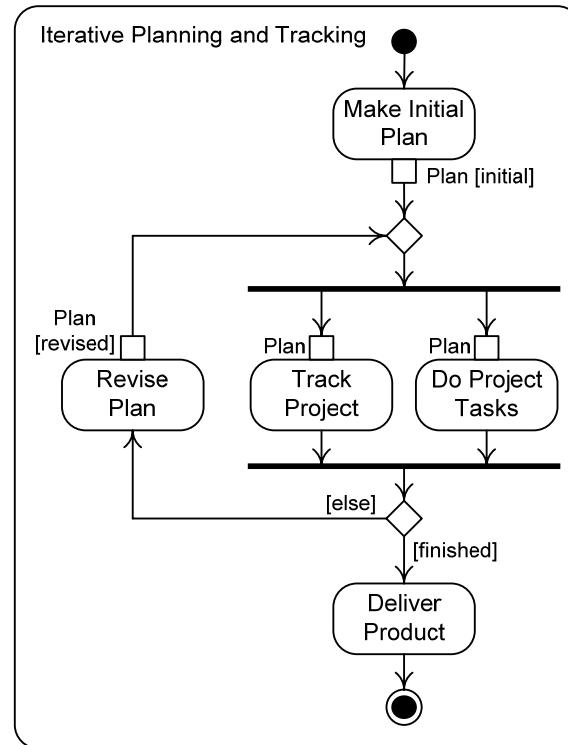


Figure 2-3-1 Iterative Planning and Tracking Process

Iterative planning and tracking is used in several popular software development methods such as the Rational Unified Process and various agile methods. The Rational Unified Process breaks development into phases during which a software product is conceived, specified, built, and tested. Each phase is performed iteratively until it is completed. Planning and tracking is based on iterations. *Agile methods* emphasize quick response to changing user needs and market demands by developing products iteratively in very small increments. In most agile methods plans are adjusted before each increment and tracking is based on increment completion.

Iterative planning and tracking is not needed for projects that are much like those that have come before. But if a project is creating something novel and innovative, iterative planning and tracking is a necessity. For this

reason, it is widely used in software development projects, and we will assume this approach in our discussion of design project management.

Design Project Decomposition

Most aspects of project management depend on the work to be done and, in particular, on how it is decomposed. An obvious way to break down a design project is to divide the work according to the generic design processes discussed in the last section. These activities produce various work products with corresponding tasks for their creation and refinement. Table 2-3-2 illustrates how this decomposition works. Many of the work products mentioned are discussed in the remainder of the book.

Work Phase		Typical Work Products
Product Design	Analysis: Design Problem	Statement of interested parties, product concept, project scope, markets, business goals Models (of the problem) Prototypes (exploring the problem)
	Analysis: Detailed Needs	Client surveys, questionnaires, interview transcripts, etc. Problem domain description Lists of needs, stakeholders Models (of the problem) Prototypes (exploring needs)
	Resolution: Product Specification	Requirements specifications Models (of the product) Prototypes (demonstrating the product)
Engineering Design	Analysis	Models (of the engineering problem) Prototypes (exploring the problem)
	Resolution: Architectural Design	Architectural design models Architectural design specifications Architectural prototypes
	Resolution: Detailed Design	Detailed design models Detailed design specifications Detailed design prototypes

Table 2-3-2 Work Products in Each Design Phase

This decomposition can be the basis for project planning, organization, staffing, and tracking, as we will see next.

Design Project Planning and Tracking

Iterative planning and tracking for a design project can be based on the work phases listed in Table 2-3-2. The initial project plan focuses on design problem analysis, with only rough plans for the remainder of the work, but expectations that the plan will be revised before product design resolution, engineering design analysis, and engineering design resolution.

Initial estimates of effort, time, and resources are as precise as possible, based on the work products to be completed. These estimates might be modeled on data about work done in the past or an analogy with similar jobs with which the planners are familiar. The estimates are then used to block out an initial schedule, allocate resources, analyze risks, and set the rules guiding the project.

Product analysis work is tracked against the initial plan. Ideally, problem analysis is complete when it is time to revise the plan, since planning the product design resolution phase requires this information. The plan may be altered during tracking to make this happen.

Once design problem analysis is complete, the scope of the entire design project will be clearer. The models, prototypes, and especially the project description are a solid basis for estimating the scope and contents of the remaining phases and give a somewhat better indication of the costs and risks of the remainder of the project. A revised plan prepared before the product design resolution phase should have much more accurate estimates, schedule, resource allocations, and risk analysis for this phase, and refined though still approximate plans for the engineering portion of the project. Iterative planning and tracking continue through the engineering design sub-phases, with more details added each time the plan is revised.

Design Project Organization, Staffing, and Leadership

The decomposition of design project work explained above can also be the basis for organizing the project. Design teams should be formed with responsibility for the design as a whole, each major phase, each sub-phase, and the production of the various work products. For example, a large company might have a division responsible for requirements and design. This division might include product design and engineering design departments. The product design department might have design teams specializing in needs elicitation and analysis, requirements specification, user interaction design, and requirements quality assurance. The engineering design department might have design teams specializing in high-level design, low-level design, and quality assurance.

Organizations are staffed to fit the decomposition of design work as well. Projects need staff to elicit and analyze needs, create prototypes, model systems, create product designs, write requirements specifications, design user interaction, make high-level and low-level engineering designs, and assure quality. The responsibilities of individual staff members are determined by the structure of the organization.

Leadership in a design project is not very different from leadership in other kinds of projects.

Design as a Project Driver

Design work extends from the start of a software development project to the coding phase, and it recurs during maintenance. Furthermore, the two major products of software design, the SRS and the design document, are the blueprints for coding and testing. Thus, design is the driving activity in software development.

This role extends also to software project management. By the time the software design is complete, enough information is available to make accurate and complete plans for the coding and testing phases. Good design work early in the life cycle is crucial for software development project success.

Section Summary

- Software development is complex, expensive, time-consuming, and done by groups, and hence it must be managed.
- Software development management is **project** management: the management of a one-time effort to achieve a particular, current goal of an organization, usually subject to specific time or cost constraints.
- Project management includes five main tasks: planning, organizing, staffing, tracking, and leading.
- **Project planning** is formulating a scheme for doing a project. It includes estimating, scheduling, resource allocation, risk analysis, and setting rules for the project.
- **Project organizing** is structuring organizational entities and assigning responsibilities and authority. Organizational structure should reflect project tasks.
- **Project staffing** is filling the positions in an organizational structure. This activity includes hiring, career development, performance evaluation, and compensation.
- **Project tracking** is observing the progress of work and adjusting work and plans accordingly.
- **Leadership** is directing and helping people doing work. Leaders must create a work environment that keeps employees motivated and productive.
- **Iterative planning and tracking** is making a rough initial project plan and refining it at set times in light of tracking data and completed work products.
- Software design work can be decomposed by work phase and further broken down by the work products completed in each work phase.
- This work breakdown is the basis for software project design planning, tracking, organization, and staffing. Iterative planning and tracking is often appropriate for design projects.
- Software development is driven by software design.

Review Quiz 2.3

1. Distinguish between operations and projects.
2. List the five main activities of project management.
3. What is a milestone?
4. What is a risk? What is risk analysis?

Chapter 2 Further Reading

Section 2.1 UML activity diagrams are presented in [Booch et al. 2005] and [Rumbaugh et al. 2004]. Flowcharts are discussed in [Yourdon 1989].

Section 2.2 Davis [1993] has a good discussion of analysis. Software engineering surveys, such as [Pressman 2001] and [Sommerville 2000], provide overviews of analysis and design.

Section 2.3 Excellent discussions of software project management include [Thayer 1997] and [Royce 1998]. Capers Jones [1998] covers estimation in depth. The Rational

Unified Process is presented in [Jacobson et al. 1999]. Agile methods are discussed briefly in [Fox 2005] and at length in [Cockburn 2002].

Chapter 2 Exercises

Section 2.1

- Find the errors: Identify at least four mistakes in the activity diagram in Figure 2-E-1.

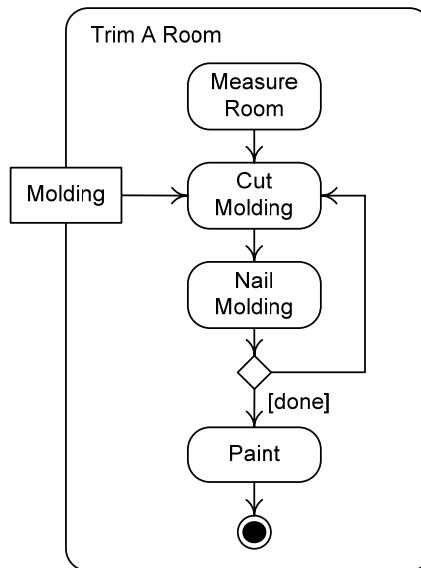


Figure 2-E-1 Activity Diagram for Exercise 1

- Draw an activity diagram modeling the process of taking a shower between (but not including) undressing and getting dressed again.
- Draw an activity diagram modeling the process of buying groceries, starting with making a shopping list.
- Draw an activity diagram modeling the process of making a sandwich.
- Draw an activity diagram modeling the process of buying a car.
- Draw an activity diagram modeling the process of finding a book in the library and checking it out.
- Draw an activity diagram modeling the process of making a schedule and registering for classes.
- Draw an activity diagram modeling what happens when a Web browser starts up and loads its home page. Include object nodes in your diagram.
- Draw an activity diagram modeling the process you went through to apply for college. Your diagram should have concurrent flows.
- Draw an activity diagram modeling the process of two people washing a car. Include concurrent actions in your model.

11. Draw an activity diagram similar to Figure 2-1-12 that models the binary search algorithm.

12. Draw an activity diagram modeling a process of your choice.

13. List two heuristics in addition to those listed in the text that you think would make activity diagrams easier to read or write.

Section 2.2 14. Categorize each of the following activities as part of software product design analysis (PA), software product design resolution (PR), software engineering design analysis (EA), or software engineering design resolution (ER):

(a) Choosing data structures and algorithms

(b) Laying out the contents of a window in a user interface

(c) Asking clients what they need in a new program

(d) Deciding which class should have a certain method

(e) Reviewing an SRS to make sure it is complete

(f) Reading an SRS to understand how a feature should work

(g) Reading an SRS to ensure all requirements are accounted for in a design document

(h) Deciding which features should go in each of several releases of a product

15. Modify the activity diagram in Figure 2-2-5 to show the major inputs and outputs of each action node in the generic software product design process.

16. Modify the activity diagram in Figure 2-2-6 to show the major inputs and outputs of each action node in the generic software engineering design process.

17. Modify the activity diagram in Figure 2-2-6 to show more of the iteration that may occur during the software engineering design process.

Section 2.3 18. Define “management.”

19. Give two examples of operations and of projects. Do not use the examples in the discussion.

20. Give two examples of how a project might go wrong. Do not use the examples in the discussion.

21. Give an example to illustrate how a task might be better estimated later in the course of a project than earlier.

Team Projects 22. Follow the software product design process to produce an SRS for one of the following simple programs. Document your work at each stage.
grep—Accept a string and a text file as input. Search the text file and print every line in the file containing the string.

wc—Accept a text file as input. Report the number of lines, words, and characters in a text file.

sort—Accept a text file as input. Print the file with the lines sorted.

23. Follow the software engineering design process using the results of the previous exercise. Document your work at each stage.

- Research Projects**
24. Plan a project to develop a product to install the software needed for computer science courses. This product should require students to log into a central Web site that knows their schedules and the software requirements of each class. The product should download and configure all the software that each student needs. Students should be able to use the product to uninstall software later if they desire.
 25. Consult software project management resources and write essays about one or more of the following topics:
 - (a) Software project cost estimation
 - (b) PERT charts for project scheduling
 - (c) Risk analysis and risk management
 - (d) Organizational structures for projects
 - (e) Software development team structures
 - (f) Project tracking and control techniques
 - (g) Hiring and retaining software developers
 - (h) Organization and team leadership

Chapter 2 Review Quiz Answers

Review Quiz 2.1

1. A process is a collection of related tasks, with well-defined inputs and outputs, for achieving some goal. The activity symbol represents a process. The tasks in a process are represented in an activity diagram by action node symbols. Process inputs and outputs are represented by activity parameter object nodes. Activity diagrams have no means for representing process goals.
2. A token is a marker used in executing the virtual machine specified by an activity diagram. The virtual machine interpretation of activity diagrams provides a precise explanation of what these diagrams mean. Hence, tokens are the key to a precise specification of the meaning of an activity diagram.
3. A control flow is the movement of tokens not containing data or objects. These tokens determine which action nodes execute and when they execute. A data flow is the movement of tokens that do contain data or objects. These tokens determine which action nodes execute and when they execute. They also carry the input or output of action nodes.
4. Flow of control through an activity diagram might be improperly specified in the following ways:
 - (a) Two or more guarded expressions attached to edges leaving a decision node are true when a token reaches the node. Which branch should be taken?
 - (b) Several edges with guards leave a decision node, but none of the guard expressions is true when a token reaches the node. Which branch should be taken?
 - (c) Two or more alternative (rather than concurrent) flows enter an action node—this leads to deadlock.
 - (d) An edge leaves a flow or activity final node. All tokens are consumed at final nodes, so how can they leave along an outgoing edge?
 - (e) An edge enters an initial node. Initial nodes do not consume tokens, so what becomes of any incoming tokens?
 - (f) A node has no edges attached to it. Nodes produce or consume tokens,

which must arrive and depart along edges. What is the role of a node with no edges attached to it?

Review Quiz 2.2

1. Design analysis is understanding what needs to be done in the design, and design resolution is figuring out and specifying a way to do it. In terms of problem solving, analysis is the activity of understanding the problem and resolution the activity of producing a solution to the problem.
2. The generic design process discussed in this section follows the general problem-solving strategy. The design problem is first understood. Design problem solution consists of repeatedly generating new candidate solutions or refining existing candidates, and then evaluating them against the design problem. Eventually, one or more satisfactory solutions should emerge. The best one can then be documented and delivered.
3. Software product design begins with an analysis whose goal is to understand the needs and desires for the new product, along with any constraints. Design resolution consists of two nested iterations: The outer iteration is aimed at making requirements more and more detailed, while the inner iteration is aimed at generating and evaluating product design alternatives. The outer iteration begins with an elicitation step to obtain and understand detailed needs and desires; it then proceeds to the inner iteration, where requirements are repeatedly generated or improved, evaluated, and selected. The inner iteration stops when adequate requirements for part of the product at some level of abstraction are obtained; the outer iteration stops when the entire product has been specified in enough detail that it can be designed and implemented. At any point it may be necessary to return to an earlier activity in the process. Once the entire specification checks out, the requirements specification document may be completed and delivered as the last step in the process.
4. Software engineering design begins with an analysis whose goal is to understand the product and constraints on its construction specified in the SRS. Design resolution consists of two main phases: architectural and detailed design. Architectural design proceeds by iteratively generating, refining, evaluating, and selecting ideas for the major system constituents, their properties, interfaces, relationships, and interactions. Once this high-level specification of the software system is adequate, detailed design begins. Detailed design specifies the internals of the major system constituents down to the level of data structures and algorithms by repeatedly generating, refining, and evaluating alternatives until acceptable specifications are produced. At any point it may be necessary to return to an earlier activity. Once the entire specification checks out, the design document may be completed and delivered, completing the process.

Review Quiz 2.3

1. Operations are standardized activities that occur continuously or at regular intervals, while projects are one-time efforts to achieve a current goal. For example, in taking a course a student reads assigned material, does homework, writes papers, and takes exams on a regular basis. These activities might be considered student operations. Occasionally, students might apply for a scholarship or graduate school, do an internship or an independent study, or prepare a presentation for a conference. These rare tasks are student projects.
2. The five main activities of project management are planning, organizing, staffing, tracking, and leading.

3. A milestone is any significant event in a project. Task or phase completion dates or work product delivery dates are typically chosen as milestones. Milestones help track project progress.
4. A risk is any occurrence with negative consequences. Risk analysis is an orderly process of identifying, understanding, and assessing risks. For example, risk analysis may reveal that there is a high probability that a key task may take longer than anticipated. Once such risks are discovered, it is prudent to try to prevent them from occurring, to detect them as soon as possible if they occur, and to prepare a response to them. In the case at hand, one might investigate the task thoroughly to see if there are ways to estimate it more accurately, to put progress checks into the schedule to detect a schedule overrun promptly, to line up additional resources or an alternative work product if the delay is too large, and to rearrange the schedule so that a delay in completing the task will not delay the project too badly.

Part II Software Product Design

The second part of this book surveys software product design as the basis for a more detailed discussion of software engineering design.

Chapter 3 describes how organizations decide what software products to develop, what information is gathered as input to the product design process, and what output is expected from product design.

Chapter 4 discusses product design analysis by exploring how needs are gathered from stakeholders and how needs are recorded and understood by product designers.

Chapter 5 looks at product design resolution, including ways that designers can generate, evaluate, and refine product design alternatives and eventually choose the design for the product. It also explains how to check product design quality.

Chapter 6 looks in detail at use case modeling for designing product function. It introduces UML use case diagrams and use case description notations.



3 Context of Software Product Design

Chapter Objectives

This chapter provides the context of software product design by discussing the activities that lead to the launch of a product development project and the input and output of the software product design activity, as indicated in Figure 3-O-1.



Figure 3-O-1 Software Product Design

The product design process is discussed in Chapters 4 and 5.

By the end of this chapter you will be able to

- Explain how markets influence product types and product types influence product design;
- Explain and illustrate the product planning process;
- Explain what a project mission statement is, why it is important, and what its contents should be;
- Explain and illustrate the various sorts of requirements that appear in the project mission statement and the SRS; and
- List the contents of an SRS.

Chapter Contents

- 3.1 Products and Markets
- 3.2 Product Planning
- 3.3 Project Mission Statement
- 3.4 Software Requirements Specification

3.1 Products and Markets

Markets Influence Products

An organization creates products for economic gain. Product development is very expensive, so an organization must be careful to create products that it can actually sell or use. A **market** is a set of actual or prospective customers who need or want a product, have the resources to exchange something of value for it, and are willing to do so. Organizations study markets and use what they learn to decide what markets to develop

products for (called **target markets**), the types of products to develop, and what functions and capabilities these products must have. Thus, the sorts of products that an organization decides to develop ultimately depend on the target markets to which it hopes to sell the products.

Products Influence Design

A lot of what happens during product design depends on what sort of product is being designed. A product's characteristics influence the decision to develop the product; the resources and time devoted to product development; the techniques, methods, and tools used to develop the product; and the distribution and support of the final product. This section illustrates this idea and introduces a simple product categorization that will be used in later discussions.

Categorizing Products

Products fall into different categories along several dimensions. A **product category** is a dimension along which products may differ. We consider three product categories: target market size, product line novelty, and technological novelty. A **product type** is a collection of products that have the same value in a particular product category. For example, consumer products are a type in the target market size category that designates all products with a mass consumer market.

Target Market Size *Target market size* is the number of customers a product is intended to serve. Table 3-1-1 summarizes the types in the target market size category.

Type	Description	Examples
Consumer	Mass consumer markets	Word processors, spreadsheets, accounting packages, computer games, operating systems
Niche Market	More than one customer but not a mass consumer market	Programs for configuration management, shipyard management, medical office records management, AquaLush
Custom	Individual customers	Systems written for one part of a company by another part, space shuttle software, weapons software

Table 3-1-1 Target Market Size Category

Designers of custom and niche-market products can usually identify individuals who can express needs and desires for a product, but this is much harder for consumer products. On the other hand, designers generally have great freedom in designing consumer products but have less with niche-market products and even less with custom products. Designers need to pay attention to competitors when designing consumer and niche-market products, but this is not usually a concern when designing custom products. Different aspects of product design are more or less important in these different categories. Consumer products place a premium on attractive user interface design, while functionality is usually more important for custom and niche-market products.

Product Line Novelty *Product line novelty* is how new a product is in relation to other products that the organization provides in its current product line. Table 3-1-2 lists the types in this category.

Type	Description	Examples
New	Different from anything else in the product line	Tax preparation product in a line of accounting products, AquaLush
Derivative	Similar to one or more existing products in the product line	Database management system for individual users in a line of systems for corporate users
Maintenance Release	New release of an existing product	Third release of a spreadsheet

Table 3-1-2 Product Line Novelty Category

Designers are highly constrained when designing maintenance releases, less constrained when designing derivative products and least constrained when designing new products. Designing a new product is a very big job, designing a derivative product is a smaller but still formidable task, and designing a new release may be relatively easy.

Technological Novelty The final product category we consider is *technological novelty*: how much new technology is incorporated in a product, from the point of view of the target market at a particular time. The types in this category are listed in Table 3-1-3. Dates are given with the examples in this table to indicate when the technology had the degree of novelty in that category.

Type	Description	Examples
Visionary Technology	New technology must be developed for the product	Mobile computing (2000), Wearable automatic lecture note-taker (2004)
Leading-Edge Technology	Proven technology not yet in widespread use	Peer-to-peer file-sharing products (2002), AquaLush (2006)
Established Technology	Widely used, standard technology	Products with graphical user interfaces (2000)

Table 3-1-3 Technological Novelty Category

Designing products with visionary or leading-edge technologies is both the most difficult and the most exciting kind of software design. It is hard to figure out what clients want and whether products with new technology will attract customers, but being in the vanguard of a new technology is fun and affords an opportunity to make a major contribution. Products with visionary technology may never be built if efforts to develop the new technology fail. Even if they are a technological success, they may still fail in the marketplace if customers don't like the new technology. Leading-edge and established technology products are more likely to succeed. Established technology products usually have more competitors than

visionary or leading-edge technology products, making competitive analysis more important in their design.

Overview Table 3-1-4 summarizes the example categories and types discussed.

Category	Types
Target Market Size	Consumer, Niche Market, Custom
Product Line Novelty	New, Derivative, Maintenance Release
Technological Novelty	Visionary, Leading Edge, Established

Table 3-1-4 Example Product Categories and Types

This overview of the dimensions along which products can be categorized is meant to illustrate the wide range of issues that affect product design and some ways these issues may be taken into account. There are many other ways to categorize products, and the characteristics of products in other categories likewise affect the design process.

Section Summary

- A **market** is a set of actual or prospective customers who need or want a product, have the resources to exchange something of value for it, and are willing to do so.
- Markets influence product development decisions; product types influence product design.
- A **product category** is a dimension along which products may differ. A **product type** is a collection of products that have the same value in a particular product category.
- Product categories considered here include target market size, product line novelty, and technological novelty. The type of a product in a category has predictable effects on design.

Review Quiz 3.1

1. Define the technological novelty product category and list its types.
2. State some ways that technological novelty affects design.
3. List three product categories and their types, different from those in the text, that you may have heard of or that you think might be important.

3.2 Product Planning

The Product Planning Process

Product development is a central business activity that often determines an organization's fate because it is expensive, complicated, and risky. Organizations must select products for development thoughtfully and manage product development carefully to maximize the likelihood that they will succeed.

We discussed project management in Chapter 2; in this section we survey the activities that result in a decision to develop a product.

A **product plan** is a list of approved development projects, with start and delivery dates. Upper management formulates and periodically revises its product plan using a process similar to the one depicted in Figure 3-2-1.

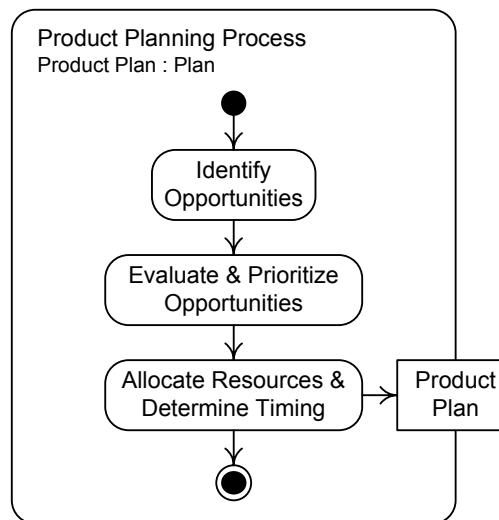


Figure 3-2-1 Product Planning Process

Although detailed study of product planning is beyond the scope of this book, we will take a brief look at the steps in this process.

Identify Opportunities

Ideas for products come from many places, in part depending on the product type. For example, consumer and niche-market product ideas may come from users, entrepreneurs, or developers, but most often they come from marketing organizations. **Marketing** is the process of conceiving products and planning and executing their promotion, distribution, and exchange with customers. Marketers (people who work in marketing) are therefore responsible for coming up with product ideas, persuading customers that a product satisfies their needs and desires, and arranging a mutually beneficial exchange of something of value for the product.

Marketers generate ideas for new products by studying customers and competitive products using surveys, focus groups, interviews, and user observation. Some marketing organizations have groups devoted to generating and evaluating ideas for new or improved products. Ideas for custom products tend to originate with clients, but various individuals in a client organization may come up with product ideas, including users, quality assurance specialists, and managers.

Ideas for maintenance releases are largely based on clients' experiences with current products, so they come from both clients and suppliers studying the products in use. Derivative products are often intended to fill out a product line and tend to originate with suppliers. Wholly new

product ideas can come from anywhere: These are the sorts of ideas that can make whole new businesses or industries. For example, Dan Bricklin and Bob Frankston created VisiCalc, the first electronic spreadsheet, in 1979. This program gave an enormous boost to the fledgling personal computer industry, as well as giving rise to an important category of PC products.

Opportunity Funnels

Smart and successful marketers harvest product ideas from markets, clients, developers, managers, business partners, inventors, and so forth. An **opportunity funnel** is a mechanism for collecting product ideas from diverse sources. An opportunity funnel includes *passive channels* for idea input, such as suggestion lines, bug-report Web pages, and awards programs for outstanding product ideas. The opportunity funnel should also have *active channels*, such as the following activities:

- Studying users to detect problems, new user needs and desires, inefficiencies, and new ways of working with products;
- Eliciting ideas through surveys, focus groups, and interviews of clients, developers, business partners, and so forth;
- Continuously studying competitive products to evaluate product strengths, weaknesses, and needed features; and
- Monitoring trends in life-styles, demographics, fashions, and new technologies to inspire new ideas and forecast potential market opportunities.

Opportunity Statements

Product ideas collected by the opportunity funnel are stated in opportunity statements and recorded in a database. An **opportunity statement** is a brief description of a product development idea. For example, an opportunity statement for a building security system might be, “Use fingerprint readers and a fingerprint database to control building entry and exit gates.”

Opportunity statements for derivative and especially maintenance products may summarize a mass of changes. For example, a new release of a student records system might state, “Upgrade StudentSoft 1.6 to incorporate user interface and platform enhancements and a retention tracking module.”

The AquaLush opportunity statement captures the essential idea that it will use moisture sensors rather than a timer to control irrigation.

Create an irrigation system that uses soil moisture sensors to control the amount of water used.

Figure 3-2-2 AquaLush Opportunity Statement

Evaluate and Prioritize Opportunities

The opportunity funnel output is a list of tens, hundreds, or thousands of product development ideas. An organization must choose the best ideas from the list for further exploration and development. Upper management makes such decisions based on five considerations:

Competitive Strategy—Companies decide how they want to compete in the marketplace. For example, some companies attempt to be technology leaders, others to be low-cost suppliers, and others to provide excellent customer support. Product ideas are evaluated to see how they fit into an organization's competitive strategy.

Market Segmentation—Markets can be divided into different parts, or *segments*, based on customer buying habits and preferences, product needs, and so forth. For example, a college bookstore might divide its market into textbook buyers, researchers, casual readers, and office supply buyers. New product ideas are evaluated with respect to the market segments an organization is interested in selling to and the market coverage of its existing products.

Technology Trajectories—Successful technologies follow a standard trajectory from initial introduction with a few users, through broader adoption and use, to complete acceptance and wide use. New technologies are often difficult to deal with and may attract only intrepid customers, but they can distinguish a product from its competitors. As technologies age, they become easier to work with but may be taken for granted by users. For example, early Web pages had relatively few images, sounds, or animations, but currently very media-intense Web pages are common. Product ideas are evaluated in terms of the trajectories of the technologies they include in light of the capabilities, interests, and future plans of the organization.

Software Reuse—It is much faster and cheaper to develop a new product by reusing parts of existing products than it is to develop the entire product from scratch. Product ideas must be evaluated with regard to (a) the extent that they can be developed from existing assets and (b) the amount of newly developed material written for them that may be reusable in later products. The first consideration is about how easily the new product can be developed, and the second is about how development of the new product will make it easier to develop future products.

Profitability—The profitability of a new product depends on the considerations already discussed, plus several unknown factors such as the eventual cost of developing the product and bringing it to market, the needs and desires of the target market when the product appears, and the competitive products that may appear before the new product is released. Managers must make educated guesses about these factors when judging the relative profitability of potential new products.

Managers must take all these considerations into account while trying to form a well-balanced and complete product line or product portfolio. The

result of these considerations will be a prioritized list of development opportunities that an organization might pursue.

Allocate Resources and Determine Timing

The list of prioritized development opportunities will probably be bigger than the organization can handle, so a careful selection must be made of those that the organization can really afford to pursue. The main consideration in making this selection is how many resources the organization can devote to development. Obviously a resource cannot be committed more than once at a given time, so the timing of resource allocation must also be taken into account.

Various tools (such as Gantt charts) can help explore alternatives. In choosing among alternatives, other considerations come into play, such as what competitors are doing, what technologies and products the market seems ready to accept, and when the product will be ready to release. For example, a high-priority project introducing an important new technology may require commitment of many resources for a long time, while several lower-priority projects may require fewer resources for shorter periods of time. If the big, expensive project is done, then no new products will be released for a while, which may make the company look like it is falling behind its competition. If the smaller projects are done instead, there will be a steady stream of new products, but the new technology in the big project will not be brought into the organization's product line.

Management must decide the best course.

The output of this final analysis and decision-making activity is a product plan listing approved development projects, with expected start and delivery dates, for several years in the future. The product plan describes each product by its opportunity statement. The product plan is revised regularly and guides the launch of various product development projects.

Shortly before launching a project, a document is produced that defines in much greater detail than the product opportunity statement what the project is to accomplish. This document, called a *project mission statement*, is discussed in greater detail in the next section.

Section Summary

- A **product plan** is a list of approved development projects with start and delivery dates. The product plan guides an organization in launching projects. It is revised periodically.
- The project planning activity consists of identifying opportunities, evaluating and prioritizing opportunities, allocating resources, and determining timing.
- An important mechanism for identifying product opportunities is an **opportunity funnel** that collects product development ideas from a wide range of sources in the form of **opportunity statements**.
- Product opportunities must be evaluated and prioritized based on an organization's competitive strategy, product market segments, product technology trajectories, software reuse possibilities, potential profitability, and development capacity.

Review Quiz 3.2

1. What is an opportunity statement?
2. Name three passive and three active opportunity funnel channels.
3. List three competitive strategies different from those in the text.
4. What is a product plan?

3.3 Project Mission Statement

Product Design Process Inputs and Outputs

The generic software product design process introduced in Chapter 2 has one input and one output, as indicated on the activity diagram reproduced in Figure 3-3-1.

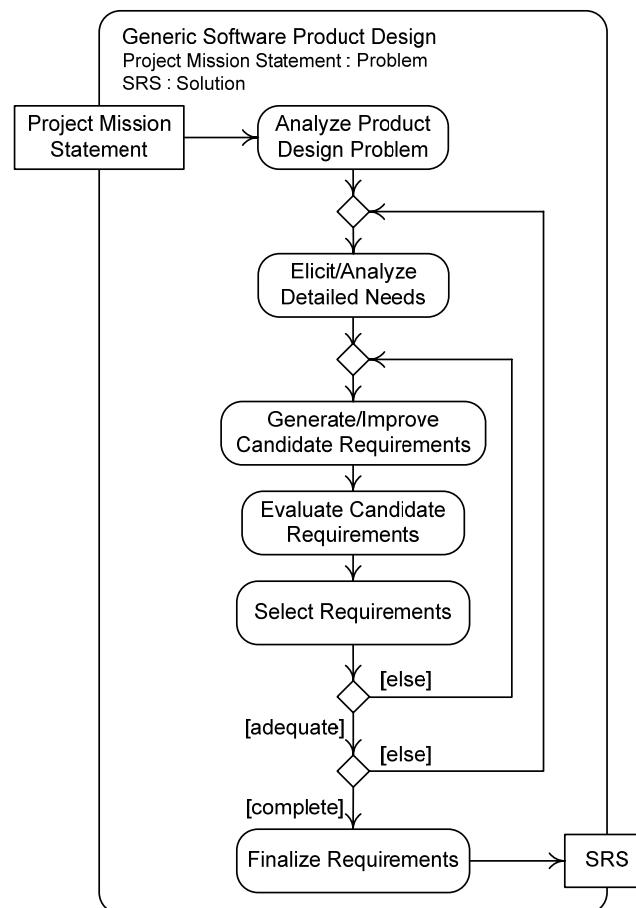


Figure 3-3-1 Generic Software Product Design Process

The input to this process is the project mission statement, and the output is the SRS. The project mission statement is also the main input to the entire development process, and hence the major input to the first activity

in the software life cycle, requirements specification. The SRS is an intermediate result of the software design process, being the major input to the engineering design process and the design life cycle activity.

The remainder of this chapter discusses these software product design inputs and outputs. The project mission statement is discussed in this section and the SRS in the next.

What Is a Project Mission Statement?

What we term the *project mission statement* has many other names, including the *project charter*, *business case document*, *project brief*, and *project vision and scope document*. No matter what its name, the following definition characterizes this document and its contents.

A **project mission statement** is a document that defines a development project's goals and limits.

The project mission statement plays two important roles:

- *Launching a Development Project*—A project mission statement is written when management determines it is time to start a project. A small, cross-functional team elaborates the product opportunity statement and delivers the resulting project mission statement to the development team to direct the team to carry out the project.
- *Stating the Software Design Problem*—The project mission statement documents the design problem; it is the major input to the software design process.

The project mission statement is a crucially important document because it drives the development effort and defines the software design problem. Unfortunately not every organization launches projects with project mission statements. If a project is begun without one, then the developers' first job is to obtain and document the information that should have been in the project mission statement. This information will appear as the first section of the SRS, discussed in the next section.

Although a project mission statement may take many forms, Figure 3-3-2 shows a reasonable outline of its contents. This template structures our discussion in this section.

1. Introduction
2. Product Vision and Project Scope
3. Target Markets
4. Stakeholders
5. Assumptions and Constraints
6. Business Requirements

Figure 3-3-2 Project Mission Statement Template

Throughout this section, we use the AquaLush project mission statement as an illustration. The complete mission statement appears in Appendix B.

Mission Statement Introduction	The introduction to the project mission statement should include any background information about the target product or the project to provide context for understanding the remainder of the document. Typically this includes information about the major business opportunity that the new product will take advantage of and the product's operating environment (the computing platform on which the software product will run).
--------------------------------	---

Figure 3-3-3 shows the AquaLush project mission statement introduction.

Timers regulate most non-agricultural irrigation or sprinkler systems: They release water for a fixed period on a regular basis. This may waste water if the soil is already wet or not provide enough water if the soil is very dry.

MacDougal Electronic Sensor Corporation (MESC) has developed a new soil moisture sensor that can be used to fundamentally change the way that irrigation systems work. Irrigation can be controlled by the soil moisture sensors so that it is skipped or suspended if the soil is sufficiently moist or extended if the soil is too dry. It is expected that this will make more efficient use of water resources as well as making irrigation more effective.

MESC hopes that this change in irrigation systems will spur the sale of its new sensor. MESC would also like to take advantage of the opportunity to sell moisture-controlled irrigation systems.

To take advantage of these two opportunities, MESC has created a new company called Verdant Irrigation Systems (VIS). VIS will develop and market moisture-controlled irrigation systems. The first product to be fielded by VIS is the AquaLush Irrigation System, a demonstration product establishing the viability of moisture-controlled irrigation systems.

Figure 3-3-3 AquaLush Mission Statement Introduction

Product Vision and Project Scope	A product vision statement is a general description of a product's purpose and form. The product vision should establish an overall idea of the product that can be accepted by and perhaps even inspire everyone with an interest in the product. The product vision statement is often a slightly rewritten version of the product opportunity statement that led to the project.
----------------------------------	--

Recall the AquaLush opportunity statement, reproduced in Figure 3-3-4.

Create an irrigation system that uses soil moisture sensors to control the amount of water used.

Figure 3-3-4 AquaLush Opportunity Statement

The AquaLush opportunity statement was rewritten and expanded somewhat into the product vision statement in Figure 3-3-5.

The AquaLush Irrigation System will use soil moisture sensors to control irrigation, thus saving money for customers and making better use of water resources.

Figure 3-3-5 AquaLush Product Vision Statement

The product vision statement may be elaborated by a list of the major product features. These features should be quite general and unobjectionable, and they should not limit the designers in their choice of solutions. Limitations are listed later, in the section on assumptions and constraints. The AquaLush major product features list appears in Figure 3-3-6.

- Monitor water usage and limit usage to amounts set by users.
- Allow users to specify times when irrigation occurs.
- Be operated from a simple central control panel.
- Have a Web-based simulator.

Figure 3-3-6 AquaLush Major Features List

The **project scope** is the work to be done in a project. Creating or greatly modifying a complex product entirely in one release is often not feasible, so projects typically are limited to realizing only part of the product vision. The project mission statement states the portion of the product vision to be addressed in the project, possibly with plans for later releases. It may be necessary to list tasks that are *not* part of the current project, as well as those that are, to make the project scope clear.

For example, the AquaLush project scope is stated in Figure 3-3-7.

The current project will create the minimal hardware and software necessary to field a viable product, along with a Web-based product simulator for marketing the product.

Figure 3-3-7 AquaLush Project Scope

Additional AquaLush product features are relegated to later projects.

Target Markets	Upper management chooses the target market segments for a new product or release during product planning. The developers must know this so they can design and build a product appropriate for its intended users.
----------------	--

The first version of AquaLush is for the largest market segment for non-agricultural irrigation systems: residential and small commercial users. These customers need automated systems to irrigate plots ranging from half an acre to about five acres. Furthermore, the product is intended for both first-time buyers and those who wish to find a more economical alternative to their current timer-based irrigation systems.

Stakeholders A **stakeholder** is anyone affected by a product or involved in or influencing its development. Developers must know who the stakeholders are so that they are all consulted (or at least considered) in designing, building, deploying, and supporting the product. There are surprisingly many stakeholders in any product. First, of course, are the eventual product users. Another important and possibly different group are the product purchasers. On the development side, stakeholders include the managers of the development, marketing, sales, distribution, and product support organizations, as well as the particular individuals directly involved in developing, marketing, selling, distributing, and supporting the product. If a product must comply with laws and regulations (such as safety or health regulations or property laws), regulators, inspectors, and perhaps lawyers may be stakeholders as well. The AquaLush stakeholders are listed in Figure 3-3-8.

Management—The Board of Directors of MacDougal Electronic Sensor Corporation, as the controlling interest in Verdant Irrigation Systems, and the CEO of Verdant Irrigation Systems.

Developers—The four-member AquaLush development team, which includes three software engineers and a mechanical engineer specializing in non-agricultural irrigation systems. Besides developing the product, this team will also support it in the field.

Marketers—The two-person VIS marketing department. There is also a contractor paid by the marketers to develop and maintain the VIS Web site, which will host the AquaLush simulator.

Users—Homeowners, groundskeepers, lawn and garden service professionals, irrigation system professionals, and small business maintenance personnel.

Purchasers—Homeowners, groundskeepers, lawn and garden service professionals, irrigation system professionals, and small business purchasing agents.

Figure 3-3-8 AquaLush Stakeholders

Assumptions and Constraints An **assumption** is something that the developers may take for granted. It is important to make assumptions explicit so that all stakeholders are aware of them and can call them into question. A **constraint** is any factor that limits developers. Of course developers must be aware of all constraints. The difference between assumptions and constraints is subtle and has to do with the difference between problems and solutions. An assumption is a feature of the problem, while a constraint is a restriction on the solution.

The difference between assumptions and constraints on the one hand and requirements on the other is that the former are provided to the designers as input, while the designers produce the latter as output.

To illustrate, an assumption of the AquaLush project is that the product will use the new moisture sensors developed by MESC—this is built into the problem that the developers have to solve. In solving this problem they are compelled to use the same core irrigation control code in both the fielded product and the Web simulation of the product. The developers could conceivably have decided to use different irrigation control software in these two programs, but this constraint rules out that design alternative.

Business Requirements

Business requirements are statements of client or development organization goals that a product must meet. Business requirements usually involve time, cost, quality, and business results. They should always be stated so that it will be clear whether they have been achieved. Usually this means that they list goals in terms of measurable quantities. The business requirements are perhaps the most important part of the project mission statement because they provide challenging, measurable goals for the development team.

Figure 3-3-9 lists examples of AquaLush business requirements.

AquaLush must achieve at least 5% target market share within one year of introduction.
AquaLush must establish base irrigation control technology for use in later products.
AquaLush must demonstrate irrigation cost savings of at least 15% per year over competitive products.
The first version of AquaLush must be brought to market within one year of the development project launch.

Figure 3-3-9 Some AquaLush Business Requirements

Business requirements state what a product must do in broad terms and the goals it must achieve to be successful from a business point of view, but they do not describe the product itself in any detail. Such detailed requirements are generated during software product design.

Section Summary

- A **project mission statement** is a document that defines a development project's goals and limits.
- The project mission statement has two major roles: It launches a development project, and it states the problem that is the input to the software design process.
- The project mission statement should include an introduction, a statement of the product vision and project scope, information about the target markets, as well as stakeholders, assumptions and constraints, and business requirements.

- The introduction states background needed to understand the mission statement.
- The **product vision statement** generally describes the product's purpose and form. It may be supplemented with a list of major features.
- The **project scope** is the work to be done in the project. A project often realizes only part of the product vision.
- **Target markets** are those market segments to which the organization intends to sell the new product. Market segments determine users, features, competitors, and so forth.
- A **stakeholder** is anyone affected by a product or involved in or influencing its development. Stakeholders must be consulted in product design and development.
- An **assumption** is something that the developers may take for granted, while a **constraint** is any factor that limits developers.
- A **business requirement** is a statement of a client or development organization goal that a product must meet. Business requirements set the targets for the developers.

Review Quiz 3.3

1. What is a project mission statement?
 2. What should be in a project mission statement?
 3. List three typical stakeholders in a software development project.
 4. Explain the difference between an assumption and a constraint.
-

3.4 Software Requirements Specification

The SRS as a Process Input and Output

As noted in the previous section, the software requirements specification (SRS) is the main output of the software product design process and the main input to the engineering design process. It is also the main output of the requirements specification activity and the main input to the design activity in the waterfall life cycle model. This section explains the types of requirements statements and the contents and format of the SRS.

First, we will clarify some terminology. Client needs, desires and development constraints always change during development, and they change even more when a product is in use. Requirements specifications must change in response. The job of creating, modifying, and managing requirements over a product's lifetime is called **requirements engineering**. The portion of requirements engineering concerned with initially establishing requirements is termed **requirements development**. The portion of requirements engineering concerned with controlling requirements changes and ensuring that changes are propagated to the remainder of the life cycle is called **requirements management**. We will focus on requirements development in this book (though we call it *software product design*).

SRS Contents

The goal of software product design is to specify software product features, capabilities, and interfaces to satisfy client needs and desires. These specifications are captured in the SRS.

The SRS should contain two kinds of information:

Problem Statement—The first portion of the SRS describes the product design problem. It is based on the project mission statement, if there is one, or developed independently, if there is not.

Product Design—The rest of the SRS records the product design as software requirements.

Requirements are of several sorts and are specified at various levels of abstraction. In this section we will discuss requirement types and levels of abstraction and then present a sample SRS template.

Types of Requirements

The previous section defined business requirements as statements of client or development organization goals that a product must meet. These requirements describe the benefits the business hopes to gain from the product, but they do not describe the product itself in detail. Requirements of the latter sort are called **technical requirements**. These specify a product the designers hope will meet business requirements, but they do not state client or developer goals.

There are three kinds of technical requirements: functional, non-functional, and data requirements.

Functional Requirements

In mathematics, a function is a mechanism for mapping arguments to results. For example, the addition function maps two numeric arguments to a single numeric result, namely the sum of the arguments. Programs work like functions, with inputs acting as arguments and outputs acting as results. Hence, we state the following definition.

A **functional requirement** is a statement of how a software product must map program inputs to program outputs.

Functional requirements are specifications of the product's externally observable behavior, so they are often called **behavioral requirements**. As such they generally specify how a software product must record, display, transform, transmit, or otherwise process data. For example, the following statements are all functional requirements:

- The traffic light control module must switch a green lamp on a traffic light face to an amber lamp on that face, never to a red lamp.
- Upon request from managers, the system must produce daily, weekly, monthly, quarterly, or yearly sales reports in HTML format.
- When a user presses the start round button, the button must be disabled until the end of the round, the round countdown clock must begin, all

round counters must be reset to 0, and the user word entry text box must be cleared and enabled.

Functional requirements are by far the most numerous in most SRS documents.

Non-Functional Requirements The SRS must also describe product characteristics that do not deal with the mapping between inputs and outputs.

A **non-functional requirement** is a statement that a software product must have certain properties.

Non-functional requirements are also called **non-behavioral requirements**. Non-functional requirements include the following sorts of requirements:

Development Requirements—These specify properties essential for meeting the needs and desires of stakeholders in the development organization, such as the product's portability, maintainability, reusability, and testability.

Operational Requirements—These specify product properties important to stakeholders interested in the use of the product, such as performance, response time, memory usage, reliability, and security.

The following statements are examples of non-functional requirements:

- The database component must be replaceable with any commercial database product supporting standard SQL queries.
- The transaction processing engine must be reusable in all products in the product line.
- The system must run without failure for at least 24 hours after being restarted, under normal conditions of use.
- The payroll system must process the payroll for all 32,000 XYZCorp employees in six hours or less.
- The case filer must provide access to a particular case file only to the registered workers for that case file and to level four and above supervisory personnel.

Data Requirements The final category of technical requirements specify the data used in a product.

A **data requirement** is a statement that certain data must be input to, output from, or stored by a product.

Data requirements describe the format, structure, type, and allowable values of data entering, leaving, or stored by the product.

The following statements are examples of data requirements:

- The Computer Assignment System must store customer names in fields recording first, last, and middle names.
- The system must display all times in time fields with the format *hh:mm:ss*, where *hh* is a two-digit military time hour field, *mm* is a two-digit military time minutes field, and *ss* is a two-digit military time seconds field.
- The system must accept product descriptions consisting of arbitrarily formatted ASCII text up to 1,024 characters in length.

Requirements Overview

This requirements taxonomy is summarized in Figure 3-4-1.

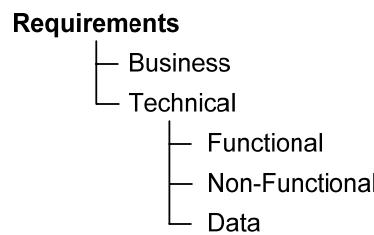


Figure 3-4-1 A Requirements Taxonomy

Business requirements help frame the design problem while technical requirements state the design solution. Every technical requirement should help to meet some business requirement.

Levels of Abstraction in Requirements

Technical requirements are also classified into three levels of abstraction: user-level, operational-level, and physical-level requirements. These levels of abstraction are important for describing the product design process, which generally proceeds from more abstract to less abstract requirements specifications.

User-Level Requirements

User-level requirements are the most abstract kind of requirement and are generally the starting point for refinements that eventually lead to statements that appear in the SRS.

A **user-level requirement** is a statement about how a product must support stakeholders in achieving their goals or tasks.

User-level requirements are stated in terms of broad stakeholder interests. These requirements state how the product will support stakeholders without giving any details about how the product will do the job.

For example, suppose users need a product to monitor some process and notify them when there is a problem. A user-level requirement may state

that a product must collect data from sensors periodically and sound an alarm if readings go out of range.

Operational-Level Requirements

The next level of abstraction in requirements is the operational level.

An **operational-level requirement** is a statement about individual inputs, outputs, computations, operations, calculations, characteristics, and so forth that a product must have or provide, without reference to physical realization.

Specifications at the operational level of abstraction have details about the data that must be stored and processed, the operations that must be performed, and the properties the product must have. These specifications abstract from record formats, user interface mechanisms, and other details about the product's physical form.

For example, a specification at this level of abstraction might include the interaction between the product and a user when adding a record to the system, including the data the user must supply and what happens if it is incorrect. The specification should not include screen layouts, which buttons need to be pressed, and so forth.

Physical-Level Requirements

Physical-level requirements are the least abstract kind of requirement.

A **physical-level requirement** is a statement about the details of the physical form of a product, its physical interface to its environment, or its data formats.

Physical-level requirements provide very specific details about the protocols that the product uses to interact with users and other systems, how it looks, how its data is recorded in various media, and so forth.

For example, user interface layout designs and input and output data record format specifications are physical-level requirements. Product interface specifications mainly consist of such requirements.

Functional, non-functional, and data requirements can be stated at any of these levels of abstraction.

Example: AquaLush Requirements

Table 3-4-2 on page 90 illustrates the different kinds of technical requirements at different levels of abstraction by listing examples from the AquaLush system.

These types and levels of abstraction in requirements specification are important in organizing both the SRS, which we consider next, and the product design process, which we consider in Chapters 4 and 5.

	Functional	Non-Functional	Data
User Level	Irrigation must occur at times set by the user.	The product must operate after a power failure as if it had not taken place.	The program must read a data file describing irrigation zones, their sensors, and their valves.
Operational Level	The program must have a function for setting irrigation time of day in hours and minutes.	Start-up data must be stored in a medium that will retain data without power.	A zone identifier must be a string beginning with "Z" followed by a positive integer value.
Physical Level	The user must set irrigation time-of-day by typing a value into a textbox in military time format.	Start-up data must be stored in an 8-MB Flash card.	Configuration files must be text files with irrigation zone descriptions. The format of this file must be...

Table 3-4-2 AquaLush Technical Requirement Examples at Each Level of Abstraction**Interaction and User Interface Design**

Interaction design is the activity of specifying products that people are able to use effectively and enjoyably. There are two interrelated sub-activities of interaction design:

- Specification of the *dialog* between a product and its users, which is essentially the design of the dynamics of the interaction; and
- Specification of the *physical form* of the product, mainly comprising the static characteristics of its appearance.

Interaction design includes many aspects of product design, including the specification of user-level and operational-level requirements. It can also be restricted in scope to include specification of the physical-level details of a product's form and behavior. The latter is called **user interface design**.

The academic and professional field of *human-computer interaction* (HCI) has traditionally focused on user interface design, while requirements engineers have traditionally left it to HCI specialists. In the last decade, a growing appreciation for the importance of considering the human use of computing products has broadened HCI to include interaction design. There is thus an overlap between the concerns of requirements engineers, who traditionally see their job as the specification of user-level and operational-level requirements, and interaction designers who wish to extend the range of their responsibilities to include product features, functions, and characteristics beyond the user interface.

It is clear that interaction design should be part of requirements development. Human interaction is an essential part of most products and hence must be part of product design. Human interaction concerns must be considered in formulating user- and operational-level requirements if high-quality products are to result. Furthermore, the physical-level requirements constituting a user interface design are refinements of the user- and operational-level requirements created during requirements development. User interface design is thus an extension of traditional requirements development activities. For these reasons, we consider

interaction design to be part of requirements development, and we include user interface design specifications in the SRS.

Extensive consideration of interaction design, or even user interface design, is beyond the scope of this book, although we will introduce two user interface modeling notations in our discussion of state diagrams in Chapter 13. Interaction design is a huge discipline with an enormous literature; standard introductory interaction design or user interface design texts are as large as this book. The section on Further Reading at the end of the chapter lists several such texts.

An SRS Template

Many SRS templates are available in books and on the Internet. Most employ requirements types and levels of abstraction to help organize the material. Templates must be adapted for the product at hand—there is no universal SRS template. A small product can be entirely specified in a single document, but a large product specification is usually broken across several documents. Some products may not have certain types of requirements, while others may distinguish special categories of requirements.

The template in Figure 3-4-3 is adapted from one recommended by the Institute of Electrical and Electronics Engineers (IEEE). It has been changed to emphasize our classification of SRS information.

- 1. Product Description
 - 1.1 Product Vision
 - 1.2 Business Requirements
 - 1.3 Users and Other Stakeholders
 - 1.4 Project Scope
 - 1.5 Assumptions
 - 1.6 Constraints
- 2. Functional Requirements
- 3. Data Requirements
- 4. Non-Functional Requirements
- 5. Interface Requirements
 - 5.1 User Interfaces
 - 5.2 Hardware Interfaces
 - 5.3 Software Interfaces

Figure 3-4-3 An SRS Template

In this template, the design problem is documented in the “Product Description” section, which contains most of the information from the project mission statement. If a project mission statement exists, it should be referenced rather than reproduced.

The product design is documented in the last four sections of the template. The sections named “Functional Requirements,” “Data Requirements,” and “Non-Functional Requirements” contain specifications mainly at the user and operational levels of abstraction. The section entitled “Interface

Requirements” contains physical-level requirements. We discuss the contents of these portions of the SRS in greater detail in later chapters.

Section Summary

- The project mission statement (stating the product design problem) is the main input to the product design process; the SRS (stating the product design solution) is its main output.
- **Requirements engineering** is the activity of creating, modifying, and managing requirements over a product’s lifetime.
- **Requirements development** is creating requirements and is mainly product design.
- **Requirements management** is controlling requirements change and ensuring that requirements are realized.
- The SRS documents the product design problem and its solution.
- **Business requirements** state client and development organization goals, while **technical requirements** state product details.
- Technical requirements categorized by content are **functional**, **non-functional**, or **data** requirements. Technical requirements categorized by level of abstraction are **user-level**, **operational-level**, or **physical-level requirements**.
- Requirements categories help organize SRS documents and the product design process.
- **Interaction design**, the activity of specifying products that people are able to use effectively and enjoyably, is an essential part of product design and hence is part of requirements development.
- SRS templates structure product design documentation but must be adapted to the product’s characteristics.

Review Quiz 3.4

1. Give three examples, different from those in the text, of business requirements.
2. Give examples, different from those in the text, of a user-level functional requirement, a user-level non-functional requirement, and a user-level data requirement.
3. Give examples, different from those in the text, of a user-level functional requirement, an operational-level functional requirement, and a physical-level functional requirement.
4. What is the difference between interaction design and user interface design?
5. List the five main sections of an SRS document, based on the template given in Figure 3-4-3.

Chapter 3 Further Reading

Section 3.1

Products, markets, and marketing are discussed in introductory marketing texts, such as [Kotler 2001].

Section 3.2

Ulrich and Eppinger [2000] provide a general introduction to product design and development. They discuss the product planning process, the opportunity funnel,

evaluating and prioritizing opportunities, and allocating resources and determining timing. The history of VisiCalc is recounted at Dan Bricklin's Web site, www.bricklin.com.

Section 3.3 Ulrich and Eppinger [2000] discuss project mission statement contents, as does Wiegers [2003]. Wiegers also discusses how to establish a product vision and project scope and mission statement contents. Robertson and Robertson [1999] discuss developing the information in the mission statement in an activity they term the "project blastoff."

Section 3.4 Wiegers [2003] discusses requirements engineering in detail. Requirements classifications can be found in [Lauesen 2002], [Robertson and Robertson 1999], and [Wiegers 2003]. Preece and her coauthors have written two excellent books on human-computer interaction and interaction design: [Preece et al. 1994] and [Preece et al. 2001]. [Cooper and Reimann 2003] provide a less academic introduction to interaction design. [Shneiderman and Plaisant 2005] is a popular text on user interface design. The IEEE standard 830 can be purchased from the IEEE and is discussed in many books, such as [Wiegers 2003]. Other SRS templates can be found in [Robertson and Robertson 1999] and [Cockburn 2001].

Chapter 3 Exercises

- Section 3.1**
1. What is a target market? What do target markets have to do with product design?
 2. Use the categories of target market size and technological novelty to classify the following products:
 - (a) Word processor
 - (b) Java compiler
 - (c) GPS-based surveyor's distance measurement tool
 - (d) Terrorism threat assessment tool for the US government
 - (e) Stress-level monitor for heart patients
 - (f) Self-programming universal remote control
- AquaLush
3. How would you classify the AquaLush product in terms of target market size, technological novelty, and product line novelty?
 4. Suppose that you plan to start a Web design business after you graduate.
 - (a) Write an opportunity statement for this business.
 - (b) What competitive strategy might you pursue?
 - (c) How would you segment the market for Web sites? Which market segments might your new company target?
 - (d) Do you know of any software you might use or reuse in this business? What sorts of software assets might you develop over time?
 5. Imagine that you own a software company that makes an expensive Computer Aided Software Engineering (CASE) tool for large-scale corporate development. You are considering developing a product for use in college software engineering courses. Write an essay in which you discuss the things you are thinking about in deciding whether to develop such a product.

- AquaLush 6. Make a product plan for Verdant Irrigation Systems covering the release of a few more products similar to AquaLush over the next several years. Explain how you arrived at your plan.
7. How does competitive strategy influence the types of products that an organization is likely to decide to develop? In particular, how do target market size, technological novelty, and product line novelty figure into these considerations?
- Section 3.3** 8. What is the role of a project mission statement?
9. *Fill in the blanks:* A _____ is a general description of a product's purpose and form. It may be elaborated by _____. The _____ is the work to be done in a project. A _____ is anyone affected by a product or involved in or influencing its development.
- AquaLush 10. Suppose that Verdant Irrigation Systems had decided that AquaLush needed to support timer-controlled watering as well as moisture-controlled watering to be competitive. How would the AquaLush project mission statement be different?
11. Write a short project mission statement for your course work this semester.
12. Suppose we are designing a smart pillbox that dispenses the correct dose of medication when it needs to be taken. Write three business requirements for such a product.
- Section 3.4** 13. Classify the following statements as business (B), functional (F), non-functional (N), or data (D) requirements; for the latter three, also classify them as user-level (U), operational-level (O), or physical-level (P) requirements:
(a) The program must process at least 3,000 calls per hour.
(b) The default credit card choice on the payment form must be "Visa."
(c) The program must notify the operator when the order quantity is greater than the quantity on hand.
(d) The product must be rated as most reliable in published product reviews within a year after release.
(e) In the daily report, the title must be in the form "Daily Report for *dd/mm/yy*," where *dd* is the current day of the month as a two digit number, *mm* is the current month as a two-digit number, and *yy* is the last two digits of the current year.
(f) The product must strap to patients' wrists and measure several bodily functions to monitor and record stress levels.
(g) The daily report of expiring drug batches must list, for each expiring drug, the drug name, manufacturer, batch number, and pharmacy bin number.
(h) The pillbox must record the medication stored in each of its hoppers in a string of 0 to 24 alphanumeric characters.
(i) The product must reduce support costs by 15% in three months.

- (j) Each elevator's default floor parking location must be an integer in the range *lowest-floor* to *highest-floor*.
- (k) An elevator in an idle state must transition to an active state if one or more floors different from the current floor are selected on its control panel, or if a call message is received from the dispatcher.
14. Suppose we are designing a smart pillbox that dispenses the correct dose of medication when it needs to be taken. Write three user-level requirements for such a product.
15. Suppose we are designing a smart pillbox that dispenses the correct dose of medication when it needs to be taken. Write three operational-level requirements for such a product.
16. Suppose we are designing a smart pillbox that dispenses the correct dose of medication when it needs to be taken. Write three physical-level requirements for such a product.
17. Discuss the level of abstraction of business requirements. How do business requirements fit into the technical abstraction levels of user level, operation level, and physical level?
- AquaLush 18. *Find the error:* What is wrong with the following technical requirement: AquaLush must be deliverable through the mail.
19. Your instructor will establish an email account for the course to use as an opportunity funnel input channel. Generate at least three product opportunities, write opportunity statements for them, and submit them to the opportunity funnel. Your ideas should be for products that (a) can be developed by small teams of students in a semester and (b) are interesting to students.
- Team Project** 20. Write a project mission statement for a product opportunity submitted to the class opportunity funnel or one of the following opportunities:
(a) A program to simulate a gas pump with several grades of gasoline, a clerk lock-out (so it will only pump gas after a clerk approves the transaction), and credit card purchase capability.
(b) A program to play a word game, such as Boggle or Scrabble.
(c) An interesting screen saver with some sort of animation.
(d) A simulation of a parking garage.
(e) A simulation of vehicle traffic in a town.
(f) A personal jukebox program.
(g) Software for a smart pillbox that notifies users when it is time for medication and dispenses the right dose.
(h) A personal calendar program that reminds users when they have a scheduled activity.
(i) A program to collect, analyze, and display data needed for the Personal Software Process (PSP). See [Humphrey 1997] for an introduction to the PSP.

Chapter 3 Review Quiz Answers

Review Quiz 3.1

1. The technological novelty category classifies products based on the newness of the technology on which they rely. The types in this category are visionary technology, leading-edge technology, and established technology.
2. The more technologically novel a product is, the riskier it is to design. Visionary technology products may never be built because the technology on which they rely may never be perfected. Visionary and leading-edge technology products use technology that is so new that it is hard to determine whether users will want them, so they may fail in the marketplace even if they succeed technologically. On the other hand, products with new technology tend to be more fun to design, especially since the designers tend to have lots of creative freedom. Products with established technology are much less risky but usually constrain designers more.
3. There are many other categories that might be used. For example, products might be classified according to the application domain (business data processing, systems, scientific and engineering, real time), level of reliability (low, medium, high, very high), level of importance (mission critical, infrastructure support, application, leisure), size (up to 10,000 lines of code; from 10,000 to 100,000; from 100,000 to 1,000,000; more than 1,000,000 lines of code), and so forth.

Review Quiz 3.2

1. An opportunity statement is a brief description of a product development idea.
2. Passive opportunity funnel channels include suggestion lines, bug-report Web pages, and award programs for product ideas. Active opportunity funnel channels include various questionnaires, surveys, focus groups, user studies, bug-report studies, competitive analysis, and monitoring style and preference trends.
3. The three competitive strategies mentioned in the text are being a technology leader, being a low-cost supplier, and providing excellent customer support. Among other strategies are being the highest-quality provider, being the most stylish provider, offering the widest range of products or features, and being the most willing to customize products.
4. A product plan is a list of approved development projects, with start and delivery dates. Note that a *product* plan is very different from a *project* plan. The former is a list of products that an organization intends to develop and the schedule for their development, while the latter is the plan for a particular effort to achieve some goal.

Review Quiz 3.3

1. A project mission statement is a document defining a software development project's goals and limits.
2. A project mission statement should include an introduction, product vision and project scope statements, information about target markets, and lists of stakeholders, assumptions and constraints, and business requirements.
3. Typical stakeholders in a software development project are users; product purchasers; managers of the sales, marketing, development, distribution, and support organizations; and individuals in sales, marketing, development, distribution, and support who will be directly involved with the product.

4. An assumption is something that the developers can take for granted about the development problem; a constraint is something that limits the range of allowable development solutions. Sometimes it can be hard to classify something as an assumption or a constraint. For example, if a product is supposed to be for a PC running Microsoft Windows, is that an assumption or a constraint? Usually it does not matter as long as stakeholders are aware of the constraint or assumption and agree to it.

Review Quiz 3.4

1. The following statements are examples of possible business requirements for the AquaLush product different from those in the text:
 - (a) AquaLush must generate \$1,300,000 of revenue in its first year on the market.
 - (b) AquaLush customer returns must be at a rate less than 2% of sales.
 - (c) AquaLush must generate no more than one customer support call per three products sold during its first year on the market.
2. The following statements are possible AquaLush user-level requirements:
Functional—AquaLush must report failed sensors and irrigation valves to the user.
Non-functional—AquaLush must continue operation as normal even when sensors or irrigation valves fail.
Data—AquaLush must record all types of hardware it uses in the running installation.
3. The following statements are possible AquaLush functional requirements:
User Level—Users must be able to set the critical moisture level for each sensor.
Operational Level—AquaLush must display the current critical moisture level for each sensor and allow users to modify this level.
Physical Level—AquaLush must display the prompt “Sensor X Critical Moisture Level: *n*,” where *n* is a value in the range 0 to 100.
4. Interaction design is the broad discipline of specifying products that users find easy, effective, and fun to use. Interaction design overlaps greatly with product design. User interface design is the narrower discipline concerned with specifying a product’s physical form and the details of its behavior when interacting with users. User interface design is a sub-field of interaction design.
5. An SRS document begins with a product description that states the design problem followed by sections on functional, non-functional, data, and interface specifications that together comprise the product design.

4 Product Design Analysis

Chapter Objectives

This chapter continues discussion of software product design by taking a deeper look at product design analysis as, indicated in Figure 4-O-1.

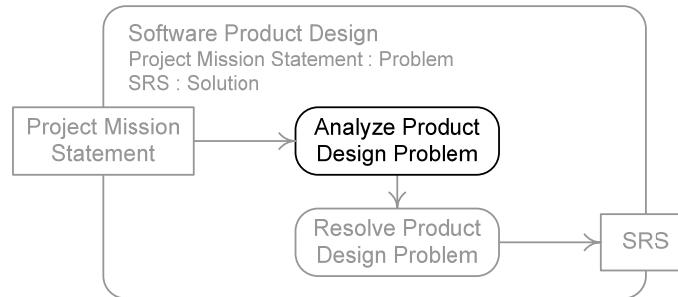


Figure 4-O-1 Software Product Design

In particular, this chapter gives a broad overview of the software product design process and then focuses on analyzing the software product design problem and eliciting, analyzing, and documenting detailed stakeholder needs.

By the end of this chapter you will be able to

- Sketch the software product design process and explain how it is essentially a top-down and user-centered process;
- List and explain several needs elicitation heuristics and techniques;
- Analyze and document needs elicitation results; and
- Check analysis results to ensure that they are correct, within scope, uniform, and complete, and that they use consistent terminology.

Chapter Contents

- 4.1 Product Design Process Overview
 - 4.2 Needs Elicitation
 - 4.3 Needs Documentation and Analysis
-

4.1 Product Design Process Overview

Process Overview

The activity diagram in Figure 4-1-1 shows the software product design process discussed in Chapter 2.

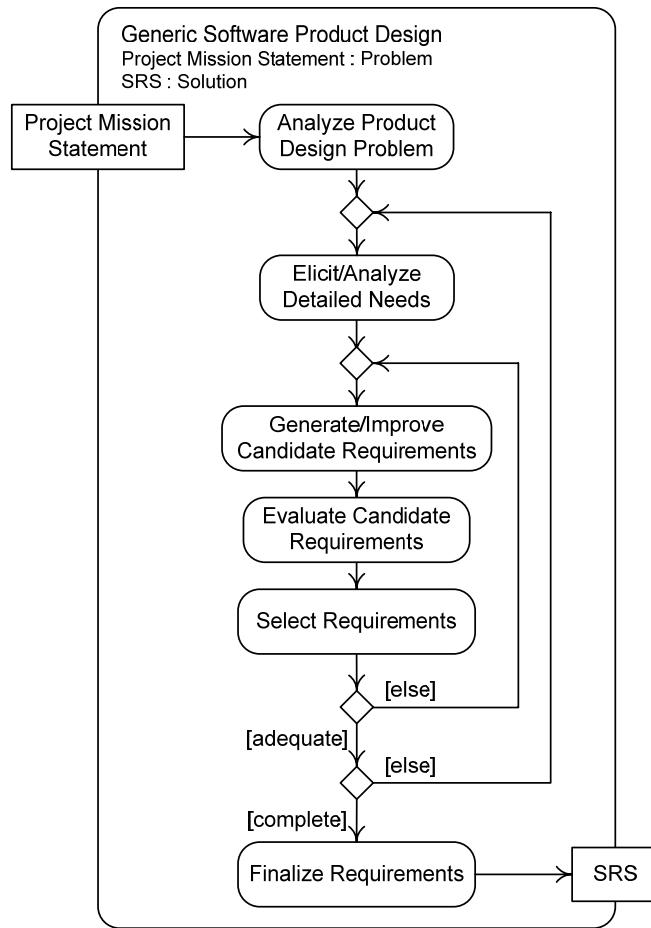


Figure 4-1-1 Generic Software Product Design Process

The first step in this process is to understand the product design problem. The nature of this task depends on whether there is an adequate project mission statement (see Chapter 3). A good project mission statement defines the product design problem, so the designers need only study the mission statement and research any parts of it that they do not understand.

If a project is begun without a good project mission statement, then the designers must discover and document all the missing information. Designers will have to consult the managers sponsoring the project, and probably most of its stakeholders, to get this information. They may also have to study competitive products. The elicitation techniques discussed in this chapter are useful for this work.

The next analysis activity in the process is comprised of eliciting and analyzing detailed needs. The project mission statement contains only business requirements that state client and development organization goals, not details about the product itself. Similarly, the product vision statement

may be supplemented with a product features list, but the items on this list are very abstract. Designers need to learn much more about stakeholder needs and desires, especially those of users and purchasers, to design a product that will meet its business requirements. Section 4.2 discusses needs elicitation techniques, and Section 4.3 discusses needs documentation and analysis.

The resolution activity proceeds by generating and refining requirements, therefore fulfilling the needs determined during analysis. Once alternative requirements are generated and stated, they are evaluated and particular requirements are selected. The last step of the software product design process is to finalize the SRS. Chapter 5 discusses these product design resolution activities and reviews the product design process.

Before turning to a deeper consideration of product design analysis, we stress two points about the product design process: It is a *top-down* and *user-centered* process. We discuss these points next.

A Top-Down Process

Product design resolution sets technical requirements at a high level of abstraction and then refines them until all product details are specified. The outer iteration pictured in Figure 4-1-1 reflects this refinement activity. During this process, user-level needs are elicited and analyzed first, and user-level functional, data, and non-functional requirements are generated, refined, and evaluated until they are adequate. The user-level requirements provide an abstract solution to the design problem. They are then refined to produce operational-level requirements, which in turn are refined to produce physical-level requirements. Refinement is complete when all physical-level functional, non-functional, and data requirements are specified.

This rigid, top-down description of product design analysis and resolution is an ideal. Designers frequently work bottom up or skip levels of abstraction. It is not uncommon for some part of a product design to be specified down to its physical-level details before other parts are specified at all. Needs elicitation must often be repeated at lower levels of abstraction to refine the problem along with its solution. Furthermore, iteration back to an earlier step when mistakes are discovered is inevitable.

Nevertheless, the overall flow of activity during product design resolution is from higher to lower levels of abstraction as product features, functions, and properties are first specified abstractly in terms of user needs and objectives and then gradually refined until all details are worked out. Product design is thus mainly a top-down process.

A User-Centered Process

User-centered design comprises the following three principles:

Stakeholder Focus—Determine the needs and desires of all stakeholders (especially users), and involve them in evaluating the design and perhaps even in generating the design.

Empirical Evaluation—Gather stakeholder needs and desires and assess design quality by collecting data (by questioning or studying stakeholders) rather than by relying on guesses (about what stakeholders need and want and how well design alternatives work).

Iteration—Improve designs repeatedly until they are adequate.

Combining these ideas suggests that iteration in the design process should include steps for empirically determining stakeholders' needs and desires as well as steps for empirical evaluations of proposed designs and final design solutions involving stakeholders.

Collecting stakeholder needs and desires is called *requirements elicitation*, *needs elicitation*, or *needs identification*, and understanding these needs is called *needs analysis* or *requirements analysis*. Confirming with stakeholders that a product design satisfies their needs and desires is called *requirements validation* or just *validation*. At a minimum, stakeholders need to be the source of data collected during needs elicitation and the authority in deciding whether a product design meets their needs in requirements validation.

Stakeholders can be involved in almost every step of product design, either as authorities answering designers' questions, as subjects of empirical studies done by designers, or as partners with designers (called *participatory design*). For example, as subjects of empirical studies, users can try out various design alternatives to see how quickly they can achieve their goals or how many errors they make. The data from such studies can then be used to help decide between design alternatives or to decide whether a design is adequate. As partners in the design process, stakeholders can help generate, improve, evaluate, and select alternatives.

Table 4-1-2 lists the roles that stakeholders can play in the activities comprising a user-driven design process.

Activity	Stakeholders' Role
Analyze Product Design Problem	Clarify project mission statement Answer questions
Elicit Needs	Answer questions Be subjects of empirical studies
Analyze Needs	Answer questions Review and validate models and documents Participate in analysis with designers
Generate/Improve Alternatives	Participate in generation and improvement
Evaluate Alternatives	Answer questions Be subject of empirical studies Participate in evaluation with designers
Select Alternatives	Participate in selection with designers
Finalize Design	Review and validate requirements

Table 4-1-2 Stakeholders' Roles in Product Design

We will discuss the roles of stakeholders in more detail as we consider each of these product design activities.

AquaLush Example

Let's illustrate this process by summarizing some of the work done to create the AquaLush requirements. In reading this narrative, it might help to match the flow of events with the process pictured in Figure 4-1-1.

The development team began by analyzing the product design problem stated in the project mission statement (see Appendix B). The team was already experienced in the irrigation product domain and aware of the new moisture sensor technology, so it readily understood the product design problem.

The development team began work on eliciting and analyzing detailed needs in the context of AquaLush business requirements. They researched competitive products, interviewed the AquaLush CEO, and conducted focus groups with purchasers, installers, and users of competitive products. These activities gave them detailed information and more ideas about stakeholder needs for a moisture-controlled irrigation product. The team analyzed these needs to check, summarize, and document their findings.

The team then conducted brainstorming sessions with use case diagrams (see Chapter 6) to generate potential user-level technical requirements. Clearly AquaLush had to have moisture-controlled irrigation, but a design alternative was to also include traditional timer-controlled irrigation to make the product less risky for customers. More focus groups with prospective purchasers and users helped to evaluate these alternatives. Enough purchasers and users were willing to buy a product without timer-controlled irrigation features that the team decided it could satisfy its business requirements without timer-controlled irrigation. The team, the CEO, and an outside consultant with expertise in irrigation products reviewed the final user-level requirements to validate them. Several omissions and confusions were found and fixed.

The team proceeded to write use case descriptions (see Chapter 6) that refined user-level requirements into operational-level requirements. The team's hardware engineers and irrigation product specialists acted as stand-ins for users in this effort. In several instances multiple use case descriptions were created to explore design alternatives. The hardware engineers and product specialists evaluated the use cases to choose between design alternatives. The team reviewed requirements generated from the use case model to ensure completeness and consistency.

The operational-level requirements were refined to create physical-level requirements for the interfaces to the user, sensors, and irrigation valves. The hardware engineers were consulted about the sensors and valves. After analysis, hardware interface requirements were specified in tables. At this point the team realized they had missed some operational-level data requirements about valve flow rates, and these were added to the use cases and specifications. Note that this iteration back to an earlier stage is not explicitly noted in Figure 4-1-1, but returning to an earlier step is always an

option in any design process. The hardware engineers validated the final version of the physical-level sensor and valve interface requirements by reviewing them.

The team studied competitive products to elicit user interface needs and to harvest good user interface ideas. User interface design alternatives were generated and documented with drawings of user interface screen diagrams and dialog maps (see Chapter 13). Prospective users tried out user interface prototypes (see Chapter 5). The designers asked users to accomplish various things with the prototypes and noted how long it took them and how many errors they made. They also interviewed the users afterwards to see how well they understood the interface and to ask for suggestions. All this information was used to choose between interface design alternatives and improve the user interface design. The team reviewed the final specification and prototype to verify the improved requirements.

Once all requirements were generated, the SRS was assembled, organized, and checked by all team members and several stakeholders to ensure that nothing had been left out, that the correct versions of all work products were present, and so forth. This final review completed the product design process.

Section Summary

- The software product design process begins with design problem analysis. This job can be easy if there is a good project mission statement or quite hard if there is not.
- The product design process is essentially top down, progressing from analysis to resolution and, within resolution, from more to less abstract requirements.
- The product design process is **user-centered**, meaning that it is stakeholder focused, relies on empirical evaluation, and is iterative.
- Stakeholders should provide input in defining the product design problem, stating detailed needs and desires (requirements elicitation), and evaluating candidate design solutions (requirements validation).
- Stakeholders can play many roles during product design, including answering questions, acting as subjects of empirical evaluations, and working with designers.

Review Quiz 4.1

1. Where do business requirements fit into the top-down product design process?
2. Name and explain the three characteristics of a user-centered product design process.
3. What is the difference between requirements elicitation and requirements validation?
4. List three roles that stakeholders can play during product design process activities.

4.2 Needs Elicitation

Needs Versus Requirements

The goal of software design is to specify the nature and composition of a software product that satisfies stakeholder needs and desires. Therefore, stakeholder needs are part of the design *problem*. Requirements, on the other hand, specify a product design and therefore constitute the design *solution*.

Often, statements of needs and requirements are similar. For example, the statement “The user needs to record the sample size, sample readings, and the time the sample was taken” expresses a need, while the statement “The product must record the sample size, sample readings, and the time the sample was taken” expresses a requirement. The similarity of these statements hides the effort that leads from one to the other. It would be nice if needs could simply be collected from stakeholders and then written down as requirements, but unfortunately there is a great deal of design work involved in understanding needs, generating designs to meet these needs, evaluating designs, and selecting a coherent set of requirements specifying an attractive and useful product. Even gathering needs is difficult, as we now discuss.

Needs Elicitation Challenges

Product design begins with business requirements that describe enterprise goals in developing a software product. For example, suppose a catalog sales company wants to increase customer satisfaction by 10% and salesperson productivity by 20% using a product to support sales personnel by providing detailed product, inventory, and delivery information. Product designers must produce technical requirements for a product meeting these business goals. How should they proceed? It seems obvious that designers should first ask stakeholders what they think the product should do and how the product should do it, and then they should make sure they understand what the stakeholders have said. This is the essence of requirements elicitation and analysis.

Unfortunately, simply asking stakeholders “What do you want?” rarely produces useful answers, for the following reasons:

- Needs and desires can only be understood in a larger context that includes understanding the problem domain. For example, designing a product for better catalog sales support requires understanding how catalog sales work and how a particular company takes orders, stores data, interacts with customers, and so forth. Designers must obtain and digest a lot of background information as a basis for their work.
- Certain stakeholders may not be readily available to designers. For example, managers may not want to release personnel from their work to talk with developers. Also, the most important stakeholders for consumer products—the consumers—may be hard to contact and may not be interested in working with designers.

- Unfocused questions usually produce a jumble of responses about different product aspects at different levels of abstraction. For example, stakeholders often answer questions about product features in terms of how they imagine the product's user interface might work, which mixes user-level, operational-level, and physical-level concerns. Investigation of the problem and of stakeholder needs and desires is better done in an orderly fashion.
- Stakeholders are often unable to explain how they do their work, what they want from a product, how they would use it, or even what they do in the absence of the product. Designers must discern needs without relying solely on verbal input.
- Even when they are able to articulate some part of their needs and desires about a product, stakeholders rarely have a clear and coherent understanding of their enterprise or of the potential new product. This is especially true of new products using visionary technology.
- Stakeholders inevitably fail to mention important points about their work and their needs because they forget them or think they are obvious. For example, a stakeholder might go into great detail about the format of product output without mentioning its contents, under the assumption that the designers already understand what data the product must provide.
- Stakeholders often misunderstand the limits and capabilities of technology. They may want something infeasible or fail to mention a need that could be met easily because they think it would be too hard to meet.

For all these reasons, designers must obtain information from stakeholders in a systematic fashion using several elicitation techniques and must document and analyze the results to ensure that needs and desires are understood correctly and completely. This is a very hard job—perhaps the single hardest job in software product development. Designers are faced with a flood of information, often contradictory, incomplete, and confusing, that they must record and make sense of as the basis for a product design. The elicitation and analysis techniques surveyed in this section are powerful tools for this job, but ultimately success depends on hard work and determination.

The main way to organize requirements elicitation is to work from the top down through levels of abstraction. Organization within each level of abstraction is achieved by focusing on particular product aspects, which depend on the product itself.

Elicitation Heuristics

Stakeholders have needs that depend on their activities: It is impossible to understand what stakeholders want or need unless you understand what they do. Consequently, designers have to understand the product problem domain as well as stakeholder needs and desires.

Understanding the problem domain must come first. Hence, we state the following heuristic:

Learn about the problem domain first. If designers don't understand the problem domain, they need to elicit, document, and analyze information about it *before* eliciting needs. For example, only designers who understand how catalog sales work, and in particular how the company in question's catalog sales process works, will be able to design a good catalog sales support system.

Another heuristic that helps elicit stakeholder needs is to concentrate first on figuring out stakeholder goals:

Determine stakeholder goals as the context of stakeholder needs and desires. What a stakeholder needs and wants is a consequence of his or her goals. For example, a user may need a product to record sample data. Why would the user need this? Because the user's goal is to monitor a manufacturing process by sampling and analyzing its output. Knowing goals can help find unstated, assumed, or forgotten needs and desires. For example, given the user's goal, the user might also like the product to help select the sample, remind the user to select the sample, or even take the sample automatically, none of which the user may have explicitly stated as a need.

Once goals are determined, it may also be useful to study the tasks supported or changed by the product. This suggests our third heuristic:

Study user tasks. For example, suppose users currently collect and measure samples by hand, record the data in a log book, use a calculator to compute statistics, enter the results on a paper graph, and study the graph to see if the process is running properly. A product to automate this task needs to provide a way to record sample data, analyze it, graph it, display the graph, and analyze the data to determine whether the process is operating normally. Deeper study of the task may lead to more needs. For example, users may occasionally analyze old data to calculate process yields and look for trends. This additional aspect of the task generates further user needs.

In summary, understanding the problem domain before eliciting needs, determining stakeholder goals as the context for needs analysis, and studying user tasks are useful needs elicitation heuristics.

Elicitation Techniques

There are many requirements elicitation techniques, some quite sophisticated. A technique's effectiveness depends largely on the stakeholders whose needs are sought. For example, interviews target particular individuals and require lots of time. They are most effective for development stakeholders, customers, and users of custom and niche-market products because these stakeholders can be identified and are willing to spend time with designers. Interviews are much less effective for consumer product customers because they are an ill-defined group with little interest in helping designers.

The following techniques are the mainstays of needs elicitation:

Interviews—An **interview** is a question and answer session during which one or more designers ask questions of one or more stakeholders or problem-domain experts. Interviews are the primary means of obtaining verbal input from committed stakeholders, such as development organization stakeholders and customers for custom and niche-market products. There are many ways to conduct interviews. Questions may be prepared beforehand or made up during the interview from a list of issues or concerns. The interview may be taped, or designers may take notes. Designers should always prepare issues or questions before an interview and be careful not to ask leading questions. Answers must be recorded in some way—it may be useful for one designer to act as scribe.

Observation—Many products automate or support work done by people, so designers need to understand how people do their work to design such products. Surprisingly, many people are unable to describe what they do accurately, so it is better to observe them while they work. Designers can watch and take notes or make videotapes. Sometimes subjects are asked to talk out loud as they work, explaining what they are doing and why, which helps designers understand what they are seeing. It is better to observe people who routinely do a task rather than those who do it only occasionally (such as supervisors). Designers should observe the same tasks several times and observe several people doing the same tasks to uncover unusual cases and variations in the way work is done. This technique, called **observation**, should be used whenever a product replaces or supports human workers. Observation is especially useful for eliciting derivative product and maintenance release needs because it can reveal many opportunities for product improvement.

Focus Groups—A **focus group** is an informal discussion among six to nine people led by a facilitator who keeps the group on topic. Requirements elicitation focus groups consist of stakeholders or stakeholder representatives who discuss some aspect of the product. Focus groups are especially useful for eliciting user-level needs and desires. Focus groups are the main technique of obtaining needs for consumer products, especially new products and those with visionary or leading-edge technologies. People may be paid to participate in focus groups to compensate them for their time.

Elicitation Workshops—An **elicitation workshop** is a facilitated and directed discussion aimed at describing the product design problem or establishing stakeholder needs and desires. Elicitation workshops are similar to focus groups but are more tightly controlled, have more precisely defined goals, and require more time and effort from participants. Workshops may be held to describe business processes, list or describe use cases (see Chapter 6), work out data formats, or specify user interface needs. Elicitation workshops are a powerful technique useful for learning about the design problem and determining needs at

all levels of abstraction, but they are appropriate only for committed stakeholders.

Document Studies—Designers can learn about the problem domain, organizational policies and procedures, work processes, and current products from studying documents. For example, designers can often learn about problem domains from books. Most companies have policies and procedures manuals describing how they do business. Often, information about work processes is available from business process reengineering or improvement studies. Existing products typically have a wealth of documentation, including development documentation, bug and problem reports, and suggestions for improvements. Document studies are helpful for learning about a problem domain and for determining requirements for derivative products or maintenance releases.

Competitive Product Studies—If a product will compete with others already on the market, studying the competition helps determine all types of requirements at all levels of abstraction. Product reviews and market studies help identify user-level requirements. Studying the products themselves reveals operational-level and physical-level requirements. Such studies should always be done for consumer products and for any product that will compete with existing products.

Prototype Demonstrations—A **prototype** is a working model of part or all of a final product. Prototypes provide a useful basis for conversations with stakeholders about features, capabilities, and user interface issues such as interaction protocols. We discuss prototypes in detail in Chapter 5. Prototypes are especially useful for products with visionary technology because they help people understand what a product with the new technology will be like.

Designers choose elicitation techniques based on the type of product being developed and the stakeholder whose needs are being elicited. In general, designers should use several techniques, favoring those that allow direct interaction with stakeholders, such as interviews, observation, and focus groups.

Elicitation Heuristics and Techniques Summaries

Figures 4-2-1 and 4-2-2 summarize the elicitation heuristics and techniques discussed in this section.

- Learn about the problem domain first.
- Determine stakeholder goals as the context of stakeholder needs and desires.
- Study user tasks.
- Use several elicitation techniques.

Figure 4-2-1 Elicitation Heuristics

Interviews—Ask stakeholders prepared questions.
Observation—Watch users at work.
Focus Groups—Hold facilitated discussions.
Elicitation Workshops—Hold facilitated directed seminars with specific goals.
Document Studies—Read and analyze relevant documents.
Competitive Product Studies—Analyze competitive products.
Prototype Demonstrations—Use prototypes to stimulate discussion and ferret out unstated needs and desires.

Figure 4-2-2 Elicitation Techniques

- Section Summary**
- Needs and desires are part of the product design problem; requirements state the product design solution.
 - Needs elicitation is hard for many reasons: the problem domain must be understood; stakeholders may not be available; and stakeholders often do not provide organized information, are inarticulate, have limited points of view, give partial information, and misunderstand technology.
 - Designers must understand the problem domain first and then elicit needs and desires.
 - Designers should determine stakeholder goals and study user tasks to help understand and determine needs.
 - Elicitation techniques include **interviews, observation, focus groups, elicitation workshops**, document studies, competitive product studies, and **prototype** demonstrations.
 - Designers should use several elicitation techniques, choosing them with an eye to product and stakeholder characteristics and favoring those that provide direct contact with stakeholders.

- Review Quiz 4.2**
1. Distinguish between needs and requirements.
 2. Why is it a good idea to determine stakeholder goals before eliciting needs?
 3. Name four reasons that designers cannot simply ask stakeholders what they want from a new product.
 4. Name and explain four requirements elicitation techniques.

4.3 Needs Documentation and Analysis

Formulating and Organizing Documentation

Elicitation techniques help designers collect information about the problem domain and stakeholder needs and desires, but they don't help organize, record, and check these findings to solidify and verify understanding. The raw data collected from interviews, observation, focus groups, workshops, competitive studies, and so forth needs to be sorted, stated clearly, and organized.

A first step is to divide the data into two categories: data about the problem domain, and data about stakeholders' goals, needs, and desires.

Documenting the Problem Domain

Data about the problem domain can be further categorized and grouped to form an organized set of notes. A useful tool in understanding the domain is a **problem domain glossary**. Most problem domains have their own terminology that designers must learn. Important facts and relationships come to light in the course of learning a domain's vocabulary.

Data about the stakeholders' organization (if there is one) can be made into an **organization chart**, which is a tree or other hierarchical display of the positions in an organization and the reporting relationships among them. It often includes the names of the individuals filling each position.

UML activity diagrams (see Chapter 2) are useful tools for organizing and documenting problem domain information about business processes or user processes. Data about processes obtained from interviews, observation, focus groups, or document studies can be represented in activity diagrams much better than in text. These process models are useful whether the new product automates part of an existing process or will simply be used in a process.

By the time designers have organized their notes, written a problem domain glossary, made an organization chart, and documented relevant processes in activity diagrams, they usually have a good understanding of the problem domain.

Documenting Goals, Needs, and Desires

Raw data about stakeholders' goals, needs, and desires can be organized into two lists: a stakeholders-goals list and a needs list.

A **stakeholders-goals list** is a catalog of important stakeholder categories and their goals.

For example, Table 4-3-1 shows part of the AquaLush stakeholders-goals list (see Appendix B for the entire list).

Note that stakeholders have been organized into groups or categories. There is usually no reason to distinguish individual stakeholders.

Furthermore, the groups are based on roles, not individuals. For example, it is likely that in most cases the same individual is both an AquaLush Operator and Maintainer, but this may not always be the case, and these roles are logically distinct, with different goals and ultimately different needs.

Stakeholder Category	Goals
Purchasers	Pay the least for a product that meets irrigation needs
	Purchase a product that is cheap to operate
	Purchase a product that is cheap to maintain
Installers	Have a product that is easy and fast to install
Operators	Irrigation can be scheduled to occur at certain times
	Irrigation schedules can be set up and changed quickly
	Irrigation schedules can be set up and changed without having to consult instructions
Maintainers	It is quick and easy to tell when the product is not working properly
	It is quick and easy to track down problems
	It is quick and easy to fix problems
	The product is able to recover from routine failures (such as loss of power or water pressure) by itself
	One sort of failure (such as loss of power or water pressure) does not lead to other failures (such as broken valves or sensors)
	The product and its parts have low failure rates

Table 4-3-1 Excerpt from the AquaLush Stakeholders-Goals List

Needs and
Needs Lists Every stakeholder need should be stated in a need statement and included in a needs list.

A **need statement** documents a single product feature, function, or property needed or desired by one or more stakeholders.

A **needs list** catalogs need statements.

Each need statement should name the stakeholder(s) with the need. The needs documented in the needs list should be stated as specific, positive, declarative sentences. This will often require interpretation of the raw data collected using elicitation techniques. Table 4-3-2 on page 112 shows examples of raw data collected from interviews or focus groups and the need statements extracted from them.

Needs lists should be hierarchically arranged. One technique for forming a hierarchy is to write need statements on cards and then sort the cards into groups. The groups can then be formed into larger groups, and so forth. This process will also catch redundant need statements, which can be eliminated.

Elicited Responses	Need Statements
The usual way to call up product information is for the customer to read the number out of the catalog.	Sales personnel need to retrieve product information using catalog identifiers.
It would be great if I could just click on the product number in the invoice and have the product information pop up in another window.	Sales personnel need to retrieve product information from customer order displays.
We do monthly and quarterly reports where we analyze how often customers request information about products.	Marketing analysts need reports about the frequency with which information is requested about each product.
I need to know how often my people are accessing product information.	Sales managers need reports about the frequency of system use by each salesperson.
Somebody has got to keep this data up-to-date. You know, we change about 20% of our stuff in every catalog.	Technical support personnel need to create, delete, update, retrieve, and display product descriptions in the product description database.
I often have trouble finding things about my account on the current Web site. I've seen sites that keep a big list of links in the left part of the screen, and that works pretty well.	Users need better Web site organization and navigation aids.

Table 4-3-2 Elicited Needs and Need Statements

As an illustration, consider the AquaLush needs list, which is presented in its entirety in Appendix B. The AquaLush needs list is organized using the SRS template headings so that there is an introduction, followed by a section about needs that result in constraints, followed by a section about functional needs, and so forth. Part of the AquaLush needs list is shown in Table 4-3-3.

Constraints	Management, Developers, and Marketers need the first version of AquaLush to be brought to market within one year of the development project launch. (2) Installers need AquaLush software to be configurable using either standard tools or tools supplied with the product. (1)
Functional Needs	Management, Developers, and Marketers need AquaLush to be mainly a moisture-controlled irrigation product. (1) Installers and Maintainers need AquaLush to allow the current time to be set. (1) Operators need AquaLush to allow them to set the moisture levels that control irrigation. (1)
Data Needs	Installers need AquaLush to record the system configuration in a persistent way so that it can be restored after power failures. (2) Maintainers need AquaLush to record the locations of failed hardware components. (3)

Table 4-3-3 Excerpt from the AquaLush Needs List

- Prioritizing Needs Designers must understand the relative importance of needs when selecting between design alternatives to trade off inconsistent needs, concentrate on the most important needs, and so forth. Priorities can be numbers on an arbitrary scale, such as the values one to five (with one being the highest priority), or qualitative ratings, such as high, medium, and low.

Designers can make ratings, but it is better if stakeholders rate needs. Stakeholders should also review the final ratings if possible.

Once assigned, priorities can be added to the needs list. The numbers in parentheses at the ends of the need statements in Table 4-3-3 are priorities on a five-point scale, with one being the highest priority.

Problem Modeling

Various models can be used to represent the problem. Models document the problem, can be reviewed with stakeholders to verify understanding, are subject to various checks for completeness and consistency, and can often be used as the basis for a solution.

Checking Needs Documentation

A major advantage of writing documents and making models is that they can be checked for errors. Among the things that can be checked are the following items:

Correctness—A statement is **correct** if it is contingent and accords with the facts. A statement is *contingent* if it can be true or false. Statements come in various forms. For example, the AquaLush stakeholders-goals list includes Operator as a stakeholder category. The presence of Operator in the list states that an AquaLush Operator is a legitimate product stakeholder. This statement is contingent because it could be true or false. Similarly, the list includes “Irrigation schedules can be set up and changed quickly” as an Operator goal. This listing makes the statement that being able to set up and change irrigation schedules quickly is an Operator goal, which is also a contingent statement.

Scope—A stakeholder goal or need is within the **project scope** if it can be satisfied using the envisioned features, functions, and capabilities of the product created by the project. For example, a potential AquaLush user who wants written reports of water usage has a desire outside the scope of the AquaLush project because AquaLush is not envisioned either to maintain this sort of data, or to have text output devices.

Terminological Consistency—Words used ambiguously (with two or more meanings) cause considerable confusion and may lead to errors. For example, a “schedule” in AquaLush could mean a collection of times when irrigation occurs, a scheme for doing irrigation (such as using a timer or a moisture sensor), or both. Furthermore, using several different terms for the same thing (synonyms) may confuse readers by suggesting that the terms have different meanings. **Terminological consistency** is simply using words with same meaning and always using the same words to refer to a particular thing.

Uniformity—A description has **uniformity** when it treats similar items in similar ways. For example, the need statement “Managers need daily, weekly, monthly, quarterly, and yearly sales reports” is not uniform with the need statement “Clerks need a digital clock displayed in the lower-left corner of the screen showing their elapsed time-on-job to the second,” because they are at much different levels of abstraction.

Completeness—Documentation is **complete** when it contains all relevant material. A stakeholders-goals list missing a stakeholder category is not complete.

A Needs Documentation Checklist Designers should inspect needs documentation before asking stakeholders for reviews. The best way to do this is with reviews that use checklists to find defects. An example of such a checklist is shown in Table 4-3-4.

Correctness
<input type="checkbox"/> Every stakeholder category in the stakeholders-goals list represents a group of legitimate stakeholders. <input type="checkbox"/> Every need statement in the needs list accurately reflects the purported stakeholder's need. <input type="checkbox"/> All need statement priorities are correct.
Scope
<input type="checkbox"/> All stakeholder goals are within the project scope. <input type="checkbox"/> Every stated need is within the project scope.
Terminology
<input type="checkbox"/> Every specialized term is defined in the problem domain glossary. <input type="checkbox"/> Specialized terms are used as defined in the problem domain glossary. <input type="checkbox"/> Terms are used with the same meaning throughout. <input type="checkbox"/> No synonyms are used.
Uniformity
<input type="checkbox"/> All need statements are at similar levels of abstraction. <input type="checkbox"/> Similar items are treated in similar ways.
Completeness
<input type="checkbox"/> Every important stakeholder category is included in the stakeholders-goals list. <input type="checkbox"/> Every relevant stakeholder goal is recorded in the stakeholders-goals list. <input type="checkbox"/> Every stakeholder goal is satisfiable by needs in the needs list. <input type="checkbox"/> Every need in the needs list is necessary to reach some stakeholder's goals. <input type="checkbox"/> Every entity mentioned in the glossary and the needs list is included in the models. <input type="checkbox"/> All needed operations are listed for every entity.

Table 4-3-4 A Sample Needs Checklist

Once the designers have done their best to scrub defects from problem domain and needs documentation, stakeholders must perform a thorough review. Having stakeholders review glossaries, stakeholders-goals lists, needs lists, and models is the best way to ensure that designers really understand the problem.

Section Summary

- The raw data from needs elicitation must be organized and documented to solidify understanding, record findings, and provide a basis for checking.
- Problem-domain data can be organized into notes, a **problem domain glossary**, and an **organization chart**.

- Stakeholder needs data can be organized into a stakeholders-goals list and a needs list.
- A **stakeholders-goals list** catalogs important stakeholder categories and their goals.
- A **need statement** documents a single product feature, function, or property needed or desired by one or more stakeholders; a **needs list** catalogs need statements.
- Raw data elicited about needs must be transformed into need statements that are positive, specific, declarative sentences that name the stakeholders with those needs.
- Needs lists should be hierarchically arranged and needs should be prioritized.
- Models often help understand the design problem and can be checked to verify understanding.
- Designers and stakeholders can check the **correctness, scope, terminological consistency, uniformity, and completeness** of elicitation results in checklist-driven reviews.

**Review
Quiz 4.3**

1. How does a domain glossary help designers?
 2. Name three characteristics of a good need statement.
 3. Name three things that might be checked to help assure the quality of needs elicitation and analysis.
-

Chapter 4 Further Reading

Section 4.1 Product design is a huge area, and our coverage of it in this book is short, so many topics in this rich area have been omitted. The following texts provide a much broader and deeper coverage of product design in general, and software product design in particular.

Ulrich and Eppinger [2000] provide a general introduction to product design and development. Cooper and Reimann [2003] discuss product design from the point of view of user interaction design. Many books cover requirements engineering in depth, including [Lauesen 2002], [Robertson and Robertson 1999], [Thayer and Dorfman 1997], and [Wiegers 2003].

Gould and Lewis [1985] introduced user-centered design; it is discussed further in [Preece, et al. 2002]. Participatory design is discussed in [Winograd 1996] and [Preece, et al. 2002]. Joint Application Design (see [Wood and Silver 1989]) is an early form of participatory design, and Extreme Programming (discussed in [Beck 2000]) incorporates participatory design as one of its tenets.

Section 4.2 Cooper and Reimann [2003] discuss the importance of goals in design, as well as in elicitation. Preece et al. [2002] discuss task description and analysis. Lauesen [2002] and Kiel and Carmel [1995] catalog elicitation techniques and discuss their use. Wiegers [2003] discusses elicitation workshops in depth. Elicitation techniques are also discussed in [Robertson and Robertson 1999].

Section 4.3 Our discussion of analysis techniques is based mainly on [Ulrich and Eppinger 2000] and [Wiegers 2003]. More analysis and checking techniques are discussed in [Wiegers 2003] and [Lauesen 2002].

Chapter 4 Exercises

The following mission statement will be used in the exercises.

Computer Assignment System (CAS)

Introduction: A group of system administrators must keep track of which computers are assigned to computer users in the community they support. Currently this is done by hand, but this is tedious, error prone, and inconvenient. System administrators want to automate this task to ease their workload.

Product Vision: The Computer Assignment System (CAS) will keep track of computers, computer users, and assignments of computers to users; answer queries; and produce reports about users, computers, and assignments.

Project Scope: Developers in the same enterprise will implement CAS. CAS will be the simplest system meeting basic system administrator needs, developed by a small team in a short time.

Target Market: Only the system administrators will use CAS.

Stakeholders: System Administrators, Software Developers, Computer Users, Accountants, and Managers.

Assumptions and Constraints:

- CAS will be accessible over the Internet to authorized users.
- Three people must develop CAS in three months or less.
- CAS must require no more than one person-week per year for maintenance.

Business Requirements:

- CAS must maintain the location, components, operational status, purchase date, and assignment of every computer in the organization.
- CAS must maintain the name, location, and title of every computer user in the organization.
- CAS users must take no more than one minute per transaction, on average, to maintain this information.
- CAS must answer queries about computers, users, and assignments.
- CAS must provide data for quarterly reports sufficient for accountants to compute equipment depreciation in preparing tax returns.

Section 4.1 1. *Fill in the blanks:* The product design process can be characterized as _____ and _____. The former means that

designers elicit needs and generate, improve, evaluate, and select requirements at high levels of _____, and then elicit needs and generate, improve, evaluate, and select requirements at successively lower levels of _____. The latter means that the design process has a _____ focus, uses _____ evaluation, and is highly _____.

2. Write a plan for involving stakeholders in the Computer Assignment System product design process. Indicate which CAS stakeholder groups will be involved in which activities in the process, and document your results in a table.

Section 4.2 3. Make a table with elicitation techniques labeling the columns. Label the rows with the following stakeholders: Development Manager, Maintenance Manager, Niche-Market Product Purchaser, Consumer Product User, New Custom Product Purchaser, and Visionary Technology Consumer Product Customer. Place X marks in table cells to indicate which techniques are appropriate for which stakeholders.

4. What application domain questions need to be answered to understand the CAS problem domain?
5. What processes would it be useful to study in determining CAS stakeholder needs?
6. What elicitation techniques would you use to gather needs and desires for the CAS, and which stakeholders would you use them on?

Section 4.3 7. Make a stakeholders-goals list for the CAS product.
 8. Make a needs list for the CAS product.
 9. Add priorities to the needs list that you made in exercise 8.
 10. Using the review checklist in Table 4-3-4, review the stakeholders-goals and needs lists you made in exercises 7 and 8.

AquaLush 11. *Find the errors:* What is wrong with each of the need statements in Figure 4-E-1?

AquaLush needs to irrigate only at specific times.
 Operators need AquaLush to support manual irrigation by allowing them to turn individual valves on and off, by displaying data about water usage and moisture levels, and by turning off all valves when irrigation is done.
 Installers need AquaLush to recognize the installed valve hardware and configure itself automatically.
 Purchasers need AquaLush to handle up to 32 valves in each irrigation area.

Figure 4-E-1 Erroneous Need Statements for Exercise 11

Team Projects 12. Form a group of three to five students. Discuss your hobbies, extra-curricular activities, and any jobs you have had in the past. From these, choose an activity that might be automated—this will be your target software product. The person with the most knowledge of this activity

will play the roles of various stakeholders with an interest in the product. The remainder of the team will write the following deliverables:

- (a) A brief mission statement (such as the one above for the CAS product)
- (b) An activity diagram describing the process to be automated and other activity diagrams describing any processes into which the product must fit
- (c) A stakeholders-goals list
- (d) A prioritized needs list

Be sure to check your deliverables before submitting them.

13. Form a team of three to five students. Suppose your team has decided to go into business with some sort of consumer software product. Go to an Internet portal where you can look at a range of consumer software products (such as Google or Yahoo) and choose a product category (for example, Diagnostic and Educational Healthcare products). Brainstorm some product opportunities and select one that looks interesting. Write a mission statement and study competitive product Web pages to elicit needs. Your deliverables should include the following items:

- (a) A mission statement
- (b) Brief synopses of the competing products you studied
- (c) A stakeholder-goals list or a prioritized needs list

Be sure to check your deliverables before submitting them.

Chapter 4 Review Quiz Answers

Review Quiz 4.1

1. Business requirements state the stakeholder and developer goals for the design effort, so they frame (part of) the design problem solved by the product design process. They are therefore the launching point for the top-down product design process.
2. The three characteristics of a user-centered product design process are stakeholder focus (stakeholders are involved in design analysis, design evaluation, and possibly design selection); empirical evaluation (surveys and observation are used to collect data that is the basis of decision making); and iteration (designs are improved repeatedly until they are adequate).
3. Requirements elicitation is determining stakeholder needs and desires prior to generating design solutions, while requirements validation is ensuring that a design meets stakeholder needs and desires once it is complete.
4. Stakeholders can have many roles during product design process activities, including answering questions during needs elicitation and analysis, serving as subjects of empirical evaluations during needs elicitation and design alternative evaluation, reviewing and validating product design documentation generated during needs analysis and design finalization, and working as partners with designers during design alternative generation, evaluation, and selection.

**Review
Quiz 4.2**

1. A need is some product feature, function, or capability that a stakeholder wants in order to achieve one or more of his or her goals; a requirement is a statement generated by designers identifying some feature, function, or capability that a product must have. Needs (partly) specify the product design problem, while requirements specify the design solution.
2. Stakeholder needs usually make sense only in the context of stakeholder goals, so it is easier to elicit needs after goals are determined.
3. Designers cannot simply ask stakeholders what they want from a new product for the following reasons: stakeholder needs and desires can be understood only in the context of the problem domain and the stakeholder's organization; some stakeholders may not be available to interview or observe; unfocused questions usually produce jumbled responses; stakeholders are often unable to articulate their needs and desires; stakeholders often do not have a firm understanding of their organization, its processes, policies, and so forth, or of the product that is under development; stakeholders may forget things or fail to mention them because they seem obvious; and stakeholders often misunderstand the limits and capabilities of technology.
4. Requirements elicitation techniques include interviews, observation, focus groups, elicitation workshops, document studies, competitive product studies, and prototype demonstrations.

**Review
Quiz 4.3**

1. Many domains have specialized terminology that designers must learn if they are to understand the problem domain and stakeholder goals, wants, and needs. Also, learning domain terminology often brings to light facts and relationships that help designers learn about the problem domain.
2. A good need statement should have the following characteristics: state a single needed or desired product feature, function, or property; be a positive, specific, simple declarative sentence; name the stakeholders with the need; and be prioritized.
3. The quality of needs elicitation and analysis can be assured by checking the following items: all stakeholder goals are within the project scope; every stated need is within the project scope; all terms used in analysis documents are used as defined in the problem domain glossary; things are always referred to by the same terms; similar entities are treated in similar ways in the needs list and analysis models; every important stakeholder category is included in the stakeholders-goals list; every relevant stakeholder goal is recorded in the stakeholders-goals list; every stakeholder goal is satisfiable by needs in the needs list; every need in the needs list is necessary to reach some stakeholder's goal; every entity mentioned in the glossary and the needs list is included in the analysis models; and all needed operations are listed for every entity.

5 Product Design Resolution

Chapter Objectives

This chapter continues discussion of the software product design process by taking a closer look at product design resolution, as shown in Figure 5-O-1. The chapter closes with a discussion of modeling, and particularly prototyping, as an especially valuable tool in product design.

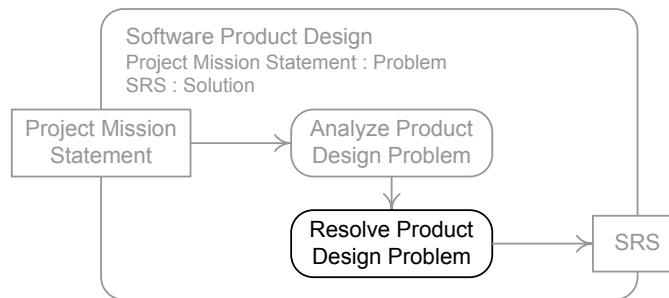


Figure 5-O-1 Software Product Design

By the end of this chapter you will be able to

- Take advantage of several sources of design ideas and use techniques for generating alternative requirements;
- List requirements specification notations and their advantages and disadvantages;
- State requirements using accepted conventions and heuristics;
- List and explain techniques for evaluating and selecting among alternative requirements;
- State the goals of product design finalization and define SRS quality characteristics;
- Differentiate review types and explain how to conduct requirements inspections;
- Explain the role of modeling in product design; and
- List kinds of prototypes and their uses in product design.

Chapter Contents

- 5.1 Generating Alternative Requirements
- 5.2 Stating Requirements
- 5.3 Evaluating and Selecting Alternatives
- 5.4 Finalizing a Product Design
- 5.5 Prototyping

5.1 Generating Alternative Requirements

Process Context

Chapter 4 discussed software product design analysis; this chapter discusses software product design resolution. The software product design process from Chapter 2 is reproduced in Figure 5-1-1 to refresh your memory.

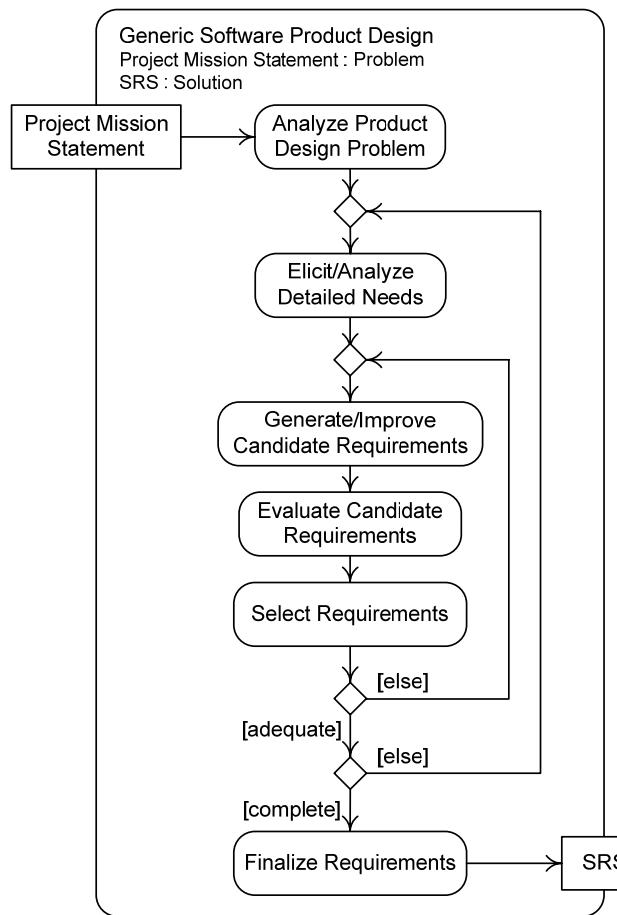


Figure 5-1-1 Generic Software Product Design Process

This chapter looks in detail at generating and improving candidate requirements, evaluating them, selecting candidates for further improvement or inclusion in the product design, and design finalization.

Generating Candidate Requirements

Once the designers understand stakeholder needs, they must create a solution. This process begins with alternative requirement generation. Common failings at this stage are considering only a few alternatives and missing entire categories of alternatives.

These failings reflect an apparent lack of creativity, but how can one become more creative? In answering this question, we first observe that creativity may be overrated as essential in design. Although occasionally people have truly original ideas, most designs copy ideas from somewhere else, and many alternative requirements can be generated by studying existing products. When some creativity is required, most people are able to be more creative than they expect they can be. There are ways to spark new ideas, and given time and the proper techniques, designers can usually come up with many good ideas.

We will now consider techniques for generating alternative requirements.

Idea Sources and Generation Techniques

Alternative requirements can come from outside the design team or within the design team. There are several external sources for design ideas:

Users and Other Stakeholders—Various stakeholders, especially users, often have interesting ideas about product design because they have been thinking about what they would like to see in a product for a long time and they have deep problem domain knowledge.

Experts—People who know a lot about problem domains often have good product ideas.

Props and Metaphors—Many good software product ideas are based on items from the real world. For example, a designer might conceive of a network design program as being like a Lego® set with pieces that snap together or a music playlist management program as being like a Rolodex®. Physical props can foster new ideas.

Competitive Products—Designers can study the competition to see what is done well enough to copy and what is done poorly enough to modify.

Similar Products—Many products from other domains are often similar enough that some of their good ideas can be reused. For example, many products display grids of values like those in a spreadsheet, though they may do things with these values that are completely unlike those performed by a spreadsheet. These products can incorporate valuable ideas from spreadsheets.

Design ideas can be generated within the team in several ways:

Team Brainstorming—The design team can have sessions devoted to brainstorming alternative requirements. No censoring of ideas should occur in these meetings, with even infeasible or silly ideas added to the list. Such sessions often spark good ideas or provide ideas for later improvement.

Individual Brainstorming—Studies have shown that individuals generate more and better ideas working alone than in groups, so team members should devote individual work time to generating alternative requirements. Individuals should do at least some of their brainstorming *before* attending any team brainstorming sessions to encourage each team member to take a fresh approach to the problem.

Modeling—Models are important for laying out, investigating, modifying, and documenting alternative requirements to communicate to other designers and for later evaluation and improvement. Most modeling techniques discussed later in this book can be used in product design resolution. In particular, prototyping is discussed in the third section of this chapter as product design tool, and use case modeling is introduced in Chapter 6 as a technique for product design resolution.

Improving Candidate Requirements	The generation and improvement step of product design resolution consists of either making up new candidate requirements or improving or refining existing candidate requirements. The starting point for candidate improvement is identifying one or more stakeholder need statements and candidate requirements that must be improved or refined (that is, made less abstract by adding details). The goal is to improve or refine the candidate requirements in accord with stakeholder needs and desires.
---	---

Refining Aqualush Requirements	To illustrate, consider the Aqualush user-level requirement that users must set irrigation parameters. Needs elicitation reveals that the parameters that need to be set include the irrigation start time and frequency, the water allocation for each irrigation cycle, and the moisture level controlling irrigation. The goal is to generate operational-level requirements for setting these parameters.
---------------------------------------	---

Setting the start time is fairly straightforward: The product must allow the user to set a time of the day for starting irrigation cycles. There are many alternatives for how irrigation frequency might be set. Possible frequency settings include daily, every other day, three days a week, weekdays, weekends, weekly on a certain day, every n^{th} day, some subset of days of the week, randomly, explicitly choosing dates from a calendar, and no doubt many others. Which of these should the product accommodate? In this case, each subset of possible frequency settings constitutes an alternative requirement. Table 5-1-2 illustrates some alternatives.

The product must allow the user to set irrigation frequency to be daily, every other day, three days per week, weekdays, or weekends.
Irrigation occurs every n^{th} day and the product must allow the user to set n in the range 1 to 14.
The product must allow the user to choose whether irrigation should occur every n^{th} day, on days chosen from a calendar, or randomly. If the first, the product must allow the user to set n in the range 1 to 14. If the second, it must allow the user to choose specific dates from a calendar for the current year. If the third, it must allow the user to set a range for random selection between 1 and 14.
The product must allow the user to choose a subset of the days of the week when irrigation should occur.

Table 5-1-2 Frequency Setting Candidate Requirements

Setting the water allocation also has some interesting alternatives. The idea is that each irrigation cycle be limited to consuming a certain amount of water to control costs and conserve resources. The product keeps track of each valve's flow rate to monitor water usage during irrigation. There are many ways that water might be allocated, and these affect the way that allocation parameters might be set. For example, each irrigation zone might get an equal share, or some zones might get more and some zones less in fixed proportions. Alternatively, the product could check the moisture sensors for each irrigation zone and apportion water according to need: The driest zones would thus get more than the wetter zones. Apportioning could occur before an irrigation cycle begins, or it could be set or modified during irrigation. Once one or more allocation policies are decided, there is the question of how much control of water allocation to give to the user. At one extreme, the user might simply set the overall allocation and leave it to the system to decide how to spread the water around. At the other extreme, the user could control the formulas used to apportion water to each valve.

All these decisions (which refine the user-level requirement that the product must limit water used in irrigation to an allocated amount) affect alternatives about how the user can specify water allocation parameters. Table 5-1-3 lists some alternatives for setting water allocation parameters.

The product must allow the user to set the overall water allocation for an irrigation cycle.
The product must allow the user to set the overall water allocation and the percentage of the overall allocation allotted to each zone.
The product must allow the user to set the overall water allocation and the percentage of the overall allocation allotted to each valve in each zone.
The product must allow the user to set the overall water allocation and the policy used to determine the percentage of the overall allocation for each zone.

Table 5-1-3 Water Allocation Setting Candidate Requirements

Finally, setting the moisture level also leads to several considerations. The first is deciding on units for specifying moisture level. A scale (for example, percent of saturation) could be used, or categories (such as parched, dry, moist, wet, or saturated) could be defined. A second consideration is that the desired moisture level could be a single critical value or several values. In the first case, any moisture reading above the critical value is wet enough, and anything below is not wet enough. In the case of several values, readings at various points on the scale could indicate degree of need for irrigation, influencing decisions about how much water to allocate for each zone. Additionally, a single moisture level might be used for an entire site, or each irrigation zone could have its own moisture setting.

There are also alternatives about the way irrigation depends on measured moisture levels. Irrigation could be stopped when a moisture sensor indicates that a target moisture level has been reached, or it could continue for some time to help ensure that the target moisture level read by the sensor is really the minimal level in the irrigation zone. Table 5-1-4 illustrates some alternatives. These all assume that moisture levels are specified as a percent of saturation.

The product must allow the user to set a single critical moisture level value for all zones.
The product must allow the user to set a critical moisture level value for each irrigation zone.
The product must allow the user to set a single critical moisture level value and the length of time to continue irrigation after the critical moisture level is reached, in the range one to five minutes.
The product must allow the user to set a critical moisture level value for each irrigation zone and a single length of time to continue irrigation after the critical moisture level is reached (in the range one to five minutes) that applies to all zones.
The product must allow the user to set critical moisture levels for light irrigation, medium irrigation, and heavy irrigation. The light irrigation level must be less than or equal to the medium irrigation level, which must be less than or equal to the heavy irrigation level.

Table 5-1-4 Moisture Level Setting Candidate Requirements

Section Summary

- Good designs depend on the generation of multiple alternative requirements, which is often not well done.
- Techniques for generating alternative requirements from sources outside the design team include consulting with users and problem-domain experts, using props and metaphors, and studying competitive and similar products.
- Techniques for generating alternative requirements within a design team include team brainstorming, individual brainstorming, and modeling.

Review Quiz 5.1

1. How does alternative requirement generation often fail?
2. How can studying competitive products help generate alternative requirements?
3. Which is more effective, team brainstorming or individual brainstorming?

5.2 Stating Requirements

Specification Notations

Once requirements are generated, they must be stated. The most widely used notation for recording software requirements specifications is natural language—plain English. The greatest advantage of stating requirements in natural language is that everyone understands it. There is also excellent tool support for it—a word processor. The greatest disadvantage of natural language is that it is not precise.

For example, suppose a requirement states that “Operation A will occur and operation B will occur, or operation C will occur.” This specification leaves open the following questions:

- Must operation A occur before operation B, or does their order not matter?
- Must operation C occur only if neither A nor B occurs, or should C occur if either one of A or B does not occur?
- May operation C occur even if both A and B occur?

This sort of imprecision propagated over an entire SRS provides an enormous source of misunderstanding and error.

Another drawback of natural language is that it is not effective for specifying complicated relationships, especially spatial and temporal relationships. For example, it is very difficult to describe screen layouts in English and awkward to detail all the steps in a process. Graphical notations are better than natural language for such tasks.

Alternatives to natural language include a large collection of semi-formal and formal notations:

Semi-Formal Notations—These notations are more precise and concise than natural language, but they are not defined with mathematical rigor and precision. These include most graphical notations used in this book, such as class diagrams, tables, and pictures. Semi-formal notations avoid much of the imprecision of natural language, are better at describing complex relationships, and are usually fairly easy for most people to understand, especially when accompanied by notes and explanations.

Formal Notations—These notations are defined with mathematical rigor and precision. Examples of such notations are mathematical and logical notations, such as set notations and first-order logic; some graphical notations, such as state diagrams; and notations invented especially for specification, such as Z. Formal notations completely avoid the imprecision of natural language, but they are difficult to learn and use and inappropriate for communication with most stakeholders.

Use of formal notations is growing among software professionals, and they may eventually become the standard for stating requirements for use by professional software developers. For now, however, the standard remains

a mixture of natural language and semi-formal graphical notations with an occasional equation or two.

Stating Requirements

Requirements specifications are read and consulted over and over again, so they must be written and laid out on the page as clearly and concisely as possible. They should follow the rules of good technical writing:

- Write complete, simple sentences in the active voice.*
- Define terms clearly and use them consistently.*
- Use the same word for a particular concept—that is, avoid synonyms.*
- Group related material into sections.*
- Provide a table of contents and perhaps an index.*
- Use tables, lists, indentation, white space, and other formatting aids to present and organize material clearly and concisely.*
- Leave margins ragged on the right, and use a kerned, medium-sized font.*

In addition to rules of good writing and layout, there are a few heuristics especially for writing software requirements, which we consider in the next few subsections.

"Must" and "Shall"

Recall that a software product requirement is a statement that a software product *must* have a certain feature, function, capability, or property. It is therefore conventional to state all requirements using the words “must” or “shall.” We can state this as a heuristic:

- Express all requirements using the words “must” or “shall.”*

For example, a requirement should be expressed in one of the following ways:

The product must display all results to three decimal places.

The product shall display all results to three decimal places.

It is not correct to express such a requirement using other auxiliary verbs or no auxiliary verb at all, as in the following examples:

- * The product will display all results to three decimal places.
- * The product should display all results to three decimal places.
- * The product displays all results to three decimal places.

Here and below, the * indicates that these are not acceptable expressions of requirements.

Verifiable Specifications

A specification is **verifiable** if there is a definitive procedure to determine whether it is met. Sometimes this property is referred to as a requirement’s **testability**. The following statements are examples of requirements that are not verifiable.

- * The product must produce reports in an acceptable amount of time.
- * The product must respond to human users quickly.

- * The product user interface must be user-friendly.
- * The product must control several drill presses.

In each case, there is no way to tell decisively whether the requirement is satisfied. The following examples show how the previous requirements might be made verifiable:

The product must produce reports in five minutes or less from the time the report is requested.

The product must respond to human users in one second or less.

Eighty percent of first-time users must be able to formulate and enter a simple query within two minutes of starting to use the program.

The product must control up to seven drill presses concurrently.

Verifiability is crucial for all stakeholders. Verifiable requirements are specific and hence capture more exactly what the product must do to satisfy a client's needs and desires. They also give engineering designers, implementers, testers, and other developers the details they need to do their jobs. Hence, an important requirements writing heuristic is

Write verifiable requirements.

Requirements Atomization

Requirements must be realized through engineering design and implementation, and products must be reviewed and tested to ensure that their requirements are met. The ability to track requirements from their expression in an SRS to their realization in engineering design documentation, source code, and user documentation and their verification in reviews and tests is called **requirements traceability**. Requirements traceability is a fundamental software quality assurance capability.

Requirements traceability depends on being able to isolate and identify individual requirements. Isolating individual requirements makes it possible to associate parts of a design, some code, a test case, and so forth with just the relevant portions of an SRS. Associating individual requirements with an identifier simplifies making links to relevant requirements. We call requirements statements that meet these traceability needs *atomic*.

A requirements statement is **atomic** if it states a single product function, feature, characteristic, or property, and it has a unique identifier.

Splitting requirements statements until each one is atomic is called **atomizing requirements**.

Consider the requirements in Figure 5-2-1 from an SRS for a product to help technical support staff track computers and their assignments to employees.

Staff must be able to add computers to the tracking system. When a computer is added, the tracking system must require the staff member to specify its type and allow the staff member to provide a description. Both these fields must be text of length greater than 0 and less than 512 characters. The tracking system must respond with a unique serial number required for all further interactions with the tracking system about the added computer.

Figure 5-2-1 A Requirements Specification Example

These paragraphs contain many individual requirements and they are not numbered. Tracing these requirements as stated would be difficult. The atomized version of these requirements appears in Figure 5-2-2.

1. The tracking system must allow staff to add computers to the tracking system.
 - 1.1 When a computer is added to the tracking system, it must require the staff member to provide type data for the added computer.
 - 1.1.1 A computer's type data must be text of length greater than 0 and less than 512 characters.
 - 1.2 When a computer is added to the tracking system, the tracking system must allow the staff member to provide description data for the added computer.
 - 1.2.1 A computer's description data must be text of length greater than 0 and less than 512 characters.
 - 1.3 The tracking system must respond to added computer input with a unique serial number identifying the computer in the tracking system.
 - 1.4 All further interactions between staff members and the tracking system about a computer must use the unique serial number assigned by the tracking system.

Figure 5-2-2 Some Atomized Requirements

The atomized requirements are simple statements of single requirements, each numbered for identification and to show relationships to other requirements. The atomized requirements are much easier to trace through the development process.

Atomization Practices	Atomizing requirements lays the foundation for requirements traceability. Each labeled statement should express a single requirement. Operational- and physical-level atomized requirements will usually be verified by single test cases and implemented by fairly small parts of the design and code. As a rule of thumb, atomic requirements statements are expressed in simple declarative sentences rather than compound sentences, lists, or paragraphs. Although any scheme for generating unique requirements identifiers will do, a hierarchical numbering scheme works well. Enumerate main requirements with whole numbers, related sub-requirements with a dotted
------------------------------	---

extension (1.1, 1.2, etc.), related sub-sub-requirements with another dotted extension (1.1.1, 1.1.2, etc.), and so on. This scheme is simple, shows connections between related requirements, and is infinitely expandable.

Non-natural-language specifications, such as equations, tables, trees, and diagrams, should be unchanged but included in the numbering scheme so that they can be cited like any other requirements.

We summarize these considerations in the following atomization heuristic:

Atomize requirements by stating each requirement in a numbered simple declarative sentence or in a numbered equation, tree, table, or diagram.

Heuristics Summary

Figure 5-2-3 summarizes our requirements specification heuristics.

- State requirements using formal or semi-formal notations when possible.
- Write complete, simple sentences in the active voice.
- Define terms clearly and use them consistently.
- Avoid synonyms.
- Group related material into sections.
- Provide a table of contents and perhaps an index.
- Use tables, lists, indentation, white space, and other formatting aids to present and organize material clearly and concisely.
- Leave margins ragged on the right and use a kerned, medium-sized font.
- Express all requirements using the words “must” or “shall.”
- Write verifiable requirements.
- Atomize requirements by stating each requirement in a numbered simple declarative sentence or in a numbered equation, tree, table, or diagram.

Figure 5-2-3 Requirements Specification Heuristics

Section Summary

- Most requirements are expressed in plain English, but natural language is vague, ambiguous, and imprecise.
- Semi-formal notations (diagrams, tables, etc.) and formal notations (logic and mathematical notations) are more precise than natural language but harder to read.
- English requirements should be written in clear, precise, and simple technical prose.
- Requirements should be expressed in sentences using the auxiliary verbs “must” or “shall.”
- Requirements should be **verifiable**.
- **Requirements traceability** is the ability to track requirements from their statement in an SRS to their realization in design, code, and user documentation and their verification in reviews and tests.

- **Atomized** requirements have unique identifiers and state a single product function, feature, characteristic, or property.
- Requirements traceability is practical only when requirements are atomized.

**Review
Quiz 5.2**

1. Give an example, different from those in the text, of a natural language requirements statement that is vague, ambiguous, or imprecise.
 2. Which of the following auxiliary verbs are conventionally used in stating requirements: “will,” “shall,” “must,” “should”?
 3. Give an example, different from those in the text, of an unverifiable requirement.
 4. Define requirements traceability.
 5. Name three advantages of atomizing requirements.
-

5.3 Evaluating and Selecting Alternatives

Evaluation and Selection Challenges

The alternative requirements generated in light of stakeholder needs and desires must be evaluated, and then the best alternatives must be selected for further improvement or inclusion in the final product design.

Alternative requirement evaluation should take several considerations into account, including the degree to which the alternative meets stakeholder needs, the priority of the needs met, and the quality of the alternative with respect to basic principles of product design.

Selection among alternatives may involve trade-offs. Different stakeholders usually have different and conflicting goals, needs, and desires, so designers must decide whose needs to meet when selecting requirements. Often a design will be good at meeting some needs but not others, even when these needs do not directly conflict. Furthermore, an alternative may do a good job of meeting stakeholder needs, but be expensive, risky, complicated, or ugly, or have some other drawback.

Evaluation Techniques

Alternative requirements do not have to be evaluated in absolute terms but only relative to one another. Various comparative evaluations can be generated using several techniques.

Two tenets of user-centered design are stakeholder focus and empirical evaluation. *Stakeholder focus* means that stakeholders should be consulted in evaluating alternative requirements, and *empirical evaluation* means that data should be collected and used to evaluate them. Stakeholders can be involved in evaluating design alternatives in two ways:

Stakeholder Surveys—Stakeholders are asked to rate various alternative requirements.

Usability Studies—Users are asked to perform various tasks with prototypes that realize various design alternatives. Measurements are taken to gauge each alternative’s effectiveness.

Both evaluation techniques are expensive, so they can only be used sparingly. This means that designers must evaluate many alternatives without stakeholder input and narrow down the alternatives that will eventually be submitted to stakeholders for evaluation.

Candidate requirements should also be evaluated in terms of the degree to which they satisfy the following basic principles of good product design:

Adequacy—Designs that meet more stakeholder needs, subject to constraints, are better.

Beauty—Beautiful designs are better.

Economy—Designs that can be built for less money, in less time, with less risk are better.

Feasibility—A design is acceptable only if it can be realized.

Simplicity—Simpler designs are better.

Alternative requirements can be rated against one another, with one alternative used as the baseline and the others evaluated using either a numeric scale, or simple quantitative ratings, such as “better,” “worse,” and “equal.”

Cost and time estimation is notoriously difficult in software development (see Chapter 2). However, even rough relative cost and time estimates (for example, alternative A is twice as expensive and will take half again as long as alternative B) can help select between alternative requirements.

Selection Decision Making

Candidate requirement selection is the crucial step in forming a design. The parties that can make these decisions are the stakeholders, the designers, or both:

Stakeholder Selection—Responsibility for selecting among alternative requirements may be turned over to users, clients, managers, product champions, or other stakeholders. Stakeholder selection of product concepts or overarching alternative requirements is appropriate when designers have documented a few leading alternatives with their costs and benefits and presented them clearly to stakeholders.

Designer Selection—Designers typically make most low-level design decisions. Ideally, designers engaged in a user-centered design process base their decisions on stakeholder needs and desires and, where possible, on empirical evidence.

Stakeholder Participation—In a participatory design process stakeholders and designers work as partners, so stakeholders share authority with designers in making design decisions.

In all cases, the design decision makers can use the selection techniques discussed next.

Selection Techniques

There are many techniques for choosing between alternative requirements:

Pros and Cons—Selectors can list each alternative's advantages and disadvantages and make a selection by consensus or by vote. This technique is fast and easy to use, but the results may depend more on the persuasiveness of individual team members than on the objective quality of alternative designs.

Crucial Experiments—If alternative requirements can be fairly evaluated on a single criterion, especially if it involves an empirical question that can be settled using stakeholder evaluations such as surveys or usability studies, then selectors can obtain the crucial data and choose the alternative(s) with the best score. Often user interface design decisions can be made in this way. This technique is easy to use, but it applies only when a single criterion can be used to select from the alternatives. Also, it may be expensive to obtain the data.

Multi-Dimensional Ranking—Usually, several criteria must be taken into account when choosing among alternative designs. In this case, a multi-dimensional ranking technique can provide an objective basis for making the selection. One such technique is a **scoring matrix**, which is a table showing alternative requirements in the columns and weighted selection criteria in the rows. Alternatives receive weighted scores for each selection criterion. The sum of the scores determines the best alternative. This technique is time consuming and is practical only with a few alternatives and no more than 10 selection criteria. We illustrate scoring matrices using an example from AquaLush in the next subsection.

Which technique is used depends on the selection that must be made. For example, when selecting among a few significant user-level requirements important in the overall product design, an expensive but objective procedure that takes many evaluation criteria into account, such as multi-dimensional ranking, is appropriate. When choosing among user interface requirements that can only be evaluated empirically, selection based on a crucial experiment is appropriate. When choosing among many relatively inconsequential alternatives, listing pros and cons and selecting by consensus is appropriate. This technique is also useful for narrowing a wide range of alternatives to a smaller set that can be considered more carefully.

Using Scoring Matrices to Select AquaLush Alternatives

Scoring matrices combine several evaluations into an overall evaluation used to select among design alternatives. The selectors must choose the evaluation criteria to be used and the relative weights accorded each criterion.

We illustrate scoring matrices by comparing several AquaLush product concepts; that is, several alternatives for the main features in AquaLush.

The three alternatives under consideration are

Moisture Controlled—In this alternative, AquaLush does moisture-controlled irrigation only. This is the simplest alternative.

Timer Controlled—In this alternative, AquaLush runs either in a moisture-controlled irrigation mode or in a timer-controlled mode similar to that used by competitive products.

Manually Controlled—In this alternative, AquaLush runs either in a moisture-controlled irrigation mode or in a manually controlled mode that allows operators to turn valves on and off individually.

The first step in making a scoring matrix is listing these alternatives across the top of the table, as shown in Table 5-3-1 on page 135.

The next step is determining the criteria used to evaluate the alternatives and the relative weight accorded each criterion. Each criterion and its weight are added as a row in the scoring matrix. In this case, the designers settled on the following criteria and weights:

Irrigation Control—Operators list being able to schedule irrigation times as important. More exploration of this need revealed that operators prefer having a lot of control over when and how irrigation occurs, with considerable interest not only in controlling irrigation using soil moisture levels, but also in timer-based and manual controls. Hence, irrigation control is listed as an important criterion, with 25% of the relative weighting assigned to it.

Reliability—Maintainers and marketers need AquaLush to be highly reliable. This criterion is added with a weight of 20%.

Ease of Use—Operators need AquaLush to have an interface that allows them to set up or change irrigation schedules in less than five minutes without a manual, so ease of use is important. This criterion is added with a weight of 20%.

Robustness—Operators also need AquaLush to operate as normally as possible in the face of valve and sensor failures. This criterion is added with a relative weight of 20%.

Risk—AquaLush must be ready for release in a year. The risk selection criterion is added to capture the possibility that a product alternative will not be finished in time. The development team is quite confident that any of the three product alternatives can be realized on schedule, so this criterion is given a low relative weight of 15%.

The next step in constructing the scoring matrix is to fill in the rating of each product for each evaluation criterion. Ratings must be positive numbers, with higher numbers indicating higher ratings. In the matrix in Table 5-3-1 ratings are made on a five-point scale. It is usually best to identify the product alternative with the middling rating for an evaluation criterion as the *reference alternative*, assigning it the middle value in the scale. The other alternatives can then be rated relative to the reference alternative for that evaluation criterion.

For example, in the AquaLush scoring matrix the Timer Controlled product is of intermediate ease of use, so it is taken as the reference alternative for this criterion and assigned a rating of three. The Moisture Controlled product has fewer features than the Timer Controlled product, so its ease of use rating is higher, at four. The Manually Controlled product requires a rather complex user interface to control individual valves, so its ease of use rating is set to two.

Evaluation criteria should be considered one at a time, with all ratings filled in for a criterion before moving on to the next one.

Once all ratings are filled in, the remainder of the work is purely mechanical. In fact, it is advisable to use a spreadsheet for scoring matrices so that these last steps can be done by the computer. Each product's score for each evaluation criterion is computed by multiplying the product's evaluation criterion rating by the evaluation criterion weight. The overall product score is then the sum of the evaluation criteria scores for that product.

The finished scoring matrix for the AquaLush alternative requirements appears in Table 5-3-1.

Evaluation Criteria		Moisture Controlled		Timer Controlled		Manually Controlled	
Description	Weight	Rating	Score	Rating	Score	Rating	Score
Irrigation Control	25%	2	0.5	3	0.75	5	1.25
Reliability	20%	3	0.6	2	0.4	2	0.4
Ease of Use	20%	4	0.8	3	0.6	2	0.4
Robustness	20%	3	0.6	3	0.6	5	1.0
Risk	15%	4	0.6	3	0.45	2	0.3
Total Score		3.1		2.8		3.35	

Table 5-3-1 AquaLush Product Alternatives Scoring Matrix

The Manually Controlled product alternative has the highest score, so it is selected as the best alternative requirement.

Prioritizing Selected Requirements Just as stakeholder needs are prioritized to help make decisions about trade-offs, requirements can be prioritized to help make decisions about which requirements to abandon if (as often happens) resources run out before the entire product can be developed. Like needs priorities, requirements priorities can be numbers or qualitative ratings.

Needs priorities, if available, can be used to assign priorities to the requirements that meet those needs. Alternatively, designers can assign priorities based on their understanding of stakeholder goals and needs. Ideally, stakeholders assign or at least review requirements priorities.

Section Summary

- Alternative requirements must be evaluated and the best ones must be selected for further improvement or inclusion in the final design.
- Evaluation techniques include stakeholder surveys, usability studies, and judgment against design principles.
- Stakeholders, designers, or both can select alternative requirements.
- Selection techniques include considering pros and cons and arriving at consensus, doing a crucial experiment, or doing multi-dimensional ranking.
- Multi-dimensional ranking can be done using **scoring matrices**.
- Once alternative requirements have been selected for inclusion in the design solution, they can be prioritized in case some must later be thrown out or deferred.

Review Quiz 5.3

1. What is the difference between a stakeholder survey and a usability study?
 2. Rank the selection techniques discussed in the text from easiest to hardest for designers to use.
 3. How is the total score for a design alternative computed in a scoring matrix?
-

5.4 Finalizing a Product Design

SRS Quality Characteristics

The final product design process step is to finalize the product design. This step is a last check to validate requirements and to ensure that the SRS is of high quality. This activity generally consists of designer and stakeholder reviews of the SRS. In this section we list what reviewers should look for when they check the SRS, and then we discuss various types of requirements reviews.

Designers should check the SRS carefully before asking stakeholders to review it, and they should leave confirmation that the requirements meet stakeholder needs and desires to the stakeholders. Designers should concentrate instead on whether the SRS specifies a good product design and does so clearly and completely. More specifically, designers should concentrate on determining whether the SRS has the following quality characteristics:

Well-Formedness—An SRS is **well formed** if it conforms to all the rules about stating requirements: All requirements should be atomized, expressed using “must” or “shall,” written in complete and simple sentences in the active voice, use consistent terminology, and so forth.

Clarity—An SRS can be well formed but still very hard to understand. An SRS is **clear** if it is easily understood.

Consistency—A set of requirements is **consistent** if a single product can satisfy them all. In other words, the SRS should not specify that the product have property P in one place and specify that it not have property P in another place. Consistency is surprisingly hard to achieve, especially for large products designed by a large team.

Completeness—An SRS is **complete** if it includes every relevant requirement. The SRS must specify every feature, function, characteristic, and property of the product that is to be realized in software. Furthermore, the design elements must be specified down to the physical level of abstraction.

Verifiability—Every specification in an SRS must be **verifiable**. One way to confirm this is to imagine (or actually write) test cases for each requirement.

Uniformity—A description has **uniformity** when it treats similar items in similar ways. A design should have similar mechanisms for manipulating similar things.

Feasibility—Designers should be reasonably confident that the requirements can be realized; that is, that the design is **feasible**.

Requirements Validation

Stakeholders should concentrate on **requirements validation**, which is the activity of confirming that a product design satisfies stakeholder needs and desires. More specifically, stakeholders should check the SRS to ensure that it has the following characteristics:

Correctness—The SRS should specify a product that meets stakeholder needs and desires. Because stakeholder needs and desires often conflict, the SRS does not have to satisfy *every* need and desire of *every* stakeholder. Rather, the SRS must specify a product that is a reasonable compromise among conflicting needs and desires. Stakeholders also need to be aware of features and capabilities that no stakeholder needs or wants that have somehow been made into requirements.

Proper Requirements Prioritization—If requirements have been prioritized, the priorities must reflect the needs and desires of stakeholders.

Requirements Reviews

As noted, designers and stakeholders usually finalize a design using some sort of review.

A **review** is an examination and evaluation of a work product or process by qualified individuals or teams.

As this definition suggests, all sorts of things can be reviewed by a variety of individuals. Although we concentrate on requirements reviews now, review techniques can be applied to other sorts of reviews as well.

There are several different types of reviews:

Desk Check—A **desk check** is an examination of a work product by an individual. A desk check might also be called *proofreading*. Requirements writers should always desk check what they have written. A checklist of items to watch for may increase this technique's effectiveness.

Walkthrough—A **walkthrough** is an informal examination of a work product by a team of reviewers. The review team meets and reads through a portion of the SRS, looking for problems and ways to improve it. Usually reviewers are expected to prepare for the meeting with a desk check of the portion of the SRS under review. This is probably the most common form of requirements review currently in use.

Inspection—An **inspection** is a formal work product review by a trained team of inspectors with assigned roles using a checklist to find defects. Inspections follow a well-defined process, and their results are documented. Inspections are the most effective review technique, so we look at them in more detail next.

Requirements Inspections

Inspections are intended to find as many defects as possible through work product review. The inspection process, materials, and roles are all designed for this purpose. Inspections are *not* intended to correct defects or to evaluate work product authors.

Inspections are hard work and very time consuming. However, finding and correcting defects early is still much cheaper and easier than finding and correcting them when the software is tested or deployed to customers. Over the years, many studies have shown that inspections are both cost effective and better at finding defects than any other technique.

Requirements inspectors play particular roles for which they must be trained. The main roles are

Moderator—Manages the inspection process by scheduling meetings, ensuring that everyone has the proper materials, facilitating meetings, reporting inspection results, and monitoring follow-up activities.

Inspector—Reads the SRS, finds defects, and points them out during the inspection meeting.

Author—Writes the portion of the SRS under inspection.

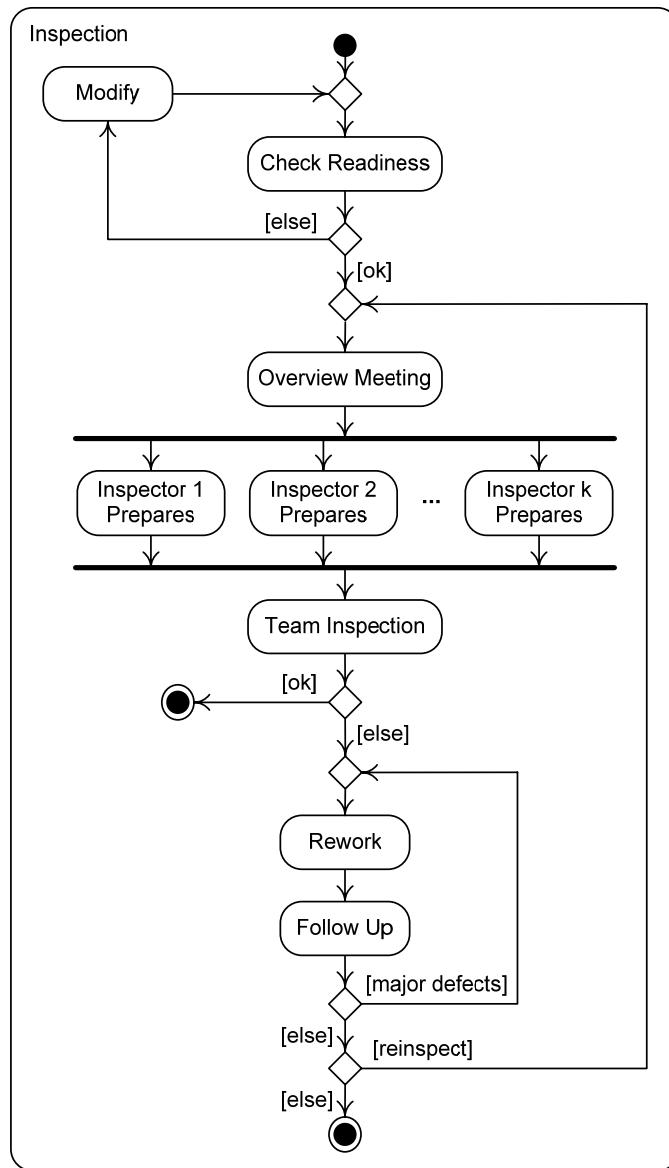
Reader—Reads or paraphrases each line of the inspected portion of the SRS during the inspection meeting.

Recorder—Notes all defects found, issues raised, time spent in the inspection meeting, and so forth.

The Inspection Process

The inspection process is an orderly sequence of activities, as pictured in the activity diagram in Figure 5-4-1 on page 139.

This process begins with an inspection readiness check of a small portion of the SRS under review. Someone (usually the Moderator) does a cursory review to see that the requirements are well formed and appear to be correct and complete. The requirements are modified until the readiness check is passed.

**Figure 5-4-1 The Inspection Process**

Next, an inspection overview meeting is held with all the participants. This meeting is short (usually no more than 20 minutes) and is used to schedule the inspection meeting, to provide any needed background information, and to supply inspection materials (the requirements, the checklist, and possibly a needs list, stakeholders-goals list, and other analysis materials). The overview may be done electronically.

Inspection preparation is crucial for successful inspections. Each Inspector studies the requirements, guided by the checklist. This activity typically takes each Inspector several hours.

The team inspection meeting begins with the Moderator ensuring that all Inspectors are ready. If some are not, the meeting is rescheduled. The Reader reads through the requirements, and the Inspectors note any defects they have found or raise issues about the specifications. The Recorder documents all findings and collects data about the duration and results of the meeting.

If the requirements are defect free, the process ends—but this almost never occurs. Usually there are many defects to correct, so the Author takes the meeting's findings and reworks the requirements to correct the defects.

The Moderator ensures that all defects have been corrected. If the work product is significantly changed or the changes are dubious, another inspection may be scheduled.

The Inspection Meeting

Inspection meetings should never last more than about two hours, and there should never be more than one scheduled per day. Also, the rate at which the material is inspected should not be too high. Data from inspection meetings should be analyzed to determine how much material should be reviewed in a single inspection and how fast material should be reviewed.

The inspection meeting is aimed at detecting defects, not fixing them or assigning blame. The Moderator is responsible for curbing discussion of requirements revisions and cutting off nasty comments aimed at the Author or other Inspectors.

Inspection Checklists

Inspections are driven by checklists that the Inspectors use to guide their review; good checklists are essential for effective inspections. Designers should modify requirements inspection checklists continuously to make them as effective as possible. If certain kinds of defects are not being detected, then a check for them can be added to the list. If a certain check is not finding many defects, that item can be removed from the list. Figure 5-4-2 shows some candidate requirements checklist entries.

Different checklists might be used for inspecting different sorts of requirements. For example, a team might use one checklist for inspecting functional requirements, another for data requirements, and another for user interface requirements.

- Every requirement is atomic.
- Every requirement statement uses “must” or “shall.”
- Every requirement statement is in the active voice.
- Terms are used with the same meaning throughout.
- No synonyms are used.
- Every requirement statement is clear.
- No requirement is inconsistent with any other requirement.
- No needed feature, function, or capability is unspecified.
- No needed characteristic or property is unspecified.
- All design elements are specified to the physical level of abstraction.
- Every requirement is verifiable.
- Similar design elements are treated similarly.
- Every requirement can be realized in software.
- Every requirement plays a part in satisfying some stakeholder’s needs or desires.
- Every requirement correctly reflects some stakeholder’s needs or desires.
- Every requirement statement is prioritized.
- Every requirement priority is correct.

Figure 5-4-2 Requirements Inspection Checklist Entries

Continuous Review

A **critical review** is an evaluation of a finished product to determine whether it is of acceptable quality. Critical reviews are mainly intended as **quality gates** that keep poor-quality work products from moving on to the next step of a process. Reviews are more useful as tools for work product improvement than as quality gates. Reviews should occur frequently during any process to catch defects as soon as possible so they can be corrected quickly and cheaply. A practice of **continuous review**—frequently evaluating work products during their creation—leads to much better results than waiting until a critical review to catch defects.

Product design finalization is a critical review of the SRS. Although this critical review must occur, it should not be the only requirements review done during design resolution. Requirements should be reviewed continuously as they are generated.

Section Summary

- Product design finalization validates requirements and ensures that the SRS is of high quality.
- Designers should check that the SRS is well formed, clear, consistent, complete, verifiable, uniform, and feasible.
- Stakeholders should check that the SRS is correct and that requirements are properly prioritized.
- Finalization is usually done through **reviews**, which may be **desk checks**, **walkthroughs**, or **inspections**.

- Inspections are the most expensive but also the most effective review technique.
- **Inspections** define roles for all participants, follow a strict process, use checklists to guide inspectors as they review requirements, and have guidelines for carrying out process activities.
- Product design finalization is a **critical review** activity.
- It is good practice to supplement critical reviews with **continuous reviews** during the development process.

Review Quiz 5.4

1. What is requirements validation?
 2. What is the difference between a desk check, a walkthrough, and an inspection?
 3. What roles do people play in inspections?
 4. What is a critical review, and how does it differ from other reviews done during a development process?
-

5.5 Prototyping

Modeling in Product Design

In our discussion of software product design in this and the last chapter, we have discussed many techniques useful at each stage of the design process. For example, we have discussed needs elicitation techniques (such as interviews and stakeholder observation), needs analysis techniques (such as making stakeholders-goals lists and conducting needs documentation reviews), design alternative generation techniques (such as team and individual brainstorming), design evaluation techniques (such as surveys and usability studies), design selection techniques (such as crucial experiments and multi-dimensional ranking), and design finalization techniques (such as requirements inspections).

Although there are several more techniques that might be used at each of these stages, the most important technique that we have not yet discussed in detail, which is applicable at every stage, is *modeling*. Models can document problem domains, stakeholder needs, and product designs. They can be analyzed to detect misunderstandings of needs and desires, inconsistencies, incompleteness, and other failings. They can prompt stakeholders to reveal additional needs and desires. Also, models serve as a vehicle for generating alternative designs, evaluating them, and choosing between them. They can even be used as part of product realization. Thus modeling is an important technique in product design.

Many kinds of models are useful in software product design. This section discusses prototypes, a special kind of dynamic model especially useful in product design. Chapter 6 is devoted to the single most important and powerful functional modeling technique: use cases.

What Are Prototypes?

Prototypes are a special kind of model. They are models because they are representations of another entity, the final product. But not every model is a prototype. For example, a plastic model car made from a kit is not a prototype of a real car. A prototype car is usually a full-size, working automobile. The crucial characteristic of a prototype is that it must *work* just as the final product does, at least in some way. Hence, we have the following definition.

A **prototype** is working model of part or all of a final product.

Prototypes can be classified along several dimensions, as discussed next.

Horizontal and Vertical Prototypes

A **horizontal prototype** realizes part or all of a product's user interface. It is called "horizontal" because it represents just one layer of the many that typically comprise a software product.

Horizontal prototypes have two main uses: They can model either a product's functionality or its user interface. When used in the former way, a horizontal prototype provides a vehicle for eliciting functional needs, exploring functional alternatives, trying out different operations, and so forth. In this case, user interface details are glossed over and the focus is on the services the product provides its users. This sort of prototype might be used in conjunction with a use case description (discussed in Chapter 6).

Alternatively, a horizontal prototype can be used as what is perhaps best called a **mock-up**, which is an executable model of a product's user interface. A mock-up can help iron out user interface details, especially in usability studies where users are observed or measured as they use a prototype. Several mock-ups may be created to study user interface design alternatives.

A **vertical prototype**, also called a **proof of concept prototype**, does some processing apart from that required for presenting the product's user interface. Such prototypes are called "vertical" because their implementation cuts across several of a software product's layers. Vertical prototypes are usually used to establish requirements feasibility or to see how some portion of a system will perform. For example, designers for a product that communicates over the Internet might write a portion of the product to see how responsive it is under various load conditions as a means of establishing performance requirements.

The diagram in Figure 5-5-1 summarizes the differences between horizontal and vertical prototypes.

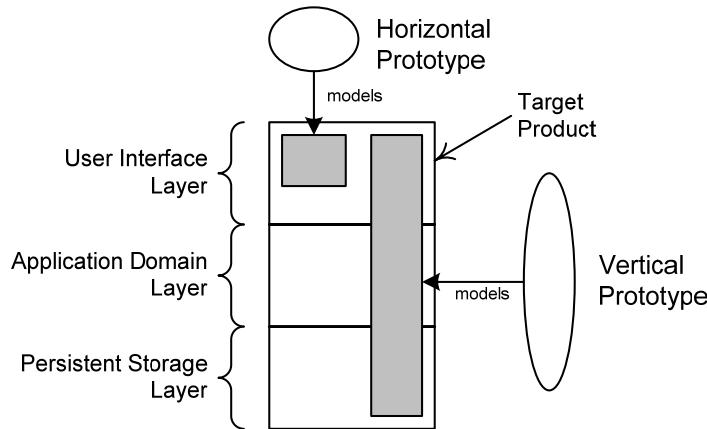


Figure 5-5-1 Horizontal versus Vertical Prototypes

Throwaway and Evolutionary Prototypes

A **throwaway** or **exploratory prototype** is developed merely as a design aid and then discarded once the design is complete. In contrast, an **evolutionary prototype** becomes part of the final product. Evolutionary prototypes are usually created as part of an iterative development effort in which the prototype is first released with only a few required features or characteristics. The prototype is enhanced and re-released several times until it has all the features and properties of the final product.

Throwaway prototypes can be built relatively quickly because they do not need to be well engineered or implemented. Evolutionary prototypes, on the other hand, must be carefully engineered and built to high standards from the beginning, and they must be flexible enough to accommodate a lot of change.

Low- and High-Fidelity Prototypes

Fidelity is how closely a prototype represents the final product. It is a spectrum ranging from representations with only the faintest resemblance to the product up to representations that are almost indistinguishable from the product.

Low-fidelity prototypes are usually **paper prototypes**—rough representations using paper, note cards, whiteboards, or cardboard boxes. Paper prototypes are extremely easy and quick to make; it takes only a few seconds to sketch a user interface screen, for example.

Paper prototypes are working models only in the sense that they are “executed” by a person explaining how the program is supposed to work. For example, Figure 5-5-2 shows a paper prototype of a program to keep track of computers, users, and assignments of computers to users.

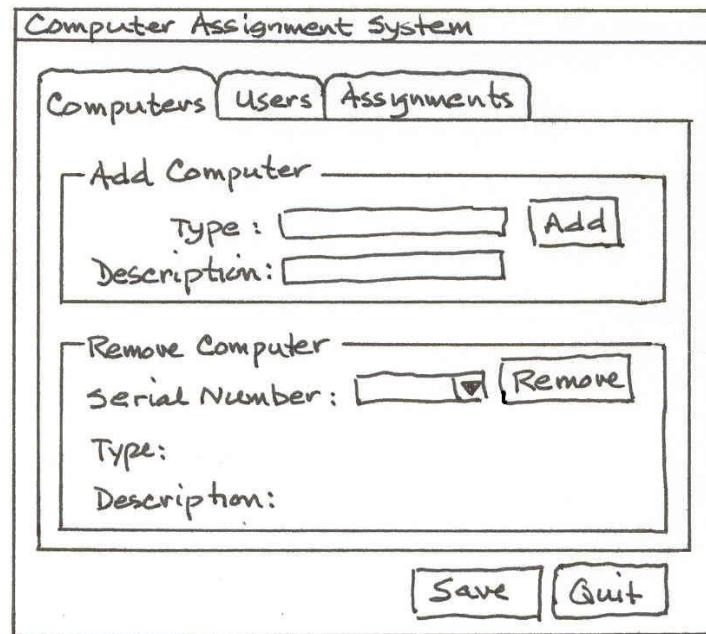


Figure 5-5-2 A Low Fidelity Prototype

This prototype would be “executed” by explaining what happens when buttons are pressed, items are selected from drop-down boxes, and so forth.

Higher-fidelity prototypes are usually **electronic prototypes**, which are programs that behave like the final product in some way. An electronic prototype can still be fairly low fidelity—for example, it might be a rough mock-up with a few buttons and labels that doesn’t respond to user input—or it can be a very high-fidelity representation that appears to behave like the final product. Prototypes require more time and effort to produce as their fidelity increases. Figure 5-5-3 shows a screen shot from a high-fidelity electronic prototype corresponding to the low-fidelity prototype in Figure 5-5-2.

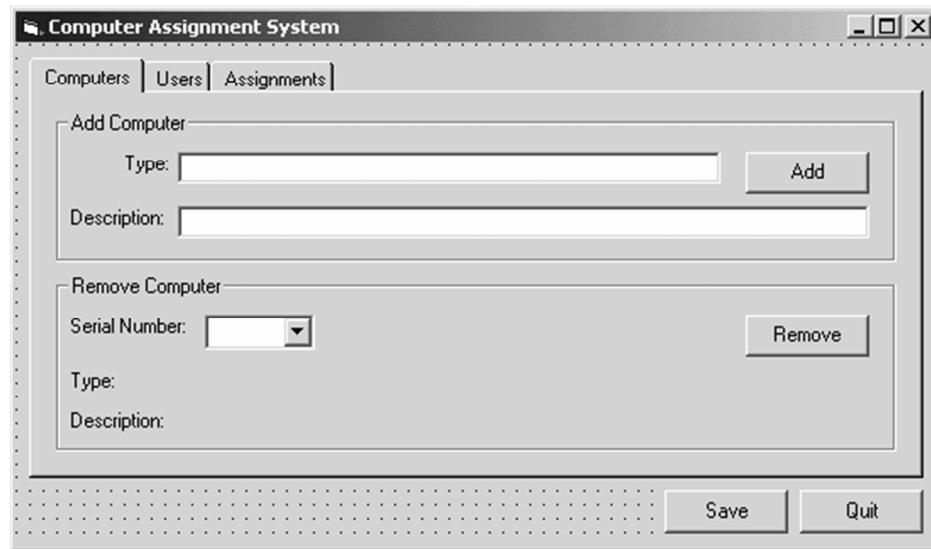


Figure 5-5-3 A High-Fidelity Prototype

Prototype Uses

The different categories of prototypes can be used for various purposes during design.

Prototyping for Needs Elicitation

Recall the many difficulties of needs elicitation arising from stakeholders' lack of understanding of their own needs, inability to express themselves, and difficulty recounting all relevant information, as well as difficulties arising from designers' having to understand a new domain, digest a great deal of information, and so forth. Prototypes can help with all of this.

Prototypes can help stakeholders articulate their needs and desires because prototypes are concrete entities that can focus and direct discussion. For example, rather than having stakeholders try to list all the things they want the product to do, designers can have stakeholders walk through an interaction with a prototype, noting what it needs to do at each step. Many needs and desires may emerge only during such exercises. Neither stakeholders nor designers may think of some things that are obvious only when actually "using" the product.

Probably the most useful sort of prototype at this early stage of design is a horizontal throwaway paper prototype—in other words, quick sketches of user interface screens. When elicitation is focused on needs at the user and operational levels of abstraction, the user interface screens themselves are not the focus of attention: They are simply vehicles for focusing on what the product is doing and what data is going into and coming out of it.

Paper prototypes are also very useful for eliciting needs and desires about the user interface. The great advantage of throwaway paper prototypes at this stage of design is that they can be made quickly, facilitating discussion during interviews, focus groups, and elicitation workshops.

In an iterative development effort, the current version of the product can be considered a high-fidelity prototype for the next version of the product, except that it lacks the new features and properties of the next release. The current product can serve as a foil for discussion about needs for new features and properties.

Prototyping for Needs Analysis	<p>Designers can make throwaway horizontal prototypes at various levels of fidelity to help analyze needs. For example, designers might create a set of screen images using a drawing tool to describe the product's function or user interface. Alternatively, designers might actually write a prototype program using a tool such as Visual Basic that allows rapid development. Such prototypes document the designers' understanding of needs and desires and can be reviewed by both stakeholders and designers. Misunderstandings, missed needs, lack of uniformity, and unneeded features may surface only when designers and stakeholders try out a prototype. At this early stage, cheap throwaway prototypes should still be used almost exclusively.</p>
Prototyping for Requirements Generation and Refinement	<p>Prototypes can help generate and refine design alternatives. Cheap prototypes are a good way to explore a completely unknown area, trying out different ideas quickly to see how they would work or how they would look. They can help explore a metaphor and may inspire other ideas. Prototypes can also be a quick way to document a design alternative. Because of the need to generate, record, and modify alternatives quickly at this stage of design, horizontal throwaway paper prototypes are again the best choice. Once a complicated design alternative that is still undergoing minor changes has solidified, it may be worthwhile to draw it up in a design tool. It is faster to make a small change to a complicated drawing in a design tool than it is to redraw the whole image by hand.</p>
Prototyping for Requirements Evaluation and Selection	<p>Stakeholders and designers are better able to evaluate a design alternative when they see how it would work, so prototypes are often needed for usability studies. Prototypes may also be needed to test the feasibility of some requirements. Hence, prototyping is important for design alternative evaluation and selection as well. Once the range of design alternatives has been narrowed to a few, it may be necessary to make higher-fidelity prototypes of the alternatives to do a thorough evaluation and comparison. This is particularly true if a user study is done, in which case an electronic prototype will almost surely be needed. Requirements feasibility is assessed at this design stage, so this is the point at which vertical prototypes may come into play.</p>

Prototyping for Design Finalization Trying out a prototype is more effective than reviewing requirements and other sorts of models, so prototypes can be a useful tool for design finalization. At this stage the product design should be nearly complete, so it would probably be cost effective to make an evolutionary horizontal prototype of fairly high fidelity. The prototype can both ensure that user interface requirements are set and provide a basis for implementation.

Prototyping Risks

There are many risks in using prototypes. The three most important are discussed here.

Except in an iterative development effort, prototypes are usually throwaways. This means that they are completely unsuitable for users. One of the worst things that can happen to a development organization is to find itself using a throwaway prototype as the basis for a product. Unfortunately, this can happen when stakeholders see a high fidelity prototype and insist that it be released as an early version of the product.

To avoid this problem, either

- Avoid making high-fidelity throwaway prototypes, or
- Make it very clear to stakeholders beforehand that the prototype only *appears* to work.

Another problem with horizontal prototypes is that stakeholders become fixated on the appearance of the user interface when they should be paying attention to product functions and properties. Using low-fidelity prototypes helps solve this problem as well. This problem can be avoided altogether if another sort of model, such as use cases, is used to resolve functional requirements.

An additional common problem occurs when stakeholders are disappointed that a final product does not look or behave exactly like the prototype. This problem can be avoided by using low-fidelity prototypes that no one would think accurately reflect the final product, making sure stakeholders know what to expect, or making sure that high-fidelity prototypes do look and behave almost exactly like the final product (by mimicking time delays or other unappealing characteristics).

Section Summary

- Modeling is a product design analysis and resolution technique useful in every stage of product design for several purposes.
- A **prototype** is a working model of part or all of a final product.
- Prototypes can be classified as **horizontal** or **vertical**, as **throwaway** or **evolutionary**, or along a continuum from low to high **fidelity**.
- Prototypes are useful in every stage of product design.
- Prototyping may lead to problems if stakeholders mistake a throwaway prototype for a product, worry about prototype appearance when they should concentrate on prototype function, or come to expect too much from the final product based on the prototype.

Review Quiz 5-5

1. Explain the difference between prototypes and other sorts of models.
 2. What is the difference between a throwaway and an evolutionary prototype?
 3. In what sense is a paper prototype a working model of a product?
 4. Would it be advisable to make high-fidelity electronic prototypes in generating product design alternatives?
-

Chapter 5 Further Reading

- Section 5.1** Ulrich and Eppinger [2000] and Preece et al. [2002] discuss alternative requirement generation techniques. McGrath [1984] discusses the relative effectiveness of individuals and groups in generating new ideas.
- Section 5.2** Every textbook about requirements development discusses conventions and heuristics for stating requirements, requirements testability, and requirements traceability, including [Lauesen 2002], [Robertson and Robertson 1999], [Thayer and Dorfman 1997], and [Wiegers 2003]. The formal specification notation Z is described in [Spivey 1989]. Harel [1987] introduced statecharts. Standard mathematical and logical notations useful in requirements specification are discussed in discrete mathematics texts such as [Epp 1995].
- Section 5.3** Preece et al. [2001] and Cooper and Reimann [2003] discuss user evaluation. Nielsen [1993] discusses usability evaluation in depth. Risk evaluation is surveyed in [Pressman 2001] and [Sommerville 2000]. Ulrich and Eppinger [2000] survey alternative requirement selection techniques and cover scoring matrices for selecting alternative requirements. Requirements prioritization and validation are discussed in [Lauesen 2002], [Robertson and Robertson 1999], and [Wiegers 2003].
- Section 5.4** SRS quality characteristics and requirements validation are discussed in most software and requirements engineering texts, including [Pressman 2001], [Robertson and Robertson 1999], [Sommerville 2004], and [Wiegers 2003]. Most software and requirements engineering texts also discuss various sorts of reviews. Gilb and Graham [1993] provide a thorough discussion of inspections.
- Section 5.5** Roberson and Robertson [1999] and Wiegers [2003] discuss prototyping extensively.

Chapter 5 Exercises

The following system descriptions are used in some of the exercises below.

Parking Garage Controller

An automated parking garage system interacts with entry and exit gates to monitor whether the garage is full or not. It also controls a sign on the street that says whether the garage is full. The entry gates are lockable, and the system can lock them when the garage is full and unlock them when it is not full. When a driver comes to an entry or exit gate, he or she must press a button to open the gate and drive into or out of the garage.

The system also interacts with a human operator. The human operator can see the status of the garage, adjust the number of spaces and the number of

occupied spaces, and set the system mode. The system modes are open (entry gates are never locked), closed (entry gates are all locked), and automatic (entry gates are locked or unlocked depending on whether the garage is full).

Locker Assignment System

Patrons of a health club are assigned lockers when they check in at the front desk. When a patron swipes his or her membership card, the Locker Assignment System issues a key card for a locker and displays the locker number to the patron. When patrons are done, they deposit their key cards in a card reader, and the Locker Assignment System releases their lockers. Patrons can also swipe their key cards to have their locker numbers redisplayed.

Patrons can have only one locker at a time. The Locker Assignment System must issue lockers for the appropriate gender. The Locker Assignment System must spread out patrons in the locker rooms as much as possible.

Pigeonhole Box Office System

A company has decided to develop a theater box office system called Pigeonhole that includes support for ticket sales. It helps sales people to see the layout of seats in the theater when discussing them with customers. Because every theater is different, the software will have to provide some sort of theater layout module so that users can set up the layout for their theaters when installing the software.

Product designers are considering three alternative requirements for this feature:

No Layout—The product does not display the seat layout at all, obviating the need for a layout module. In this case salespeople will need paper layouts to see where seats are.

Simple Layout—The product displays a very rudimentary layout with matrices of labeled rectangles for banks of seats. Colors will indicate sold and unsold seats. The layout module is quite simple.

Fancy Layout—The product displays an accurate seat layout with colors indicating different price levels and sold and unsold seats. The layout module is very sophisticated.

The following evaluation criteria can be used to decide between these three alternatives:

Cost—The relative cost of developing the layout module and the user interface to support seat layout display to ticket salespeople.

Ease of Setup—How easy it is for users to set up the software for use.

Ease of Use—How easy it is for users to conduct ticket sales.

Error Prevention—How well the design helps prevent errors during ticket sales (selling the wrong seat, selling the same seat twice, etc.).

Beauty—How attractive users find the program's interface.

Simply Postage Kiosk System

- 1 The Simply Postage Kiosk is a self-service postage kiosk that dispenses first-class postage.
 - 1.1 The machine may be set up in any location that has an Internet connection (either dialup or broadband).
 - 1.2 The kiosk has a touch screen that allows the user to select some number of 39-cent stamps to be purchased.
 - 2.1 The user first touches a start button on the greeting page and is then taken to the quantity page from which a selection is made.
 - 2.1.1 The number of stamps that can be dispensed must be 4, 8, 12, 16, or 20.
 - 2.2 Upon choosing a quantity, the user must be prompted to swipe a credit card.
 - 2.2.1 If 20 seconds pass without a credit card swipe, the system must cancel the transaction and redisplay the greeting page.
 - 2.3 Once the credit card is swiped, the Kiosk establishes a connection with a credit card center to approve the credit card charge.
 - 2.3.1 While processing the credit card, the system must display a processing message.
 - 2.3.2 The credit card center either (a) responds with a charge approval, (b) responds with a charge approval failure, or (c) does not respond within 20 seconds.
 - 2.3.2.1 In case (a), the Kiosk must dispense stamps.
 - 2.3.2.2 In case (b), the Kiosk must display a message that the credit card charge was not approved. After displaying this message for four seconds, the Kiosk returns to the greeting screen.
 - 2.4 If the credit card charge is approved, the system Kiosk must print and dispense stamps in sets of four.
 - 2.4 After each set of four stamps is printed and pushed out of the machine to the user, the user must be prompted to retrieve the stamps from the dispenser.
 - 2.4.1 When all requested stamps are dispensed, the screen displays a thank you message. After four seconds, the Kiosk must again display the greeting page.
 - 3 In case of any problems with the Kiosk, hardware or software, the system must display an out-of-service message and not respond to any screen presses.
 - 4 The Kiosk will also display an out-of-service message when there are less than 20 stamps left. This will ensure that the current transaction can be processed prior to the system being out of service.

Section 5.1

1. *Fill in the blanks:* Ideas for alternative requirements can come from _____ the design team or _____ the team. The former sources include users and other stakeholders, _____, _____ products, and _____ products. The latter include team and individual _____, props and metaphors, and _____.
2. Give two examples of popular software products that are based, at least in part, on items from the real world.
3. Give an example of a product whose design you think you could have improved had you been consulted as a product user by the design team.

Section 5.2

4. Given an atomized software requirements specification, what information might one add to a design model, source code, or a test case to achieve requirements traceability?
5. Atomize the following requirements drawn from the SRS used in the example from this section:

The system may be queried. A query may contain a user number or a serial number. If the query contains a user number, all equipment assigned to that user is reported. If the query contains a serial number, the assignment for that computer is reported. All query results show the name, office, and user number of a user, followed by the serial numbers and types of all computers assigned to that user, and the date of each assignment. If a user is not assigned a computer or a computer is not assigned to a user, this is reported.
6. Rewrite the requirements for the Parking Garage Controller (above) so they are atomized, are verifiable, use the appropriate auxiliary verbs, and are clear and precise. Resolve ambiguities and unspecified details as you see fit.
7. Rewrite the requirements for the Locker Assignment System (above) so they are atomized, are verifiable, use the appropriate auxiliary verbs, and are clear and precise. Resolve ambiguities and unspecified details as you see fit.
- AquaLush 8. *Find the errors:* What is wrong with each of the following atomized requirements statements?
 - (a) 5.1.1 AquaLush should turn off all valves when it restarts after a power failure.
 - (b) 6.2.1 AquaLush must record its state as either *manual* or *automatic* in persistent store. AquaLush must also record the current irrigation time in persistent store.
 - (c) 3.2.1 An irrigation cycle must be stopped by AquaLush when the water allocation is exhausted.
 - (d) 4.1.2 AquaLush must record irrigation start time as a time of day and a set of days of the week.
 - (e) 3.1.8 AquaLush must allow users to reset things.
 - (f) 6.8.1 AquaLush must always display good error messages.

- Section 5.3**
- 9. *Fill in the blanks:* Evaluation of alternative requirements should include _____ in some way. For example, they might be asked about their opinions of various alternative requirements in _____ or they might be the subjects of observation or experimentation in _____.
 - 10. Use a spreadsheet to make a selection matrix for the Pigeonhole Box Office alternative requirements described previously. Weight each evaluation criterion equally. Rate each product for each evaluation criterion as you see fit.
 - 11. Use a spreadsheet to make a selection matrix for the Pigeonhole Box Office alternative requirements described previously. Decide on appropriate weights for each evaluation criterion. Rate each product for each evaluation criterion as you see fit.
 - 12. Are you satisfied with the selection resulting from the selection matrices you made in exercises 10 and 11? If not, why do you think the result was not satisfactory?
- Section 5.4**
- 13. Explain the difference between correctness and verifiability.
 - 14. Make a list of an inspection Moderator's responsibilities.
 - 15. Based on the discussion in the text, make a short list of guidelines for conducting inspections.
 - 16. *Find the errors:* Using the checklist in Figure 5-4-2, desk check the Simply Postage Kiosk System requirements, and make a list of every defect you find.
- Section 5.5**
- 17. *Fill in the blanks:* Because they can be made so quickly and easily, _____ prototypes are useful in those stages of software product design when many ideas are considered rapidly. They are especially useful in needs _____ and requirements _____.
 - 18. Make a horizontal paper prototype of the user interface for the operator of the Parking Garage Controller. Write a short explanation to go along with your screen sketches.
 - 19. Make a horizontal paper prototype to illustrate how the Pigeonhole Box Office product supports regular ticket sales. Write a short narrative to go along with your screen sketches.
 - 20. Make an additional horizontal paper prototype to illustrate an alternative design for regular ticket sales in the Pigeonhole Box Office product.
 - 21. Make a horizontal paper prototype to illustrate how the Pigeonhole Box Office product supports season ticket sales. Write a short narrative to go along with your screen sketches.
 - 22. Make an additional horizontal paper prototype to illustrate an alternative design for season ticket sales in the Pigeonhole Box Office product.

23. Make a horizontal paper prototype to illustrate how users enter all the show information needed to support ticket sales into the Pigeonhole Box Office product.

Team Projects

24. Form a team of three people. Use individual and team brainstorming techniques to come up with as many product ideas as you can to satisfy each of the following need statements. In all cases, assume that you are designing a software product that uses one or more existing devices that the stakeholders already own.
 - (a) Students need to be reminded that they have class in five minutes.
 - (b) Professors need help learning or remembering students' names.
 - (c) Students need a way to find other students to get help on a particular assignment, to study with, or to prepare for a particular test.
 - (d) Students need a way to find others willing to share rides to and from campus on weekends or over school breaks.
 - (e) Graduating seniors who want to sell dorm furnishings (refrigerators, lofts, etc.) and incoming freshmen who want to buy them need a way to find out about one another.
25. Form a team of two people. Obtain a portion of a requirements document from your instructor. Write a requirements checklist and use it to conduct a requirements inspection.
26. Rewrite the Simply Postage Kiosk System requirements so that they are of high quality.

Chapter 5 Review Quiz Answers

Review Quiz 5.1

1. Alternative requirement generation typically fails because designers consider only a few alternatives, or they fail to consider entire categories of alternatives.
2. Competitive products often have features, functions, and characteristics that are good and should be copied, as well as features, functions, and characteristics that are bad and should be entirely avoided or somehow improved. Studying such products thus provides alternative requirements and directly and indirectly stimulates thinking about other alternatives.
3. Studies have shown that individual brainstorming is more effective than team brainstorming.

Review Quiz 5.2

1. Consider the statement "The product must have an open access mode in which every user will not need to log in." This statement can be interpreted to mean "The product must have an open access mode in which no user will need to log in" or to mean "The product must have an open access mode in which not every user will need to log in."
2. Conventionally the auxiliary verbs "shall" and "must" are used to state requirements.
3. An example unverifiable requirement is "The product must process transactions quickly." This requirement is not verifiable because it is not clear how to determine which transactions need to be done quickly or whether a

transaction has been processed quickly enough. This requirement can be made verifiable by listing the transactions involved and quantifying the required processing rate for each transaction type.

4. Requirements traceability is the ability to track requirements from their statement in an SRS to their realization in design documentation, source code, and user documentation and their verification in reviews and tests.
5. The advantages of atomizing requirements include providing the basis for requirements traceability; helping root out inconsistencies and incompleteness in an SRS, and helping analysts understand the SRS thoroughly.

**Review
Quiz 5.3**

1. In a stakeholder survey, stakeholders are asked to rate various alternative requirements. In a usability study, alternative requirements are evaluated by measuring users as they perform various tasks with a prototype realizing alternative requirements.
2. The alternative requirement selection techniques discussed in the text, ranked from easiest to hardest to use, are pros and cons, crucial experiments, and multi-dimensional ranking. Doing a crucial experiment may be more difficult than multi-dimensional ranking if the experiment's data is harder to collect than the rating information needed for multi-dimensional ranking.
3. The total score for an alternative requirement in a scoring matrix is computed by summing the scores for that alternative for each evaluation criterion. Each evaluation criterion score is computed by multiplying the alternative requirement's evaluation criterion rating by the evaluation criterion's weight.

**Review
Quiz 5.4**

1. Requirements validation is the activity of confirming with stakeholders that a product design satisfies their needs and desires.
2. A desk check is a review of a work product that a person does alone, while walkthroughs and inspections are group activities. A walkthrough is an informal review of a work product wherein participants have no set roles, do not follow a strict process, and are not guided by a checklist. In contrast, an inspection is a formal review following a well-defined process, with each participant playing a role and reviewers following a checklist as they study the work product.
3. Inspection roles include Moderator, Reader, Author, Inspector, and Recorder.
4. A critical review is an evaluation of a finished product to determine whether it is of acceptable quality. Critical reviews are intended to *evaluate* a work product. In contrast, reviews done during work product creation as part of a practice of continuous review are intended to detect (and sometimes to correct) defects, so their aim is to *improve* a work product.

**Review
Quiz 5.5**

1. A prototype is a model that works in some way like the final product that it represents. Full-sized working models of manufactured goods are often made before the manufacturing process is set up to make sure that the finished product will be as expected—these are prototypes. In software product design, prototypes can range from models drawn on paper that are “executed” by people, to programs that are almost indistinguishable from the final product, though they may be engineered quite differently.

2. A throwaway prototype is intended only as a tool in the design process, not as an artifact to be released for use. An evolutionary prototype is intended from the first to be part of the final product.
3. A paper prototype is a working model of a product only in the sense that people can use it to help simulate the real product.
4. Generating product design alternatives needs to be done quickly and cheaply. Otherwise, designers will tend to generate far fewer alternatives, damaging the design process. Because high-fidelity electronic prototypes take considerable time and effort to create, it would not be a good idea to use them to generate design alternatives.

6 Designing with Use Cases

Chapter Objectives This chapter discusses use case modeling in product design. Figure 6-O-1 shows where use case diagrams are most useful in the product design resolution process.

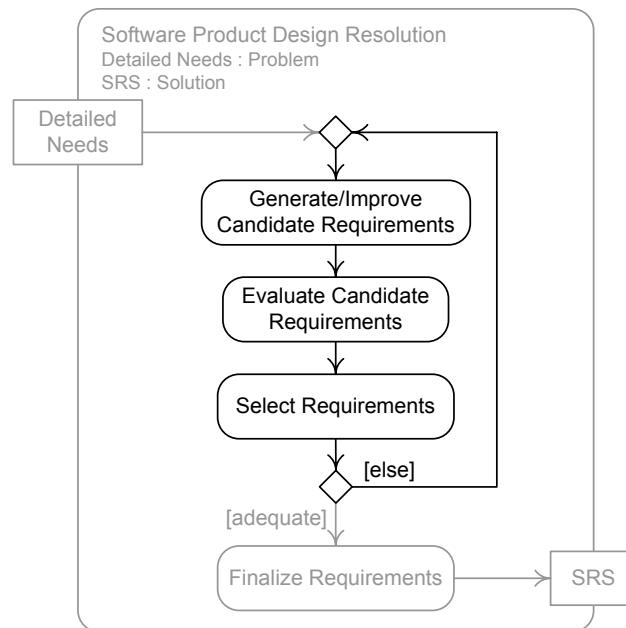


Figure 6-O-1 Software Product Design Resolution

By the end of this chapter you will be able to

- Explain and illustrate actors, use cases, and scenarios;
- Explain what use case diagrams are and state and explain rules and heuristics for their use;
- State what use case descriptions are and list the information that may be included in them;
- Draw use case diagrams and write use case descriptions;
- Explain what a use case model is; and
- Employ use case models to generate, refine, evaluate, and select product design alternatives.

Chapter Contents 6.1 UML Use Case Diagrams
6.2 Use Case Descriptions
6.3 Use Case Models

6.1 UML Use Case Diagrams

Refining Functional Requirements

Functional requirements are usually specified at a high level of abstraction and then refined. User-level functional requirements state what a product must do to support users in achieving their goals or carrying out tasks. These requirements are refined into operational-level requirements that specify the data passed between the product and its environment, the order in which this exchange takes place, the computations that transform inputs into outputs, and what happens when something goes wrong. Operational-level requirements are eventually refined into physical requirements that specify the details of a product's form and behavior.

In this chapter, we consider how to resolve user-level and operational-level functional requirements.

Why Use Case Modeling Works

If designers have carefully collected and analyzed stakeholder needs, then they can construct an initial list of user-level requirements by simply transforming the needs list into requirements statements, filtering out those needs that do not have to do with product function in the process. However, the resulting requirements will likely be inconsistent, incoherent, and too ambitious for the project. Consequently, designers must use the techniques discussed in Chapter 5 to generate and select a collection of user-level requirements forming a coherent product concept.

Designers can then refine each user-level requirement based on detailed needs elicited from stakeholders until they are able to form a complete collection of operational-level requirements. Again, the techniques discussed in the last chapter can be used for this work.

The problem with this approach to generating and refining requirements is that the requirements are considered in isolation as specifications of individual product functions and features. When designers and stakeholders try to imagine the product as a whole, they may have trouble combining individual requirements into a coherent idea of what the product will do and how it will work. They will also be unlikely to notice inconsistencies, missing requirements, redundancies, and other problems. This is especially true for large and complex products.

An entirely different and very powerful approach to resolving functional requirements is to focus on how the product under development will interact with users. The focus on the product in use structures design thinking; connects individual actions into coherent, extended activities; and makes it easy to assess product features against stakeholder goals and tasks. Having specific interactions to focus on also makes it easier to come up with design alternatives and to judge alternatives against one another. This approach models product and user interactions as use cases.

**Use Cases
and Actors**

A use case characterizes a way of using a product or represents a dialog between a product and its environment. Thus, a use case captures an interaction between the product and one or more external entities with which it interacts.

A **use case** is a type of complete interaction between a product and its environment.

The collection of use cases for a product should characterize all its externally observable behavior.

A use case must be complete in the sense that it forms an entire and coherent transaction—the sort of thing that we would be inclined to name. Making a cash withdrawal at an ATM machine, placing a call on a telephone, or printing a file are examples of complete interactions that would qualify as use cases.

The external agents with which a product interacts are called *actors*.

An **actor** is a type of agent that interacts with a product.

The product itself is never an actor. Actors may be human or non-human. They may interact with a product by exchanging data with it, invoking one of its operations, or having one of their own operations invoked by the product.

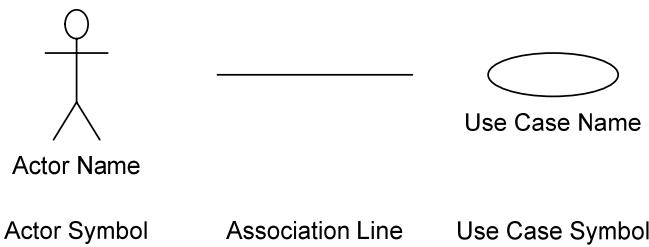
Actors are abstractions of actual individual users, systems, or devices typifying the roles played in product interactions. Some examples of actors are Dispatcher, Clerk, Printer, Communications Channel, and Inventory System.

**Use Case
Diagrams**

UML includes a notation for representing use cases, actors, and the participation of actors in use cases: the use case diagram.

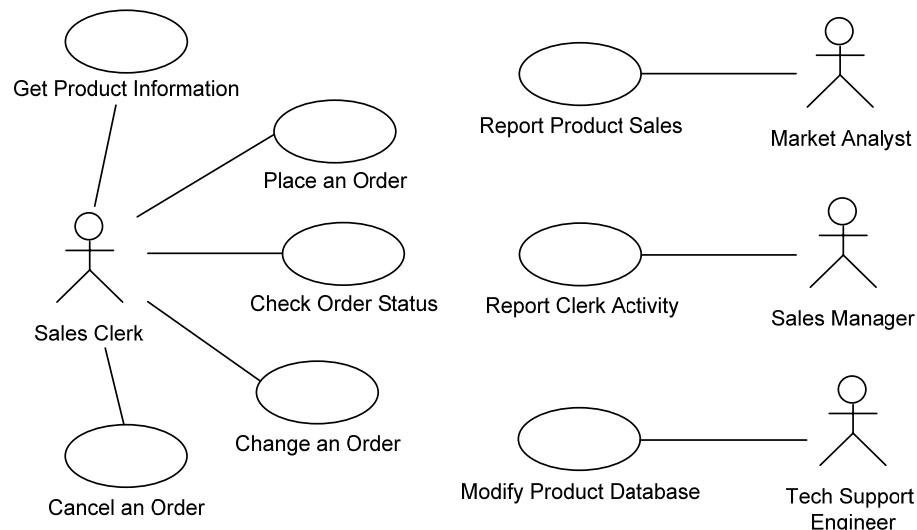
A **use case diagram** represents a product's use cases and the actors involved in each use case.

Use cases are represented in use case diagrams by ovals with the use case name either below or within the oval. Each actor is represented by a stick figure with the actor's name below it. An *association line* connects actors with the use cases in which they participate. The line does not have a label, arrowheads, or any textual annotation. These symbols are shown in Figure 6-1-1.

**Figure 6-1-1 Use Case Diagram Elements**

In this section we discuss use case diagrams employing only these three symbols; UML offers more symbols for elaborating and adorning use case diagrams, but these three are sufficient for most needs. Advanced use case diagram features are discussed in the sources cited for Further Reading at the end of the chapter.

The use case diagram in Figure 6-1-2 illustrates this notation. This example shows the use cases for a Catalog Sales Support System.

**Figure 6-1-2 Sales Support System Use Case Diagram**

This use case diagram indicates that the Catalog Sales Support System has four actors, three of whom participate in only one use case. The **Sales Clerk** actor participates in five use cases.

UML permits drawing a rectangle around the use cases, with the actors outside, to represent the system. This *system box* can be labeled with the system name. The system box is not needed though, and in fact it is rather misleading because it suggests that use cases are entities within a system even though they represent interactions between a system and its actors.

Scenarios Use cases are types of interactions, which means they are abstractions of interactions between the product and specific individuals. This contrasts with scenarios.

A **scenario** is an interaction between a product and particular individuals.

Use cases abstract scenarios that are *instances* of the same kind of interaction between a product and various actors. To illustrate, consider the scenario in Figure 6-1-3.

Mary Smith inserts her bank card into an active ATM machine. The ATM prompts her for her PIN, and she types 2384. The machine displays a transaction menu. Mary chooses a balance inquiry on her checking account. The ATM reports that she has \$1329.67 in her account and again displays the transaction menu. This time Mary chooses to end the interaction, and the ATM releases her card. Mary removes it and the ATM returns to its ready state.

Figure 6-1-3 A Balance Transaction Scenario

A use case abstracts scenarios such as the one in Figure 6-1-3 to provide a general specification for like interactions. Figure 6-1-4 describes an ATM Balance Inquiry use case:

A balance inquiry begins when a Customer inserts his or her bank card into an active ATM machine in its ready state. The machine prompts for the Customer's PIN. The Customer types the PIN. If the PIN is incorrect, the ATM displays an error message and asks the Customer to try again. The Customer gets three tries. After the third failure, the ATM keeps the card, displays a message telling the Customer to see a teller, and returns to its ready state after 20 seconds. If the Customer enters a valid PIN, the machine presents a transaction menu. The Customer chooses a balance inquiry on either checking or savings. The ATM displays the current balance and redisplays the transaction menu. This continues until the Customer chooses to terminate the interaction. The ATM releases the bank card. The Customer removes the bank card, and the machine returns to its ready state. If the bank card is not removed within 20 seconds, the machine retains the bank card.

Figure 6-1-4 ATM Balance Inquiry Use Case

Note that the use case generalizes the scenario. It describes the interaction between the product and actors rather than individuals; it abstracts details that vary between scenarios, such as the PIN and the particular balance inquiry made; and it includes all paths that the interaction can take in different scenarios.

Using Scenarios

Scenarios help designers envision how a product may be used. Designers can work up a collection of scenarios to explore ideas about how a product will be used, the features it will provide, the individuals who will interact with it, and how particular interactions might transpire. These scenarios can then be analyzed to abstract use cases, actors, and the flow of actions during an interaction.

Making Use Case Diagrams

There are many ways to make use case diagrams. We will first consider an approach based on scenarios and documents produced from product design analysis.

The process begins by generating scenarios for the envisioned product. The collection of scenarios should cover the main features of the product. It may include episodes of unsuccessful interactions or problems. The size of the collection depends on how complicated the product is and how thoroughly the designers explore their ideas for how the product might work.

Use case diagram construction starts by abstracting the individuals in the scenarios as actors in the use case diagram. This activity is supplemented by examining the stakeholders-goals list (discussed in Chapter 4) to identify stakeholders who directly interact with the product. For example, the stakeholders who directly interact with AquaLush are **Maintainer**, **Operator**, and possibly **Installer**.

Designers must be careful not to forget non-human actors. Designers can often find such actors by reviewing the project mission statement and the needs list and listing specific devices or systems that interact with the product. Individuals can be generalized as actors. For example, the AquaLush software must interact with irrigation valves and sensors, so these are AquaLush actors.

Stakeholders, devices, and systems involved with the product at one remove (or more) are not actors. For example, if a cashier or clerk interacts with a product on a patron's behalf, and the patron never directly interacts with the product, then the patron is not an actor, although the clerk and cashier are.

Designers draw actors around the edges of the use case diagram as the actors are identified.

The next step is to identify use cases. Scenarios usually correspond to single use cases, but several scenarios may be instances of a single use case, especially if they narrate both successful and failed interactions. For example, a scenario like the one in which Mary Smith forgets her PIN and so cannot complete a balance inquiry is an instance of the **ATM Balance Inquiry** use case, not a separate use case. Designers must compare scenarios to figure out the best ways to abstract them into a set of use cases. As use cases are identified, they are named and added to the diagram along with association lines connecting them with participating actors.

Designers can next consider each actor in turn and consult the needs list to make sure that there are use cases to meet every actor's needs. Several actors may participate in a single use case, so the same use case may come up several times as each actor is considered.

The activity diagram in Figure 6-1-5 summarizes this process.

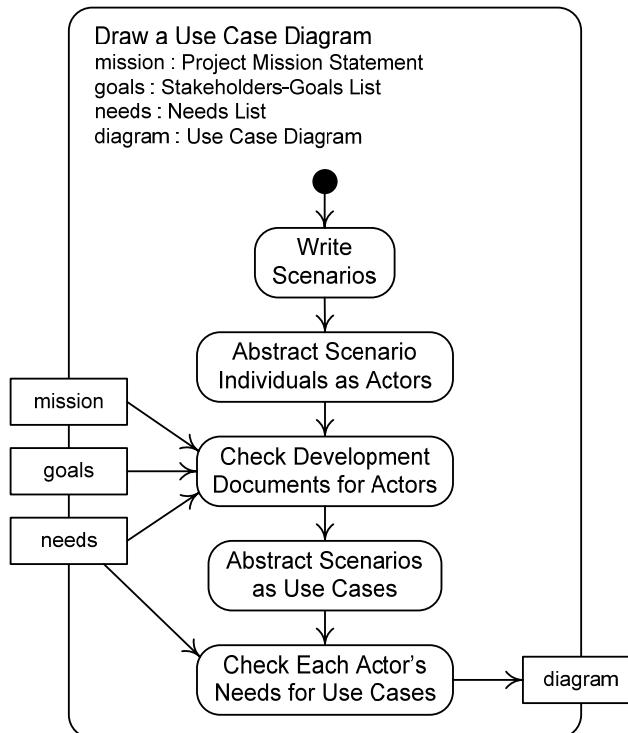


Figure 6-1-5 A Use Case Diagramming Process

Event Lists An alternative way to create a use case diagram is to make a list of all the events, both internal and external, to which the product must respond in some way. This is called an **event list**. For example, AquaLush must respond to the following events:

- An operator sets the time of day or irrigation parameters.
- An operator turns automatic irrigation on or off.
- A maintainer asks for a report of failed valves and sensors.
- A maintainer indicates that some failed valves or sensors are repaired.
- The program starts up.
- The clock reaches the time for automatic irrigation to begin.

The next step is to invent use cases to handle all the events and add them to the diagram. For example, possible AquaLush use cases to handle the

events in this list are: Set Irrigation Parameters, Toggle Irrigation, Report Failures, Make Repairs, Start Up, and Irrigate.

The final step is to consider each use case, determine the actors involved in it, and add them to the diagram along with association lines. For example, the AquaLush **Irrigate** use case involves **Valves** and **Sensors**, so these would be added to the diagram and associated with this use case. The result of such an exercise is pictured in Figure 6-1-6.

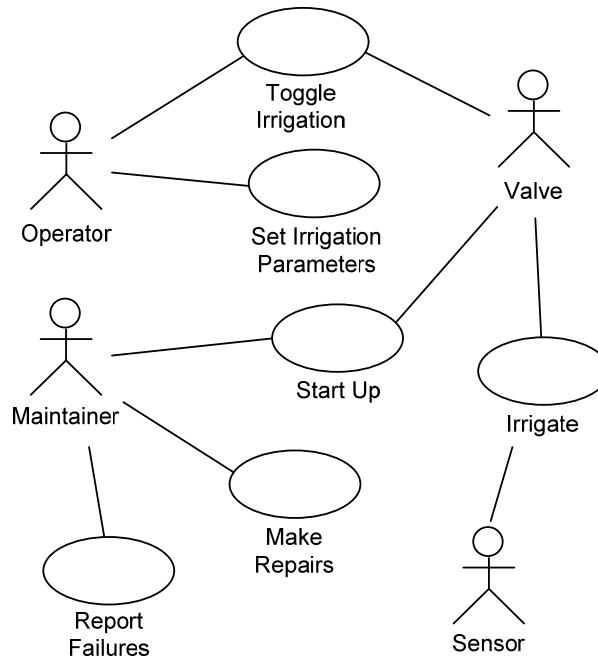


Figure 6-1-6 An AquaLush Use Case Diagram

Use Case Briefs

It may be useful to supplement a use case diagram with short descriptions of the actors and use cases, called use case or actor **briefs**. These actor and use case briefs can be only a sentence or two in length. For example, the following briefs are for the actors and use cases in Figure 6-1-6:

Maintainer—A person responsible for repairing irrigation valves and sensors.

Operator—A person who supervises and controls irrigation at a site.

Valve—An irrigation valve controlled by AquaLush.

Sensor—A soil moisture sensor read by AquaLush.

Toggle Irrigation—Turn automatic irrigation on or off.

Set Irrigation Parameters—Set the current time and date, the time and date for automatic irrigation, the critical moisture levels for each irrigation zone, the water allocation, and so forth.

Report Failures—Provide the identifiers and locations of failed sensors and valves.

Make Repairs—Tell AquaLush that a sensor or valve is repaired and can now be used in irrigation.

Start Up—Initialize the system when it is first turned on or comes on after a power failure.

Irrigate—Automatically irrigate the site at the irrigation time under the control of the moisture sensors.

Use Case Diagram Formation Rules

Every use case diagram must have at least one actor and at least one use case, and each use case must be associated with at least one actor. Every association must have an actor at one end and a use case at the other. Associations never connect actors to actors or use cases to use cases. Every use case and actor must be named, but association lines must not.

Use Case Diagram Heuristics

The following heuristics guide creation of good use case diagrams:

Never make the product an actor. Only agents outside the product that directly interact with it are actors.

Name actors with noun phrases. Actors are types of individuals that interact with a product, so actor names should be noun phrases, but (usually) not proper names. Actor names often identify roles, as in **Sales Representative**, **Dock Superintendent**, or **Billing System**.

Name use cases with verb phrases. Use cases are types of interactions between actors and a product, so they should be named by verb phrases—for example, **Record Sale**, **Enter Manifest**, or **Obtain Billing Record**. The exception to this rule is when an interaction is an activity with a standard name. For example, we might name a use case **Registration** rather than **Register for Classes**, because registration is the name for this activity.

Use case modelers often have a hard time deciding on use case size, making them either too big (**Manage a Raffle**) or too small (**Enter Password**). The following heuristics can help designers choose use cases with good granularity:

Achieve a stakeholder's goal in a use case. If a use case does not achieve some stakeholder's goal, it is probably too small. **Enter Password** does not achieve a stakeholder's goal, so it is too small to be a use case.

Exceptions to this rule are usually small activities that are common to many use cases, such as logging into a system. Although logging in does not (usually) achieve a stakeholder goal, it is often a prerequisite for so many use cases that it is easier to make it a separate use case than to include it as the first step in many others.

Make use cases that can be finished in a single session. A use case that cannot be completed in a single session is probably too big. Something like **Manage**

a Raffle would presumably take many sessions over days or weeks to complete, so it is too big to be a use case.

Make use cases of uniform size and complexity. Mixing large and complex use cases with small and simple ones is often an indication that the use cases need reorganization. Split the big ones and combine the small ones to make them more or less the same size and complexity.

In general it is best to draw a use case on a single page, an observation we embody as the next heuristic. A large product may have tens or even hundreds of use cases—too many to fit on a page. The second of the following heuristic helps to spread a large use case diagram over several pages:

Draw use case diagrams on one page. If a diagram is too big, it can be split across several pages using some organizational principle to determine the contents of each page.

Organize use cases by actor, problem domain categories, or solution categories. When organizing by actor, all the use cases involving a particular actor should be put in one diagram (use cases involving several actors will appear in several diagrams). For example, if use cases in a banking system are organized by natural divisions in the problem domain, then use cases for investment accounts might be in one group, those for non-commercial checking and savings accounts in another group, and those for commercial accounts in a third group. If they are organized by natural divisions in the product, then use cases for deposits and share purchases might be in one group; those for withdrawals, fund transfers, and share sales in a second group; and those for account queries and reports in a third group.

Checking Use Case Diagrams

Once a diagram is drafted, designers can check that diagram in the following ways:

- Review the stakeholders-goals list to make sure that all stakeholders who are actors appear on the diagram.
- Review the needs list to ensure that no actors and use cases have been forgotten.
- Review the constraints and limitations in the project mission statement to make sure that they are satisfied.
- Generate an event list and check that every event is handled by some use case.
- Check that the collection of use cases covers all externally visible program behavior.
- Make sure that the diagram is well formed by checking that every actor and use case symbol is labeled, and that no association line is labeled.
- Check that the actors are named with noun phrases, use cases are named with verb phrases, use cases are of uniform size, and so forth.

Designers can easily create a checklist based on these and other considerations for use in use case diagram reviews.

When to Employ Use Case Diagrams

Use case diagrams catalog the actors that interact with a product and the use cases that the product supports. They may be supplemented with actor and use case briefs, but use case diagrams supply only a small amount of information about a product.

As we will see in the third section of this chapter, use case diagrams are nevertheless useful for generating user-level functional design alternatives and summarizing these alternatives for refinement and evaluation. They also summarize user-level functional requirements and serve as an index for *use case descriptions*, which model operational-level requirements in great detail. Together use case diagrams and use case descriptions provide a powerful modeling tool for generating, exploring, refining, and documenting user- and operational-level functional requirements. To summarize, **use case models** have two parts:

Use Case Diagram—A static model of the use cases supported by a product. It names product use cases and the actors involved in each use case.

Use Case Descriptions—A dynamic model of the interactions between the product and the actors in a use case. A product's use case descriptions together form a dynamic model of product behavior.

We discuss use case descriptions in Section 6.2 and use case models and their application to product design in Section 6.3.

Heuristics Summary

Figure 6-1-7 summarizes use case diagram heuristics.

- Never make the product an actor.
- Name actors with noun phrases.
- Name use cases with verb phrases.
- Achieve a stakeholder's goal in a use case.
- Make use cases that can be finished in a single session.
- Make use cases uniform in size and complexity.
- Draw use case diagrams on a single page.
- Organize use cases by actor, problem domain categories, or solution categories.

Figure 6-1-7 Use Case Diagram Heuristics

Section Summary

- Use case modeling is a superior technique for generating and refining user- and operational-level functional requirements.
- An **actor** is a type of agent that interacts with a product. Actors represent human roles, systems, or devices that interact directly with the product.

- A **use case** is a type of complete interaction between a product and one or more actors. Use cases abstract **scenarios**, and they must be whole and complete interactions.
- **Use case models** consist of use case diagrams and use case descriptions.
- A **use case diagram** represents the use cases supported by a product, the actors that interact with the product, and the associations between actors and use cases.
- Use case diagrams can be created by listing actors and then figuring out the use cases involving each one.
- Use case diagrams are useful during user-level functional requirements generation, refinement, and evaluation, as well as for documenting functional requirements.

**Review
Quiz 6.1**

1. Why should the product never be an actor in a use case diagram?
 2. Is it ever a good idea to name a use case with a noun phrase?
 3. What can designers do to check that a use case diagram contains all the use cases that it should?
 4. Give three examples, different from those in the text, of activities that are too small to be use cases.
-

6.2 Use Case Descriptions

**What Is a
Use Case
Description?**

Use case diagrams show the use cases that a product supports, and they may be accompanied by brief actor and use case descriptions. However, use case diagrams are static models that catalog types of interactions at a high level of abstraction. The aim of use case descriptions is to provide a dynamic model showing the details of the interaction between a product and its actors in each use case. They might also be thought of as stating a contract about product behavior.

A **use case description** is a specification of the interaction between a product and the actors in a use case.

Use case descriptions must say what each party does, with attention to the order of actions and responses to actions. Various responses are typically possible, depending on the actions that came before. Thus, use case descriptions must sometimes specify complex activity flows.

Use case descriptions may be expressed in a variety of notations. There is no standard use case description notation. In particular, UML does not specify any sort of notation for them. Any notation that can detail an interaction between several agents can be used to describe use cases, including UML activity diagrams, UML interaction diagrams, data flow diagrams, flow charts, and programming languages. The preferred use case

description notation is text, usually formatted according to a standard template. We use text for use case descriptions, and we will introduce a standard use case template in this section.

Creating use case descriptions is a form of technical writing, and good use case descriptions should have the virtues of good technical writing: They should be clear, easy to read, complete, and unambiguous.

Use Case Description Contents

Use case descriptions can be written to include or abstract many details about an interaction:

Use Case Name and Number—This is basic bookkeeping information that should always be included.

Actors—The agents participating in the use case.

Stakeholders and Needs—Use cases must meet stakeholders' needs, so it is useful to list the stakeholders whose needs are met by the use case, along with the needs themselves.

Preconditions—A **precondition** of an activity or operation is an assertion that must be true when the activity or operation begins. A use case description's readers and writers may assume that its preconditions are true. A use case does not check that its preconditions are true.

Postconditions—A **postcondition** of some activity or operation is an assertion guaranteed to be true when that activity or operation finishes. Postconditions are what everyone can count on when the use case is done. Postconditions must ensure that stakeholder needs have been met.

Trigger—The **trigger** is the event that causes a use case to begin.

Basic Flow—The **basic flow** documents a typical successful flow of action in the dialog between the product and its actors. It must begin at the triggering event and continue until successful completion of the use case.

Extensions—Alternative action flows typically occur, either because of normal variations in the interaction or because of errors. These alternative flows are documented as **extensions** to the main flow.

Extensions can begin anywhere in the use case after the trigger and may lead to successful or unsuccessful use case completion, or back to the basic flow. An extension may also be an alternative to another extension.

The use case description in Figure 6-2-1 on page 170 illustrates all these elements. This use case is for a program that supports system administrators by recording all the computer users they serve, the computers they are responsible for, and assignments of computers to computer users. The program produces reports used by accountants to inventory and depreciate capital equipment, so they are stakeholders in this product, though not actors.

Use Case 1: Modify a Computer Record**Actors:** Administrator**Stakeholders and Needs:**

Administrator—To modify the database.

Computer Users—To have accurate data in the database.

Accountants—To have accurate and complete data in the database.

Preconditions: The Administrator is logged in and has a computer identifier.**Postconditions:** The database is modified only if all correctness and completeness checks on the modified record succeed and the Administrator confirms the changes. Computer record edits are always saved unless the Administrator cancels the transaction.**Trigger:** Administrator initiates a computer record modification transaction.**Basic Flow:**

1. Administrator initiates the transaction and enters the computer identifier.
2. CAS displays all data for the indicated computer.
3. Administrator edits the data for the computer.
4. CAS verifies the changes and asks for confirmation that they should be accepted.
5. Administrator confirms the changes.
6. CAS modifies its database and informs the Administrator that the transaction is complete.

Extensions:

- *a Administrator cancels the operation: The use case ends.
- 1a The computer identifier is invalid:
 - 1a1. CAS alerts the Administrator of the problem.
 - 1a2. Administrator may Make a Query and correct the problem, and activity resumes.
- 3a Administrator directs CAS to execute a held transaction (see 6a):
 - 3a1. CAS modifies its database to complete the held transaction and informs the Administrator that the transaction is complete, and the use case ends.
- 4a CAS detects invalid or incomplete data:
 - 4a1. CAS alerts the Administrator to the problem.
 - 4a2. Administrator corrects the problem and activity resumes.
- 6a Administrator does not confirm the changes: The use case ends.
- 6a CAS is unable to modify its database:
 - 6a1. CAS records the transaction for later completion, informs the Administrator of the problem, and asks whether the transaction should be held.
 - 6a2. Administrator confirms that the transaction should be held.
 - 6a3. CAS verifies that it is holding the transaction and the use case ends.

Figure 6-2-1 A Use Case Description**Use Case Description Formats**

There are two kinds of textual use case description formats, with endless variations. By far the most popular alternative is a narrative as shown in Figure 6-2-1, with or without numbered steps, detailing the interaction. The other sort separates each actor's activities into columns of a table. This latter approach makes it very clear who is doing what but uses a lot of space. Variations on these themes involve the amount of detail included in the description, the arrangement of fields, and numbering conventions.

The format used in Figure 6-2-1, and the one we adopt in this book, is a slight variation of Alistair Cockburn's "fully dressed" format. Some aspects of this format—particularly the following features—may not be self evident:

- Underlined text refers to another use case. In Figure 6-2-1, the underlined phrase Make a Query indicates that the Administrator may run the Make a Query use case before continuing.

- The Extensions section uses a complicated numbering scheme. The identifier for an extension begins with the identifier of the step where the alternative flow diverges, followed by a letter designating the alternative. The extension heading states the condition that leads to the alternative—a colon terminates this condition. The next steps have identifiers consisting of the extension identifier followed by sequence numbers for the steps in the alternative flow. For example, in the **Modify a Computer Record** use case of Figure 6-2-1, extension 6a is an activity flow that diverges from the basic flow at step 6 under the condition that CAS is unable to modify its database. Steps 6a1 through 6a3 constitute the alternative flow resulting from this condition. Indentation is used to make the structure of the extensions easier to read. Had there been other alternative flows at step 6 of the basic flow, they would have been labeled 6b, 6c, and so on. The asterisk (*) is a wildcard symbol that stands for any action step identifier. Hence, the extension labeled *a is an alternative that can occur at any step. Sequences of steps can also be indicated using a dash, so 3-6a is an alternative that can occur at steps 3, 4, 5, or 6.

Extensions could be written with `if` statements, but complicated alternatives would be harder to read than they are using the scheme described here.

Use Case Description Template

Figure 6-2-2 shows our use case description template. The italic text describes how to fill in each field of the template.

Use Case number: <i>name</i> Actors: <i>actorList</i> Stakeholders and Needs: <i>stakeholder—needsList.</i> <i>...</i> <i>stakeholder—needsList.</i> Preconditions: <i>what is assumed at the start.</i> Postconditions: <i>what is guaranteed at the end.</i> Trigger: <i>the event that starts the use case.</i> Basic Flow: <i># stepDescription</i> <i>...</i>	Extensions: <i>extensionIdentifier condition:</i> <i>extensionIdentifier # stepDescription</i> <i>...</i> <i>...</i>
--	---

Figure 6-2-2 Use Case Description Template

The pound sign (#) stands for a sequential step number. The *extensionIdentifier* is formed as described above.

How to Write Use Case Descriptions

Use case descriptions are written for each use case in a use case diagram. If a collection of scenarios is available then the scenarios that are instances of the use case being described are valuable resources. Designers might write additional scenarios to explore extensions. The stakeholders-goals and needs lists are also useful for filling in the Stakeholders and Needs field of the template.

The best way to write a use case description is to fill in the template from top to bottom. The use case diagram already shows the use case name and actors. Use case numbers are arbitrary and can be assigned beginning at one.

The human actors in a use case are almost always stakeholders, and their needs are often to accomplish the task of the use case. Other stakeholders who do not interact directly with a product may also have needs that must be met in a use case. Checks and tests in a use case are often there to meet such needs. For example, use cases may have to include actor identity checks to protect the privacy or financial needs of offstage stakeholders, as well as the need for accurate and complete data. Listing stakeholders and their needs helps description writers remember them and helps readers understand why the use case works as it does.

The needs list should be reviewed when writing each use case, and stakeholder needs should be considered in light of the interaction that occurs in the use case. If the use case can affect whether a stakeholder need is met or not, then that stakeholder and need should be listed in the Stakeholders and Needs section.

Preconditions must be true before a use case begins. For example, a use case precondition might be that a certain actor is logged in. Although many things may be true before a use case begins, only those that matter to the interaction should be listed in the Preconditions section. Often, it is easier to fill in preconditions as they become clear when writing the Basic Flow and Extensions sections.

Postconditions must be true after a use case ends, whether successfully or not. For example, a use case postcondition might guarantee that a transaction is not done unless a qualified actor confirms it. Many things may be true after a use case ends, but only those important for meeting stakeholder needs should be listed in the Postconditions section. Each need in the Stakeholders and Needs field should be considered to come up with use case postconditions.

Determining the Trigger and Basic Flow

The trigger is the event that starts a use case. Often, the trigger is also the first step in the use case. For example, a **Buy a Book** use case for a Web-based bookstore might be triggered when the **Buyer** begins the purchasing transaction, which is also the first step in the use case. But sometimes, the trigger precedes the first step. In **AquaLush**, the **Irrigate** use case is triggered when the current time reaches the irrigation start time, but the first step of the use case is that **AquaLush** reads the **Sensors** until it finds one with a reading below its critical moisture level; it then opens its associated **Valves**.

The use case template fields above the Basic Flow section set the scene for the heart of the description, which is specification of the interaction, in all its variation, that occurs in the use case. In writing the basic flow, choose a common, simple activity flow from the trigger through successful use case completion, and write down its steps in simple declarative sentences.

Often, a successful scenario is a good source for the steps in the basic flow. The steps can assume the preconditions and should achieve the postconditions. The example Basic Flow in Figure 6-2-3 is from a use case description for the Fingerprint Access System, a program that uses fingerprints to monitor and control entry to and exit from a secure facility.

Use Case: Enter Secure Facility

Trigger: Patron presses finger on Fingerprint Reader.

Basic Flow:

1. Fingerprint Reader scans the fingerprint and sends its image to the Fingerprint Access System.
2. Fingerprint Access System validates the fingerprint and unlocks the Entry Gate.
3. Entry Gate signals the Fingerprint Access System that it has cycled.
4. Fingerprint Access System locks the Entry Gate, logs an entry event, and increments its occupant count.

Figure 6-2-3 Example Basic Flow

Note that each step describes the actions of a single actor or the product itself, and that each clearly indicates which entity is active. Steps will typically describe either a communication sent between the product and actors, or some processing internal to the product needed to meet stakeholder needs.

The action steps in the basic flow can be supplemented with directions about doing the steps iteratively or concurrently.

Writing Extensions

Once the basic flow is defined, the extensions can be specified. These alternatives are called **extensions** because they extend the activity flow in a different direction from a **branch point**, which is simply a place where the action flow may diverge. Thus, in writing an extension, the branch point must be marked, the condition that causes the branch must be stated, and the alternative activity flow must be described.

The first task, then, is to brainstorm all the branch points and conditions in the basic flow. Scenarios for failed or alternative interactions are excellent sources for branch points and conditions. Designers also need to consider each basic flow step in turn, and for each one list all the conditions that lead to a different activity flow, including errors or faults. The following questions help with this brainstorming activity:

- Is there an alternative way to finish the use case from this step?
- What happens if the actor enters bad data?

- What happens if the actor does not respond at this point?
- What happens if entered data fails a validation?
- How can the product fail at this step?

The result of brainstorming should be a list of extension points and conditions causing alternate flows. The list in Figure 6-2-4 is the result of brainstorming the basic flow in Figure 6-2-3.

- 1a Fingerprint Reader fails to scan the fingerprint.
 - 2a Fingerprint fails validation.
 - 2b Entry Gate fails to unlock.
 - 2c Entry Gate sends a cycle signal before being unlocked.
 - 3a Entry Gate cycle signal never arrives.
 - 3b Fingerprint Reader sends a fingerprint image.
 - 4a Entry Gate fails to lock.
 - 4b Unable to write to the log.

Figure 6-2-4 Candidate Branch Points and Conditions

The next step is to rationalize the brainstormed list of branch points and conditions. Conditions that the product cannot detect or cannot or should not do anything about should not be considered further. In the example from Figure 6-2-4, the Fingerprint Access System cannot detect Fingerprint Reader and Entry Gate failures, so all these can be removed from the list. Only one person is allowed to enter the facility at a time, so extraneous fingerprint images are ignored during the use case—this means that condition 3b is ignored. Poorly stated conditions should also be rewritten. In this case, the vague condition “Entry Gate cycle signal never arrives” is rewritten into a testable condition.

The final list appears in Figure 6-2-5.

- 2a Fingerprint fails validation.
 - 2b Entry Gate sends a cycle signal before being unlocked.
 - 3a Entry Gate cycle signal does not arrive within 30 seconds of unlocking the Entry Gate.
 - 4a Unable to write to the log.

Figure 6-2-5 Rationalized Branch Points and Conditions

These branch points and conditions can be added to the Extensions section.

The steps in each extension should now be written. Each one can be treated as if it were a separate use case, with its entry condition as trigger, and its completion of the original use case or recovery from failure as its goal state. Scenarios for failed or alternative interactions are once again an

excellent resource for writing extensions. The basic flow of the extension is written first, followed by extensions to the extension, if any. Figure 6-2-6 shows the final version of the basic flow and extensions of the Enter Secure Facility use case example we have been considering.

Use Case: Enter Secure Facility

Trigger: Patron presses finger on Fingerprint Reader.

Basic Flow:

1. Fingerprint Reader scans the fingerprint and sends its image to the Fingerprint Access System.
2. Fingerprint Access System validates the fingerprint and unlocks the Entry Gate.
3. Entry Gate signals the Fingerprint Access System that it has cycled.
4. Fingerprint Access System locks the Entry Gate, logs an entry event, and increments its occupant count.

Extensions:

2a Fingerprint fails validation:

- 2a1. Fingerprint Access System logs a failed entry event and the use case ends.

2b Entry Gate sends a cycle signal before being unlocked:

- 2b1. Fingerprint Access System logs an anomalous entry gate signal event and continues where it was interrupted.

3a Entry Gate cycle signal does not arrive within 30 seconds of unlocking the Entry Gate:

- 3a1. Fingerprint Access System locks the Entry Gate and logs an aborted entry event, and the use case ends.

2*1, 3a1, 4a Unable to write to the log:

- 4a1. Fingerprint Access System locks the Entry Gate and writes a logging failure message to the console, ending the use case.

Figure 6-2-6 A Basic Flow and Extensions

We summarize the process for writing use case descriptions in the activity diagram shown in Figure 6-2-7 on page 176.

Use Case Description Heuristics

The following heuristics provide guidance in writing good use case descriptions:

Fill in the use case template from top to bottom. The fields appear in the order they do because that is the easiest way to fill them in. However, expect lots of revisions as realizations are made during the writing process.

Obtain the use case and actor names from the use case diagram, if there is one. This ensures that the use case diagram and use case descriptions are consistent.

Make human actors stakeholders whose needs include completion of the task done by the use case. Other stakeholders may need to be included as well, but this provides a start to the list.

Write simple declarative sentences in the active voice. Above all, use cases should be easy to understand. This is accomplished in part by writing simple declarative sentences in the active voice.

Make actors or the product the subject of each step description. It must be clear which entity carries out every step of an interaction. Making either

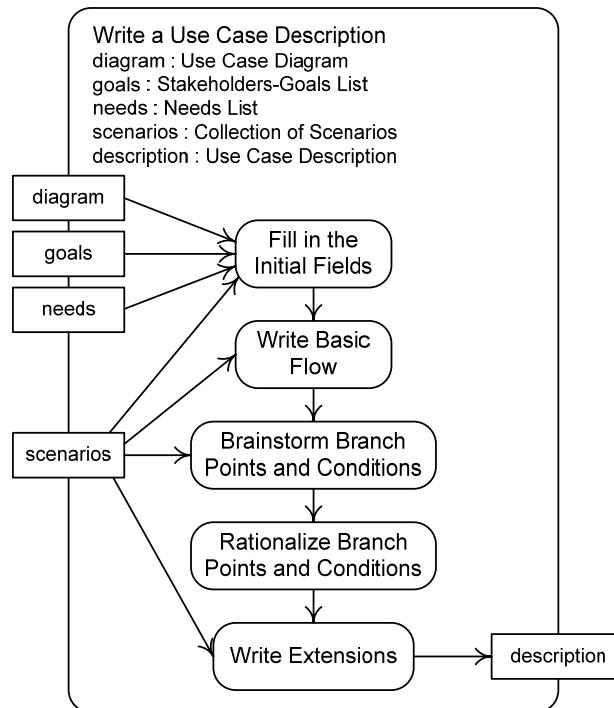


Figure 6-2-7 A Use Case Description Writing Process

actors or the product the subjects of most sentences in the description usually does this.

Write a basic flow with three to nine steps. A very short use case description often indicates either that the use case is too small or that the level of abstraction in the description is too high. A long description suggests that the use case is too big or that it contains too much detail. The basic flow in a use case description should be from three to nine steps, and each step should be only one or two sentences.

Avoid sequences of steps by the actors or the product. Use cases are interactions, so usually actors and the product take turns in the flow of activity. When several steps in a row are done by a single party, it often means that intervening actions by the other party have been left out.

Don't specify physical details. Use cases should be at the operational level of abstraction, so avoid specifying physical-level details—especially user interfaces. For example, specify that a human actor starts an operation, not that the actor presses a button or selects something from a menu.

Impose minimal order on activities. Use case descriptions naturally impose an order on the interaction between a product and its environment, but they should not over-specify this order. For example, a product needs a mailing address before it can mail something, but the order in which the mailing address fields are filled in does not matter. The use case

description must say that the address is obtained before mailing the item, but it should not specify the order in which the mailing address fields are filled in.

Proofread the description. It is a good idea to reread use case descriptions looking for grammatical and typographic errors, checking the numbering scheme, making sure that the preconditions are really needed, and ensuring that all postconditions are satisfied no matter how the use case ends.

The Purpose of Use Case Descriptions

Use case descriptions refine user-level functional requirements into operational-level functional requirements. There are many ways to add details during refinement, and use case descriptions can capture these design alternatives. The alternative descriptions can be evaluated and refined, and eventually a description can be chosen for the final design. Thus, use case descriptions are useful for functional design generation, refinement, evaluation, and documentation. We consider employing use case descriptions for specifying design alternatives in the next section.

Heuristics Summary

Figure 6-2-8 summarizes use case description heuristics.

- Fill in the use case template from top to bottom.
- Obtain use case and actor names from the use case diagram, if there is one.
- Make human actors stakeholders whose needs include completion of the task done by the use case.
- Write simple declarative sentences in the active voice.
- Make actors or the product the subject of each step description.
- Write a basic flow with three to nine steps.
- Avoid sequences of steps by the actors or the product.
- Don't specify physical details.
- Impose minimal order on activities.
- Proofread the description.

Figure 6-2-8 Use Case Description Heuristics

Section Summary

- A **use case description** is a specification of the interaction between a product and the actors in a use case.
- Use case description fields may include the use case name and number, the actors, relevant stakeholder needs, **preconditions** and **postconditions**, the **triggering** event, the interaction's **basic flow**, and its **extensions**.
- Writing a use cases description can begin by filling in the contextual fields (name, number, stakeholder needs, preconditions, and postconditions).
- The next step can be writing the triggering event and then writing the steps in a typical, simple, successful interaction as the basic flow.

- Branch points and conditions can be brainstormed and rationalized, and an extension can be written for each branch point, with the condition as a trigger.
- Use case descriptions are useful for operational-level requirements generation, evaluation, refinement, and documentation.

**Review
Quiz 6.2**

1. What notations can be used for use case descriptions?
 2. What are preconditions and postconditions?
 3. Why must a use case always have at least one stakeholder with a need relevant to the use case?
 4. List three heuristics for writing good use case descriptions.
-

6.3 Use Case Models

**Why Make Use
Case Models?**

As noted in Chapter 4, eliciting and documenting stakeholder needs for a product is only the first step in producing software requirements specifications. Needs are often in conflict, and designers must decide which needs to satisfy. A collection of needs, even when converted into the grammatical form of requirements specifications, does not constitute a coherent and complete description of a product. Many decisions must be made about how needs are satisfied and the form of the product.

This means that requirements development is really a product design activity. Like any design activity, product design should be done using an iterative process that generates and refines design alternatives, evaluates them, and selects the best ones for further refinement or as the final design solution.

Capturing a product design in a long list of requirements specifications makes it hard for designers to think about their designs. Use case models alleviate this problem by representing a product design in terms of coherent interactions between actors and the product. Use case diagrams catalog these interactions, while use case descriptions add interaction details. Together, they model a product's observable behavior at the user and operational levels of abstraction in a way that makes it fairly easy for designers to create and improve their designs.

With this in mind, designers make use case models to help represent their products during the design process. Use case models capture design alternatives during the generation and refinement step, and they represent alternatives that can be evaluated and chosen for further refinement or as a final design.

In this section we illustrate how to design with use cases by considering some design alternatives for AquaLush. Following a top-down process, we first consider two user-level design alternatives represented by use case diagrams. We then choose a use case and generate alternative operational-level specifications in the form of use case descriptions.

Designing with Use Case Diagrams

A use case diagram models a design alternative for the interactions that a product will support. Different alternatives may have different sets of actors, different use cases, and different overall product functionality. A good initial collection of design alternatives might be generated by having every design team member create several different diagrams and then comparing and contrasting them in a team meeting.

Use case diagram models can be evaluated in the many ways discussed in the last chapter. For example, each diagram can be accompanied with a list of unmet needs, development costs, time and risk estimates, conformance to design constraints, and ratings of feasibility, simplicity, and beauty. Stakeholders can be asked to evaluate them as well. The diagrams can also be evaluated for notational correctness and their quality as models, but such defects are easily corrected.

Use case diagrams are high-level models of product function, and as such they represent important design alternatives that merit thorough consideration when choosing among them. Designers can usually winnow down the alternatives to a few by discussing the pros and cons of each diagram or by using a simple scoring matrix. Stakeholders should be consulted for their input on major competing alternatives. Scoring matrices are probably a good method to use for the final decision.

To illustrate how use case diagrams can model alternative designs, consider the pair of diagrams in Figures 6-3-1 and 6-3-2 for the AquaLush product.

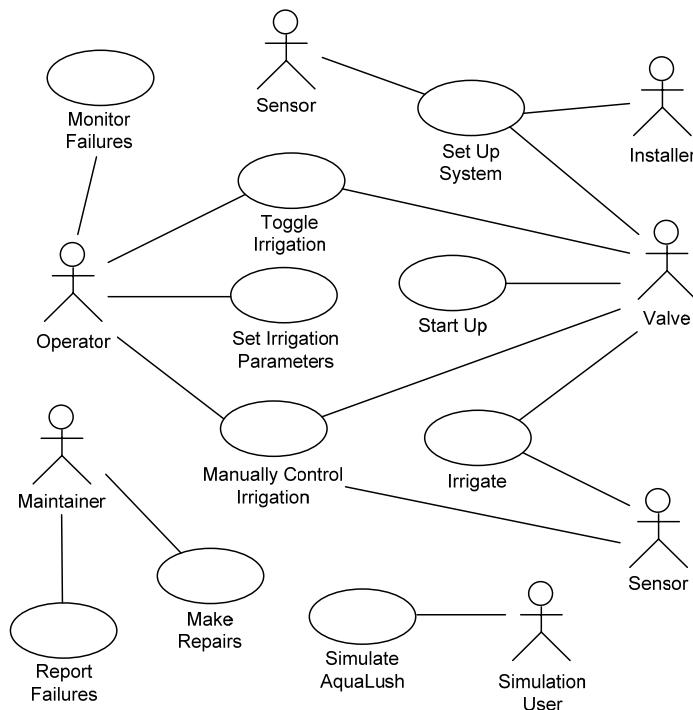


Figure 6-3-1 AquaLush Use Case Diagram A

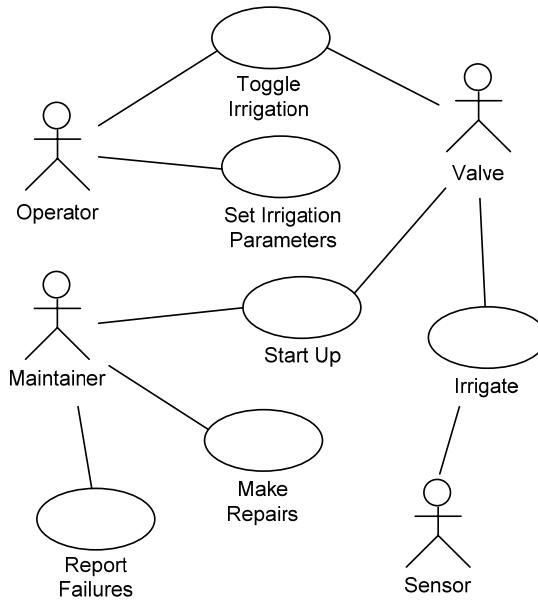
**Figure 6-3-2 AquaLush Use Case Diagram B**

Diagram A shows a “full-featured” product design and Diagram B a “minimal system” design. Alternative A shows AquaLush supporting installers in setting up the system, while alternative B envisions installers configuring AquaLush using some other product (such as a text editor creating a configuration file). In alternative A, AquaLush recovers from power failures on its own, while in alternative B the Maintainer must participate in this activity. Design A allows the Operator to choose between manual and automatic irrigation modes. In automatic mode, AquaLush controls irrigation; in manual mode, the Operator can open and close individual Valves. Alternative B operates only in automatic mode, though irrigation can be turned off completely. In alternative A the product alerts the Operator to valve and sensor failures as they occur, unlike alternative B. Alternative A also includes a Web simulation use case; alternative B does not. Several use cases covering core functions are common to both designs.

In evaluating these alternatives, note first that alternative A meets every need in the AquaLush needs list and satisfies all business requirements and then some; alternative B fails to meet several needs and does not meet the business requirement to develop a Web product simulation. On the other hand, design B would be much easier to realize than design A, making its development significantly cheaper, faster, and less risky and making the product more maintainable. Design B is also somewhat simpler and more elegant than design A.

Neither of these alternatives is ideal. Design A has more features than are necessary, while design B does not satisfy an important business requirement and does not meet enough stakeholder needs. A better design

incorporates the best features of these alternatives. Appendix B has a use case diagram showing the ultimate design.

Designing with Use Case Descriptions

After the interactions supported by a product are cataloged in a use case diagram, the interactions can be refined in use case descriptions. Different descriptions of the same use case represent design alternatives at a lower level of abstraction. These alternatives can be generated, refined, evaluated, and selected using the techniques we have discussed.

For example, consider the alternative descriptions of the AquaLush Manually Control Irrigation use case presented in Figures 6-3-3 and 6-3-4.

Use Case A: Manually Control Irrigation

Actors: Operator, Valve, Sensor

Stakeholders and Needs:

Operator—To schedule irrigation for certain times, to continue operating as normally as possible in the face of Valve and Sensor failures, to irrigate the site.

Maintainers—To detect and record Valve failures, to recover from power failures without Maintainer intervention.

Preconditions: AquaLush is in manual mode.

Postconditions: All Valve and Sensor failures are recorded.

Persistent store failures are reported.

Trigger: Operator selects a non-empty set of closed Valves and directs that they be opened.

Basic Flow:

1. Operator selects a non-empty set of closed Valves and directs that they be opened.
2. AquaLush opens each Valve in the set and displays to the Operator for each open Valve: its location, how long it has been open, how much water is has used, and the moisture level of its associated Sensor. AquaLush also shows the total water used since the start of the use case. The following Operator action and AquaLush responses may be done in any order, and repeatedly.
 3. Operator selects a set of open Valves and directs that they be closed.
 4. AquaLush closes the indicated Valves and removes them from the Valve status display.
 5. Operator selects a set of closed Valves and directs that they be opened.
 6. AquaLush opens the selected Valves and adds them to the Valve status display.
 7. The Operator indicates that he or she is finished.
 8. AquaLush acknowledges that manual irrigation is finished and closes all Valves.

Extensions:

2a,4a,6a, 8a A Valve fails:

- 2a1. AquaLush tries twice more to manipulate the Valve, and if it succeeds, the use case continues.
- 2a2. AquaLush alerts the Operator of the failure and records that this Valve failed in its persistent store, and the use case continues.

2b A Sensor fails:

- 2b1. AquaLush tries twice more to read the Sensor, and if it succeeds the use case continues.
- 2b2. AquaLush alerts the Operator of the failure and records that this Sensor failed in its persistent store.

2*2a AquaLush cannot write to its persistent store:

- 2*2a1. AquaLush alerts the Operator of the failure, and the use case continues.

Figure 6-3-3 Manually Control Irrigation, Alternative A

Use Case B: Manually Control Irrigation**Actors:** Operator, Valve**Stakeholders and Needs:**

Operator—To schedule irrigation for certain times; to continue operating as normally as possible in the face of Valve and Sensor failures; to irrigate the site.

Maintainers—To detect and record Valve failures; to recover from power failures without Maintainer intervention.

Preconditions: AquaLush is in manual mode.**Postconditions:** All Valve failures are recorded.

Persistent store failures are reported.

Trigger: Operator selects a non-empty set of closed Valves for manual control.**Basic Flow:**

1. Operator selects a non-empty set of closed Valves for manual control.
 2. Operator specifies how long each Valve in the set is to be open, and directs that manual irrigation begin.
 3. AquaLush opens each valve in the set.
 4. Every minute, AquaLush checks how long the Valves have been open and closes the Valves that have been open the specified amount of time.
- The use case ends when all Valves are closed.

Extensions:

3,4a A Valve fails:

3,4a1. AquaLush tries twice more to manipulate the Valve, and if it succeeds goes on as before.

3,4a2. AquaLush alerts the Operator to the failure and records that this Valve failed in its persistent store.

*a AquaLush cannot write to its persistent store:

*a1. AquaLush alerts the Operator of the failure and continues operation.

Figure 6-3-4 Manually Control Irrigation, Alternative B

These descriptions illustrate only two of the many alternatives that might be generated for manually controlled irrigation. Alternative A relies on the Operator to completely control each Valve, but AquaLush supplies lots of data about irrigation. Alternative B is easier for the Operator because after the Operator has chosen which Valves to open and for how long, AquaLush does the rest. Alternative A gives the Operator more control, but it is less convenient, slightly harder to implement, and slightly more prone to failure because it relies on more hardware (the Sensors). Alternative B is simpler and more convenient but does not help as much to conserve water and save money. Stakeholders might be consulted about the relative importance of convenience and control in evaluating these (and other) alternatives.

Designers might combine these factors in a scoring matrix or consider pros and cons when choosing between these alternatives.

Extracting Requirements from Use Case Models

Use case models do not include atomized requirements statements, but such statements can be extracted from them. Alternatively, use case models can serve as surrogates for requirements statements. Designers can study a use case model and write atomized requirements statements for a product behaving just as the model prescribes. Engineering designers can also study a use case model and design an implementation for programs that will behave as the model prescribes.

Both these activities require some effort. Particular functions, features, and properties may appear in several use cases or be assumed by one or more use cases, making it hard to extract requirements statements from a use case model. Use case models are a kind of functional decomposition, while engineering designs are either object-oriented decompositions or functional decompositions made in different ways to take implementation concerns into account. In either case, making an engineering design based on use cases is difficult.

On the whole, it is better to extract atomized requirements statements than to pass on the use case model to engineering designers as a surrogate for true requirements specifications, for the following reasons:

- Traceability requires atomized requirements specifications, which use case models do not provide.
- Products sometimes have functions that are not exhibited in use case diagrams. Any complete operation that a product does without interacting with its environment will not appear in a use case. For example, AquaLush might read a configuration file provided by the system installer when it starts up without interacting with any actors. This is an important product function that is not captured in a use case. Requirements statements are needed to document such “silent” functions.
- The exercise of extracting requirements statements provides yet another check of the design that may ferret out errors or opportunities to improve it.
- It is better to spend the extra time and effort during product design to make engineering design easier—this task is already hard enough.

We conclude that atomized requirements statements should be extracted from use case models.

Use case descriptions detail interactions but abstract from physical details, so requirements statements extracted from use case descriptions are at the operational level of abstraction. These are the bulk of the SRS functional requirements section. The use cases (or corresponding user-level requirements) can help organize these requirements in the SRS.

We conclude, then, that designing with use cases involves formulating a use case model and then extracting operational-level requirements statements from the use case descriptions. This process has characteristics of the product design process discussed earlier: It is a top-down, user-centered process, and it generates, refines, evaluates, and selects design alternatives. The coherent framework provided by interactions makes it much easier to design with use cases than to write requirements statements directly.

One final point: Use case models represent product functions, so the requirements statements extracted from them are functional. Functions usually rely on data, so studying use cases can suggest data requirements as well. Non-functional requirements may also become obvious from use case models.

Use-Case-Driven Development

An iterative development process will build system functionality gradually through several rounds of analysis, design, coding, testing, and evaluation. Use cases can help organize and direct iterative development. At each iteration, one or more use cases are selected for implementation until all the use cases are implemented and the system is complete. This approach is called *use-case-driven iterative development*.

Use-case driven development begins with the creation of a complete collection of use cases. The next step is to prioritize the use cases and estimate the cost and time needed to implement them. Generally the core parts of the product and those that provide the most important functionality to stakeholders should be given highest priority. It is also advisable to tackle the most difficult and challenging problems first. The following characteristics should increase a use case's priority:

- The use case embodies an interaction that realizes high-priority stakeholder needs.
- The use case includes core system functionality.
- Implementing the use case will require putting the main elements of the system architecture in place.
- Implementing the use case is expected to be technically challenging.

Use cases can then be scheduled into iterative development cycles based on their priority, cost, and time to implement. This decision usually involves trading off the level of desired functionality (represented by priorities) with cost and time.

Section Summary

- A **use case model** consists of a use case diagram and a use case description for each use case in the diagram.
- Use case diagrams are good tools for iteratively generating, evaluating, selecting, and refining alternative designs for the set of interactions supported by a product.
- Operational-level requirements statements constituting the bulk of the functional requirements portion of an SRS can be extracted from use case descriptions.
- Data and non-functional requirements may also become clear from studying use case models.
- Designing with use cases is usually a better way to create functional requirements than directly generating them by refining requirements from stakeholder needs.
- Use case models can be used for analyzing and designing the interactions between any entity and its environment.
- Use cases can also drive iterative development.

Review Quiz 7.3

1. Why are use case diagrams and user-level requirements at the same level of abstraction?
2. In what sense is designing with use cases a top-down process?

3. Why is it unnecessary to extract user-level requirements from use case diagrams?
-

Chapter 6 Further Reading

- Section 6.1** Use cases are discussed in most requirements development surveys, including [Lauesen 2002], [Robertson and Robertson 1999], and [Wiegers 2003]. Any UML discussion treats use case diagrams, often in depth; the following books are recommended: [Bennett et al. 2001], [Booch et al. 2005], [OMG 2003], and [Rumbaugh et al. 2004]. [Cockburn 2001] is an excellent in-depth treatment of use cases and is the basis for much of our discussion of use case descriptions, though Cockburn's terminology and method are slightly different from ours.
- Section 6.2** Cockburn [2001] goes into great detail about writing good use case descriptions; his process, format, and heuristics are the basis of the presentation in this section. Another discussion of use case descriptions can be found in [Wirfs-Brock and McKean 2003].
- Section 6.3** Use-case-driven development is discussed in [Jacobson et al. 1999] and [Cockburn 2001]. Use-case-driven development is closely related to extreme programming [Beck 2000] and other similar methods.

Chapter 6 Exercises

The following product descriptions are used in the exercises.

Computer Assignment System (CAS)

A group of system administrators must keep track of computers assigned to computer users in the community they support. A planned Computer Assignment System (CAS) will keep track of computers, computer users, and assignments. Developers in the same enterprise will implement CAS.

The major stakeholders are System Administrators, the Computer Users, the CAS Developers, Accountants, and the Managers of the development group and the system administration group.

CAS has the following business goals:

- Three people must develop CAS in three months or less.
- CAS must require no more than one person-week per year for maintenance.
- CAS must maintain the location, components, operational status, purchase date, and assignment of every computer in the organization.
- CAS must maintain the name, location, and title of every computer user in the organization.
- CAS users must take no more than an average of one minute per transaction to maintain this information.

Managers need CAS to meet its business requirements. Managers will not use CAS directly.

The Computer Users need CAS to maintain accurate data so that they will not be bothered with straightening out confusions.

The Developers need CAS to be easy to build and maintain. Developers need to build but not use CAS.

The Accountants need reports about computers, their costs, and their purchase dates so that they can compute capital expenditures, depreciation, and so forth.

The System Administrators need CAS to meet its business requirements. They also need reports listing all information CAS maintains about computers, computer users, and assignments, sorted by computer or by computer user. In addition, they need CAS to support queries about individual computers or individual computer users.

Fingerprint Access System (FAS)

A physical security equipment manufacturer will develop a system to control access to secure facilities using fingerprints. The product is intended for secure facilities such as military outposts, embassies, and research and development laboratories.

A computer connected to fingerprint readers will electronically control entry and exit gates. Personnel wishing to enter or leave the facility will place a finger on the fingerprint reader at a gate, and the gate will be unlocked to let them through.

The product under consideration is the software that controls this system; hence, the special hardware (fingerprint readers and gates) should be treated as outside the product.

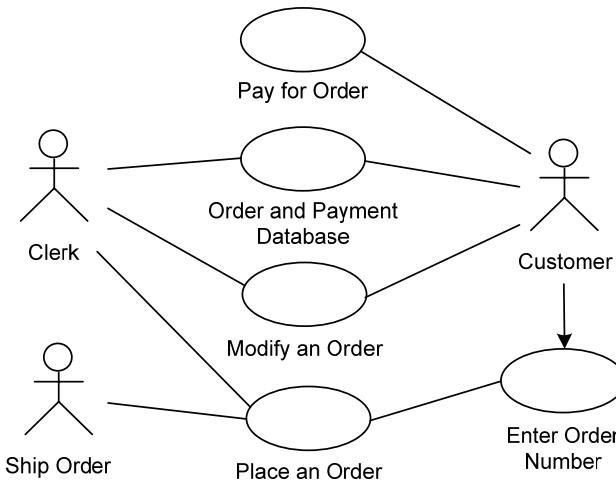
The stakeholders in this product include Security Managers, the Fire Marshall, Commuters (the people going through the gates), Developers, Sales and Marketing, Product Support, and company Management.

Stakeholders have the following needs:

- Commuters need to get through gates at least half as fast as with competitive systems.
- Security Managers need the fingerprint recognition error rate (false positives versus false negatives) to be adjustable.
- Security Managers need logs showing all entries and exits and all failed or aborted attempts at entry or exit.
- Security Managers need reports of all current facility occupants.
- Security Managers need to be able to ask FAS whether a particular individual is in the facility.
- The Fire Marshall needs Commuters to be able to leave the facility unimpeded during an emergency.
- The Fire Marshall needs emergency personnel to be able to enter a facility unimpeded during an emergency.

Section 6.1

1. Consider a software system that sells products over the Web. Classify the following activities as probably too big (B), too small (S), or the right size (R) to be use cases in this system:
 - (a) Enter Credit Card Number
 - (b) Set Printing Parameters
 - (c) Buy an Item
 - (d) Manage Web Site
 - (e) Select Mailing Address
 - (f) Modify Shopping Cart
 - (g) Search for an Item
2. *Find the errors:* Name five things wrong with the use case diagram in Figure 6-E-1.

**Figure 6-E-1 An Erroneous Use Case Diagram for Exercise 2**

3. List two heuristics in addition to those listed in the text that you think would make use case diagrams easier to read or write.
4. List two heuristics in addition to those listed in the text that you think would make use case descriptions easier to read or write.
5. Write a checklist to guide review of use case diagrams.
- AquaLush 6. Create a use case diagram for AquaLush that includes use cases for timer-controlled as well as moisture-controlled irrigation.
- CAS Use the description of the Computer Assignment System to do the next four exercises.
 7. Write three scenarios describing interactions between individuals and the Computer Assignment System.
 8. List and write brief descriptions of the actors involved in the Computer Assignment System.
 9. Draw a use case diagram for the Computer Assignment System.

10. Write briefs for the use cases in the use case diagram you made in the last exercise.
- FAS** Use the description of the Fingerprint Access System to do the next four exercises. Assume whatever hardware devices and characteristics you like and explain them briefly as a context for your answers.
 11. Write three scenarios describing interactions between individuals and the Fingerprint Access System.
 12. List and write brief descriptions of the actors involved in the Fingerprint Access System.
 13. Draw a use case diagram for the Fingerprint Access System.
 14. Write briefs for the use cases in the use case diagram you made in the last exercise.
- Section 6.2**
 15. Must every use case have preconditions? Why or why not?
 16. Must every use case have postconditions? Why or why not?
 17. Write a checklist to guide review of use case descriptions.
 18. Write use case descriptions for two use cases supported by the Computer Assignment System.
 19. Write use case descriptions for two use cases supported by the Fingerprint Access System. Assume whatever hardware devices and characteristics you like and explain them briefly as a context for your descriptions.
 20. How might use case diagrams and use case descriptions be employed during product design analysis?
- Section 6.3**
 21. Draw an activity diagram illustrating the process of designing with use case models.
 22. Draw three use case diagrams that present alternative user-level designs for the Computer Assignment System.
 23. Write three descriptions for a use case in one of the diagrams you made for the last exercise that present alternative operational-level designs for the use case.
 24. Draw three use case diagrams that present alternative user-level designs for the Fingerprint Access System. Assume whatever hardware devices and characteristics you like and explain them briefly as a context for your diagram.
 25. Write three descriptions for a use case in one of the diagrams you made for the last exercise that present alternative operational-level designs for the use case.
- Team Projects**
 26. Make a complete use case model for the Computer Assignment System. Use this model to write functional requirements for this system.
 27. Make a complete use case model for the Fingerprint Access System. Assume whatever hardware devices and characteristics you like and

explain them briefly as a context for your model. Use this model to write functional requirements for this system.

- Research Project**
28. Consult other books that discuss use cases and create templates for alternative use case description formats. When might these various formats be useful?

Chapter 6 Review Quiz Answers

Review Quiz 6.1

1. The product can never be an actor in a use case diagram because by definition actors are agents external to the product that interact with it.
2. A use case can be named by a noun phrase when the noun phrase is used for a well-known or well-established activity. For example, a use case to check an account balance might be called “Balance Inquiry” rather than “Check Account Balance.”
3. Designers can check a use case diagram for completeness by reviewing the product needs list to ensure that all needs are satisfied or that any unsatisfied needs have been left out or overlooked.
4. Some examples of activities that are too small to be use cases are: sending a device a signal or another system a message, receiving a signal or message from a device or another system, and displaying a message to a user.

Review Quiz 6.2

1. Any notation that can show the activity flow in an interaction can be used for use case descriptions, including UML activity diagrams, UML interaction diagrams, flow charts, programming languages, and natural language.
2. A precondition is a statement that must be true before some activity or operation occurs if the activity or operation is to complete successfully. A postcondition is a statement guaranteed to be true after some activity or operation completes successfully.
3. If no product stakeholder has a need relevant to a use case, there is no reason for that use case to be supported by the product. A product is supposed to meet stakeholder needs and desires, and all its features and functions should contribute to this goal.
4. Guidance for writing good use case descriptions is supplied by the following heuristics: write simple declarative sentences in the active voice, make actors or the product the subjects of most sentences, avoid specifying user interface details, avoid over-specifying the order of activities in the use case, and proofread the description.

Review Quiz 6.3

1. Use case diagrams catalog the interactions supported by a product, all of which presumably help stakeholders achieve goals. User-level requirements specify functions, features, or capabilities that a product must have to help stakeholders achieve goals. Thus, both use case diagrams and user-level requirements specify product aspects for helping stakeholders achieve goals, and both are at the same level of abstraction.
2. Designing with use cases is a top-down process because it works from higher to lower levels of abstraction. It begins by constructing a use case diagram,

which abstracts interaction details, and continues with writing descriptions that refine the use cases listed in the use case diagram.

3. A use case is constructed from the stakeholders-goals list and the needs list, and operational requirements are extracted from the use case model and become the contents of the SRS functional requirements section. As a result, there is no need to generate user-level requirements.

Part III Software Engineering Design

The third part of this book covers core material in software engineering design.

Chapter 7 discusses how to understand the engineering design problem, emphasizing the use of conceptual models made with the UML class diagram notation.

Chapter 8 begins an extended discussion of engineering design resolution by distinguishing architectural and detailed design. It also introduces fundamental engineering design principles.

Chapter 9 is the first of two chapters examining architectural design. It focuses on architectural modeling notations.

Chapter 10 discusses the architectural design resolution process.

Chapter 11 is the first of three chapters about mid-level design. It focuses on making static class models using UML class diagrams.

Chapter 12 introduces UML sequence diagrams for dynamic mid-level interaction modeling.

Chapter 13 introduces UML state diagrams for dynamic mid-level state modeling, completing the discussion of mid-level design.

Chapter 14 completes our survey of engineering design by discussing low-level design processes, notations, and heuristics. It also touches on low-level design evolution and the transition to the implementation phase.



7 Engineering Design Analysis

Chapter Objectives This chapter begins our discussion of engineering design with its first step: engineering design analysis, as shown in the diagram in Figure 7-O-1.

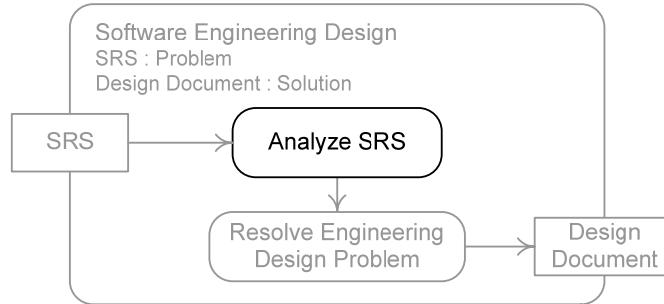


Figure 7-O-1 Software Engineering Design

The main analysis model that we consider is the analysis class model, which motivates introduction of UML class and object diagrams.

By the end of this chapter you will be able to

- State the context of engineering design analysis and explain the role of modeling in engineering design analysis;
- Say what class and object models and diagrams are;
- Explain the difference between conceptual models, design class models, and implementation class models;
- State some uses for conceptual models;
- Read and write basic UML class diagrams that include class symbols with names, attribute and operation specifications, and associations with multiplicities;
- Read and write basic UML object diagrams that include object symbols with names, attributes with values, and links;
- State rules and heuristics for making correct and readable UML class and object diagrams; and
- List the steps in a conceptual modeling process and make conceptual models as UML class diagrams.

Chapter Contents
7.1 Introduction to Engineering Design Analysis
7.2 UML Class and Object Diagrams
7.3 Making Conceptual Models

7.1 Introduction to Engineering Design Analysis

Context of Engineering Design Analysis

In Chapter 1 we distinguished software product design (the activity of specifying software features, capabilities, and interfaces to satisfy client needs and desires) from software engineering design (the activity of specifying software systems, sub-systems, and their constituent parts and workings to meet the specifications for a software product). This chapter marks the beginning of our discussion of the latter topic. In terms of the software life cycle, this chapter is the start of our discussion of the activities of the design phase.

In Chapter 2 we introduced a generic software engineering design process, reproduced as Figure 7-1-1 on page 195.

This activity diagram shows engineering design analysis as the first step of engineering design, followed by the generation, improvement, evaluation, and selection activities of architectural and detailed design. Engineering design analysis is the first step of a much larger activity.

The diagram also shows that engineering design requires an SRS as its input. Generally, a software product design is specified in an SRS document, which may be supplemented with various models and prototypes. The discussion in this chapter and the next assumes that, at a minimum, an SRS has been supplied as analysis input. It also discusses how to use other product design artifacts, such as use case models.

Analysis is the activity of breaking down a design problem to understand it, as discussed in Chapter 2. The goal of engineering design analysis is to understand an engineering design problem. This problem is posed by a software product design and engineering design constraints specified in the SRS and product design models. Unfortunately, engineering designers may not have a good SRS or product design models to analyze. If not, they must do whatever part of product design remains undone, using the approaches and techniques discussed in the previous chapters.

Although engineering designers may be able to understand the problem they need to solve just by reading the SRS and auxiliary product design models, usually it helps to engage the material actively. Modeling is one of the best ways to do this. Modeling during analysis also serves as a good test of understanding and provides further documentation for input to design resolution. For these reasons, engineering design analysis activities consist mainly of studying the SRS and product design models and producing new models of the problem. In the course of this activity, inconsistencies and incompleteness in the SRS often come to light. If so, engineering designers must ask product designers for clarification or elaboration. This may lead product designers to redo part of the design, which may in turn lead to discussion and consultation with stakeholders. Analyzing the SRS and product design models can improve both their quality and the quality of the product design itself, besides laying the foundation for engineering design resolution.

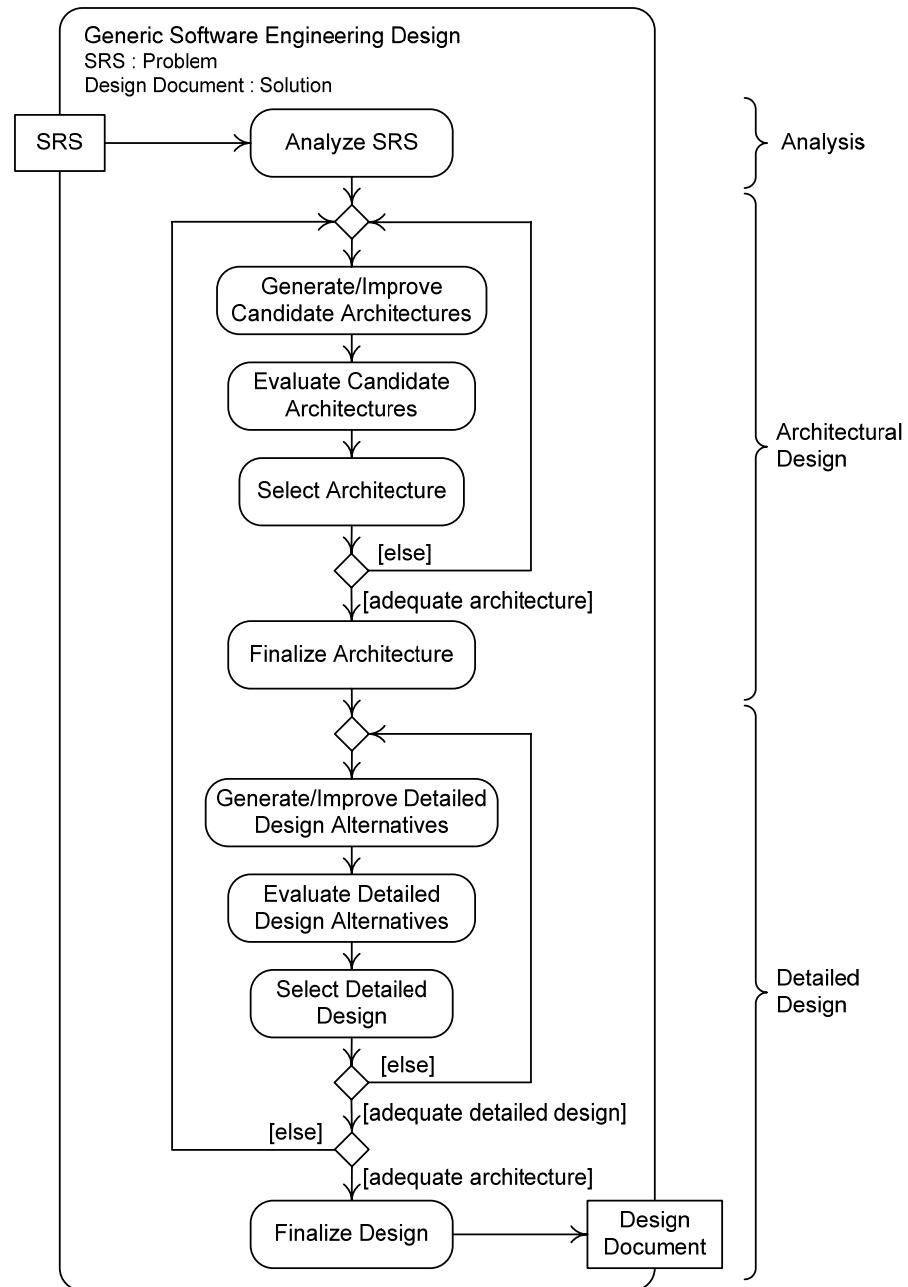


Figure 7-1-1 Generic Software Engineering Design Process

The outputs of engineering design analysis embody the understanding gleaned from studying and elaborating the SRS and product design models. Usually these outputs are models.

Analysis Models

An **analysis model** is any representation of a design problem. Thoroughly understanding a design problem usually requires understanding both its static and dynamic aspects, so both kinds of models are important in analysis. There are many static and dynamic models useful for engineering design analysis, but our discussion in this chapter focuses on object-oriented modeling. The section on Further Reading at the end of the chapter provides references for non-object-oriented modeling.

The most useful dynamic model for engineering design analysis is the use case model, discussed in Chapter 6. If a use case model has not been constructed during product design, making one during engineering design analysis is often helpful.

The most useful object-oriented static model for engineering design analysis is the *analysis class model*, often called the *conceptual model*. Conceptual models may also be used during product design. Most of this chapter covers conceptual modeling and basic UML class diagrams as a conceptual modeling notation.

Class and Object Models and Diagrams

The central artifacts in object-oriented analysis and design are representations of the classes and objects making up a software system. Either classes or objects may be represented in a model, yielding two sorts of models.

A **class model** is a representation of classes in a problem or a software solution.

An **object model** is a representation of objects in a problem or a software solution.

People tend to refer to class and object models interchangeably, mainly because classes and objects are closely related. Nevertheless, object models and class models are very different, and we distinguish carefully between them in this book.

Class and object models are usually diagrams. **Class diagrams** are graphical forms of class models. Class diagrams typically include class names, and they may also show a lot of additional information, including attributes, operations, responsibilities, associations, visibilities, attribute data types, and operation signatures.

Object diagrams are graphical forms of object models. Object diagrams typically include each object's name and class. They may also show attributes, their values, and various relational links between objects. In practice, object diagrams are not as important as class diagrams and are used relatively infrequently.

To illustrate, consider the class diagram of musical recordings in Figure 7-1-2.

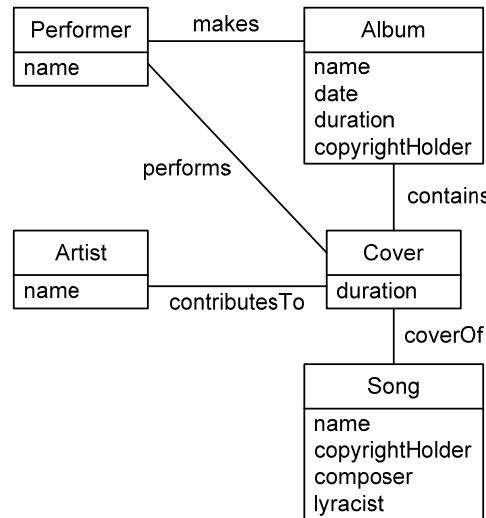


Figure 7-1-2 Musical Performance Class Diagram

This diagram represents classes as boxes with two compartments. The top compartment shows the name of the class, and the bottom compartment shows the attributes of the class. For example, an **Album** has a **name**, a **date** when it was made, a **duration**, and a **copyrightHolder**. Lines between classes represent relations between their instances. In this diagram, **Performers** make **Albums** that contain **Covers** of **Songs**. **Artists** may contribute to **Covers** as well. For example, Eric Clapton (a **Performer**) made *Unplugged* (an **Album**), which contains a **Cover** of the Song “San Francisco Bay Blues,” written by Jesse Fuller. Among the **Artists** that contributed to this **Cover** was Ray Cooper on percussion.

There are many class and object modeling notations; the most widely used are CRC cards and UML class and object diagrams. *CRC cards* are note cards recording information about classes (C), their responsibilities (R), and the other classes with which they collaborate (C) to achieve computational goals (hence CRC cards). CRC cards are a form of class model mainly used for analysis. They have the virtues of notational simplicity, changeability, and ease of use, but they are not as expressive as class diagrams.

UML class and object diagrams are now the *de facto* standard notation for class and object modeling. The next section of this chapter introduces UML class and object diagrams; they are discussed further in later chapters.

Types of Class Models

At least three kinds of class models are used in object-oriented analysis and design:

Analysis Class Models or Conceptual Models—**Conceptual models** represent the important entities or concepts in the problem, their attributes, and the most important associations between them. These models are usually expressed using class diagrams or CRC cards.

Design Class Models—**Design class models** represent the classes in a software system and their attributes, operations, and associations in abstraction from language and implementation details. Implementation details left out of design class models typically include data types, visibility, accessibility, and details having to do with particular environments, such as specific classes from class libraries. Class diagrams are used for design class modeling.

Implementation Class Models—**Implementation class models** represent the classes in a software system and include some or all of the implementation details left out of design class models. These models guide programmers writing code. Class diagrams are used for implementation class modeling.

The goal of conceptual modeling is to understand and document the structure of the *problem*, while the goal of design and implementation class modeling is to specify the structure of the software *solution*. The distinction between design and implementation class models is fuzzy because it is a matter of degree of abstraction. The difference between conceptual models and design and implementation class models is sharp because it rests on the clear distinction between a problem and its solution. Thus, we emphasize the following remark.

Conceptual models are about real-world entities in the problem domain and not about software.

It is essential that designers focus on the problem when creating conceptual models: There should be nothing in a conceptual model having to do with a computerized software solution to the problem. Conceptual models are the basis for later design solutions. A conceptual model subverted by solution considerations can degrade the final design.

Using Conceptual Models

Use case descriptions capture the interactions between a product and its environment. Use case models represent product behavior; they do not show the static structure of the entities in the problem domain. Conceptual models complement use case models by representing the static structure of the problem. Conceptual models are analysis tools that can be used for both product and engineering design analysis. As we will see, a conceptual model can also be used as the pivotal artifact in transforming an engineering analysis model into an engineering resolution model.

Conceptual models can be used during product design for the following tasks:

Understanding the Problem Domain—Fundamental to understanding any problem domain is knowing the important entities in the domain, their major characteristics, and their relations to one another. Product designers can work with problem-domain experts to construct a conceptual model as a way for the designers to learn about the domain.

Setting Data Requirements—Conceptual models can help determine the entities, their characteristics, and the relations between them that a product must manipulate or track. Data requirements statements at all levels of abstraction can be extracted from the model.

Validating Requirements—An SRS can be reviewed to ensure that data and functional requirements are consistent with a conceptual model.

Conceptual models can be used during engineering design for the following tasks:

Understanding a Product Design—A software product interacts with entities and stores and manipulates data about those entities. Conceptual models represent these entities and their responsibilities, attributes, and relations. This helps engineering designers understand the problem they must solve.

Under-Girding Engineering Design Modeling—Conceptual models can often be transformed into design models, thus transforming a description of the problem into a specification for its solution. This technique is one of the most powerful in object-oriented design, but it only works if one begins with a good conceptual model.

Section Summary

- Engineering design analysis is the first step of engineering design and is followed by the engineering design resolution activities of architectural and detailed design.
- The inputs to engineering design resolution are the product design and engineering design constraints, which are documented in the SRS and any models constructed during product design.
- The main activities of engineering design analysis are studying the SRS and any auxiliary models and constructing engineering design models.
- **Analysis models** represent a design problem. Use case models are valuable dynamic analysis models; conceptual models are valuable static analysis models.
- A **class (object) model** represents the classes (objects) in a problem or a software solution; **class (object) diagrams** are a graphical kind of class (object) models.
- **Conceptual models** show the important entities in a problem, their attributes, and the most important relations between them.
- **Design class models** show the classes in a software system and their attributes, operations, and associations in abstraction from language and implementation details.
- **Implementation class models** show details left out of design class models.
- Conceptual models are useful for understanding problem domains, data requirements, and product design. They are also helpful for validating requirements and as the basis for engineering design.

**Review
Quiz 7.1**

1. What must engineering designers do if they do not have a complete, correct, and consistent product design when beginning their work?
 2. What are the most useful static and dynamic models for engineering design analysis?
 3. Give an example of something that should not appear in a conceptual model but might appear in a design class model.
 4. Give an example of something that should not appear in a design class model but might appear in an implementation class model.
-

7.2 UML Class and Object Diagrams

**Class and
Object
Modeling
Notations**

UML class diagrams are preferred for making class models, including conceptual models. We introduce only the portions of the UML class diagram notation most useful for conceptual modeling in this section; more advanced features are introduced in Chapter 11.

Object models are much less useful than class models and are relatively rarely used. However, UML object diagrams are a simple variation of UML class diagrams, so it is easy to introduce the former with the latter.

Names

A **name** in UML is a character string that identifies a model element. A name can be either simple or composite. A **simple name** is a sequence of letters, digits, or punctuation characters such as the underscore, for example, `util`, `Vector`, `last_chance`, or `java`. Simple names usually start with a letter. A **composite name** is a sequence of one or more simple names separated by a delimiter, conventionally the double colon in UML, for example, `java::util::Date`. Usually the last element in a composite name designates an entity, such as a class or use case. The remaining elements—those separated from the last element by the delimiter—designate a container of some sort, such as a package or a directory. For example, the pathname `java::util::Date` designates a class; `Date` is the class name and `java::util` is a package containing the `Date` class, so `java::util::Date` is a particular class in a certain package.

UML has a very permissive name policy. UML simple names may contain any character, including blanks, colons, and commas. Many punctuation characters have special meanings in UML, so allowing them in simple names is a potent source of confusion. We therefore adopt a much more restrictive policy regarding simple names: In this book all simple names are strings of letters, and underscores.

Classes A **class** is an abstraction of a set of objects with common operations and attributes. In UML, classes are represented by rectangles divided vertically into three or more *compartments* numbered starting from one at the top. The first compartment is the *class name compartment*, the second is the *attributes compartment*, and the third is the *operations compartment*. Other compartments do not have names. Compartments may be *suppressed*, meaning that they may be omitted from the class symbol. This does not mean that these compartments are empty, but rather that the modeler decided to abstract the compartment on the diagram.

The first compartment in a class symbol must contain the name of the class represented by the symbol. The second compartment must contain a list of the class's attributes and the third a list of the class's operations. Additional compartments may be added as needed to supply additional information. For example, a fourth compartment might list class responsibilities. Additional compartments may contain arbitrary text. Compartment names may be added at the top of a compartment if desired; this is a good practice when using the fourth and later compartments. Figure 7-2-1 shows some examples of class symbols.

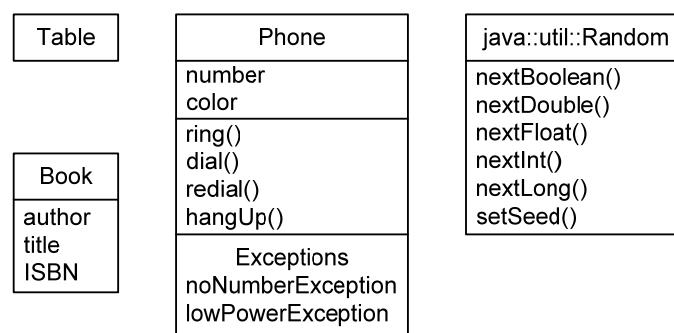


Figure 7-2-1 Class Symbol Examples

In Figure 7-2-1, the attributes and operations compartments are both suppressed in the **Table** class symbol, the operations compartment is suppressed in the **Book** class symbol, and the attributes compartment is suppressed in the **java::util::Random** class symbol.

Attributes An **attribute** is a data item held by an object or a class. Attribute types and initial values may be specified along with their names, which is often useful in conceptual modeling. The form of this specification is described in Figure 7-2-2.

Class Symbol Attribute Specification Format

name : type [multiplicity] = initial-value

Where:

name—The name of the attribute; it must be a simple name and cannot be suppressed.

type—The name of some data type (such as `integer`) or class (such as `Shape`); its format is not specified by UML and it may be suppressed. If the *type* is suppressed, then the colon is omitted.

multiplicity—The number of values stored in the attribute. The *multiplicity* is specified as a comma-separated list of ranges of non-negative numbers. Each range is specified in the form $n..k$, where $n \leq k$ (such as $1..1$ or $0..5$). A range with the same lower and upper bounds can be abbreviated to a single number (so $1..1$ can be abbreviated as simply 1 , for example). The asterisk (*) stands for an unlimited upper bound, so the range $1..*$ means “one or more.” Finally, the range $0..*$ (zero or more) is abbreviated *. The *multiplicity* may be suppressed, in which case it is 1 by default. If it is suppressed, the square brackets are omitted.

initial-value—The value assigned to the attribute when the class is created. This is a literal whose format is not specified by UML. An *initial-value* is optional. If it is not specified, the equal sign is omitted.

Figure 7-2-2 Class Symbol Attribute Specification Format

This notation provides a lot of flexibility in specifying attributes and their characteristics. The following are examples of attribute specifications with explanations:

`weight : float` specifies that the `weight` attribute is of type `float`.

`toBuy : string[*] = ()` specifies a collection of `strings` of arbitrary size (a set or list of `strings`, in other words) initially set to contain no values.

`ToDo : string[0..10]` specifies a collection of 0 to 10 `strings` (a set or list of no more than 10 `strings`, in other words).

`size : integer = 128` specifies an `integer` attribute holding a single value set to 128 when an instance is created.

Besides being able to suppress the attributes compartment and certain aspects of attribute specification, entire attribute specifications may be suppressed; in other words, not all attributes need to be listed.

- | | |
|------------|---|
| Operations | An operation is an object or class behavior. Operation signatures can be specified in class symbols. The form of this specification is detailed in Figure 7-2-3. |
|------------|---|

Class Symbol Operation Specification Format

$$\textit{name}(\textit{parameter-list}) : \textit{return-type-list}$$

Where:

name—The name of the attribute; it must be a simple name and cannot be suppressed.

parameter-list—The names and types of the parameters of this operation, specified in the form

$$\textit{direction param-name : param-type} = \textit{default-value}$$

In this specification, *direction* is one of the values *in*, *out*, *inout*, or *return* and indicates that the parameter is used to send data into, out of, or both into and out of the operation, or to return data from the operation. The *direction* can be suppressed and is *in* by default. The *param-name* is the parameter's name; it must be a simple name. The *param-type* is the parameter's data type or class; the format of *param-type* is not specified by UML. Neither *param-name* nor *param-type* may be suppressed. The *default-value* is the initial value of the formal parameter if no argument value is supplied (as in C++, for example). It is a literal whose format is not specified by UML. A *default-value* is optional; if there is none, the equals sign is omitted.

return-type-list—A comma-separated list of the types of the values returned by the operation, in a format not specified by UML. If the operation has no return value or returns void, the colon and the list are omitted.

The *parameter-list* and the *return-type-list* may be suppressed together.

Figure 7-2-3 Class Symbol Operation Specification Format

In some cases, there is no way to tell whether an optional item has been suppressed or simply does not exist. For example, an operation whose *parameter-list* and *return-type-list* are suppressed is indistinguishable from one with no parameters and no return value.

Some examples of operation specifications appear in the following list:

`getWeight() : float` specifies that the `getWeight` operation has no parameters and returns a `float` value.

`calibrateLightMeter(in lumens : double)` specifies that the `calibrateLightMeter()` operation takes a single `double` parameter called `lumens` that sends data only into the operation, and that it has no return value.

`getFontData() : integer, integer, string` specifies that the `getFontData()` operation has no parameters and returns two `integer` values and one `string` value.

Figure 7-2-4 shows example class symbols specifying various attributes and operations.

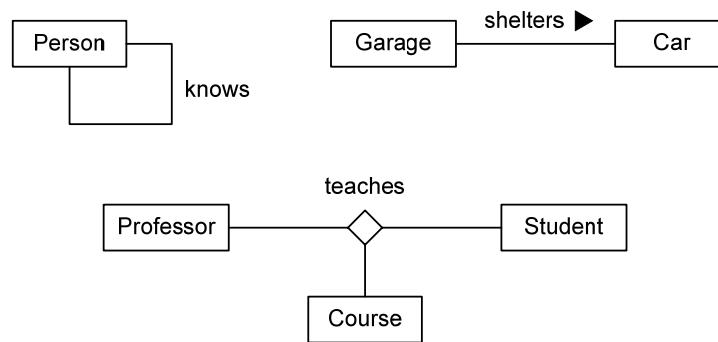
Player	WaterHeaterController
roundScore : int = 0 totalScore : int = 0 words : String[*] = () resetScores() setRoundScore(in size : int) findWords(in board : Board) getRoundScore() : int getTotalScore() : int getWords() : String[*]	mode : HeaterMode = OFF occupiedTemp : int = 70 emptyTemp : int = 55 setMode(newMode : Mode = OFF) setOccupiedTemp(newTemp : int) setEmptyTemp(newTemp : int) clockTick(out ack : Boolean)

Figure 7-2-4 Attribute and Operation Specifications

UML class symbol attributes and operation lists can specify even more than we have discussed, but these additional capabilities are not particularly relevant for conceptual modeling. We discuss the full expressive power of UML attribute and operation specifications in Chapter 11.

Associations In UML class diagrams, lines represent associations between classes. An **association** between classes exists when there is a relation between instances of the associated classes. Association lines may start and end at the same class to show a relation that the instances of the class bear to one another. Associations may also correspond to relations between instances of three or more classes. In this case, a small diamond is used as the nexus from which association lines radiate to the classes involved.

Association lines may be unlabeled, or they may show the association name and, optionally, the direction in which the name is read. The name is placed near the center of the line. A small name-direction arrow can indicate how to read the association. Figure 7-2-5 illustrates this notation.

**Figure 7-2-5 Various Association Notations**

Note that any association can be read in either direction. For example, the associations in Figure 7-2-5 can be read in the following ways:

A Person *knows* another Person or a Person is *known by* another Person.

A Garage *shelters* a Car or a Car is *sheltered by* a Garage.

A Professor *teaches* Students in a Course, Students in a Course are *taught by* a Professor, or a Course with Students is *taught by* a Professor.

The model is the same no matter how its associations are read. However, it is usually easier to understand the model if associations are named with verbs in the active voice; for example, *teaches* rather than *is taught by*.

Each end of an association line may be adorned with a **rolename** describing the part played by instances of the class at that end of the association. Figure 7-2-6 illustrates rolenames.

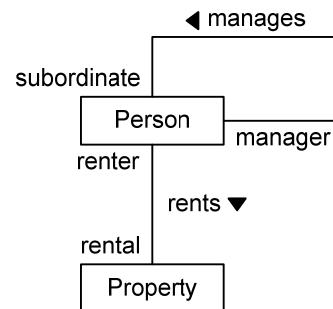


Figure 7-2-6 Association Roles and Rolenames

Rolenames can appear at neither, one, or both ends of an association line.

Association Multiplicity	<p>Each end of an association line may also have a multiplicity indicating how many instances of the class at that end may be associated with a single instance of the class at the other end of the association. The multiplicity notation for associations is the same as the notation for attributes discussed previously—a comma-separated list of ranges of non-negative numbers. Multiplicities express constraints on the number of instances that may participate in the relation represented by the association line.</p> <p>Multiplicities are read and written as follows: Choose a class at one end of the association as the <i>source class</i>; the class at the other end is the <i>target class</i>.</p>
--------------------------	--

The multiplicity at the target class end of an association is the number of instances of the target class that can be associated with a single instance of the source class.

The multiplicity at the other end of the association line is determined by reversing the source and target classes.

To illustrate this rule, consider the relation between students and dorm rooms at a university. A student may live in a dorm room or live off

campus, and a dorm room may be empty or, let's suppose, have up to three occupants. First let's take the **Student** class as the source and the **DormRoom** class as the target. A single **Student** occupies zero or one **DormRooms**, so the multiplicity on the **DormRoom** end of the **occupies** association is 0..1. Second, switching the source and target classes, a single **DormRoom** is occupied by zero to three **Students**, so the multiplicity on the **Student** end of the association is 0..3.

Another way to think about assigning multiplicities is that they are made by reading an association in various ways. For example, when read from **Student** to **DormRoom**, the multiplicity is 0..1 because a **Student** *occupies* zero or one **DormRooms**. When read from **DormRoom** to **Student**, the association multiplicity is 0..3 because a **DormRoom** is *occupied by* zero to three **Students**.

Figure 7-2-7 illustrates this result along with a few other associations and multiplicities.

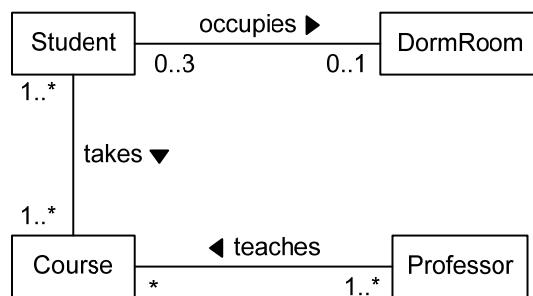


Figure 7-2-7 Association Multiplicities

Multiplicities are read in the same way they are written. For example, Figure 7-2-7 shows that a single **Professor** teaches zero or more **Courses** and a single **Course** is taught by one or more **Professors**.

Multiplicities may be suppressed at either end of an association. There is no default association multiplicity.

Additional information may be added to association ends; we will discuss some of these additional items in later sections.

Class Diagram Formation Rules

Class diagrams must conform to the following rules:

Class symbols must have at least a name compartment. All compartments except the name compartment may be suppressed. A diagram that focuses on class associations and their multiplicities, for example, may suppress all but the class name compartments.

Compartments must be in order. The name compartment must be first, followed by the attributes compartment, the operations compartment, and other compartments, in that order. Although any but the first compartment may be suppressed, when they appear, they must be in the right order.

Attributes and operations must be listed one per line. The attributes and operations lists are vertical lists with nothing to separate their members except appearance on a new line.

Attribute and operation specifications must be syntactically correct. The syntax previously discussed for attribute and operation specifications must be followed. It is easy to confuse UML syntax with programming language syntax. For example, in UML an int class attribute named count would be specified `count : int`, while in Java or C++ it would be `int count`.

Class Diagram Heuristics

The following heuristics help to make better class diagrams and more readable class models:

Name classes, attributes, and roles with noun phrases. Classes, attributes, and roles are things, so they should be named with noun phrases. For example, consider creating a model for a physical campus, which calls for a buildings class. The obvious name for such a class is `Building`, a noun. `Building` attributes that might be important include `name`, `abbreviation`, `address`, `type`, `location`, and `photograph`. All of these are nouns. A `Person` can occupy a `Building`. Rolenames for the participants in this relation might be `occupant` and `occupiedBuilding`, both noun phrases.

Name operations and associations with verb phrases. Operations are behaviors of class instances; phrases that refer to behaviors and other actions and activities are verb phrases. Similarly, associations represent relations between instances of classes. A relation is expressed by a verb phrase. Continuing with the campus building example from the previous paragraph, operations in the `Building` class will probably include some for fetching attributes (such as `getName()`), setting attributes (such as `setType()`), and computing something (such as `countOccupants()`). These names are verb phrases. Associations in the model might include `occupies` (a `Person` occupies a `Building`), `isAdjacentTo` (one `Building` isAdjacentTo another), and `eatsIn` (a `Person` eatsIn a `Building`). These association names are also verb phrases.

Often, verb phrases involving “is” are abbreviated in association names. For example, `isAdjacentTo` may be abbreviated to simply `adjacentTo`.

Capitalize class names but not attribute, operation, association, and rolenames. The convention in most object-oriented design and programming is to capitalize class names but not attribute and operation names. As we will see, roles are implemented as attributes, so rolenames are treated like attribute names.

Center class and compartment names but left-justify other compartment contents. This is merely a convention that makes the contents of class symbols easier to read.

Stick to binary associations. Associations involving more than two classes are much harder to understand than binary associations; think about assigning multiplicities, for example. Fortunately, most relations captured in class diagrams are binary, and all non-binary relations can be

reduced to several binary relations if necessary, so there is no need to use associations involving more than two classes.

Prefer association names to rolenames. Many interesting relations have English names, but relatively few roles do. For example, the relation between a building and an organization that is based in it is expressed by the term “houses,” as in “Porter Hall houses the Physics Department.” However, there is no good term for the roles of either the building or the housed organization. Rolenames are also often not very useful; saying that a building houses a department is enough to understand the roles played by the participants in the relation. The only time that rolenames are really necessary is when an association holds between instances of the same class. For example, the `parentOf` association from `Person` to itself has different multiplicities at its two ends. It is important to know which end is the parent end and which is the child end to understand the multiplicity constraints.

Place association names, rolenames, and multiplicities on opposite sides of the line. This heuristic is merely to improve readability. Of course, it is impossible to place all three labels (association names, rolenames, and multiplicities) on opposite sides of a single line when all three are present. In this case, the rolenames and multiplicities go on opposite sides and the association name goes wherever it fits best.

Class Diagram Uses

Class diagrams are the central static modeling tool in object-oriented software design. They are useful during product design for making conceptual models that represent the important entities in the problem, their characteristics, and their relationships (conceptual modeling is discussed in the next section). Conceptual models can help designers understand and document the problem domain. During product design analysis, they help elicit and analyze stakeholder needs and desires, especially about data requirements. They can also help designers generate, improve, evaluate, select, and document data requirements during product design resolution.

Conceptual modeling is the main tool for understanding the static structure of the engineering design problem during engineering design analysis. Class models are then used extensively in engineering design resolution in generating, improving, evaluating, and selecting the classes and class structures in the software solution. We discuss class modeling for these purposes throughout the remainder of the book.

UML Object Diagrams

UML object diagrams resemble UML class diagrams, which makes sense because objects are instances of classes. An object symbol is a rectangle divided vertically into two compartments: the first (top) compartment is the *object name compartment* and the second (bottom) is the *attributes compartment*. Unlike class symbols, object symbols can have only two compartments. The second compartment may be suppressed.

The name compartment must contain the name of the object in the format shown in Figure 7-2-8.

Object Symbol Name Compartment Format

$$\underline{\text{object-name}} : \underline{\text{class-name}}$$

[stateList]

Where:

object-name—The name of the object; it must be a simple name. The *object-name* may be suppressed.

class-name—The name of the class of which the object is an instance. This is a UML name (a simple name or pathname). The *class-name* may be suppressed. If so, the colon does not appear.

stateList—A comma-separated list of the names of the current states of the object. The states must all be able to occur together. The *stateList*, along with the square brackets, is optional.

The *object-name* and *class-name* may not be suppressed simultaneously.

Figure 7-2-8 Object Symbol Name Compartment Format

The diagram in Figure 7-2-9 illustrates these possibilities.

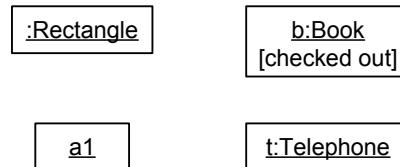


Figure 7-2-9 Object Symbols

The attributes compartment contains a list of instance attributes and their values in the format specified in Figure 7-2-10.

Object Symbol Attribute Specification Format

$$\text{attribute-name} = \text{value}$$

Where:

attribute-name—The name of an attribute of the object.

value—A literal (whose format is not specified by UML) indicating the value of the attribute at some instant when the model represents the object. The value must be of the type indicated for the attribute in the class.

Particular attributes and values may be suppressed simultaneously.

Figure 7-2-10 Object Symbol Attribute Specification Format

Figure 7-2-11 shows some examples.

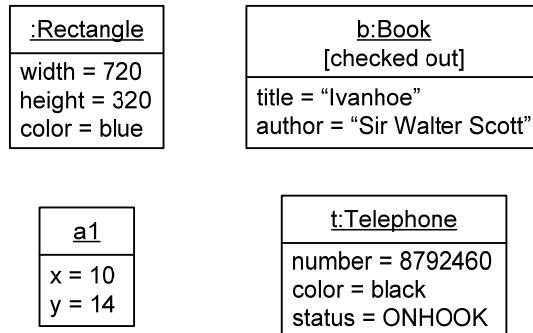


Figure 7-2-11 Object Symbols with Attribute Values

An object symbol represents an instance of a class at a particular moment. The corresponding class symbol shows the common features of all instances of the class, so all the object symbol must show are the things that vary between instances, namely, their names and the values of their attributes. Consequently, this is all that can be represented in an object symbol.

Object Links

Recall that an association holds between classes when there is a relation between the instances of the associated classes. An object diagram can show that particular objects are elements of a relation by connecting them with a **link**. A link is said to be an *instance* of an association. Links are to associations what objects are to classes. Links are shown using a solid *link line*. The name of the instantiated association can be placed underlined near the center of the link line. Association rolenames can be placed at the ends of the link line. Note that links never have multiplicity because they always show that a particular object is related to another particular object. Figure 7-2-12 illustrates a few links between objects.

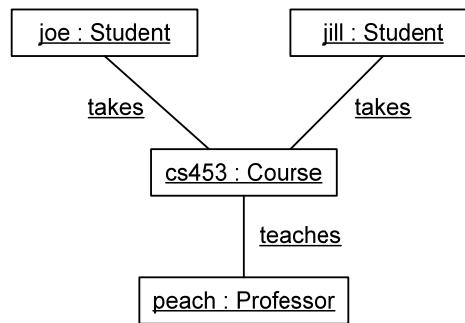


Figure 7-2-12 Some Linked Objects

Object Diagram Formation Rules and Heuristics

Object diagrams derive from class diagrams so there are few additional formation rules for them beyond those mentioned previously, such as underlining the object name and never placing multiplicities on links. The following rules are the only heuristics for object diagrams:

Name object instances with noun phrases.

Don't capitalize object names.

Object Diagram Uses

Object diagrams show the state of one or more objects at a moment during program execution, so they are useful for demonstrating what happens to objects as a program runs. This can be helpful during engineering design when trying to understand how objects change during execution. Object diagrams are thus useful for evaluating design alternatives during engineering design resolution.

Heuristics Summary

Figure 7-2-13 summarizes class and object diagram heuristics.

- Name classes, objects, attributes, and roles with noun phrases.
- Name operations and associations with verb phrases.
- Capitalize class names but not object, attribute, operation, association, and rolenames.
- Center class, object, and compartment names but left-justify other compartment contents.
- Stick to binary associations and links.
- Prefer association names to rolenames.
- Place association and link names, rolenames, and multiplicities on opposite sides of the association or link line.
- Never adorn link lines with multiplicities.

Figure 7-2-13 Class and Object Diagram Heuristics

Section Summary

- A **name** is a character string that identifies model elements, and it can be either a **simple name** or a **composite name**.
- A **class symbol** is rectangle divided vertically into three or more compartments: the class name, attributes, operations, and optional compartments.
- The class name compartment must be present and contain the name of the class.
- The suppressible attributes compartment contains a list of attribute names, types, multiplicities, and initial values.
- The suppressible operations compartment contains a list of operation names, parameter directions, parameter names, parameter types, parameter default values, and a list of return types.
- **Associations** represent relations between instances of classes. Association lines represent associations. They may be adorned with association names, name direction arrows, **rolenames**, and **multiplicities**.

- An *object symbol* is a rectangle divided vertically into a name compartment and a suppressible attributes compartment.
- The name compartment must contain an underlined object name; the attributes compartment must contain attribute names and current values.
- A **link** is an instance of an association. A link line may connect object symbols to show that the objects are elements of a relation. Link lines may have underlined association names and rolenames, but no multiplicities.
- Class diagram formation rules and heuristics apply to object diagrams as well.

**Review
Quiz 7.2**

1. Suppose a class symbol has two compartments displayed. List all possibilities for the contents of the first and second compartments.
 2. Give an example of a syntactically correct attribute specification that suppresses or omits as much as possible. Give an example of a syntactically correct attribute specification that suppresses or omits nothing.
 3. Give examples of associations that a **Person** class bears to itself with each of the following multiplicities: *, 1..*, 0..1.
 4. Name three heuristics for drawing good class diagrams.
 5. Suppose that an object diagram shows that objects **a1:A** and **a2:A** are both linked to object **b:B** in relation **R**. What can you deduce about the multiplicity of association **R**?
-

7.3 Making Conceptual Models

**The Process of
Conceptual
Modeling**

A **problem concept** is an important entity in a problem, and a conceptual model represents the important entities in a problem, their responsibilities or attributes, the important relationships between them, and perhaps some of their behaviors. In object-oriented conceptual modeling, classes represent concepts, so the important characteristics are recorded as attributes. Associations represent relations. Important concept behaviors, if recorded, are shown as operations.

Although conceptual models can be made at any level of abstraction, they generally represent problem entities at an intermediate level of abstraction corresponding roughly to operational-level requirements and use case models. User and other interfaces tend to draw on standard concepts such as buttons, windows, and standard data types, so there is not much use in modeling physical-level problem concepts.

The conceptual modeling process starts with information about the target product and produces a conceptual model—in our case, a UML class diagram. The first step of this process is to identify the important problem concepts (classes). In the second step, important characteristics of these concepts (attributes) are added to each class. During the third step, important relationships (associations) between concepts are added to the model. In the final step, constraints on the relationships (multiplicities) are added. The general process for making a conceptual model as a class model

is shown in the activity diagram in Figure 7-3-1. We discuss each of these steps in turn in this section.

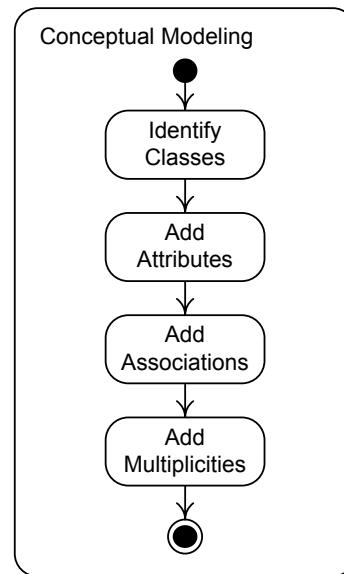


Figure 7-3-1 Conceptual Modeling Process

Identifying Classes Concepts are things of importance in the problem, and nouns and noun phrases refer to things. A good way to begin looking for classes, then, is to read through the product design and make a list of significant nouns and noun phrases. The SRS, except for the physical design, and the use case model, if any, should both be examined. This list of candidate classes should include the following kinds of things:

- Physical entities, individuals, roles, groups, and organizations;
- Things in the real world managed, tracked, recorded, or otherwise represented in the product; and
- People, devices, or systems with which the product interacts—in a use case model, this would be all the actors.

Do not add use cases themselves to the list; use cases represent interactions that are generally the result of the collaborative activity of several entities, not entities themselves.

The list of candidate classes can now be rationalized and put into a class diagram. Review the list and look for the following elements:

- Nouns or noun phrases designating characteristics of other entities in the list. For example, *size*, *height*, *width*, *weight*, and similar nouns may be in the list, but these are obviously attributes of some other entity. Add them to the appropriate classes as attributes. If in doubt about whether something should be a class or an attribute, make it a class.

- Noun phrases referring to activities or behaviors. For example, the list might include phrases such as *tax calculation* or *item inspection*. If such an item is an important behavior clearly belonging to a particular class, add it to the class as an operation.
- Entities that are essentially the same but have different names. Choose the best name and make a single class, keeping all attributes of the combined entities.
- Entities that do not directly interact with the product. A conceptual model can include such things, but the model is a better basis for engineering design if they are omitted.
- Entities that are irrelevant to the problem. Sometimes things are mentioned in specifications, explanations, or examples that have little importance to the problem, are particular values, or are other things that do not make good entities. Eliminate them from further consideration.
- Vague noun phrases. Define items precisely or remove them from further consideration.
- Implementation entities such as collections, files, processes, data types, and so forth. Such classes are not appropriate for a conceptual model, so remove them.

This simplification activity may lead to the discovery of new classes as the problem is better understood.

This step results in a draft class diagram. The classes may have a few attributes and some operations, but no associations. To illustrate this step, consider the following short description of a product that we will use as the basis for making a conceptual model.

Caldera Example

Caldera is a smart water heater controller that attaches to the thermostat of a water heater and provides more efficient control of the water temperature to save money and protect the environment. Caldera sets the water heater thermostat high when hot water is much in demand and sets it low when there is not much demand. For example, Caldera can be told to set the thermostat high on weekday mornings and evenings and all day on weekends, and low during the middle of weekdays and at night.

Furthermore, Caldera can be told to set the thermostat high all the time in case of illness or other need, or be told to set the thermostat low all the time in case of vacation or some other prolonged absence from the house.

The homeowner can specify values for the following Caldera parameters:

Low Temp—Temperature when little or no hot water is needed.

High Temp—Temperature when much hot water is needed.

Weekend Days—Days when the thermostat will be set to *High Temp* all day long; on all other days it will vary between *Low Temp* and *High Temp*.

Peak Times—From one to three time periods on a 24-hour clock during which the thermostat will be set to *High Temp* on non-*Weekend Days*. On

Weekend Days, the thermostat will be set to *High Temp* during the entire period between the earliest time and the latest time set in *Peak Times*.

Mode—One of the following Caldera states:

Stay Low Mode—Thermostat is set to stay at *Low Temp*.

Stay High Mode—Thermostat is set to stay at *High Temp*.

Normal Mode—Thermostat is changed between *Low Temp* and *High Temp* on a regular schedule, as explained above.

Caldera has its own internal clock that it checks every second to determine how to set the water heater thermostat.

The noun phrases in this description are listed in Table 7-3-2 classified according to their use in the class model.

Noun Phrases	Comment
Water heater controller, thermostat, homeowner, clock	Concepts
Mode, Low Temp, High Temp, Weekend Days, Peak Times	Attributes of water heater controller
Time	Attribute of clock
Caldera, money, environment, weekday, morning, evening, need, vacation, day, week, night, middle, illness, absence, house, parameter, value, one, three, schedule, second	Irrelevant noun phrases (parts of explanations, examples, etc.)
Water heater, water temperature, hot water	Indirect connection to program

Table 7-3-2 Disposition of Noun Phrases

For the first step of the process, we can take the results of this analysis and draw the first draft of the Caldera conceptual model shown in the class diagram in Figure 7-3-3.

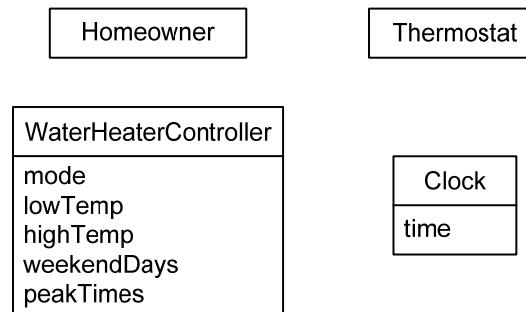


Figure 7-3-3 Caldera Conceptual Model, Draft 1

Adding Attributes

The second step is to add obvious and important attributes of each class, along with their types, multiplicities, and initial values. The SRS and the use case model are again consulted for this activity. The following guidelines help with adding attributes:

- Adjectives and modifiers sometimes give clues about class attributes. For example, a requirement that a product “must search the smallest blocks first” suggests that blocks must have a **size** attribute.
- Attribute names should be taken from the problem domain. For example, a **Radio** has a **volume** setting, not a **loudness** setting.
- Attribute type, multiplicity, and initial value information should be included only if it is specified in the problem.
- Attributes should not be added for object identification unless an identifier is part of the problem. For example, a **Student** class should not have an identifier attribute unless a student number is part of the problem.
- Be careful not to add attributes that specify implementation details, such as data structures or types. For example, it may seem clear that a **Playlist** class needs a **size** attribute even if one is not mentioned in the problem. However, adding it is making an engineering design resolution decision; alternatively, the **Playlist** size could be computed whenever it is needed rather than recorded in the class.

As a rule, operations should not be recorded in conceptual models because conceptual models are supposed to concentrate on the entities in the problem, their characteristics, and their relationships. Adding operations unnecessarily constrains engineering design. If certain behaviors are essential to particular entities in the problem, they can be added as operations in the class model.

The result of the second step of the conceptual modeling process is a revised draft of the model showing classes, their attributes, and perhaps a few operations as well.

Continuing with our Caldera example, the **WaterHeaterController** and **Clock** attributes seem to have been found along with the classes. The **mode** has a special **ModeType**, and **lowTemp** and **highTemp** are **Temperatures**. The user tells Caldera which days the thermostat is set to **highTemp** all day: These are the weekend days. A user home all the time, on vacation or ill, may want the thermostat set to **highTemp** every day, so the **weekendDays** attribute must hold zero to seven **Day** values. Note that the multiplicity of **weekendDays** should be **0..7** rather than just **7**. The former says that **weekendDays** can hold zero to seven values, while the latter says it must hold exactly seven values, and we need the former. Similarly, we see that **peakTimes** must hold one to three **TimePeriods**, so its multiplicity should be **1..3**. The **Clock** attribute **time** is self explanatory, so we have not designated a type for it. The **Homeowner** has no interesting attributes. The **Thermostat** must have a temperature setting attribute, so we add this into the model; it is of type **Temperature**.

Figure 7-3-4 shows our current draft of the model.

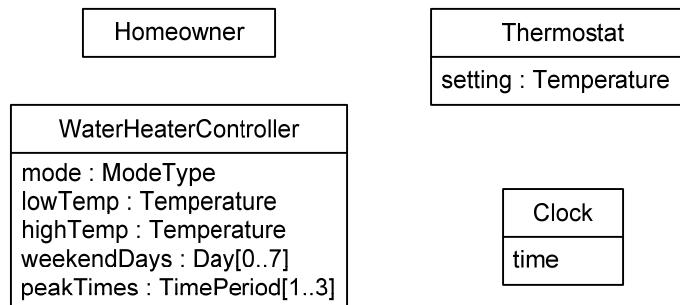


Figure 7-3-4 Caldera Conceptual Model, Draft 2

Adding Associations The third step of the process is to add associations. Once again the SRS and use case model should be reviewed, this time looking for transitive verbs and prepositions describing relationships between entities represented by classes in the model. Labeled association lines can be added to the class diagram as relations are found.

The following sorts of verb phrases and prepositions often indicate important relationships that should be modeled:

- Physical or organizational proximity, such as “below” or “under”;
- Control, coordination, and influence relationships, such as “reports to,” “consults,” “monitors,” “controls,” or “supervises”;
- Creation, destruction, and modification relationships, such as “initiates,” “edits,” “completes,” or “disposes of”;
- Communication relationships, such as “sends,” “signals,” or “interrupts”;
- Ownership or containment relationships, such as “owns,” “contains,” “is part of,” “inside,” or “has.” The latter word is used with many meanings, so care must be taken to decipher the relationship.

The associations resulting from this activity should be reviewed and adjusted in the following ways:

- The greatest danger is that too many associations will be noted. There are often dozens of relationships among the classes in a model: Include only those most important to the problem. In general, do not draw more than one association line between any two classes, and do not make more associations than classes.
- Sometimes multiple association lines between the same classes really represent the same relationships under different names. If so, keep the best name or determine if a better name can be devised, then combine the association lines.

- Break any associations involving three or more classes into binary associations.
- Check association names to make sure they are descriptive, and add directional arrows if it is not clear which way to read the labels.
- Add rolenames where they are needed to make the model clear.

The following relationships are important in the Caldera example: the Homeowner specifies WaterHeaterController parameters, the WaterHeaterController checks the Clock every second, and the WaterHeaterController sets the Thermostat. Associations capturing these relations are shown in the diagram in Figure 7-3-5.

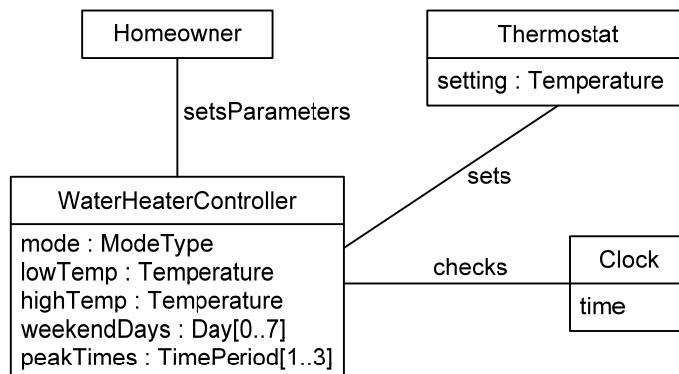


Figure 7-3-5 Caldera Conceptual Model, Draft 3

At completion of this step of the process, the class model contains classes, attributes, associations, and possibly some operations. The only task that remains is adding multiplicities.

Adding Multiplicities

Adding multiplicities to a conceptual class model is the easiest step. Simply take each associated pair of classes in turn, choose one end as the source (the other end will be the target), and ask the question: How many instances of the target class can be related to a single instance of the source class? The multiplicity that answers this question is added to the target end of the association line. Reversing the roles of the target and source and then proceeding as before determines the other multiplicity.

The product design may need to be consulted to assign multiplicities. If a multiplicity cannot be determined, it may be that it does not matter in the problem, or that it does matter but the product designers failed to specify it. If it does not matter, then the multiplicity can remain unstated. If it does matter, then the product designers need to be consulted to work out this specification.

The Caldera model association multiplicities are not very interesting: There is one of everything except the Homeowner, and presumably any number of

them may set parameters. The final version of the Caldera conceptual model is shown in Figure 7-3-6.

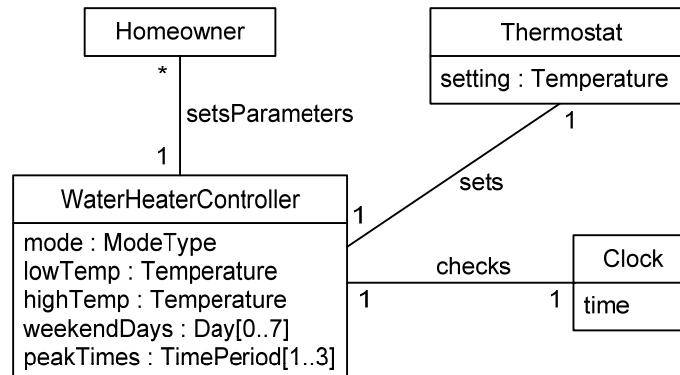


Figure 7-3-6 Caldera Conceptual Model, Final Draft

Heuristics Summary

Figure 7-3-7 summarizes the conceptual modeling process heuristics discussed in this section.

Identifying Classes

- Make a list of nouns and noun phrases from the SRS and other product design artifacts.
- Include use case actors but not use cases.
- Rationalize the list by removing vague or redundant noun phrases and those denoting attributes, operations, indirectly interacting entities, irrelevant entities, and implementation entities.

Adding Attributes and Operations

- Deduce attributes from adjectives and modifiers.
- Use attribute names, types, and multiplicities from the problem.
- Add operations sparingly.

Adding Associations

- Deduce associations from verb phrases and prepositions.
- Have at most one association between any two classes.
- Don't have more associations than classes.
- Combine equivalent associations.
- Break associations between three or more classes into binary associations.
- Make sure association names are descriptive.
- Add rolenames to make the model clear.

Figure 7-3-7 Conceptual Modeling Process Heuristics

- Section Summary**
- A **problem concept** is an important entity in a problem.
 - A **conceptual model** represents problem concepts, their characteristics, and their relationships.
 - A four-step process for making conceptual models is (1) Identify the classes, (2) Add attributes, (3) Add associations, and (4) Add multiplicities.

- Review Quiz 7.3**
1. What sorts of noun phrases tend to indicate potential classes for a conceptual model?
 2. What sorts of verb phrases tend to indicate potential associations for a conceptual model?
 3. Why should operations rarely appear in a conceptual model?
-

Chapter 7 Further Reading

- Section 7.1** The main alternative approach to object-oriented analysis and design is structured analysis and design. A useful structured analysis static modeling tool is the context diagram. Useful dynamic analysis modeling tools are data flow diagrams and state diagrams. Yourdon [1989] discusses these in depth. Although introduced in the context of object-oriented analysis and design methods, use case models can be used equally well in structured analysis or object-oriented analysis efforts.
- The distinction between conceptual, design, and implementation class models is based on [Fowler and Scott 1997].
- Section 7.2** Many books discuss UML class and object diagrams, but they often contain errors or are outdated. The best sources are the UML specification itself ([OMG 2003]); books by the originators of UML ([Booch et al. 2005] and [Rumbaugh et al. 2004]); and comprehensive recent publications, such as [Bennett et al. 2001].
- Section 7.3** The basic technique for identifying concepts and relations is based on Abbot [1983], who first proposed using parts of speech as guideposts. This technique has been adopted widely and is discussed in [Bennett et al. 2001] and [Wirfs-Brock and McKean 2003].

Chapter 7 Exercises

The following product descriptions will be used in the exercises.

- Grow Light** A maker of expensive novelty gifts is designing a special grow light for houseplants. This device will monitor the hours of sunlight each day, along with the sunlight's intensity, using a light meter. When light levels are low, the grow light will come on to augment the natural light. When the sun goes down, the grow light will come on for as long as necessary to provide an equivalent minimum number of hours of daylight.
- The user interface to this device is an on-off switch and a dial specifying the minimum number of hours of daylight for the plant. The device also has a 24-hour clock accurate to the second.

Arboretum Management System

An arboretum has extensive plantings that must be maintained and monitored. An Arboretum Management System keeps track of all plant stocks, the location and value of each, and the contributors who supplied the funds to purchase them.

The arboretum has dozens of beds of varying shapes and locations. The layout of the arboretum is maintained in a map database that records the shapes of all beds and the contents and approximate locations of all plantings in each bed. Gardeners may redraw the beds when they are changed. Gardeners may also query map images for a display of the quantity and location of the contents of a bed.

Whenever a plant is purchased, its species, color, size, value, donator, and annuality (perennial or annual) is recorded. Its location (bed or greenhouse) is also recorded. Plants are removed from the database when they die.

Gardeners can query the system to determine the locations of all plants of a particular kind or of all plants donated by a certain contributor.

When requested, the system produces a complete report of all plants owned by the arboretum.

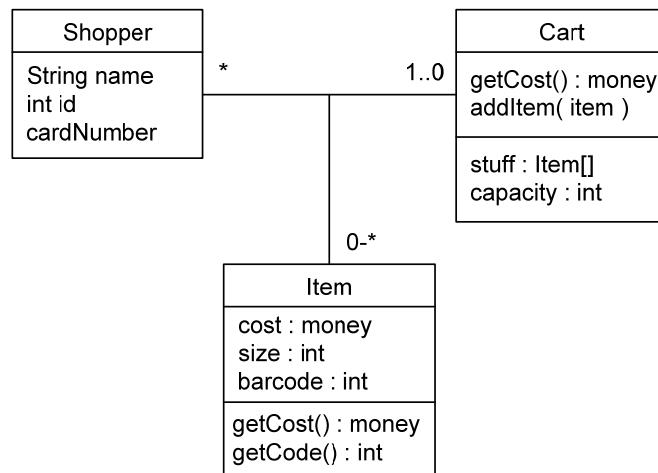
Section 7.1

1. *Fill in the blanks:* The first step of engineering design is _____ . The inputs to this activity include the _____ , perhaps supplemented with _____ , both produced during product design. In terms of the software life cycle, engineering design occurs during the _____ phase. A good way to engage the problem during this activity is to _____ , which produces the main outputs of this activity.
2. Explain the difference between a class model and a class diagram.
3. Explain the difference between analysis class models, design class models, and implementation class models.

Section 7.2

4. Using the restrictive policy that simple names are strings of letters, numbers, and underscores, which of the following terms are simple names (S), composite names (C), or neither (N)? Assume that the path delimiter is the double colon “::”.
 - (a) java
 - (b) java.util.Vector
 - (c) Java::Compiler
 - (d) Java Compiler
 - (e) 3.14
 - (f) PI
 - (g) Math::PI

5. *Find the errors:* Which of the following terms are correctly formed UML attribute specifications?
 - (a) height
 - (b) HEIGHT
 - (c) Rectangle::height
 - (d) float height
 - (e) height : float[*]
 - (f) height = float[5]
 - (g) height : [5]
 - (h) height : float[]
 - (i) height = 0
6. *Find the errors:* Which of the following expressions are correctly formed UML operation specifications?
 - (a) crunch
 - (b) crunch()
 - (c) crunch(v = 5)
 - (d) crunch(v : Cereal)
 - (e) crunch() : boolean
 - (f) crunch();
 - (g) boolean crunch(Cereal v)
 - (h) crunch(Cereal v = 5) : boolean
 - (i) crunch : boolean
7. *Find the errors:* Which of the following expressions are correctly formed UML multiplicity specifications?
 - (a) 0..1
 - (b) 0,1,2,4,8
 - (c) 1..0
 - (d) *..10
 - (e) *..*
 - (f) 0..5,10..20
 - (g) 1,*
 - (h) 0..1, 5..*
8. How would you determine the multiplicity at the ends of an association representing a relation between the instances of three classes?
9. Make a UML class diagram describing the `java.awt.Insets` class (You can see the documentation for this class at <http://java.sun.com>).
10. Make a UML class diagram whose classes correspond to the main parts of a computer (CPU, monitor, keyboard, and so forth). Your diagram should include at least one attribute and one operation for each class. Add associations corresponding to the connections among these parts.
11. *Find the errors:* Identify at least five ways in which the diagram in Figure 7-E-1 does not conform to the rules for making UML class diagrams. Redraw the diagram so that it is correct.

**Figure 7-E-1 An Erroneous Class Diagram for Exercise 11**

12. Answer the following questions based on the diagram in Figure 7-E-1:
 - (a) What is the multiplicity of `capacity` in the `Cart` class?
 - (b) How many parameters does `getCost()` in the `Item` class have?
 - (c) Could the `Cart` class have additional attributes that do not appear in the diagram?
 - (d) Suppose that an association is added between `Cart` and `Item` named `contains` that holds when a `Cart` instance contains an `Item` instance. What multiplicities should constrain this association?
13. List two heuristics different from those in the text that you think would make class diagrams easier to read or write.
14. Write a checklist to aid in reviewing class diagrams.
- Section 7.3** 15. Make a list of noun phrases in the Grow Light product description and use them to produce a draft conceptual model of this problem. Your draft should include classes and perhaps some attributes.
16. Add attributes, associations, and multiplicities to the draft you produced in the last exercise to complete a conceptual model of the Grow Light product.
17. Make a list of noun phrases in the Arboretum Management System product description and use them to produce a draft conceptual model of this problem. Your draft should include classes and perhaps some attributes.
18. Add attributes, associations, and multiplicities to the draft you produced in the last exercise to complete a conceptual model of the Arboretum Management System product.

- Team Projects** 19. Form a team of two. Independently make conceptual models of the Fingerprint Access System described in the Exercises section of Chapter 6. Write an essay describing how your models differ and why.

- AquaLush** 20. Form a team of at least two. Independently make conceptual models of the AquaLush system without looking at the conceptual model in Appendix B. Compare and contrast the conceptual models made by team members and create one representing the best efforts of the team. Finally, compare and contrast the team's model with the one in Appendix B. Which one is better? Did this exercise help you understand the AquaLush engineering design problem?
- Research Project** 21. Consult books about CRC cards. Make a conceptual model of one of the following systems using CRC cards:
- (a) Caldera
 - (b) Grow Light
 - (c) Arboretum Management System
 - (d) AquaLush

Chapter 7 Review Quiz Answers

Review Quiz 7.1

1. If engineering designers do not have a complete, correct, and consistent product design provided by product designers, they have no choice but to complete the product design themselves. It is impossible to figure out how to realize a product without knowing exactly what the product is.
2. The most useful static model for object-oriented engineering analysis is an analysis class model. The most useful dynamic model is a use case model.
3. Examples of things that should not appear in a conceptual model but might appear in a design class model are those that have to do with the software system rather than the problem. Specific examples include class attributes that only have to do with the implementation, such as counters, references, and convenience variables; characteristics of attributes that only have to do with the implementation, such as whether they are public or private or what their data structure is; operations that only have to do with implementation, such as constructors, destructors, or finalizers and attribute get and set methods; or aspects of operations that only have to do with implementation, such as whether they are public or private or what sort of concurrency they support.
4. Examples of things that should not appear in a design class model but might appear in an implementation class model are those having to do with the implementation of a program in a particular environment and language. Some examples are particular classes from class libraries, such as `java.lang.Integer`; language-specific data types, such as `unsigned long`; operations, such as `finalize()`; initializer expressions, such as `={2,4,6}`; and so forth.

Review Quiz 7.2

1. The first compartment of a class symbol must always be the name compartment because it cannot be suppressed. The second compartment might be the attributes compartment, the operations compartment if the attributes compartment is suppressed, or an optional compartment if both the attributes and operations compartments are suppressed.
2. A syntactically correct attribute specification that suppresses or omits as much as possible lists only the attribute name; for example, `length`. An attribute specification that suppresses or omits nothing would have a name, type, multiplicity, and initial value, such as `length : integer[2] = (0,0)`.

3. A Person may love zero or more Persons (multiplicity *). A Person can see one or more Persons at any moment (1..*). A Person is married to zero or one Persons at any moment (0..1).
4. The following heuristics help with drawing good class diagrams: name classes, attributes, and roles with noun phrases; name operations and associations with verb phrases; capitalize class names but not attribute, operation, association, and rolenames; center class and compartment names but left-justify other compartment contents; stick to binary associations; prefer association names to rolenames; place association names, rolenames, and multiplicities on opposite sides of the line.
5. If objects $a_1:A$ and $a_2:A$ are both linked to object $b:B$ in relation R , then the multiplicity of association R on the A end must include two and on the B end must include one. This still leaves infinitely many possibilities for the multiplicity of R .

**Review
Quiz 7.3**

1. Noun phrases that tend to indicate potential classes for a conceptual model include those designating physical entities; individuals; roles; groups; organizations; things in the real world managed, tracked, recorded, or otherwise represented in the product; and people, devices, or systems with which the product interacts.
2. Verb phrases that tend to indicate potential associations for a conceptual model include those that express physical or organizational proximity, control, coordination, influence, creation, destruction, modification, communication, ownership, or containment relationships.
3. Conceptual models are supposed to represent the important entities in the problem along with their main characteristics and relations. Operations are peripheral to this goal. Furthermore, specifying operations in the conceptual model may unnecessarily constrain the engineering design. Consequently, operations should rarely appear in conceptual models.

8 Engineering Design Resolution

Chapter Objectives

Once the engineering design problem is understood, it must be solved. The diagram in Figure 8-O-1 depicts the place of engineering design resolution in the software engineering design process.

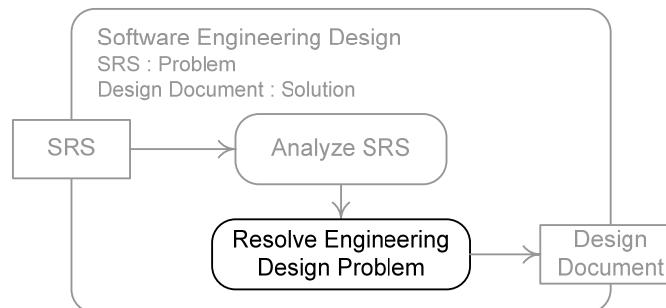


Figure 8-0-1 Software Engineering Design

This chapter introduces the main steps in the resolution activity and explores the principles that guide engineers in evaluating designs and making design decisions.

By the end of this chapter you will be able to

- Distinguish architectural and detailed design;
- List the items comprising architectural and detailed design specifications;
- State and explain basic engineering design Principles of Feasibility, Adequacy, Economy, and Changeability; and
- State and explain principles used to evaluate particular aspects of an engineering design, including the principles of Small Modules, Information Hiding, Least Privilege, Cohesion, Coupling, Reuse, Simplicity, and Beauty.

Chapter Contents

- 8.1 Engineering Design Resolution Activities
 - 8.2 Engineering Design Principles
 - 8.3 Modularity Principles
 - 8.4 Implementability and Aesthetic Principles
-

8.1 Engineering Design Resolution Activities

Architectural and Detailed Design

Engineering design resolution is traditionally divided into two major phases or activities: architectural design and detailed design. We characterize these activities by the following definitions.

Architectural design is the activity of specifying a program's major parts; their responsibilities, properties, and interfaces; and the relationships and interactions among them.

Detailed design is the activity of specifying the internal elements of all major program parts; their structure, relationships, and processing; and often their algorithms and data structures.

Engineering design resolution and its major activities can be understood in terms of *black boxes*, or units with external form and behavior but hidden internal workings. During product design, the entire program is treated as a black box, and only its external form and behavior are specified. During architectural design the program black box is “opened”—its high-level structure is determined, and its major constituents and their interactions are specified. However, the major program constituents are treated as black boxes during architectural design. In detailed design, each major constituent is opened and its internal structure and behavior are specified. Depending on the size of the program, there may be other layers of black boxes considered during detailed design. Ultimately, all black boxes are opened and the entire workings of the program are specified.

In the remainder of this section we consider what happens during architectural and detailed design in more depth. Later sections discuss the design principles governing engineering design resolution.

Architectural Design Specification

The central goal of architectural design is to determine the high-level structure of the program, or its **software architecture**. As we will see in the next chapter, there are both technical and organizational considerations that influence architectural design. For now, we consider the items that may appear in a software architecture specification:

Decomposition—Depending on the size of the product, the major parts into which a product is divided can range in scope from single classes to entire large collections or packages. For example, the AquaLush architecture divides the program into five layers, each one a module containing several program units (layered architectures are discussed in Chapter 15). This decomposition is pictured in Figure 8-1-1 on page 228.

Responsibilities—Each module must be placed in charge of certain data and activities. For example, the AquaLush **Startup** layer configures the product when it begins execution. The **User Interface** layer is in charge of tracking the user interface state and changing it in response to user and product actions. The **Irrigation** layer tracks the state of the valves and sensors, holds program parameters, and controls irrigation. The **Device Interface** layer provides a uniform interface to the hardware portion of the product, such as the sensors, valves, clock, keypad, and display. The **Simulation** layer implements virtual devices used in the AquaLush Web simulation; it is present only in the simulation.

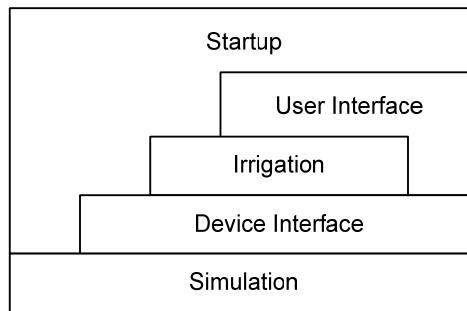


Figure 8-1-1 AquaLush Architectural Constituents

Interfaces—An **interface** is a boundary across which entities communicate. An **interface specification** is a description of the mechanisms that an entity uses to communicate with its environment. Somewhat confusingly, we often refer to an interface specification as simply an “interface.” Architectural constituents must have their interfaces specified so that they can be developed independently. For example, the interface specification for the AquaLush **Device Interface** layer says that it provides virtual valves with `open()` and `close()` operations that throw valve failure exceptions. Developers of the other parts of the system can then rely on the fact that they can communicate with virtual valves using these operations.

Collaborations—When units cooperate to achieve computational goals, they typically do so according to some protocol or course of interaction. For example, the AquaLush **Irrigation** layer must keep track of the current time, which is kept by a virtual clock in the **Device Interface** layer. One way the **Irrigation** layer can track the time is for it to continually poll the virtual clock. Another way is for the virtual clock to notify the **Irrigation** layer every minute; the **Irrigation** layer can then query the virtual clock for the current time.

Relationships—An architectural design may specify that parts communicate in certain ways or that there are dependencies or other relationships between them. For example, the modules or layers in the AquaLush architecture may use only modules directly below them in the layer diagram of Figure 8-1-1. The **Startup** layer may use any layer; the **User Interface** layer may use only the **Irrigation** and **Device Interface** layers; the **Irrigation** layer may use only the **Device Interface** layer; and the **Device Interface** layer may use only the **Simulation** layer (if it is present).

Properties—Many products have requirements about the time operations can take; the resources, such as processor time or memory, that can be consumed; the rate at which transactions must be completed; and their reliability. Products can satisfy their requirements only if their constituents have certain properties, which must be specified. For example, suppose that a product must perform a transaction in one second or less, and suppose that three of its sub-systems must cooperate

to process a transaction. Each sub-system must complete its part of transaction processing in a certain amount of time, such as a third of a second. The property of completing transaction processing in a third of a second or less is a property specification.

States and State Transitions—Units may have important states that affect their externally observable behavior. If so, these states and the transitions among them should be specified. For example, the Aqualush Irrigation layer has two main states: manual mode and automatic mode. It switches between these states based on input from the User Interface layer.

This list of kinds of specifications can be remembered using the acronym **DeSCRIPTR**, where *De* stands for *Decomposition* and the other letters stand for *States*, *Collaborations*, *Responsibilities*, *Interfaces*, *Properties*, *Transitions*, and *Relationships*. An architectural specification does not have to supply all this information for every constituent. There is no need to document obvious or trivial aspects of a design. For example, a unit may not have interesting or important states, so there is no need to document its states and state transitions. However, some information—in particular, about responsibilities and interfaces—are almost always important.

An architectural design must specify both the structure and the behavior of the program at a high level of abstraction. Static and dynamic models, which we discuss in later chapters, are often used for this purpose.

Detailed Design Specification

Detailed design fills in the design specifications left open after architectural design. The main goal of detailed design is to specify the program in sufficient detail so that programmers can implement it. Detailed design specification includes most of the following characteristics:

Decomposition—Each major program module must be decomposed into program units, such as compilation units, packages, classes, modules, and operations. For example, the Aqualush Irrigation module is decomposed into Irrigator, IrrigationCycle, AutoCycle, ManualCycle, Zone, Sensor, and Valve classes.

Responsibilities—As with architectural constituents, program units are also in charge of certain data and behaviors. For example, the Valve class is responsible for keeping data about valves, including their flow rate, location, status, and water allocation during automatic irrigation. It is also responsible for opening and closing valves and monitoring water usage during automatic irrigation.

Interfaces—Program unit specifications describe the public aspects of the unit that other units can use to interact with it.

Collaborations—Units collaborate to achieve computational goals by calling operations and passing data. These interactions must be specified.

Relationships—Inheritance relationships, class associations, program unit dependencies, interface realization relationships, and similar relationships need to be specified in a detailed design.

Properties—Program units may have to satisfy certain specifications to meet needs for reliability, efficiency, memory utilization, and so forth. If so, then these must be stated.

States and State Transitions—Some units, such as objects, have states that are important for understanding the way they work. Detailed design should specify interesting state-based program unit behavior.

Packaging and Implementation—Program units and their internal elements typically reside in various containers. The allocation of units to containers may be specified, along with the scope and visibility of program elements, though this may be left to the programmers to determine. For example, AquaLush is implemented in Java. The AquaLush detailed design specifies the Java packages in the program, the classes in each package, and the visibility of each class in each package.

Algorithms, Data Structures, and Types—Detailed design is the least abstract design activity, but it is still done at a level of abstraction higher than program code (although the boundary between detailed design and coding is fuzzy). A detailed design may specify a program down to the level of descriptions of individual data types and structures, loops, branches, and low-level string and arithmetic operations. Most often, however, such details are left to the coders, and detailed design stops just above the level of data structures and algorithms.

These specifications include all the DeSCRIPTR elements plus four more items: *Packaging*, *Algorithms*, *Implementation*, and *Data structures and types*, which can be remembered with the acronym **PAID**. As in the case of architectural design specifications, not every program unit in a detailed design must be specified with all the DeSCRIPTR-PAID information; only the interesting and non-trivial specifications that the designers do not want to leave for the coders must be included.

Section Summary

- **Architectural design** is the activity of specifying a program's decomposition into major parts and their states, collaborations, responsibilities, interfaces, properties, state transitions, and relationships (**DeSCRIPTR**).
- **Detailed design** is the activity of specifying the internal elements of all major program parts, possibly including their decomposition, states, collaborations, responsibilities, interfaces, properties, state transitions, relationships, packaging, algorithms, implementation, and data structures and types (**DeSCRIPTR-PAID**).

Review Quiz 8.1

1. Explain product design, architectural engineering design, and detailed engineering design in terms of black boxes.
2. What does the acronym DeSCRIPTR-PAID stand for?

8.2 Engineering Design Principles

Criteria for Design Evaluation

During engineering design resolution, design alternatives are generated and evaluated. Evaluation is made in light of several **design principles** or **tenets**, which are statements about characteristics that make designs better. Software engineering design shares many design principles with other design disciplines because of the common design goal of creating high-quality, long-lived products. For example, any design should meet stakeholder needs, be buildable, not cost too much, and so forth. We call principles stating characteristics that make a design better able to meet stakeholder needs and desires **basic design principles**.

Other design principles specify characteristics of programs that we know from experience are essential for achieving high-quality products. These principles are mostly unique to software engineering design because they are about the characteristics of well-constructed programs. For example, we have learned over the years that software is more robust and maintainable when it is built from fairly small parts that hide their internal processing details from each other. This peculiar feature of programs is of no interest to software product designers or users, but it is very important to software engineering designers and programmers. Because such principles are crucial for building high-quality programs, we call them **constructive design principles**.

In the next few sections we discuss the most important engineering design principles, beginning with basic design principles and moving to constructive design principles. These principles will be used and discussed extensively as we consider design problems in later chapters.

Basic Principles

The primary goal of software engineering design is to specify the structure and behavior of programs satisfying software product specifications. Good programs must have certain characteristics at delivery and must be maintainable over time. Specifically, at delivery a good program must satisfy its requirements and conform to its design constraints. To be deliverable, a program must be designed so that it can be built for an acceptable amount of time, for an acceptable amount of money, and with acceptable risk. For a program to be maintainable over time, it must be changeable. These considerations give rise to four basic design principles: Feasibility, Adequacy, Economy, and Changeability.

Feasibility The bottom line for any design is that it must specify something that can be built and will work, which leads to the **Principle of Feasibility**.

A design is acceptable only if it can be realized.

Adequacy Once designers determine that a design specifies something that can be built, the next concern is whether the result will do what it is supposed to do; that is, whether the design meets stakeholder needs and desires subject to constraints. It may not be possible to meet all stakeholder needs, to meet them all fully, or to satisfy all constraints. Some designs may meet more needs than others or satisfy more constraints than others. Hence, adequacy is a matter of degree. This leads to the following **Principle of Adequacy**.

Designs that meet more stakeholder needs and desires, subject to constraints, are better.

Economy Software development is expensive, time consuming, and risky. A good design should specify a program that can probably be built, tested, and deployed on time and within budget. This is the justification for the **Principle of Economy**.

Designs that can be built for less money, in less time, with less risk, are better.

The Principles of Adequacy and Economy may be in conflict: A designer may be unable to specify a program meeting stakeholder needs and satisfying constraints that can be built on time and within budget, with reasonable risk. The designer must then work with stakeholders to make trade-offs among needs, constraints, time, money, and risk.

Changeability Approximately 70% of lifetime software product costs are for maintenance. Designs that make a product easy to change will have a tremendous influence on cumulative life cycle costs. The **Principle of Changeability** acknowledges this fact.

Designs that make a program easier to change are better.

These basic design principles state fundamental criteria for evaluating the overall quality of a design. Unfortunately, they are not very helpful in determining whether particular aspects of a design are good or bad, or in deciding between design alternatives at low levels of abstraction. The principles discussed in the next two sections give more specific guidance in evaluating design details.

Summary of Basic Design Principles

Figure 8-2-1 summarizes the basic design principles.

Feasibility—A design is acceptable only if it can be realized.
Adequacy—Designs that meet more stakeholder needs and desires, subject to constraints, are better.
Economy—Designs that can be built for less money, in less time, with less risk, are better.
Changeability—Designs that make a program easier to change are better.

Figure 8-2-1 Basic Engineering Design Principles

Section Summary

- **Design principles** are evaluative criteria that state characteristics of good designs.
- **Basic design principles** state desirable design characteristics based on meeting stakeholder needs and desires.
- **Constructive design principles** state desirable engineering design characteristics based on past software development experience.

Review Quiz 8.2

1. For what purposes are design principles used?
2. What is the difference between a basic design principle and a constructive design principle?
3. Why is changeability an important basic design principle?

8.3 Modularity Principles

Constructive Principles

Constructive design principles provide evaluative criteria based on technical details of engineering designs that developers have found through experience to be characteristic of good designs. Designs that rate highly according to constructive design principles also tend to rate highly according to basic design principles.

Constructive design principles fall into the following categories:

Modularity Principles—We have learned over the years that high-quality programs are invariably constructed from self-contained, understandable parts, or *modules*, with well-defined interfaces. Several important constructive design principles state criteria for judging whether designs specify good modules.

Implementability Principles—A design may be more or less easy to build. Ease of construction affects both adequacy and economy.

Aesthetic Principles—Beauty is widely acknowledged as an important design principle, and aesthetic concerns clearly contribute to design quality.

We consider constructive design principles in the modularity category in this section and principles in the implementability and aesthetic categories in the next section.

Modularity Modularity is defined as follows.

A **modular** program is composed of well-defined, conceptually simple, and independent units that communicate through well-defined interfaces.

Modular programs have several clear advantages over non-modular programs:

- Modular programs are easier to understand and explain because their parts make sense and can stand on their own. It is easier to understand and explain all the details of a whole if its parts can be considered in isolation.
- Modular programs are easier to document because their parts make sense and because each part can be documented independently.
- Programming individual modules is easier because the programmer can focus on individual small, simple problems rather than a large, complex problem.
- Modular programs are easier to change because modifications are usually restricted to only a few parts, and a change to one part has little or no effect on other parts.
- Testing and debugging individual modules is easier because they can be dealt with in isolation. Fewer and simpler tests are needed than are needed for the entire program. Bugs are easier to locate and understand, and they can be fixed without fear of introducing problems outside the module.
- Well-composed modules are more reusable because they are more likely to comprise part of a solution to many problems. In addition, a good module should be easy to extract from one program and insert into another.
- Modules are easier to tune for better performance because all the relevant data and processing are available and segregated from the rest of the program.

Modularity is probably the single most important characteristic of a well-designed program. But what exactly is a module?

What Is a Module?

Programs are wholes with a hierarchy of parts. At the top level of decomposition, programs have several major, large parts. These are in turn composed of somewhat smaller parts, and those of even smaller parts, down to the level of individual statements. In different contexts, units at any non-leaf level of this hierarchy may be considered design modules.

A **module** is a program unit with parts.

We can, at each level of the parts hierarchy, specify what we regard as a module along with its immediate parts. The *immediate parts* of a whole are the parts directly below the whole in the parts hierarchy. For example, we can say that a car has a chassis and a drive train (two immediate parts); that the drive train has an engine, a transmission, and wheels (three immediate parts); that the chassis has a frame, a body, and an interior (three immediate parts); and so on.

An exact specification of what counts as an immediate part depends on where a module is in the parts hierarchy. At the highest level of abstraction, an entire program is a module whose immediate parts are the sub-programs of which it is composed. A program is a module whose parts are its architectural constituents. The immediate parts of a sub-program module might be classes, functions, packages, or compilation units. The immediate parts of a class module are attributes and operations. The immediate parts of a compilation unit are declarations or lines of code. The immediate parts of a sub-program are lines of code.

At a low-enough level of abstraction are parts that we no longer consider decomposable; we eventually come to atomic parts. Thus, a line of code is generally not considered a module because its pieces, the lexical elements of a programming language, are not considered parts with respect to design decomposition. The lowest level or atomic elements in design decomposition are individual lines of code, and therefore the bottom-level modules are blocks or sub-programs.

Modularity Principles

Forming good modules is essential in software design. Several considerations come into play in making design decisions about what should go into a module, how modules should be packaged, what portions of a module's contents should be accessible, and so forth.

Modularity principles are design principles that serve as guides and evaluative criteria in forming modules. We present five modularity principles.

Small Modules

The simplest rule of thumb in forming modules is to keep them small. Large modules are hard to understand, explain, document, write, test, debug, change, and reuse. As a result, our first modularity principle is the following **Principle of Small Modules**.

Designs with small modules are better.

One might ask, “What counts as a small module?” As noted previously, a module is a whole with immediate parts, so this question becomes, “How many immediate parts does a small module have?” The answer to this

question depends on where the module is in the parts hierarchy. For example, a convention in procedural programming is that sub-programs have no more than about 60 lines of code. (This rule apparently originated from the fact that there are 60 lines on two pages of open fan-folded line-printer paper.) Another is that compilation units have no more than about 500 lines of code. Modules under these limits are small.

Many other such rules are based on the capacity of human short-term memory, which is said to be seven, plus or minus two, items. Thus, some people recommend that classes have seven (plus or minus two) public operations or that functions be decomposed into no more than seven (plus or minus two) sub-functions.

All such heuristics about module size are reasonable but may be violated when the occasion demands. The ultimate goal is to keep a module, wherever it occurs in the parts hierarchy, small enough to be a conceptually simple, comprehensible unit. When a module gets large enough to run afoul of a size heuristic, that is a signal for the designer or implementer that the module may be too hard to understand. In such cases the designer should consider dividing the module into two or more pieces. Some modules that do not violate size restrictions but are difficult to understand anyway should also be divided to make them more comprehensible. The bottom line is that designers must rely on their experience, engineering judgment, and the recommendations of others in deciding whether a module is a conceptually simple, comprehensible unit.

Information Hiding

Information hiding is a practice of very long standing in software design; David Parnas first proposed it in 1972. Although Parnas proposed it as a guide in decomposition, information hiding has much wider application. We will begin by defining the term.

Information hiding is shielding the internal details of a module's structure and processing from other modules.

For example, most modern languages allow local variables to be declared in sub-programs, and these variables can be accessed only from within those sub-programs. This mechanism *hides* the local variables inside their sub-programs.

Information hiding is good for both the module that hides its internals and the rest of the program:

- It is good for the hiding module because modification of module internals from the outside is prohibited. This protects the module from damage by errant or malicious external code and makes module repair, enhancement, and reuse easier.
- It is good for the rest of the program because it enforces disciplined use of the module. A module can be used without understanding its internals, thus reducing complexity and making programming much

easier. The interface to the module is fixed and stable, making it easier to understand and use. It is also easy to replace. Perhaps most importantly, the module may be changed without breaking the rest of the program.

For these reasons, the following **Principle of Information Hiding** is of fundamental importance in software design.

Each module should shield the details of its internal structure and processing from other modules.

All details about how a module works should be hidden. Specific examples of information best hidden include the following items:

- Use of data and operations from other modules, such as the attributes, constants, and operations from other classes, and details of how they are used;
- Internal organization, such as collaborations with other objects or the control flow within an operation;
- Internal data representations, such as data types and data structures;
- Algorithms, such as choices of alternatives for sorting, searching, insertion, deletion, and so forth;
- Volatile (likely to change) design decisions, such as sizes, capacities, waiting times, and so forth; and
- Non-portable code and data, such as code specific to certain operating systems, and data such as file paths and URLs.

Every module must also have public information, comprising its interface to the rest of the program, that cannot be hidden. Examples of public information include the following items:

- Names and properties of data provided by a module to its clients, such as public attributes, constants, and types;
- Names, parameters, and return types (that is, the signature) of operations provided by a module to its clients;
- The behavior of operations provided by a module, in particular transformation of input data to output data, events that invoke the operation, any events that the operation produces, and any side effects that the operation may have;
- Module error and exception conditions and behavior; and
- The other modules with which a module interacts as a client.

As an illustration of the Principle of Information Hiding, suppose we have an application that must track the contents of several bulk storage units, such as silos, tanks, or bins. We can create a `BulkStore` class to model such storage units. Instances of this class are responsible for knowing how much they can hold (`totalCapacity`), how much they currently hold (`currentQuantity`), and how much more they can hold (`remainingCapacity`). There is an

invariant relationship between these three quantities captured by the equation `totalCapacity = currentQuantity + remainingCapacity`. As a result of this relationship, the `BulkStore` class can maintain attributes tracking any two or all three of these quantities, giving the four alternative designs illustrated in Figure 8-3-1.

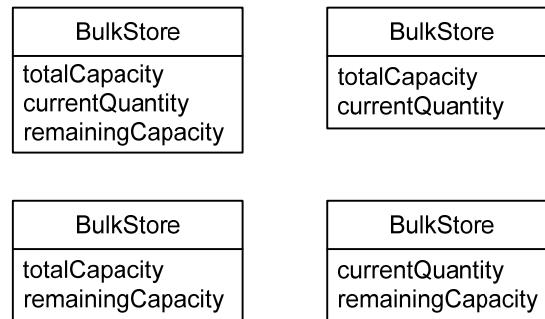


Figure 8-3-1 BulkStore Design Alternatives

Other parts of a program using the `BulkStore` class should not depend on which of these four alternatives the designers of the `BulkStore` class choose. Nor should the designers of the `BulkStore` class have to worry about whether other parts of the program assume one of these alternatives. This design decision should be hidden. This is done by making the attributes private and providing three public operations to query `BulkStore` instances about their `totalCapacity`, `currentQuantity`, and `remainingCapacity`. The `totalCapacity` is set at object creation, and the `currentQuantity` and `remainingCapacity` are altered by `add()` and `remove()` operations of the `BulkStore` class. The class interface is then fixed, and designers may make or change their decisions about which attributes are actually maintained by the class without fear of affecting the rest of the program.

Figure 8-3-2 shows the *public* parts of the `BulkStore` class. Note that these interface operations can be added to any of the previous four alternatives.

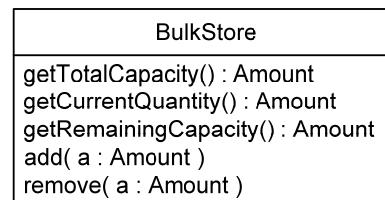


Figure 8-3-2 BulkStore Interface

- | | |
|-----------------|--|
| Least Privilege | Information hiding is about a module restricting access to the details of its internal processing. A complementary principle is that a module should not have access to resources that it does not need. This is called the Principle of Least Privilege and it is stated as follows. |
|-----------------|--|

Modules should not have access to unneeded resources.

A module with access to unneeded resources is harder to understand because it is not clear why the module has access to those resources. Such a module is less reusable because it (apparently) requires the presence of other (actually unneeded) resources. Perhaps most importantly, a module with access to unneeded resources can be the source of serious security problems.

The following examples illustrate violations of the Principle of Least Privilege:

- Modules that import packages or modules they do not need;
- Modules with unneeded access to files, programs, databases, or other computers;
- Classes with references to objects that are never accessed; and
- Operations with parameters for passing data they don't need.

Coupling	Coupling is a historically important concept originally proposed for procedural design that has proven important for object-oriented design as well.
----------	--

Coupling is the degree of connection between pairs of modules.

The strength of module coupling depends on module communication. Pairs of modules that communicate extensively or communicate in undesirable ways are *strongly coupled* or *tightly coupled*. Pairs that communicate only a little and in desirable ways are *weakly coupled*. Pairs that do not communicate at all are *not coupled* or *decoupled*.

To illustrate coupling, suppose that a program has two classes that work together to display financial data. The **Holdings** class maintains a list of investments, and the **DataDisplay** class makes charts and graphs about investments. One way that these classes can work together is for the **Holdings** class to respond to changes in its data or to requests for different displays by telling the **DataDisplay** class what to display. For example, suppose that the data in the **Holdings** class changes. The **Holdings** class might then have to call **DataDisplay** operations such as `displayInvestment()`, `displayTotalInvestments()`, or `graphInvestments()`.

The **Holdings** and **DataDisplay** classes are tightly coupled in this design: The **Holdings** class must be aware of what the **DataDisplay** class can display, and it must call the right operations at the right times to update the display.

An alternative communication strategy is for the **Holdings** class to notify the **DataDisplay** class whenever its data changes. The **DataDisplay** class is then responsible for querying the **Holdings** class to obtain the new data and for displaying it in the right way. Coupling is greatly reduced in this design.

alternative because the **Holdings** class does not need to know what the **DataDisplay** class does; the **Holdings** class only needs to know how to notify the **DataDisplay** class.

Strongly coupled modules are hard to change independently because programmers must worry about the data exchanged between them. Strongly coupled modules are also hard to understand, explain, and document because they cannot be considered independently of the modules to which they are coupled. Finally, strongly coupled modules are harder to test, debug, and maintain than weakly coupled modules because they cannot be treated in isolation.

Modularity is enhanced when coupling is minimized. Hence, we have the **Principle of Coupling**.

Module coupling should be minimized.

If two designs are equal in all other ways, then the one with the least coupling is better.

The notion of coupling was originally a way of characterizing good sub-program design. Different levels of sub-program coupling were defined, ranging from the tight coupling that results when one sub-program depends on the private details of another (such as accessing local variables or branching into another sub-program's code), to the loose coupling that results from passing data between sub-programs using parameters and function return values. We now realize that the Principle of Coupling has a much broader application than just to sub-program modules: Module coupling should be minimized at all levels of the parts hierarchy.

This breadth of application makes it difficult to characterize coupling levels precisely. Nevertheless, we can make a few observations about coupling strength:

- Failure to hide information leads to tight coupling and should always be avoided. Tight coupling always results when a module has access to details of another module's processing.
- Modules that communicate through a global entity such as a global variable, a file, or a global class are more tightly coupled than modules that communicate directly. Such modules are linked strongly to the global entity and hence to one another. Such communication should be avoided.
- Modules that communicate using special data types or data structures are more tightly coupled than modules that pass simple, standard data values. Relying on special data types or data structures links modules more tightly because they both must be changed if one of them needs to alter their common data format. Special data types or data structures should be avoided in module communication.

- When modules communicate only through public module interfaces, their coupling strength is proportional to the number of messages and the amount of data passed between them.

A simple heuristic for evaluating coupling strength between a pair of modules is to ask, “Could I use each of these modules in some other program without the other?” If the answer is that each could be used without the other, then the modules are decoupled. If the answer is that each could be used without the other provided that some substitute could be found to supply the services they provide each other, then they are loosely coupled. If the answer is that at least one could not be used without the other, then they are tightly coupled, and some way may need to be found to reduce their connection.

Although module coupling should be minimized, greater or lesser degrees of coupling will depend on the problem. For example, in a word processing program, modules implementing a paragraph type and a sentence type would inevitably be more strongly coupled than would modules implementing a word type and a printer interface. The former pair must rely on one another’s services for edits and formatting, while the latter have no intrinsic connection at all. Engineering judgment must be brought to bear in determining whether the strength of coupling between two modules is appropriate.

Cohesion	Cohesion was introduced along with coupling as a way to judge procedural design, but it has also proven important for object-oriented design.
----------	---

Cohesion is the degree to which a module’s parts are related to one another.

Cohesive modules hold data on a single topic or related topics, perform a single job, carry out a responsibility, or have a clear role or purpose. Cohesive modules have parts that belong together. A module all of whose parts are strongly related is said to be *cohesive*, or to have *high cohesion*; a module without strongly related parts is *non-cohesive* or has *low cohesion*.

As an illustration of cohesion, consider the **Holdings** class discussed previously to illustrate coupling. One way to implement this class is to have it call **DataDisplay** operations to update the display whenever investment data changes. To do this, the **Holdings** class must include code to manage the display, and it may also include data to help it keep track of the state of the display so that it can call the right **DataDisplay** operations at the right times. This design lacks cohesion because the **Holdings** class contains code and data unrelated to its main responsibility of maintaining data about investments. The alternative design in which the **Holdings** class contains no code or data about managing the display, but instead keeps track of objects it must notify when its data changes, is much more cohesive.

Besides making the module more understandable, cohesion is also beneficial because changes to highly cohesive modules are less likely to affect other modules. Everything that needs to be changed in a highly cohesive module should be present in the module, so other modules are unaffected. In other words, it is easier to hide information in a highly cohesive module. Thus, highly cohesive modules are easier to code, test, debug, maintain, explain, and document. This leads to the **Principle of Cohesion**.

Module cohesion should be maximized.

The Principle of Cohesion fits well with the Principle of Coupling: A highly cohesive module should be able to do most of its work on its own so communication with other modules is minimized, reducing coupling.

Cohesion was introduced with coupling to characterize good procedural program design. Various levels of cohesion were defined based on the functionality placed in a module. Low-cohesion modules included functionality by accident or because of temporal coincidence or logical similarity, while high-cohesion modules performed a single action or did a single operation on a data structure. As with coupling, we now apply the Principle of Cohesion to modules across the parts hierarchy and evaluate levels of cohesion based on more than sub-program functionality.

Module cohesion is determined by the following considerations:

- Cohesion is highest in modules that have a single clear, logically independent responsibility or role. Cohesion degrades as unrelated responsibilities or tasks are added to a module. For example, a module that interacts with a user, such as one that collects data in a form, has high cohesion if that is its only role. Its cohesion decreases if it carries out complex data validations and decreases drastically if it also processes the data or modifies data structures. These latter responsibilities cloud the central role of the module—getting data from a user—and should belong to other modules.
- One practical and long-standing practice for achieving cohesion is to form modules that implement data types. A **data type** has two elements: a set of values (the **carrier set** of the type) and a collection of operations for manipulating and examining values in the carrier set. Modules formed to implement data types store representations of values from the carrier set and realize the operations of the data type; they do nothing else. Such modules are highly cohesive.
- As discussed previously, a module is a software unit with parts that may themselves be modules. One way to increase cohesion is to build a module hierarchy reflecting the levels of abstraction in a program. We discuss this point in greater detail when we consider architectural design.
- If the module hierarchy reflects the levels of abstraction in a program, then cohesive modules should not have parts that cross levels of

abstraction. Crossing abstraction levels is confusing and often makes it harder to hide information. For example, suppose that a module implements a water heater device interface in a smart house control program. The operations in this module include turning the water heater on and off and setting its thermostat. Including operations in the module interface to control the heating elements within the water heater crosses the boundary to a lower level of abstraction, resulting in low module cohesion.

One of the great advantages of the object-oriented analysis and design approach is that it encourages designers to create units that mimic real-world entities. Real-world entities have appropriate responsibilities, so modeling them in software generally produces cohesive modules.

Summary of Modularity Principles

Figure 8-3-3 summarizes the modularity principles.

<p><i>Small Modules</i>—Designs with small modules are better. <i>Information Hiding</i>—Each module should shield the details of its internal processing from other modules. <i>Least Privilege</i>—Modules should not have access to unneeded resources. <i>Coupling</i>—Module coupling should be minimized. <i>Cohesion</i>—Module cohesion should be maximized.</p>
--

Figure 8-3-3 Modularity Principles

Section Summary

- A **modular** program has well-defined, conceptually simple, and independent units interacting through well-defined interfaces.
- A **module** is a software unit with parts.
- **Modularity principles** provide guidance in evaluating whether a program's modules are well formed.
- **Information hiding** is shielding the internal details of a module's structure and processing from other modules.
- **Coupling** is the degree of connection between pairs of modules.
- **Cohesion** is the degree to which a module's parts are related to one another.

Review Quiz 8.3

1. What are the immediate parts of a class?
2. How small is a small module?
3. Give three examples of things that should be hidden inside a module.
4. How are coupling and cohesion related?
5. What is a data type and how is it related to cohesion?

8.4 Implementability and Aesthetic Principles

Other Constructive Principles

The modularity principles discussed in the previous section offer guidance in deciding whether a program is well-modularized, but that is not all there is to making or evaluating a design. Implementability principles help designers determine whether a design is buildable, while aesthetic principles help designers determine whether a design is beautiful. We consider each category in turn.

Implementability Principles

As mentioned earlier, software development is expensive, risky, and time consuming. Designs that are easier to implement will usually be cheaper and faster to realize and more likely to work. **Implementability principles** thus help achieve design economy by providing criteria for judging whether a design can be built cheaply, quickly, and successfully.

There are three implementability principles: Simplicity, Designing with Reuse, and Designing for Reuse.

Simplicity

Simpler designs can usually be created, documented, coded, tested, and debugged faster than more complex designs, as well as being less error prone. Clearly, simplicity is a major contributor to design implementability, which leads to the **Principle of Simplicity**.

Simpler designs are better.

People drawn to software development tend to enjoy the challenge of solving complex problems. Unfortunately, fascination with complexity leads some designers to make their creations more complex than necessary.

Several legitimate concerns, such as efficiency and reusability, can also lead designers to make overly complex designs. Designers often choose more complex design alternatives for the sake of efficiency even when no performance problems are expected. To illustrate, suppose that an application must occasionally search some text. One design alternative is to build and maintain an index as the text grows, which entails problems of data structure and algorithm design and often requires complex updating strategies, but provides extremely fast searching. Another alternative is simply to search the entire text when the need arises. This is simple, but results in slower searching. In many cases, the small size of the text or the infrequency of text searching makes the simpler solution perfectly acceptable. An argument for the more complex solution on the grounds of efficiency is specious. The more complex solution is faster, but if performance is not a problem, then the simpler solution is best.

Designing with and for Reuse Reuse is a common way to get the most out of an individual's or an organization's resources. Reuse is important in software as well.

Software reuse is the use of existing artifacts to build new software products.

Items reused during software development may range from project plans through product requirements, design elements, code units, test plans, and test cases. Anything that can be reused is termed an **asset**.

Reuse has great productivity and quality benefits, and software design offers many opportunities for software reuse. There are two aspects of reuse that figure in software design:

- Designers can reuse existing assets. This is called *design with reuse*.
- Designers can create new work products intended for reuse. This is called *design for reuse*.

Design with reuse increases design implementability, leading to the **Principle of Design with Reuse**.

Designs that reuse existing assets are better.

Although quality and productivity gains come from designing with reuse, this can be done only if reusable assets are available. Hence, an investment must first be made to create reusable assets, leading to the corollary **Principle of Design for Reuse**.

Designs that produce reusable assets are better.

As a rule, making work products highly reusable is much more difficult and expensive than creating work products that only solve the current problem. On the other hand, designing with reuse saves a great deal of effort and expense and usually repays the costs of designing for reuse in the long run.

Aesthetic Principles Many scientists believe that one way to tell whether a scientific theory is true is by how beautiful it is; an ugly theory cannot be true. In many design disciplines, beauty is a main goal of design. Are aesthetic principles important in software design too? If so, **aesthetic principles** state that certain designs are better because they are more beautiful or pleasing to the mind or the senses.

Little work has been done to explore the nature of beauty in software design. The most extensive treatment of this topic is by David Gelernter, who argues that beauty is indeed a fundamental principle of software

design. He proposes that beauty in computing is a compound of simplicity and power. We have already mentioned simplicity as a design principle in its own right. According to Gelernter, simplicity is also a factor in aesthetics.

Gelernter defines *power* as the ability to accomplish a wide range of tasks or to get a lot done. For example, we rate the power of a computer in terms of its processing speed because faster computers can get more done than slower computers. Any design that satisfies requirements and conforms to design constraints gets done what needs to be done, so for a given problem, any adequate design is as powerful as any other. But a simpler design will be better. This is why power must be combined with simplicity in evaluating beauty, and why more beautiful software designs are superior to less beautiful ones.

Applying this understanding of beauty in software design, we would say, for example, that binary search is a beautiful algorithm because it is both very simple and very powerful, being able to search a list of size n in time proportional to $\log_2 n$.

Beauty	The most beautiful designs are those in which extremely simple mechanisms are able to accomplish a great deal. For example, quicksort is a simple algorithm that is among the fastest algorithms that sort by comparing and exchanging values: This makes it beautiful. The Principle of Beauty generalizes this observation.
---------------	--

Beautiful (simple and powerful) designs are better.

There is considerable question about whether beauty is only a combination of simplicity and power or whether some other factors are present. Some designs are simple and quite powerful yet seem awkward or crude. One might say this about balanced binary tree insertion and deletion algorithms, for example. Beauty has been an object of philosophical investigation for millennia, and the nature of beauty is still an open question. Undoubtedly Gelernter is right that power and simplicity have something to do with it, however, so we accept his formulation as a working definition.

Conflicts Between Principles

It is almost impossible to satisfy every design principle simultaneously. For example, simpler designs that are easy to implement often do not hide information and are less reusable. Product requirements and organizational goals usually dictate the relative importance of design principles. For instance, a product whose failure may result in loss of life or property must have adequacy as its dominant basic design principle, while an organization

working toward high levels of reuse may value modularity principles and design for reuse over simplicity.

No matter what priority is given to individual design principles, however, engineering judgment and accepted practice are the ultimate bases for resolving conflicts between them. For example, suppose that a module implementing a data type, such as a class, has a private non-local variable, such as an attribute, accessible to all the module's operations. The operations accessing this variable are coupled through it. Providing access operations for the variable and having all other operations in the module access the variable through these operations can lower coupling, though it is not as simple.

Although the Principle of Coupling supports the use of access operations, accepted practice is to do the simpler thing and access the variable directly rather than through access functions. Engineering judgment supports this practice: In a cohesive module implementing a data type (such as a well-designed class) the non-local variables and operations form a conceptually interconnected unit that must be understood, implemented, maintained, and reused as a whole. Thus, there is little to be gained by decoupling operations connected through access to a common non-local variable inside a module. These operations are conceptually connected in any case. Design principles are guides for engineering judgment, not arbiters of design decisions.

Summary of Principles

Figure 8-4-1 summarizes the implementability and aesthetic principles.

Implementability Principles

Simplicity—Simpler designs are better.

Design with Reuse—Designs that reuse existing assets are better.

Design for Reuse—Designs that produce reusable assets are better.

Aesthetic Principle

Beauty—Beautiful (simple and powerful) designs are better.

Figure 8-4-1 Implementability and Aesthetic Principles

Design Principles Taxonomy

The taxonomy of engineering design principles discussed in this chapter is summarized in Figure 8-4-2.

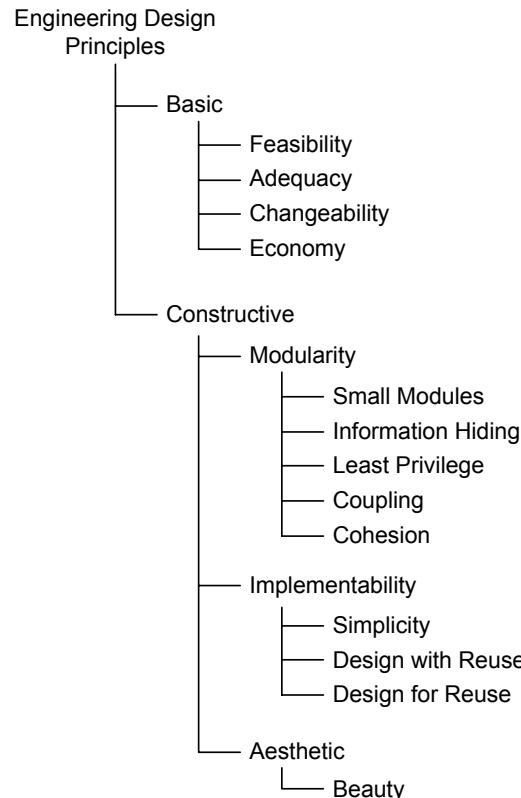


Figure 8-4-2 Taxonomy of Engineering Design Principles

Section Summary

- **Implementability principles** state that designs that can be realized more quickly, cheaply, and successfully are better.
- **Software reuse** is the use of existing artifacts to build new software products.
- **Aesthetic principles** state that designs that are more beautiful or pleasing to the mind or senses are better.

Review Quiz 8.4

1. How are design for reuse and design with reuse related?
2. What is beauty in software design?
3. Give an example, different from the text, of ways that design principles can be in conflict.

Chapter 8 Further Reading

Section 8.1

Good introductions to architectural design include [Shaw and Garlan 1996] and [Bass et al. 1998].

Section 8.3

Riel [1996] discusses heuristics for forming good object-oriented modules, including some based on the capacity of human short-term memory. Parnas

[1972] introduced information hiding. See [Stevens, Myers, and Constantine 1974] for the original discussion of coupling and cohesion.

Section 8.4 Gelernter [1998] discusses beauty in software design.

Chapter 8 Exercises

- Section 8.1**
1. Classify each of the following statements as an architectural (A) or detailed (D) design specification:
 - (a) The `find()` operation must use a binary search algorithm.
 - (b) The program must separate the user interface code from the rest of the code.
 - (c) The client must complete processing in no more than one second and the server must complete processing in no more than half a second.
 - (d) The `Register` class must maintain clients in a hash table.
 - (e) All utilities in the graphics library must be thread-safe.
 - (f) All `Car` objects must register themselves with the `Clock` object, and the `Clock` object will notify the `Car` objects when a second has passed.
 - (g) All observers of the `Clock` class must implement the `ClockUser` interface.
- Section 8.2**
2. What is the difference between a principle and a heuristic?
 3. Make a list of the modules and their immediate parts at each level of the parts hierarchy of a Java program.
 - (a) What sizes are considered acceptable for modules at each level of the parts hierarchy?
 - (b) How is information hidden in modules at each level?
 - (c) How is access to resources provided at each level?
- Section 8.3**
4. For each *modularity* design principle, discuss how the use of the principle affects a design in terms of the *basic* design principles.
 5. Consider the advantages and disadvantages of the four design alternatives for the `BulkStore` class mentioned in the discussion of information hiding. Choose one of the alternatives, and defend your choice.
 6. Each of the following heuristics helps make a good module according to some modularity principle. Identify the principle.
 - (a) Make all attributes of a class protected or private.
 - (b) A class should capture exactly one key abstraction.
 - (c) Make sure an operation needs all its parameters.
 - (d) Spin off unrelated data into another class.
 - (e) Minimize the number of classes with which a class collaborates.
 - (f) Most of the operations in a class should use most of the attributes most of the time.
 - (g) Model the real world whenever possible.
 - (h) Do not change the state of an object without going through its public interface.

- Section 8.4**
7. For each *implementability* and *aesthetic* design principle, discuss how the use of the principle affects a design in terms of the *basic* design principles.
 8. Each of the following heuristics helps make a good module according to some implementability or aesthetic principle. Identify the principle.
 - (a) Minimize the number of operations in a class.
 - (b) Eliminate irrelevant classes from your design.
 - (c) Do not nest control structures more than seven levels deep.
 - (d) Never make a class to do a job that a class in a standard library already does.
 - (e) Eliminate duplicated code whenever you find it.
 - (f) Provide get and set methods for all attributes in a class that clients might be interested in, even if they are not all used in the current program.
 - (g) Do not reuse variables to hold different data.
 - (h) Avoid operations with only a single line of code.
 - (i) Keep operation parameters to five or less.
 9. Suppose that a module containing several other modules, such as a package containing several classes, has a variable accessible throughout the module. Comment on the acceptability of this arrangement in light of design evaluation principles. Propose an alternative arrangement and compare these design alternatives.
 10. Make a table laying out the constructive principles discussed in this chapter by category. Name and state each principle in the table. For each principle, indicate which of the basic design principles it best helps achieve.
 11. Choose five pairs of constructive design principles from different categories and discuss whether each pair is mutually supportive or in conflict.
 12. Can you propose any other aesthetic principles? Among those you might consider are principles of symmetry, balance, and harmony.
 - Team Projects**
 13. Form a team of five to write an essay giving at least one example illustrating each modularity principle.
 - Research Projects**
 14. Find some code from a previous project and identify sub-program or operation modules. Analyze the coupling between pairs of modules and list whether each pair is strongly, loosely, or not coupled. How might the strongly coupled modules be made loosely coupled? Can the strength of loose coupling between any pairs be reduced?
 15. Find some code from a previous project and identify sub-program or operation modules. Analyze the cohesion of each module and characterize it as high or low. How might the non-cohesive modules be made cohesive? Can any module be made more cohesive by reassigning some of its data or processing to some other module?
 16. Find some code from a previous project and analyze it to find ways it might be changed to simplify it or to reuse some existing assets. What

effects do these changes have on the solution with respect to modularity principles?

17. Find some code from a previous project, or code from a book or the Internet, that you think is particularly ugly or beautiful. Why do you think it is ugly or beautiful? If the code is ugly, propose an alternative that you think is more beautiful. Is it a better design?

Chapter 8 Review Quiz Answers

Review Quiz 8.1

1. A black box is a device whose external behavior is observable but whose internal workings are hidden. During product design, the program is treated as a black box whose external behavior is specified in the design. In architectural design, the black box of the program is opened and its internal details are specified in terms of major constituents that are treated as black boxes. During detailed design, the remaining black boxes are opened and their internals are specified.
2. DeSCRIPTR-PAID stands for Decomposition, States, Collaborations, Responsibilities, Interfaces, Properties, Transitions, Relationships, and Packaging, Algorithms, Implementation, and Data structures and types. This acronym captures the sorts of documentation that might be needed in detailed design program unit specifications. DeSCRIPTR applies to architectural design specifications as well.

Review Quiz 8.2

1. Design principles are used to evaluate designs, either when comparing design alternatives or for deciding whether a design is adequate. Designers aware of design principles will use them in generating designs. There is no point in generating a design alternative that does not meet basic design principles because it will be unacceptable.
2. A basic design principle states an evaluative criterion based on the ability of a design to meet stakeholder needs, while a constructive design principle states an evaluative criterion based on engineering properties that have been shown over time to be characteristic of high-quality programs.
3. Changeability is an important design principle because so much of the cost of a software product over its lifetime (around 70%) occurs during the maintenance phase of the software life cycle. Programs that are easier to change will be considerably cheaper over the long run, which is an important concern for many stakeholders.

Review Quiz 8.3

1. The immediate parts of a class are its attributes and operations.
2. What counts as a small module depends on what kind of module it is. For example, a compilation unit should have no more than about 500 lines of code, and a class should have no more than about seven public operations.
3. Among the things that should be hidden inside a module are the following items: use of data and operations from other modules, internal organization, internal data representations, algorithms, volatile decisions, and non-portable code and data.
4. Coupling and cohesion were both introduced in the early 1970s as fundamental principles of structured design. Besides this historical relationship, the

principles of coupling and cohesion tend to be mutually reinforcing. Modules that are loosely coupled are often highly cohesive, and vice versa. Perhaps most telling, modules that violate either principle tend to violate the other: Tightly coupled modules tend to not be very cohesive, and modules that lack cohesion tend to be tightly coupled to one or more other modules.

5. A data type is a set of values (the carrier set of the type) and operations for manipulating and examining those values. Modules that implement data types tend to be highly cohesive.

**Review
Quiz 8.4**

1. Design with reuse is possible only if reusable assets are available, and good reusable assets are created only if they are designed for reuse. On the other hand, taking the time and effort to design for reuse pays off only if the reusable assets are actually used again, which happens only when programs are designed with reuse.
2. It has been suggested that beauty in software design is a combination of simplicity and power, where power is the ability to get a lot done. Thus an algorithm such as heapsort is beautiful because it is quite simple, but also very powerful (because it can sort an array of n elements in time proportional to $n \log_2 n$).
3. Suppose that you are writing a program with a graphical user interface that accepts several input values from users, computes results when a “Compute” button is pressed, and displays the results in text and graphics. Suppose that the GUI portion of the program is separated from the part of the program that computes the results from the input data, forming two architectural units, called the **GUI** and the **Calculator**. One way that the **GUI** and **Calculator** can interact when there are results to display is that the **Calculator** can call one or more **GUI** display operations. This is simple, but it couples the **Calculator** to the **GUI** through the display operations. Alternatively, the **GUI** can register with the **Calculator** as an agent interested in its result. When the **Calculator** has new results, it notifies all registered units. The **GUI** then queries the **Calculator** for its results and displays them. The second alternative is more complicated than the first, but it greatly reduces the coupling between the **GUI** and the **Calculator**; in fact, the **Calculator** now only needs to know that the **GUI** is a unit that it must notify when it has results. In this case the Principles of Simplicity and Coupling are in conflict.

8 Engineering Design Resolution

Chapter Objectives

Once the engineering design problem is understood, it must be solved. The diagram in Figure 8-O-1 depicts the place of engineering design resolution in the software engineering design process.

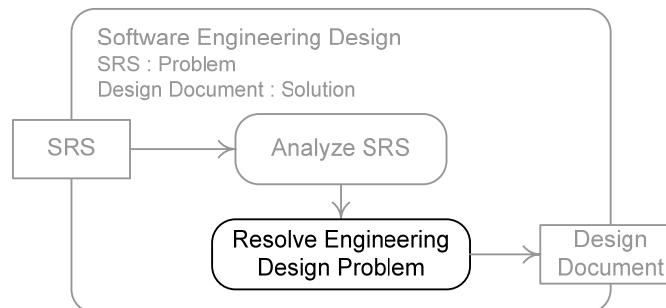


Figure 8-0-1 Software Engineering Design

This chapter introduces the main steps in the resolution activity and explores the principles that guide engineers in evaluating designs and making design decisions.

By the end of this chapter you will be able to

- Distinguish architectural and detailed design;
- List the items comprising architectural and detailed design specifications;
- State and explain basic engineering design Principles of Feasibility, Adequacy, Economy, and Changeability; and
- State and explain principles used to evaluate particular aspects of an engineering design, including the principles of Small Modules, Information Hiding, Least Privilege, Cohesion, Coupling, Reuse, Simplicity, and Beauty.

Chapter Contents

- 8.1 Engineering Design Resolution Activities
 - 8.2 Engineering Design Principles
 - 8.3 Modularity Principles
 - 8.4 Implementability and Aesthetic Principles
-

8.1 Engineering Design Resolution Activities

Architectural and Detailed Design

Engineering design resolution is traditionally divided into two major phases or activities: architectural design and detailed design. We characterize these activities by the following definitions.

Architectural design is the activity of specifying a program's major parts; their responsibilities, properties, and interfaces; and the relationships and interactions among them.

Detailed design is the activity of specifying the internal elements of all major program parts; their structure, relationships, and processing; and often their algorithms and data structures.

Engineering design resolution and its major activities can be understood in terms of *black boxes*, or units with external form and behavior but hidden internal workings. During product design, the entire program is treated as a black box, and only its external form and behavior are specified. During architectural design the program black box is “opened”—its high-level structure is determined, and its major constituents and their interactions are specified. However, the major program constituents are treated as black boxes during architectural design. In detailed design, each major constituent is opened and its internal structure and behavior are specified. Depending on the size of the program, there may be other layers of black boxes considered during detailed design. Ultimately, all black boxes are opened and the entire workings of the program are specified.

In the remainder of this section we consider what happens during architectural and detailed design in more depth. Later sections discuss the design principles governing engineering design resolution.

Architectural Design Specification

The central goal of architectural design is to determine the high-level structure of the program, or its **software architecture**. As we will see in the next chapter, there are both technical and organizational considerations that influence architectural design. For now, we consider the items that may appear in a software architecture specification:

Decomposition—Depending on the size of the product, the major parts into which a product is divided can range in scope from single classes to entire large collections or packages. For example, the AquaLush architecture divides the program into five layers, each one a module containing several program units (layered architectures are discussed in Chapter 15). This decomposition is pictured in Figure 8-1-1 on page 228.

Responsibilities—Each module must be placed in charge of certain data and activities. For example, the AquaLush **Startup** layer configures the product when it begins execution. The **User Interface** layer is in charge of tracking the user interface state and changing it in response to user and product actions. The **Irrigation** layer tracks the state of the valves and sensors, holds program parameters, and controls irrigation. The **Device Interface** layer provides a uniform interface to the hardware portion of the product, such as the sensors, valves, clock, keypad, and display. The **Simulation** layer implements virtual devices used in the AquaLush Web simulation; it is present only in the simulation.

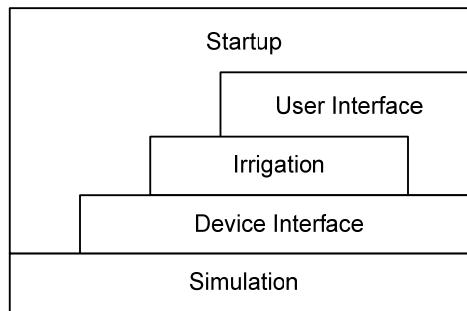


Figure 8-1-1 AquaLush Architectural Constituents

Interfaces—An **interface** is a boundary across which entities communicate. An **interface specification** is a description of the mechanisms that an entity uses to communicate with its environment. Somewhat confusingly, we often refer to an interface specification as simply an “interface.” Architectural constituents must have their interfaces specified so that they can be developed independently. For example, the interface specification for the AquaLush **Device Interface** layer says that it provides virtual valves with `open()` and `close()` operations that throw valve failure exceptions. Developers of the other parts of the system can then rely on the fact that they can communicate with virtual valves using these operations.

Collaborations—When units cooperate to achieve computational goals, they typically do so according to some protocol or course of interaction. For example, the AquaLush **Irrigation** layer must keep track of the current time, which is kept by a virtual clock in the **Device Interface** layer. One way the **Irrigation** layer can track the time is for it to continually poll the virtual clock. Another way is for the virtual clock to notify the **Irrigation** layer every minute; the **Irrigation** layer can then query the virtual clock for the current time.

Relationships—An architectural design may specify that parts communicate in certain ways or that there are dependencies or other relationships between them. For example, the modules or layers in the AquaLush architecture may use only modules directly below them in the layer diagram of Figure 8-1-1. The **Startup** layer may use any layer; the **User Interface** layer may use only the **Irrigation** and **Device Interface** layers; the **Irrigation** layer may use only the **Device Interface** layer; and the **Device Interface** layer may use only the **Simulation** layer (if it is present).

Properties—Many products have requirements about the time operations can take; the resources, such as processor time or memory, that can be consumed; the rate at which transactions must be completed; and their reliability. Products can satisfy their requirements only if their constituents have certain properties, which must be specified. For example, suppose that a product must perform a transaction in one second or less, and suppose that three of its sub-systems must cooperate

to process a transaction. Each sub-system must complete its part of transaction processing in a certain amount of time, such as a third of a second. The property of completing transaction processing in a third of a second or less is a property specification.

States and State Transitions—Units may have important states that affect their externally observable behavior. If so, these states and the transitions among them should be specified. For example, the Aqualush Irrigation layer has two main states: manual mode and automatic mode. It switches between these states based on input from the User Interface layer.

This list of kinds of specifications can be remembered using the acronym **DeSCRIPTR**, where *De* stands for *Decomposition* and the other letters stand for *States*, *Collaborations*, *Responsibilities*, *Interfaces*, *Properties*, *Transitions*, and *Relationships*. An architectural specification does not have to supply all this information for every constituent. There is no need to document obvious or trivial aspects of a design. For example, a unit may not have interesting or important states, so there is no need to document its states and state transitions. However, some information—in particular, about responsibilities and interfaces—are almost always important.

An architectural design must specify both the structure and the behavior of the program at a high level of abstraction. Static and dynamic models, which we discuss in later chapters, are often used for this purpose.

Detailed Design Specification

Detailed design fills in the design specifications left open after architectural design. The main goal of detailed design is to specify the program in sufficient detail so that programmers can implement it. Detailed design specification includes most of the following characteristics:

Decomposition—Each major program module must be decomposed into program units, such as compilation units, packages, classes, modules, and operations. For example, the Aqualush Irrigation module is decomposed into Irrigator, IrrigationCycle, AutoCycle, ManualCycle, Zone, Sensor, and Valve classes.

Responsibilities—As with architectural constituents, program units are also in charge of certain data and behaviors. For example, the Valve class is responsible for keeping data about valves, including their flow rate, location, status, and water allocation during automatic irrigation. It is also responsible for opening and closing valves and monitoring water usage during automatic irrigation.

Interfaces—Program unit specifications describe the public aspects of the unit that other units can use to interact with it.

Collaborations—Units collaborate to achieve computational goals by calling operations and passing data. These interactions must be specified.

Relationships—Inheritance relationships, class associations, program unit dependencies, interface realization relationships, and similar relationships need to be specified in a detailed design.

Properties—Program units may have to satisfy certain specifications to meet needs for reliability, efficiency, memory utilization, and so forth. If so, then these must be stated.

States and State Transitions—Some units, such as objects, have states that are important for understanding the way they work. Detailed design should specify interesting state-based program unit behavior.

Packaging and Implementation—Program units and their internal elements typically reside in various containers. The allocation of units to containers may be specified, along with the scope and visibility of program elements, though this may be left to the programmers to determine. For example, AquaLush is implemented in Java. The AquaLush detailed design specifies the Java packages in the program, the classes in each package, and the visibility of each class in each package.

Algorithms, Data Structures, and Types—Detailed design is the least abstract design activity, but it is still done at a level of abstraction higher than program code (although the boundary between detailed design and coding is fuzzy). A detailed design may specify a program down to the level of descriptions of individual data types and structures, loops, branches, and low-level string and arithmetic operations. Most often, however, such details are left to the coders, and detailed design stops just above the level of data structures and algorithms.

These specifications include all the DeSCRIPTR elements plus four more items: *Packaging*, *Algorithms*, *Implementation*, and *Data structures and types*, which can be remembered with the acronym **PAID**. As in the case of architectural design specifications, not every program unit in a detailed design must be specified with all the DeSCRIPTR-PAID information; only the interesting and non-trivial specifications that the designers do not want to leave for the coders must be included.

Section Summary

- **Architectural design** is the activity of specifying a program's decomposition into major parts and their states, collaborations, responsibilities, interfaces, properties, state transitions, and relationships (**DeSCRIPTR**).
- **Detailed design** is the activity of specifying the internal elements of all major program parts, possibly including their decomposition, states, collaborations, responsibilities, interfaces, properties, state transitions, relationships, packaging, algorithms, implementation, and data structures and types (**DeSCRIPTR-PAID**).

Review Quiz 8.1

1. Explain product design, architectural engineering design, and detailed engineering design in terms of black boxes.
2. What does the acronym DeSCRIPTR-PAID stand for?

8.2 Engineering Design Principles

Criteria for Design Evaluation

During engineering design resolution, design alternatives are generated and evaluated. Evaluation is made in light of several **design principles** or **tenets**, which are statements about characteristics that make designs better. Software engineering design shares many design principles with other design disciplines because of the common design goal of creating high-quality, long-lived products. For example, any design should meet stakeholder needs, be buildable, not cost too much, and so forth. We call principles stating characteristics that make a design better able to meet stakeholder needs and desires **basic design principles**.

Other design principles specify characteristics of programs that we know from experience are essential for achieving high-quality products. These principles are mostly unique to software engineering design because they are about the characteristics of well-constructed programs. For example, we have learned over the years that software is more robust and maintainable when it is built from fairly small parts that hide their internal processing details from each other. This peculiar feature of programs is of no interest to software product designers or users, but it is very important to software engineering designers and programmers. Because such principles are crucial for building high-quality programs, we call them **constructive design principles**.

In the next few sections we discuss the most important engineering design principles, beginning with basic design principles and moving to constructive design principles. These principles will be used and discussed extensively as we consider design problems in later chapters.

Basic Principles

The primary goal of software engineering design is to specify the structure and behavior of programs satisfying software product specifications. Good programs must have certain characteristics at delivery and must be maintainable over time. Specifically, at delivery a good program must satisfy its requirements and conform to its design constraints. To be deliverable, a program must be designed so that it can be built for an acceptable amount of time, for an acceptable amount of money, and with acceptable risk. For a program to be maintainable over time, it must be changeable. These considerations give rise to four basic design principles: Feasibility, Adequacy, Economy, and Changeability.

Feasibility The bottom line for any design is that it must specify something that can be built and will work, which leads to the **Principle of Feasibility**.

A design is acceptable only if it can be realized.

Adequacy Once designers determine that a design specifies something that can be built, the next concern is whether the result will do what it is supposed to do; that is, whether the design meets stakeholder needs and desires subject to constraints. It may not be possible to meet all stakeholder needs, to meet them all fully, or to satisfy all constraints. Some designs may meet more needs than others or satisfy more constraints than others. Hence, adequacy is a matter of degree. This leads to the following **Principle of Adequacy**.

Designs that meet more stakeholder needs and desires, subject to constraints, are better.

Economy Software development is expensive, time consuming, and risky. A good design should specify a program that can probably be built, tested, and deployed on time and within budget. This is the justification for the **Principle of Economy**.

Designs that can be built for less money, in less time, with less risk, are better.

The Principles of Adequacy and Economy may be in conflict: A designer may be unable to specify a program meeting stakeholder needs and satisfying constraints that can be built on time and within budget, with reasonable risk. The designer must then work with stakeholders to make trade-offs among needs, constraints, time, money, and risk.

Changeability Approximately 70% of lifetime software product costs are for maintenance. Designs that make a product easy to change will have a tremendous influence on cumulative life cycle costs. The **Principle of Changeability** acknowledges this fact.

Designs that make a program easier to change are better.

These basic design principles state fundamental criteria for evaluating the overall quality of a design. Unfortunately, they are not very helpful in determining whether particular aspects of a design are good or bad, or in deciding between design alternatives at low levels of abstraction. The principles discussed in the next two sections give more specific guidance in evaluating design details.

Summary of Basic Design Principles

Figure 8-2-1 summarizes the basic design principles.

Feasibility—A design is acceptable only if it can be realized.
Adequacy—Designs that meet more stakeholder needs and desires, subject to constraints, are better.
Economy—Designs that can be built for less money, in less time, with less risk, are better.
Changeability—Designs that make a program easier to change are better.

Figure 8-2-1 Basic Engineering Design Principles

Section Summary

- **Design principles** are evaluative criteria that state characteristics of good designs.
- **Basic design principles** state desirable design characteristics based on meeting stakeholder needs and desires.
- **Constructive design principles** state desirable engineering design characteristics based on past software development experience.

Review Quiz 8.2

1. For what purposes are design principles used?
2. What is the difference between a basic design principle and a constructive design principle?
3. Why is changeability an important basic design principle?

8.3 Modularity Principles

Constructive Principles

Constructive design principles provide evaluative criteria based on technical details of engineering designs that developers have found through experience to be characteristic of good designs. Designs that rate highly according to constructive design principles also tend to rate highly according to basic design principles.

Constructive design principles fall into the following categories:

Modularity Principles—We have learned over the years that high-quality programs are invariably constructed from self-contained, understandable parts, or *modules*, with well-defined interfaces. Several important constructive design principles state criteria for judging whether designs specify good modules.

Implementability Principles—A design may be more or less easy to build. Ease of construction affects both adequacy and economy.

Aesthetic Principles—Beauty is widely acknowledged as an important design principle, and aesthetic concerns clearly contribute to design quality.

We consider constructive design principles in the modularity category in this section and principles in the implementability and aesthetic categories in the next section.

Modularity Modularity is defined as follows.

A **modular** program is composed of well-defined, conceptually simple, and independent units that communicate through well-defined interfaces.

Modular programs have several clear advantages over non-modular programs:

- Modular programs are easier to understand and explain because their parts make sense and can stand on their own. It is easier to understand and explain all the details of a whole if its parts can be considered in isolation.
- Modular programs are easier to document because their parts make sense and because each part can be documented independently.
- Programming individual modules is easier because the programmer can focus on individual small, simple problems rather than a large, complex problem.
- Modular programs are easier to change because modifications are usually restricted to only a few parts, and a change to one part has little or no effect on other parts.
- Testing and debugging individual modules is easier because they can be dealt with in isolation. Fewer and simpler tests are needed than are needed for the entire program. Bugs are easier to locate and understand, and they can be fixed without fear of introducing problems outside the module.
- Well-composed modules are more reusable because they are more likely to comprise part of a solution to many problems. In addition, a good module should be easy to extract from one program and insert into another.
- Modules are easier to tune for better performance because all the relevant data and processing are available and segregated from the rest of the program.

Modularity is probably the single most important characteristic of a well-designed program. But what exactly is a module?

What Is a Module?

Programs are wholes with a hierarchy of parts. At the top level of decomposition, programs have several major, large parts. These are in turn composed of somewhat smaller parts, and those of even smaller parts, down to the level of individual statements. In different contexts, units at any non-leaf level of this hierarchy may be considered design modules.

A **module** is a program unit with parts.

We can, at each level of the parts hierarchy, specify what we regard as a module along with its immediate parts. The *immediate parts* of a whole are the parts directly below the whole in the parts hierarchy. For example, we can say that a car has a chassis and a drive train (two immediate parts); that the drive train has an engine, a transmission, and wheels (three immediate parts); that the chassis has a frame, a body, and an interior (three immediate parts); and so on.

An exact specification of what counts as an immediate part depends on where a module is in the parts hierarchy. At the highest level of abstraction, an entire program is a module whose immediate parts are the sub-programs of which it is composed. A program is a module whose parts are its architectural constituents. The immediate parts of a sub-program module might be classes, functions, packages, or compilation units. The immediate parts of a class module are attributes and operations. The immediate parts of a compilation unit are declarations or lines of code. The immediate parts of a sub-program are lines of code.

At a low-enough level of abstraction are parts that we no longer consider decomposable; we eventually come to atomic parts. Thus, a line of code is generally not considered a module because its pieces, the lexical elements of a programming language, are not considered parts with respect to design decomposition. The lowest level or atomic elements in design decomposition are individual lines of code, and therefore the bottom-level modules are blocks or sub-programs.

Modularity Principles

Forming good modules is essential in software design. Several considerations come into play in making design decisions about what should go into a module, how modules should be packaged, what portions of a module's contents should be accessible, and so forth.

Modularity principles are design principles that serve as guides and evaluative criteria in forming modules. We present five modularity principles.

Small Modules

The simplest rule of thumb in forming modules is to keep them small. Large modules are hard to understand, explain, document, write, test, debug, change, and reuse. As a result, our first modularity principle is the following **Principle of Small Modules**.

Designs with small modules are better.

One might ask, “What counts as a small module?” As noted previously, a module is a whole with immediate parts, so this question becomes, “How many immediate parts does a small module have?” The answer to this

question depends on where the module is in the parts hierarchy. For example, a convention in procedural programming is that sub-programs have no more than about 60 lines of code. (This rule apparently originated from the fact that there are 60 lines on two pages of open fan-folded line-printer paper.) Another is that compilation units have no more than about 500 lines of code. Modules under these limits are small.

Many other such rules are based on the capacity of human short-term memory, which is said to be seven, plus or minus two, items. Thus, some people recommend that classes have seven (plus or minus two) public operations or that functions be decomposed into no more than seven (plus or minus two) sub-functions.

All such heuristics about module size are reasonable but may be violated when the occasion demands. The ultimate goal is to keep a module, wherever it occurs in the parts hierarchy, small enough to be a conceptually simple, comprehensible unit. When a module gets large enough to run afoul of a size heuristic, that is a signal for the designer or implementer that the module may be too hard to understand. In such cases the designer should consider dividing the module into two or more pieces. Some modules that do not violate size restrictions but are difficult to understand anyway should also be divided to make them more comprehensible. The bottom line is that designers must rely on their experience, engineering judgment, and the recommendations of others in deciding whether a module is a conceptually simple, comprehensible unit.

Information Hiding

Information hiding is a practice of very long standing in software design; David Parnas first proposed it in 1972. Although Parnas proposed it as a guide in decomposition, information hiding has much wider application. We will begin by defining the term.

Information hiding is shielding the internal details of a module's structure and processing from other modules.

For example, most modern languages allow local variables to be declared in sub-programs, and these variables can be accessed only from within those sub-programs. This mechanism *hides* the local variables inside their sub-programs.

Information hiding is good for both the module that hides its internals and the rest of the program:

- It is good for the hiding module because modification of module internals from the outside is prohibited. This protects the module from damage by errant or malicious external code and makes module repair, enhancement, and reuse easier.
- It is good for the rest of the program because it enforces disciplined use of the module. A module can be used without understanding its internals, thus reducing complexity and making programming much

easier. The interface to the module is fixed and stable, making it easier to understand and use. It is also easy to replace. Perhaps most importantly, the module may be changed without breaking the rest of the program.

For these reasons, the following **Principle of Information Hiding** is of fundamental importance in software design.

Each module should shield the details of its internal structure and processing from other modules.

All details about how a module works should be hidden. Specific examples of information best hidden include the following items:

- Use of data and operations from other modules, such as the attributes, constants, and operations from other classes, and details of how they are used;
- Internal organization, such as collaborations with other objects or the control flow within an operation;
- Internal data representations, such as data types and data structures;
- Algorithms, such as choices of alternatives for sorting, searching, insertion, deletion, and so forth;
- Volatile (likely to change) design decisions, such as sizes, capacities, waiting times, and so forth; and
- Non-portable code and data, such as code specific to certain operating systems, and data such as file paths and URLs.

Every module must also have public information, comprising its interface to the rest of the program, that cannot be hidden. Examples of public information include the following items:

- Names and properties of data provided by a module to its clients, such as public attributes, constants, and types;
- Names, parameters, and return types (that is, the signature) of operations provided by a module to its clients;
- The behavior of operations provided by a module, in particular transformation of input data to output data, events that invoke the operation, any events that the operation produces, and any side effects that the operation may have;
- Module error and exception conditions and behavior; and
- The other modules with which a module interacts as a client.

As an illustration of the Principle of Information Hiding, suppose we have an application that must track the contents of several bulk storage units, such as silos, tanks, or bins. We can create a `BulkStore` class to model such storage units. Instances of this class are responsible for knowing how much they can hold (`totalCapacity`), how much they currently hold (`currentQuantity`), and how much more they can hold (`remainingCapacity`). There is an

invariant relationship between these three quantities captured by the equation `totalCapacity = currentQuantity + remainingCapacity`. As a result of this relationship, the `BulkStore` class can maintain attributes tracking any two or all three of these quantities, giving the four alternative designs illustrated in Figure 8-3-1.

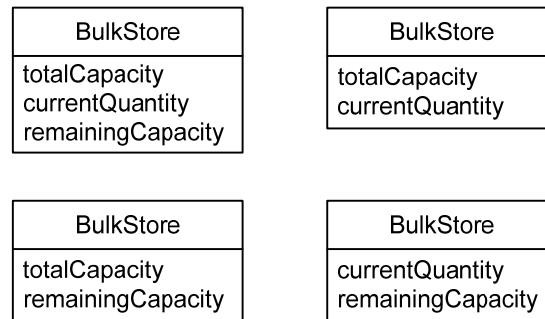


Figure 8-3-1 BulkStore Design Alternatives

Other parts of a program using the `BulkStore` class should not depend on which of these four alternatives the designers of the `BulkStore` class choose. Nor should the designers of the `BulkStore` class have to worry about whether other parts of the program assume one of these alternatives. This design decision should be hidden. This is done by making the attributes private and providing three public operations to query `BulkStore` instances about their `totalCapacity`, `currentQuantity`, and `remainingCapacity`. The `totalCapacity` is set at object creation, and the `currentQuantity` and `remainingCapacity` are altered by `add()` and `remove()` operations of the `BulkStore` class. The class interface is then fixed, and designers may make or change their decisions about which attributes are actually maintained by the class without fear of affecting the rest of the program.

Figure 8-3-2 shows the *public* parts of the `BulkStore` class. Note that these interface operations can be added to any of the previous four alternatives.

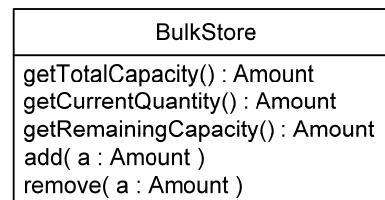


Figure 8-3-2 BulkStore Interface

- | | |
|-----------------|--|
| Least Privilege | Information hiding is about a module restricting access to the details of its internal processing. A complementary principle is that a module should not have access to resources that it does not need. This is called the Principle of Least Privilege and it is stated as follows. |
|-----------------|--|

Modules should not have access to unneeded resources.

A module with access to unneeded resources is harder to understand because it is not clear why the module has access to those resources. Such a module is less reusable because it (apparently) requires the presence of other (actually unneeded) resources. Perhaps most importantly, a module with access to unneeded resources can be the source of serious security problems.

The following examples illustrate violations of the Principle of Least Privilege:

- Modules that import packages or modules they do not need;
- Modules with unneeded access to files, programs, databases, or other computers;
- Classes with references to objects that are never accessed; and
- Operations with parameters for passing data they don't need.

Coupling	Coupling is a historically important concept originally proposed for procedural design that has proven important for object-oriented design as well.
----------	--

Coupling is the degree of connection between pairs of modules.

The strength of module coupling depends on module communication. Pairs of modules that communicate extensively or communicate in undesirable ways are *strongly coupled* or *tightly coupled*. Pairs that communicate only a little and in desirable ways are *weakly coupled*. Pairs that do not communicate at all are *not coupled* or *decoupled*.

To illustrate coupling, suppose that a program has two classes that work together to display financial data. The **Holdings** class maintains a list of investments, and the **DataDisplay** class makes charts and graphs about investments. One way that these classes can work together is for the **Holdings** class to respond to changes in its data or to requests for different displays by telling the **DataDisplay** class what to display. For example, suppose that the data in the **Holdings** class changes. The **Holdings** class might then have to call **DataDisplay** operations such as `displayInvestment()`, `displayTotalInvestments()`, or `graphInvestments()`.

The **Holdings** and **DataDisplay** classes are tightly coupled in this design: The **Holdings** class must be aware of what the **DataDisplay** class can display, and it must call the right operations at the right times to update the display.

An alternative communication strategy is for the **Holdings** class to notify the **DataDisplay** class whenever its data changes. The **DataDisplay** class is then responsible for querying the **Holdings** class to obtain the new data and for displaying it in the right way. Coupling is greatly reduced in this design.

alternative because the **Holdings** class does not need to know what the **DataDisplay** class does; the **Holdings** class only needs to know how to notify the **DataDisplay** class.

Strongly coupled modules are hard to change independently because programmers must worry about the data exchanged between them. Strongly coupled modules are also hard to understand, explain, and document because they cannot be considered independently of the modules to which they are coupled. Finally, strongly coupled modules are harder to test, debug, and maintain than weakly coupled modules because they cannot be treated in isolation.

Modularity is enhanced when coupling is minimized. Hence, we have the **Principle of Coupling**.

Module coupling should be minimized.

If two designs are equal in all other ways, then the one with the least coupling is better.

The notion of coupling was originally a way of characterizing good sub-program design. Different levels of sub-program coupling were defined, ranging from the tight coupling that results when one sub-program depends on the private details of another (such as accessing local variables or branching into another sub-program's code), to the loose coupling that results from passing data between sub-programs using parameters and function return values. We now realize that the Principle of Coupling has a much broader application than just to sub-program modules: Module coupling should be minimized at all levels of the parts hierarchy.

This breadth of application makes it difficult to characterize coupling levels precisely. Nevertheless, we can make a few observations about coupling strength:

- Failure to hide information leads to tight coupling and should always be avoided. Tight coupling always results when a module has access to details of another module's processing.
- Modules that communicate through a global entity such as a global variable, a file, or a global class are more tightly coupled than modules that communicate directly. Such modules are linked strongly to the global entity and hence to one another. Such communication should be avoided.
- Modules that communicate using special data types or data structures are more tightly coupled than modules that pass simple, standard data values. Relying on special data types or data structures links modules more tightly because they both must be changed if one of them needs to alter their common data format. Special data types or data structures should be avoided in module communication.

- When modules communicate only through public module interfaces, their coupling strength is proportional to the number of messages and the amount of data passed between them.

A simple heuristic for evaluating coupling strength between a pair of modules is to ask, “Could I use each of these modules in some other program without the other?” If the answer is that each could be used without the other, then the modules are decoupled. If the answer is that each could be used without the other provided that some substitute could be found to supply the services they provide each other, then they are loosely coupled. If the answer is that at least one could not be used without the other, then they are tightly coupled, and some way may need to be found to reduce their connection.

Although module coupling should be minimized, greater or lesser degrees of coupling will depend on the problem. For example, in a word processing program, modules implementing a paragraph type and a sentence type would inevitably be more strongly coupled than would modules implementing a word type and a printer interface. The former pair must rely on one another’s services for edits and formatting, while the latter have no intrinsic connection at all. Engineering judgment must be brought to bear in determining whether the strength of coupling between two modules is appropriate.

Cohesion	Cohesion was introduced along with coupling as a way to judge procedural design, but it has also proven important for object-oriented design.
----------	---

Cohesion is the degree to which a module’s parts are related to one another.

Cohesive modules hold data on a single topic or related topics, perform a single job, carry out a responsibility, or have a clear role or purpose. Cohesive modules have parts that belong together. A module all of whose parts are strongly related is said to be *cohesive*, or to have *high cohesion*; a module without strongly related parts is *non-cohesive* or has *low cohesion*.

As an illustration of cohesion, consider the **Holdings** class discussed previously to illustrate coupling. One way to implement this class is to have it call **DataDisplay** operations to update the display whenever investment data changes. To do this, the **Holdings** class must include code to manage the display, and it may also include data to help it keep track of the state of the display so that it can call the right **DataDisplay** operations at the right times. This design lacks cohesion because the **Holdings** class contains code and data unrelated to its main responsibility of maintaining data about investments. The alternative design in which the **Holdings** class contains no code or data about managing the display, but instead keeps track of objects it must notify when its data changes, is much more cohesive.

Besides making the module more understandable, cohesion is also beneficial because changes to highly cohesive modules are less likely to affect other modules. Everything that needs to be changed in a highly cohesive module should be present in the module, so other modules are unaffected. In other words, it is easier to hide information in a highly cohesive module. Thus, highly cohesive modules are easier to code, test, debug, maintain, explain, and document. This leads to the **Principle of Cohesion**.

Module cohesion should be maximized.

The Principle of Cohesion fits well with the Principle of Coupling: A highly cohesive module should be able to do most of its work on its own so communication with other modules is minimized, reducing coupling.

Cohesion was introduced with coupling to characterize good procedural program design. Various levels of cohesion were defined based on the functionality placed in a module. Low-cohesion modules included functionality by accident or because of temporal coincidence or logical similarity, while high-cohesion modules performed a single action or did a single operation on a data structure. As with coupling, we now apply the Principle of Cohesion to modules across the parts hierarchy and evaluate levels of cohesion based on more than sub-program functionality.

Module cohesion is determined by the following considerations:

- Cohesion is highest in modules that have a single clear, logically independent responsibility or role. Cohesion degrades as unrelated responsibilities or tasks are added to a module. For example, a module that interacts with a user, such as one that collects data in a form, has high cohesion if that is its only role. Its cohesion decreases if it carries out complex data validations and decreases drastically if it also processes the data or modifies data structures. These latter responsibilities cloud the central role of the module—getting data from a user—and should belong to other modules.
- One practical and long-standing practice for achieving cohesion is to form modules that implement data types. A **data type** has two elements: a set of values (the **carrier set** of the type) and a collection of operations for manipulating and examining values in the carrier set. Modules formed to implement data types store representations of values from the carrier set and realize the operations of the data type; they do nothing else. Such modules are highly cohesive.
- As discussed previously, a module is a software unit with parts that may themselves be modules. One way to increase cohesion is to build a module hierarchy reflecting the levels of abstraction in a program. We discuss this point in greater detail when we consider architectural design.
- If the module hierarchy reflects the levels of abstraction in a program, then cohesive modules should not have parts that cross levels of

abstraction. Crossing abstraction levels is confusing and often makes it harder to hide information. For example, suppose that a module implements a water heater device interface in a smart house control program. The operations in this module include turning the water heater on and off and setting its thermostat. Including operations in the module interface to control the heating elements within the water heater crosses the boundary to a lower level of abstraction, resulting in low module cohesion.

One of the great advantages of the object-oriented analysis and design approach is that it encourages designers to create units that mimic real-world entities. Real-world entities have appropriate responsibilities, so modeling them in software generally produces cohesive modules.

Summary of Modularity Principles

Figure 8-3-3 summarizes the modularity principles.

Small Modules—Designs with small modules are better.

Information Hiding—Each module should shield the details of its internal processing from other modules.

Least Privilege—Modules should not have access to unneeded resources.

Coupling—Module coupling should be minimized.

Cohesion—Module cohesion should be maximized.

Figure 8-3-3 Modularity Principles

Section Summary

- A **modular** program has well-defined, conceptually simple, and independent units interacting through well-defined interfaces.
- A **module** is a software unit with parts.
- **Modularity principles** provide guidance in evaluating whether a program's modules are well formed.
- **Information hiding** is shielding the internal details of a module's structure and processing from other modules.
- **Coupling** is the degree of connection between pairs of modules.
- **Cohesion** is the degree to which a module's parts are related to one another.

Review Quiz 8.3

1. What are the immediate parts of a class?
2. How small is a small module?
3. Give three examples of things that should be hidden inside a module.
4. How are coupling and cohesion related?
5. What is a data type and how is it related to cohesion?

8.4 Implementability and Aesthetic Principles

Other Constructive Principles

The modularity principles discussed in the previous section offer guidance in deciding whether a program is well-modularized, but that is not all there is to making or evaluating a design. Implementability principles help designers determine whether a design is buildable, while aesthetic principles help designers determine whether a design is beautiful. We consider each category in turn.

Implementability Principles

As mentioned earlier, software development is expensive, risky, and time consuming. Designs that are easier to implement will usually be cheaper and faster to realize and more likely to work. **Implementability principles** thus help achieve design economy by providing criteria for judging whether a design can be built cheaply, quickly, and successfully.

There are three implementability principles: Simplicity, Designing with Reuse, and Designing for Reuse.

Simplicity

Simpler designs can usually be created, documented, coded, tested, and debugged faster than more complex designs, as well as being less error prone. Clearly, simplicity is a major contributor to design implementability, which leads to the **Principle of Simplicity**.

Simpler designs are better.

People drawn to software development tend to enjoy the challenge of solving complex problems. Unfortunately, fascination with complexity leads some designers to make their creations more complex than necessary.

Several legitimate concerns, such as efficiency and reusability, can also lead designers to make overly complex designs. Designers often choose more complex design alternatives for the sake of efficiency even when no performance problems are expected. To illustrate, suppose that an application must occasionally search some text. One design alternative is to build and maintain an index as the text grows, which entails problems of data structure and algorithm design and often requires complex updating strategies, but provides extremely fast searching. Another alternative is simply to search the entire text when the need arises. This is simple, but results in slower searching. In many cases, the small size of the text or the infrequency of text searching makes the simpler solution perfectly acceptable. An argument for the more complex solution on the grounds of efficiency is specious. The more complex solution is faster, but if performance is not a problem, then the simpler solution is best.

Designing with and for Reuse Reuse is a common way to get the most out of an individual's or an organization's resources. Reuse is important in software as well.

Software reuse is the use of existing artifacts to build new software products.

Items reused during software development may range from project plans through product requirements, design elements, code units, test plans, and test cases. Anything that can be reused is termed an **asset**.

Reuse has great productivity and quality benefits, and software design offers many opportunities for software reuse. There are two aspects of reuse that figure in software design:

- Designers can reuse existing assets. This is called *design with reuse*.
- Designers can create new work products intended for reuse. This is called *design for reuse*.

Design with reuse increases design implementability, leading to the **Principle of Design with Reuse**.

Designs that reuse existing assets are better.

Although quality and productivity gains come from designing with reuse, this can be done only if reusable assets are available. Hence, an investment must first be made to create reusable assets, leading to the corollary **Principle of Design for Reuse**.

Designs that produce reusable assets are better.

As a rule, making work products highly reusable is much more difficult and expensive than creating work products that only solve the current problem. On the other hand, designing with reuse saves a great deal of effort and expense and usually repays the costs of designing for reuse in the long run.

Aesthetic Principles Many scientists believe that one way to tell whether a scientific theory is true is by how beautiful it is; an ugly theory cannot be true. In many design disciplines, beauty is a main goal of design. Are aesthetic principles important in software design too? If so, **aesthetic principles** state that certain designs are better because they are more beautiful or pleasing to the mind or the senses.

Little work has been done to explore the nature of beauty in software design. The most extensive treatment of this topic is by David Gelernter, who argues that beauty is indeed a fundamental principle of software

design. He proposes that beauty in computing is a compound of simplicity and power. We have already mentioned simplicity as a design principle in its own right. According to Gelernter, simplicity is also a factor in aesthetics.

Gelernter defines *power* as the ability to accomplish a wide range of tasks or to get a lot done. For example, we rate the power of a computer in terms of its processing speed because faster computers can get more done than slower computers. Any design that satisfies requirements and conforms to design constraints gets done what needs to be done, so for a given problem, any adequate design is as powerful as any other. But a simpler design will be better. This is why power must be combined with simplicity in evaluating beauty, and why more beautiful software designs are superior to less beautiful ones.

Applying this understanding of beauty in software design, we would say, for example, that binary search is a beautiful algorithm because it is both very simple and very powerful, being able to search a list of size n in time proportional to $\log_2 n$.

Beauty	The most beautiful designs are those in which extremely simple mechanisms are able to accomplish a great deal. For example, quicksort is a simple algorithm that is among the fastest algorithms that sort by comparing and exchanging values: This makes it beautiful. The Principle of Beauty generalizes this observation.
---------------	--

Beautiful (simple and powerful) designs are better.

There is considerable question about whether beauty is only a combination of simplicity and power or whether some other factors are present. Some designs are simple and quite powerful yet seem awkward or crude. One might say this about balanced binary tree insertion and deletion algorithms, for example. Beauty has been an object of philosophical investigation for millennia, and the nature of beauty is still an open question. Undoubtedly Gelernter is right that power and simplicity have something to do with it, however, so we accept his formulation as a working definition.

Conflicts Between Principles

It is almost impossible to satisfy every design principle simultaneously. For example, simpler designs that are easy to implement often do not hide information and are less reusable. Product requirements and organizational goals usually dictate the relative importance of design principles. For instance, a product whose failure may result in loss of life or property must have adequacy as its dominant basic design principle, while an organization

working toward high levels of reuse may value modularity principles and design for reuse over simplicity.

No matter what priority is given to individual design principles, however, engineering judgment and accepted practice are the ultimate bases for resolving conflicts between them. For example, suppose that a module implementing a data type, such as a class, has a private non-local variable, such as an attribute, accessible to all the module's operations. The operations accessing this variable are coupled through it. Providing access operations for the variable and having all other operations in the module access the variable through these operations can lower coupling, though it is not as simple.

Although the Principle of Coupling supports the use of access operations, accepted practice is to do the simpler thing and access the variable directly rather than through access functions. Engineering judgment supports this practice: In a cohesive module implementing a data type (such as a well-designed class) the non-local variables and operations form a conceptually interconnected unit that must be understood, implemented, maintained, and reused as a whole. Thus, there is little to be gained by decoupling operations connected through access to a common non-local variable inside a module. These operations are conceptually connected in any case. Design principles are guides for engineering judgment, not arbiters of design decisions.

Summary of Principles

Figure 8-4-1 summarizes the implementability and aesthetic principles.

Implementability Principles

Simplicity—Simpler designs are better.

Design with Reuse—Designs that reuse existing assets are better.

Design for Reuse—Designs that produce reusable assets are better.

Aesthetic Principle

Beauty—Beautiful (simple and powerful) designs are better.

Figure 8-4-1 Implementability and Aesthetic Principles

Design Principles Taxonomy

The taxonomy of engineering design principles discussed in this chapter is summarized in Figure 8-4-2.

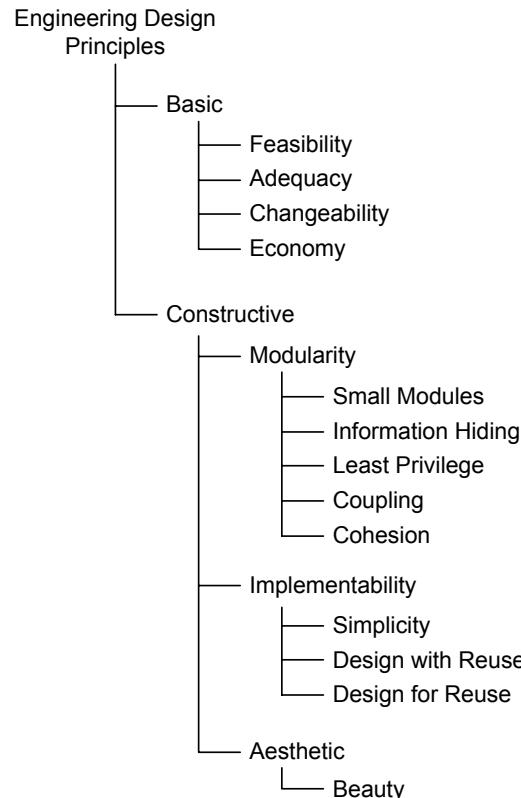


Figure 8-4-2 Taxonomy of Engineering Design Principles

Section Summary

- **Implementability principles** state that designs that can be realized more quickly, cheaply, and successfully are better.
- **Software reuse** is the use of existing artifacts to build new software products.
- **Aesthetic principles** state that designs that are more beautiful or pleasing to the mind or senses are better.

Review Quiz 8.4

1. How are design for reuse and design with reuse related?
2. What is beauty in software design?
3. Give an example, different from the text, of ways that design principles can be in conflict.

Chapter 8 Further Reading

Section 8.1

Good introductions to architectural design include [Shaw and Garlan 1996] and [Bass et al. 1998].

Section 8.3

Riel [1996] discusses heuristics for forming good object-oriented modules, including some based on the capacity of human short-term memory. Parnas

[1972] introduced information hiding. See [Stevens, Myers, and Constantine 1974] for the original discussion of coupling and cohesion.

Section 8.4 Gelernter [1998] discusses beauty in software design.

Chapter 8 Exercises

- Section 8.1**
1. Classify each of the following statements as an architectural (A) or detailed (D) design specification:
 - (a) The `find()` operation must use a binary search algorithm.
 - (b) The program must separate the user interface code from the rest of the code.
 - (c) The client must complete processing in no more than one second and the server must complete processing in no more than half a second.
 - (d) The `Register` class must maintain clients in a hash table.
 - (e) All utilities in the graphics library must be thread-safe.
 - (f) All `Car` objects must register themselves with the `Clock` object, and the `Clock` object will notify the `Car` objects when a second has passed.
 - (g) All observers of the `Clock` class must implement the `ClockUser` interface.
- Section 8.2**
2. What is the difference between a principle and a heuristic?
 3. Make a list of the modules and their immediate parts at each level of the parts hierarchy of a Java program.
 - (a) What sizes are considered acceptable for modules at each level of the parts hierarchy?
 - (b) How is information hidden in modules at each level?
 - (c) How is access to resources provided at each level?
- Section 8.3**
4. For each *modularity* design principle, discuss how the use of the principle affects a design in terms of the *basic* design principles.
 5. Consider the advantages and disadvantages of the four design alternatives for the `BulkStore` class mentioned in the discussion of information hiding. Choose one of the alternatives, and defend your choice.
 6. Each of the following heuristics helps make a good module according to some modularity principle. Identify the principle.
 - (a) Make all attributes of a class protected or private.
 - (b) A class should capture exactly one key abstraction.
 - (c) Make sure an operation needs all its parameters.
 - (d) Spin off unrelated data into another class.
 - (e) Minimize the number of classes with which a class collaborates.
 - (f) Most of the operations in a class should use most of the attributes most of the time.
 - (g) Model the real world whenever possible.
 - (h) Do not change the state of an object without going through its public interface.

- Section 8.4**
7. For each *implementability* and *aesthetic* design principle, discuss how the use of the principle affects a design in terms of the *basic* design principles.
 8. Each of the following heuristics helps make a good module according to some implementability or aesthetic principle. Identify the principle.
 - (a) Minimize the number of operations in a class.
 - (b) Eliminate irrelevant classes from your design.
 - (c) Do not nest control structures more than seven levels deep.
 - (d) Never make a class to do a job that a class in a standard library already does.
 - (e) Eliminate duplicated code whenever you find it.
 - (f) Provide get and set methods for all attributes in a class that clients might be interested in, even if they are not all used in the current program.
 - (g) Do not reuse variables to hold different data.
 - (h) Avoid operations with only a single line of code.
 - (i) Keep operation parameters to five or less.
 9. Suppose that a module containing several other modules, such as a package containing several classes, has a variable accessible throughout the module. Comment on the acceptability of this arrangement in light of design evaluation principles. Propose an alternative arrangement and compare these design alternatives.
 10. Make a table laying out the constructive principles discussed in this chapter by category. Name and state each principle in the table. For each principle, indicate which of the basic design principles it best helps achieve.
 11. Choose five pairs of constructive design principles from different categories and discuss whether each pair is mutually supportive or in conflict.
 12. Can you propose any other aesthetic principles? Among those you might consider are principles of symmetry, balance, and harmony.
 - Team Projects**
 13. Form a team of five to write an essay giving at least one example illustrating each modularity principle.
 - Research Projects**
 14. Find some code from a previous project and identify sub-program or operation modules. Analyze the coupling between pairs of modules and list whether each pair is strongly, loosely, or not coupled. How might the strongly coupled modules be made loosely coupled? Can the strength of loose coupling between any pairs be reduced?
 15. Find some code from a previous project and identify sub-program or operation modules. Analyze the cohesion of each module and characterize it as high or low. How might the non-cohesive modules be made cohesive? Can any module be made more cohesive by reassigning some of its data or processing to some other module?
 16. Find some code from a previous project and analyze it to find ways it might be changed to simplify it or to reuse some existing assets. What

effects do these changes have on the solution with respect to modularity principles?

17. Find some code from a previous project, or code from a book or the Internet, that you think is particularly ugly or beautiful. Why do you think it is ugly or beautiful? If the code is ugly, propose an alternative that you think is more beautiful. Is it a better design?

Chapter 8 Review Quiz Answers

Review Quiz 8.1

1. A black box is a device whose external behavior is observable but whose internal workings are hidden. During product design, the program is treated as a black box whose external behavior is specified in the design. In architectural design, the black box of the program is opened and its internal details are specified in terms of major constituents that are treated as black boxes. During detailed design, the remaining black boxes are opened and their internals are specified.
2. DeSCRIPTR-PAID stands for Decomposition, States, Collaborations, Responsibilities, Interfaces, Properties, Transitions, Relationships, and Packaging, Algorithms, Implementation, and Data structures and types. This acronym captures the sorts of documentation that might be needed in detailed design program unit specifications. DeSCRIPTR applies to architectural design specifications as well.

Review Quiz 8.2

1. Design principles are used to evaluate designs, either when comparing design alternatives or for deciding whether a design is adequate. Designers aware of design principles will use them in generating designs. There is no point in generating a design alternative that does not meet basic design principles because it will be unacceptable.
2. A basic design principle states an evaluative criterion based on the ability of a design to meet stakeholder needs, while a constructive design principle states an evaluative criterion based on engineering properties that have been shown over time to be characteristic of high-quality programs.
3. Changeability is an important design principle because so much of the cost of a software product over its lifetime (around 70%) occurs during the maintenance phase of the software life cycle. Programs that are easier to change will be considerably cheaper over the long run, which is an important concern for many stakeholders.

Review Quiz 8.3

1. The immediate parts of a class are its attributes and operations.
2. What counts as a small module depends on what kind of module it is. For example, a compilation unit should have no more than about 500 lines of code, and a class should have no more than about seven public operations.
3. Among the things that should be hidden inside a module are the following items: use of data and operations from other modules, internal organization, internal data representations, algorithms, volatile decisions, and non-portable code and data.
4. Coupling and cohesion were both introduced in the early 1970s as fundamental principles of structured design. Besides this historical relationship, the

principles of coupling and cohesion tend to be mutually reinforcing. Modules that are loosely coupled are often highly cohesive, and vice versa. Perhaps most telling, modules that violate either principle tend to violate the other: Tightly coupled modules tend to not be very cohesive, and modules that lack cohesion tend to be tightly coupled to one or more other modules.

5. A data type is a set of values (the carrier set of the type) and operations for manipulating and examining those values. Modules that implement data types tend to be highly cohesive.

**Review
Quiz 8.4**

1. Design with reuse is possible only if reusable assets are available, and good reusable assets are created only if they are designed for reuse. On the other hand, taking the time and effort to design for reuse pays off only if the reusable assets are actually used again, which happens only when programs are designed with reuse.
2. It has been suggested that beauty in software design is a combination of simplicity and power, where power is the ability to get a lot done. Thus an algorithm such as heapsort is beautiful because it is quite simple, but also very powerful (because it can sort an array of n elements in time proportional to $n \log_2 n$).
3. Suppose that you are writing a program with a graphical user interface that accepts several input values from users, computes results when a “Compute” button is pressed, and displays the results in text and graphics. Suppose that the GUI portion of the program is separated from the part of the program that computes the results from the input data, forming two architectural units, called the **GUI** and the **Calculator**. One way that the **GUI** and **Calculator** can interact when there are results to display is that the **Calculator** can call one or more **GUI** display operations. This is simple, but it couples the **Calculator** to the **GUI** through the display operations. Alternatively, the **GUI** can register with the **Calculator** as an agent interested in its result. When the **Calculator** has new results, it notifies all registered units. The **GUI** then queries the **Calculator** for its results and displays them. The second alternative is more complicated than the first, but it greatly reduces the coupling between the **GUI** and the **Calculator**; in fact, the **Calculator** now only needs to know that the **GUI** is a unit that it must notify when it has results. In this case the Principles of Simplicity and Coupling are in conflict.

9 Architectural Design

Chapter Objectives

This is the first of two chapters surveying architectural design. This chapter introduces the topic and discusses architectural design activities and notations. The place of architectural design in the software engineering design resolution process is shown in Figure 9-O-1.

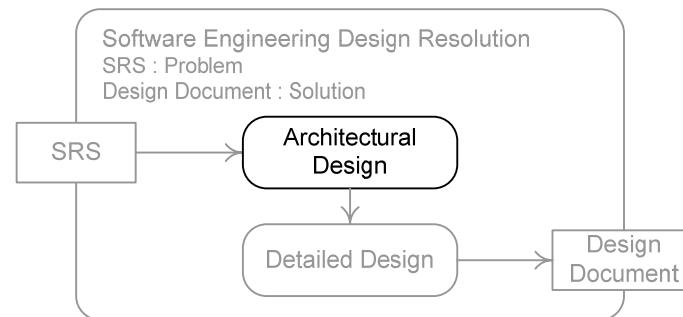


Figure 9-O-1 Software Engineering Design Resolution

By the end of this chapter you will be able to

- State the role of software architecture in the engineering design process and explain the architectural design process;
- List and explain the contents of a software architecture design template;
- Explain what quality attributes are and why they are important in architectural design;
- Specify module interface syntax, semantics, and pragmatics;
- Read and write architectural models using box-and-line diagrams;
- Read and write UML diagrams that include notes, constraints, properties, stereotypes, and dependency relations;
- Read and write design models using UML package and component diagrams; and
- Read and write UML deployment diagrams depicting physical architecture.

Chapter Contents

- 9.1 Introduction to Architectural Design
- 9.2 Specifying Software Architectures
- 9.3 UML Package and Component Diagrams
- 9.4 UML Deployment Diagrams

9.1 Introduction to Architectural Design

The Context of Architectural Design

Engineering design resolution has two phases: architectural design and detailed design. Architectural design is a problem-solving activity whose input is the product description in an SRS and whose output is the abstract specification of a program realizing the desired product. Architectural design thus sits between software product design and detailed design in the software design process.

But in fact, the architectural design activity is not as cleanly separated from product design and detailed design as the previous paragraph suggests. Some architectural design occurs during product design for the following reasons:

- Product designers must judge the feasibility of their designs, which may be difficult without some initial engineering design work.
- Stakeholders must be convinced that their needs will be met, which may be difficult without demonstrating how the engineers plan to build the product.
- Designers and stakeholders must trade off requirements to create a feasible product that can be built on schedule and within budget. Trade-offs may not be clear without exploring alternative software architectures.
- Project planners must have some idea about what software must be built to create schedules and allocate resources.

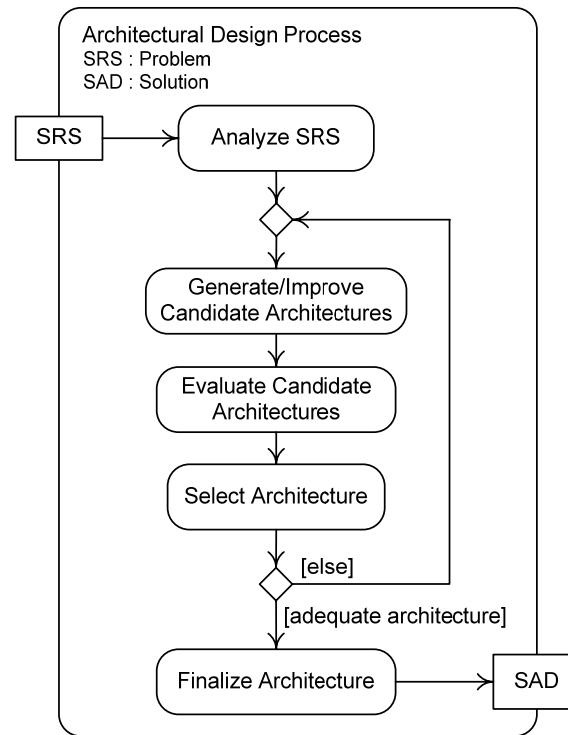
Consequently, engineering design work often begins during product design, proceeds in parallel with it, and influences product design decisions.

The boundary between architectural and detailed design is even less clear. In Chapter 8, we noted that a software architecture specifies a program's major constituents, their responsibilities and properties, and the relationships and interactions among them. Detailed design refines the architecture by specifying the internal details of the major program constituents and fleshing out the details of their properties, relationships, and interactions. This picture is vague, and, in particular, we may wonder:

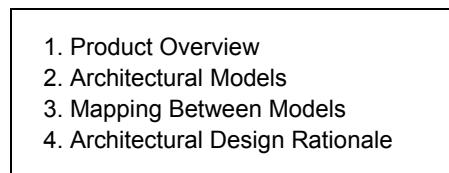
- What comprises a “major” constituent?
- How abstract should architectural specifications be?

There are no definitive answers to either of these questions. Some architects lean toward quite abstract specifications of a few top-level constituents, while others insist on detailed constituent specifications through several layers of abstraction.

Problem Context	<p>The circumstances of the design problem also dictate different amounts of detail in a software architecture. In a very small program consisting of only a handful of classes or modules interacting in simple ways, the software architecture is hardly distinguishable from the detailed design, so it is appropriate for the architecture to be simple and quite abstract. However, a very large and complicated system with hundreds or thousands of classes, distributed over many machines, interacting with many peripheral devices, and with demanding non-functional requirements, demands a detailed high-level specification that is carefully worked out and analyzed. Experience and engineering judgment must dictate the level of detail provided in an architectural specification.</p>
Organizational Context	<p>The organization in which architectural design takes place influences the architectural design process. An organization has investments, such as code libraries, standards and guidelines, software tools, and people with particular skills, that software architects are expected to make the most of in their designs. Organizations also have structures that may influence designers. For example, an organization may have groups that specialize in developing certain kinds of software, such as user interfaces, database systems, middleware, and networks. Architects may have to design programs whose constituents can be farmed out to existing development groups. Finally, the skills, experience, and preferences of architects themselves obviously influence the designs they produce.</p> <p>Incidentally, software architectures have a reciprocating influence on the organizations that create and use them. An organization may make investments in tools, technologies, methods, and people needed to build a program according to a software architecture. Groups may be formed to implement an architecture's major parts or sub-systems. Architects learn and grow as they solve new problems and learn what works and what does not.</p>
The Architectural Design Process	<p>The architectural design process is a straightforward application of the generic design process to the problem of architectural design. The activity diagram in Figure 9-1-1 reproduces and slightly elaborates part of the engineering design process activity diagram from Chapter 2.</p> <p>As indicated in the diagram, the input to this process is a software requirements specification (SRS), and its output is a software architecture document. A software architecture document (SAD) is simply a document that specifies the architecture of a software system. We consider the contents of the SAD next.</p>

**Figure 9-1-1 Architectural Design Process****SAD Contents**

The contents of a SAD vary depending on the program being designed and the needs of the development team. A small system with a simple architecture and a good SRS may need only a few architectural design models and minimal supporting material, while a complex system may require much more. The template shown in Figure 9-1-2 is appropriate for documenting the software architectures of small- to medium-sized systems.

**Figure 9-1-2 Software Architecture Document Template**

The template has the following sections:

Product Overview—This section either summarizes the product vision, stakeholders, target market, assumptions, constraints, and business

requirements or refers readers to the project mission statement. It is present because readers would have difficulty understanding the software architecture without knowing anything about the target product.

Architectural Models—This section presents the architecture, using a variety of models to represent different aspects or views. It typically uses the design notations discussed later in this chapter.

Mapping Between Models—Sometimes it is difficult to connect different architectural models. This section uses tables and textual explanations to help readers see these connections.

Architectural Design Rationale—This section explains the main design decisions made in arriving at the architecture. Architects have the time and energy to discuss only a few of the many decisions made during architectural design in the SAD. In deciding which decisions to discuss, architects should select those that took a lot of time and effort to make, are crucial for fulfilling important requirements, are puzzling at first glance, or will be hard to change later. Architects should explain the factors affecting each decision, the design alternatives considered, evaluations of the design alternatives, and the reasoning behind the final choice.

The section on architectural models must contain enough information that the models can be understood and used as a basis of detailed design and eventual implementation. In particular, it is important to include appropriate DeSCRIPTR information, as discussed in Chapter 8. A graphic, table, or short textual description generally does not supply all the needed information and must be supplemented by other specifications, usually written in English. Additional information may help readers understand the design. In particular, it may help to include a design rationale listing design alternatives and explaining why the architects chose the alternative they did.

Appendix B contains the AquaLush SAD. It uses the template shown in Figure 9-1-2 and illustrates the information contained in each section.

Quality Attributes

Software architecture is crucial not only for satisfying a product's functional requirements, but also for satisfying its non-functional requirements. Recall from Chapter 5 that non-functional requirements specify properties or characteristics that a software product must have. These are called quality attributes.

A **quality attribute** is a characteristic or property of a software product independent of its function that is important in satisfying stakeholder needs and desires.

Quality attributes fall into two categories: *development attributes* and *operational attributes*. Development attributes include properties important to development organization stakeholders, such as the following characteristics:

Maintainability—A product's **maintainability** is the ease with which it can be corrected, improved, or ported. Sometimes more specific kinds of maintainability attributes are used, such as modifiability or portability.

Reusability—A product's **reusability** is the degree to which its parts can be used in another software product. A product designed for reuse will have higher reusability.

Operational quality attributes include the following properties:

Performance—A program's **performance** is its ability to accomplish its function within limits of time or computational resources. Programs often must respond to external events within a certain time or must do their jobs using small amounts of memory or processor time.

Availability—A program's **availability** is its readiness for use. A Web server, for example, may need to be available for all but a few minutes a day.

Reliability—A program's **reliability** is its ability to behave in accord with its requirements under normal operating conditions. Any program that handles money or can endanger humans must have high reliability.

Security—A program's **security** is its ability to resist being harmed or causing harm by hostile acts or influences.

Programs can have a wide variety of architectural structures and still satisfy a product's functional requirements, but various architectural structures make it easier or harder to satisfy non-functional requirements.

Furthermore, architectures that increase the ability of a program to satisfy some non-functional requirements may decrease its ability to satisfy others. Software architects must consider alternative structures that enable a program to satisfy its functional requirements and select those that allow it to best satisfy its non-functional requirements.

To illustrate, consider a program responsible for matching fingerprints read from scanners against a database to allow people into and out of a secure facility. Besides its functional requirements, this program has some obvious non-functional requirements. For example, it must respond quickly, it must be available the entire time people are entering or leaving the facility, it must match fingerprints fairly reliably, and it must resist attackers.

Many software architectures can meet the functional requirements of such a program, but some will be better able to meet the non-functional requirements. Higher availability can be achieved by having redundant program units, such as a backup database, but this is more complicated, which may decrease reliability. On the other hand, redundant databases may speed processing so this architectural alternative may improve performance. Fingerprint matching can be made more or less reliable

depending on the algorithm used, but the more reliable algorithms may take more time, which affects performance. The program may be made more secure by adding units for data encryption and decryption, but this makes it more complicated (decreasing reliability) and slower (decreasing performance).

Architectural Design Challenges

Software architects must record their designs somehow to assist their thinking, share their ideas with others, evaluate their designs, and document them. Architectural constituents are large and abstract, and several kinds of models and notations are needed to represent software architectures fully. We consider how to specify software architectures in the next section of this chapter.

Architectural design, like all design, makes demands on the creativity of designers. Where do ideas for architectures come from? How can architectures be improved? We consider some answers to these questions in Chapter 10.

Software architects must somehow assess the ability of a software architecture to meet its requirements, even though the architecture is an abstract specification and there is no software to run. This is one of the most challenging aspects of architectural design, but there are techniques for evaluating architectures. We survey them in Chapter 10 as well.

A product's architecture should be validated before moving on to detailed design. The last section of Chapter 10 discusses how to validate software architectures with reviews and inspections.

Section Summary

- Architectural design may begin during product design, and it can provide information to product designers and stakeholders that is important for making product design decisions.
- There is no clear boundary between architectural and detailed design.
- There are no accepted standards for the level of abstraction of architectural specifications.
- A **quality attribute** is a software product property independent of a program's function that is important for satisfying stakeholder needs and desires.
- Quality attributes are specified in non-functional requirements.
- Software architects must specify architectural structures that meet both functional and non-functional requirements, paying special attention to the way that alternative structures affect quality attributes.

Review Quiz 9.1

1. How do development organizations affect architectural design, and how do software architectures affect development organizations?
2. What information should be included in a software architecture document?
3. What is the difference between operational and development quality attributes?
4. Name and describe three quality attributes.

9.2 Specifying Software Architectures

Notational Variety

Software architectures are specified using notations that describe static program structure and dynamic program behavior: Both static and dynamic design models are needed in architectural design.

A wide variety of notations can be used to represent software architectures, including several described elsewhere in this book. In particular

- UML activity diagrams, discussed in Chapter 2, can be used to describe the processes that units follow as they interact.
- UML use case diagrams and use case descriptions, presented in Chapter 6, can be used to describe the interactions between architectural constituents.
- UML class diagrams can describe the static architectural structure of small programs. These are discussed in Chapters 7 and 11.
- UML interaction diagrams can describe the communication between architectural constituents. These are discussed in Chapter 12.
- UML state diagrams can describe the states and state changes of major program units. State diagrams are covered in Chapter 13.

In this section we discuss various textual notations for architectural specifications. We also present box-and-line diagrams, a design notation especially useful for describing software architectures. In the next section we discuss notations that can be used in any UML diagram, along with UML package, component, and deployment diagrams, which are useful for both architectural and detailed design modeling.

Table 9-2-1 catalogs how the various notations mentioned previously are used for DeSCRIPTR aspects of architectural specification.

Type of Specification	Useful Notations
Decomposition	Box-and-line diagrams, class diagrams, package diagrams, component diagrams, deployment diagrams
States	State diagrams
Collaborations	Sequence and communication diagrams, activity diagrams, box-and-line diagrams, use case models
Responsibilities	Text, box-and-line diagrams, class diagrams
Interfaces	Text, class diagrams
Properties	Text
Transitions	State diagrams
Relationships	Box-and-line diagrams, component diagrams, class diagrams, deployment diagrams, text

Table 9-2-1 Notations for Architectural Specifications

Textual Specifications

Text is particularly important for specifying responsibilities, interfaces, properties, and relationships. We consider each in turn.

Specifying Responsibilities and Relationships

A unit's responsibilities are usually indicated in part by its name and in part by the symbols used to represent it in various diagrams. Similarly, unit relationships are often indicated by various connectors and by the names of the relationships. Names and symbols provide only part of the information about what a unit is supposed to do and how it is related to other units, though; supplementary information is usually needed.

There is no standard format for specifying responsibilities or relationships. As in all technical writing, responsibility and relationship specifications should be spelled out in simple, clear, and precise sentences. Grammar and spelling should be correct, and the material should be formatted to aid the reader. The AquaLush SAD in Appendix B contains many examples of textual responsibility specifications.

Specifying Interfaces

As mentioned in Chapter 8, an **interface** is a communication boundary between entities, and an **interface specification** describes the mechanism that an entity uses to communicate with its environment. Interface specifications, as descriptions of a means of communication, include specification of three characteristics:

Syntax—The **syntax** of a communications medium specifies the elements of the medium and the ways they may be combined to form legitimate messages. For example, the syntax of a programming language describes the lexical elements in the language (keywords, operators, and so forth), along with rules dictating how these may be combined to form legitimate programs.

Semantics—The **semantics** of a communications medium specify the meanings of messages. For example, the semantics of Java specify that the meaning of the statement `x = x+4` is that the variable `x` has its value increased by four.

Pragmatics—The **pragmatics** of a communications medium specify how messages are used in context to accomplish certain tasks. For example, Java `Collection` objects can store only reference values, not values of primitive types. Thus, for example, `int` values cannot be stored in an `ArrayList`. However, values of primitive types can be wrapped in objects and then stored in `Collections`. For example, an `int` value can be placed in an `Integer` object that can be stored in an `ArrayList`. This is part of the pragmatics of Java `Collections`.

An interface specification should cover the syntax, semantics, and pragmatics of the communications between a module and its environment. Both the messages that can be sent to the module and the messages that the module sends need to be specified.

Sometimes the pragmatics of several messages are best explained together because the module expects a certain **protocol**—that is, a certain ordering

or timing of the messages sent to it. Even when a module does not have a protocol, examples of how to use it are helpful additions to an interface specification. Finally, an interface specification can include an explanation of why the interface is designed as it is.

These considerations lead to the template shown in Figure 9-2-2 for specifying module interfaces.

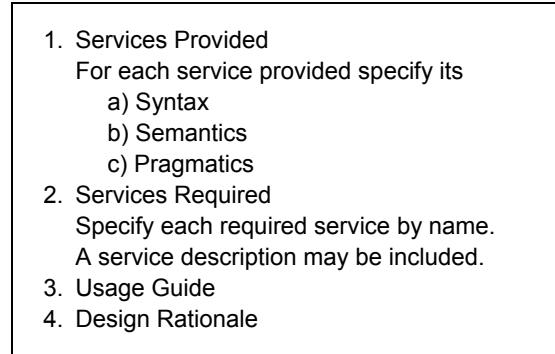


Figure 9-2-2 Interface Specification Template

There are many ways to specify syntax, semantics, and pragmatics. Syntax may be specified very abstractly by simply stating the names, inputs, and outputs of messages. UML class diagram operation specifications provide a more detailed notation. Programming languages provide a very detailed notation for syntactic specification.

One popular method for specifying semantics and pragmatics is to use preconditions and postconditions. A **precondition** is an assertion that must be true at the initiation of an activity or operation, while a **postcondition** must be true at the completion of an activity or operation. Together, pre- and postconditions show how the state of a computation changes when an operation executes, which explains the operation's semantics. Preconditions also state the context in which an operation can be used, which is part of its pragmatics. Postconditions should also state what happens when an operation is used even though its preconditions are violated. This is part of the operation's semantics.

AquaLush Interface Specification Example

Let us consider part of the AquaLush SAD to illustrate textual interface specification. AquaLush has a major constituent called the Device Interface layer. This constituent contains virtual devices that implement interfaces to physical devices (or simulations of them), such as valves, sensors, the parts of the control panel, and so forth. Every virtual device has an interface to which all device drivers of that sort must conform. For example, the virtual valve device has an interface to which all valve device drivers must conform. The advantages of having a device interface layer are discussed in Chapter 10. For now, we focus on one virtual device and its interface: the clock device.

The AquaLush virtual `ClockDevice` must keep track of the day of the week and the time of the day. A component, such as a master clock, that needs the current day or time of day can query the `ClockDevice` to obtain this information. The day of the week and the time of the day can also be set. A component that wants to be notified by the `ClockDevice` that time has passed can register itself as the device's `TickListener`. The `ClockDevice` sends messages to its `TickListener` every minute to notify it that time has passed. The `ClockDevice` needs to be told that time has passed by another entity that really keeps track of the time: a real (or simulated) clock of some sort. A real clock object must notify the `ClockDevice` that time has passed at least every minute. It also needs to be able to determine the current actual day and time, but we assume that this feature is supported by the programming language so we do not include it as an interface requirement.

This rather vague English description is made more precise in the interface specification for a `ClockDevice` shown in Figure 9-2-3 on page 264.

Specifying Properties

Non-functional requirements specify properties or quality attributes for a program as a whole that are then propagated to its parts. It is often difficult to specify quality attributes precisely. For example, maintainability is very hard to specify precisely—stating that a program or its parts must be “easy to change” says almost nothing. Specifications can be made more precise by introducing a target metric. For example, an SRS might specify that new editor features can be added with no more than one person-month of effort on average. This is much better, but how can such a program property be propagated to individual parts during design?

One way to make property specification more precise and easier to work with is to characterize them with a collection of scenarios. In Chapter 6, a **scenario** was defined as a specific interaction between a product and particular individuals that instantiates a use case. Use cases are episodes of interaction between a program and its actors. Scenarios for quality attributes are specific interactions between a program and any entity, including its developers and maintainers. Collections of scenarios typify the interactions relevant to a particular quality attribute. Studying these scenarios helps specify properties and can be the basis for architectural evaluation, which is discussed in the next chapter.

Returning to the maintenance example, the requirements that new editor features can be added with no more than one person-month of effort can be elaborated by creating a set of scenarios (typically three to five) of specific editor features that stakeholders believe are likely to be requested. Architects can consider how each scenario would be implemented to generate sets of modification scenarios for architectural constituents. These sets of scenarios can then be used to specify constituent properties. We will consider scenarios in greater detail in Chapter 10.

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the precondition is violated.

Set the time of the day	Syntax:	<code>setTime(milTime : int)</code>
	Pre:	milTime is a legitimate military time specification.
	Post:	The clock device is reset to milTime.
Get the time of the day	Syntax:	<code>getTime() : int</code>
	Pre:	None.
	Post:	The current time is returned, accurate to the minute, in military time format.
Set the day of the week	Syntax:	<code>setDay(d : Day)</code>
	Pre:	None.
	Post:	The clock device is reset to day d.
Get the day of the week	Syntax:	<code>getDay() : Day</code>
	Pre:	None.
	Post:	The current day of the week is returned.
Set the ClockDevice listener	Syntax:	<code>setListener(I : TickListener)</code>
	Pre:	None.
	Post:	TickListener I will start to receive notifications of the passage of time every minute. Any previous TickListener is replaced.

Services Required

This layer requires a `Day` enumeration type. The `ClockDevice` requires some sort of real or simulated device that notifies it when one minute has passed.

Usage Guide

The `Clock`, not the `ClockDevice`, is the main time and time notification service provider. The `Clock` uses the `ClockDevice` for time notification. To use a `ClockDevice`:

1. Create a new `ClockDevice` object. It should be unique.
2. Register the `ClockDevice` with the simulated hardware or software entity that supplies it with time notifications.

The day of the week and the time of the day may be adjusted at any time.

Design Rationale

AquaLush requirements are satisfied by a clock that keeps track of the day of the week and the time of the day, accurate to one minute. Hence the `ClockDevice` has only these features.

The real clock time increment does not matter as long as it is no greater than one minute.

Figure 9-2-3 AquaLush ClockDevice Interface

Box-and-Line Diagrams

Perhaps the most widely used architectural design notation has no precise specification and indeed hardly qualifies as a notation at all. It consists of various symbols or icons, called *boxes*, usually connected with various sorts of *lines*. Such graphics are called **box-and-line diagrams**. Figure 9-2-4 is an example.

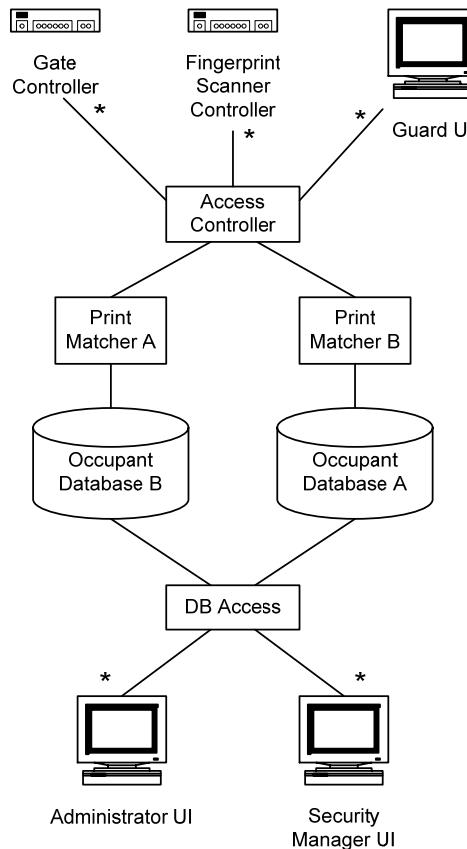


Figure 9-2-4 A Box-and-Line Diagram

This diagram is a static model of the Fingerprint Access System (FAS), a program that helps maintain physical security by controlling and monitoring access to a building using fingerprint scans. The program compares fingerprint scans to those recorded in a database and opens access gates for people whose fingerprints match. A guard can also control the gates. Administrators maintain the database, and security managers can query the database to determine who is in the building. The program's data store is replicated to increase availability and performance.

The boxes in this diagram all refer to software or data store components. The lines indicate interaction relationships between components. The asterisks at the ends of some lines indicate that several instances of such

components may be involved in the interaction. For example, several **Gate Controllers** may interact with the **Access Controller**. The box shapes classify the components as user interfaces, device controllers, internal components, or data stores.

Because box-and-line diagrams have no conventions about the meanings of symbols, it is a very good idea to include a legend with every box-and-line diagram. A legend for the FAS diagram in Figure 9-2-4 appears in Figure 9-2-5.

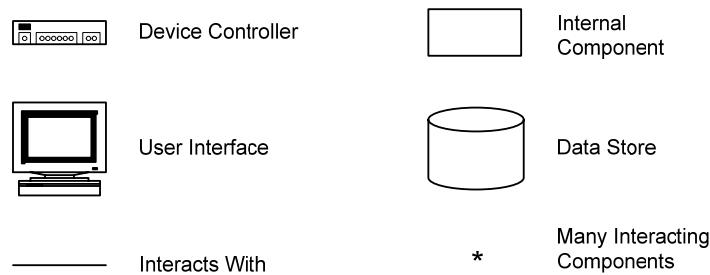


Figure 9-2-5 A Box-and-Line Diagram Legend

Even if a diagram uses a standard notation, marking the notation type on the diagram in place of the legend helps avoid confusion.

Sometimes physical relationships between boxes represent relationships in a box-and-line diagram. For example, it is common to represent layered modules using adjacency. The diagram in Figure 9-2-6 illustrates this case.

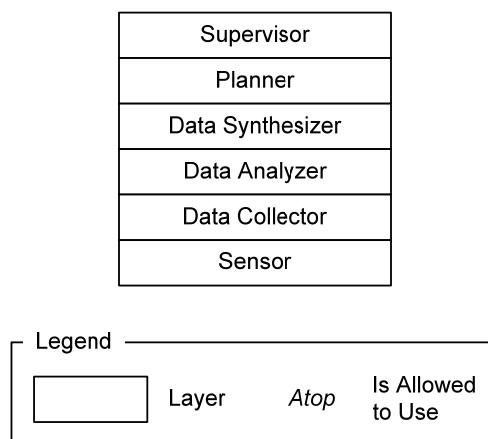


Figure 9-2-6 A Box-and-Line Diagram Showing Layers

Layers are allowed to interact only with the layers below them, so this diagram models the program quite clearly.

Box-and-Line Diagram Uses	As box-and-line diagrams are so loosely specified, they can be used for both static and dynamic modeling, and they can represent any aspect of a program that the architect desires. In practice, they tend to be used mostly for static models that show decomposition into major constituents or subsystems, along with interaction relationships.
Box-and-Line Diagram Heuristics	<p>We have already mentioned the advisability of including a legend explaining the symbols in a box-and-line diagram. In addition, the following heuristics can help make better box-and-line diagrams:</p> <p><i>Make box-and-line diagrams only when no standard notation can meet modeling needs.</i> There is no need to invent notations when good standard notations already exist. On the other hand, few standard notations are intended for architectural modeling, though they can be used for this purpose. A notation not meant to model architectures but used for that purpose is often not as good as a box-and-line drawing. For example, UML packages (discussed later in this chapter) can be used to represent architectural layers, but this notation takes longer to draw and is not as easy to read as the stacked boxes in a diagram such as Figure 9-2-6. Consequently, architectural layers are most often, and appropriately, shown using box-and-line diagrams.</p> <p><i>Keep the boxes and lines simple.</i> People sometimes get carried away with fancy graphics, but often there is no need to use many fancy symbols in box-and-line diagram. Keeping things simple usually makes the diagram easier to create, read, change, and reproduce, often with no loss of descriptive power.</p> <p><i>Make symbols for different things look different.</i> This fundamental rule of technical communication is especially relevant for box-and-line drawings because the architect is responsible for the notation as well as the design. Diagrams without distinct symbols are harder to read and more likely to be misinterpreted. For example, suppose two different kinds of relationships (such as an interaction relationship and a decomposition relationship) are represented by a solid line and a slightly thicker solid line. Readers might not notice the difference between the lines, and they might have a hard time recognizing which is which even if they do. It would be better to use very different line styles (such as a solid line and a dashed line), to add special symbols at the ends of the lines (such as arrowheads at the ends of interaction lines only), or to follow UML practice and put stereotypes on symbols to distinguish them.</p> <p><i>Use symbols consistently in different diagrams.</i> Even though each box-and-line diagram can have its own legend explaining its symbols, both the modelers and the model readers will have a hard time keeping symbols straight in different diagrams if they vary too much. By the same token, it is good practice to adopt notational conventions from standard diagrams for box-and-line diagrams. For example, an architect might always use solid lines with labels for relationships on static models, following UML practice for associations.</p>

Use grammatical conventions to name elements. Noun phrases name things, and verb phrases name actions or activities. Diagram elements that represent things should be named with noun phrases, and elements that represent actions, relationships, or interactions should be named with verb phrases.

Don't mix static and dynamic elements. A common mistake when making box-and-line diagrams is to add a few dynamic elements (such as data or control flows) to a static model. This usually messes up the model.

Decide before making the model whether it will be a static or dynamic model, and add only elements in accord with this decision.

Heuristics Summary

Figures 9-2-7 and 9-2-8 summarize the heuristics discussed in this section.

- Write good technical prose when specifying architectures.
- Use a template to specify interfaces.
- Specify the syntax, semantics, and pragmatics of interfaces.
- Use preconditions and postconditions to specify semantics and pragmatics.
- Elaborate quality attributes with scenarios.

Figure 9-2-7 Architectural Specification Heuristics

- Make box-and-line diagrams only when no standard notation can meet modeling needs.
- Keep the boxes and lines simple.
- Make symbols for different things look different.
- Use symbols consistently in different diagrams.
- Include a legend explaining the symbols in the diagram.
- Use grammatical conventions to name elements.
- Don't mix static and dynamic elements.

Figure 9-2-8 Box-and-Line Diagram Heuristics

Section Summary

- Several different kinds of notations are used to specify software architectures.
- Text is often used to specify responsibilities and relationships, interfaces, and properties.
- **Interface specifications** should state the **syntax, semantics, and pragmatics** of services provided and required by a unit.
- **Scenarios** can be used to help articulate component properties.
- **Box-and-line diagrams** use symbols or icons (*boxes*) connected by various kinds of *lines* to model software.

**Review
Quiz 9.2**

1. What diagrams can be used to model collaborations between program parts?
 2. Give an example from ordinary English to illustrate the difference between syntax, semantics, and pragmatics.
 3. Give an example from everyday life of a protocol governing communication between two people.
 4. What are preconditions and postconditions?
 5. What sorts of symbols can appear in box-and-line diagrams?
-

9.3 UML Package and Component Diagrams

**UML Common
Notations**

Several UML notations can be used in any UML diagram. We discuss them here because they are often found in UML models of software architectures and because some of them are needed to discuss UML package and component diagrams.

Notes A **note** is a dog-eared box connected to any model element by a dashed line. A note may contain arbitrary text. Notes usually contain comments, but they may also contain constraints.

Extension Mechanisms UML provides several mechanisms for extending the notation. These extension mechanisms allow symbols to take on special meanings or give UML the ability to model things that its basic notation does not support. The main ways the extend UML are the following mechanisms:

Constraints—A **constraint** is a statement that must be true of the entities designated by one or more model elements; in other words, it is a condition or restriction on the target of the model. Constraints are written inside curly brackets in a format not specified by UML. Constraints may appear near a name or model element. Constraints that apply to two model elements may be placed next to a dashed line connecting the elements; constraints applying to more than two elements may be placed in a note connected by dashed lines to all the elements involved.

Properties—A **property** is a characteristic of the entity designated by a model element. Properties are specified in comma-separated lists of tagged values enclosed in curly brackets. A *tagged value* is a name-value pair connected by an equals sign. For example, the property `{version=2.2, synchronized=true}` associates the value 2.2 with the tag `version` and the value true with the tag `synchronized`. If a tag has the Boolean value true then the value and the equals sign can be omitted. For example, the previous property can be abbreviated as `{version=2.2, synchronized}`. Properties can appear next to the elements that they describe or be connected to them with dashed lines.

Stereotypes—A **stereotype** is a UML model element given more specific meaning. Special icons, colors, or other graphic features can represent stereotyped elements, but placing a stereotype keyword before or above the model element name is the usual way to make one. *Stereotype keywords* are words placed between balanced guillemots, which look like double angle brackets but are distinct symbols. For example, «uses» is a stereotype keyword.

Figure 9-3-1 contains an example of a note, a constraint, a property, and a stereotype.

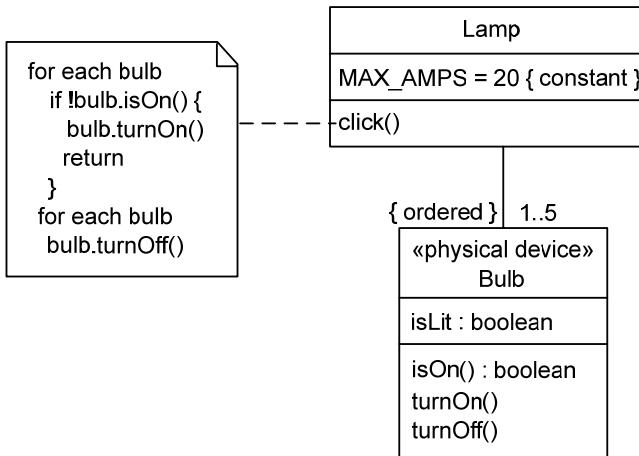


Figure 9-3-1 Stereotypes, Notes, Properties, and Constraints

The **Bulb** class is stereotyped as a «physical device», which means (let us suppose) that it is a special kind of class that controls a physical device. The **MAX_AMPS** attribute in the **Lamp** class has the property of being a constant. The **{ ordered }** constraint specifies that the collection of **Bulb** instances held by a **Lamp** instance must be ordered. The note contains pseudocode that specifies the **click()** operation's behavior.

Dependencies A dependency is a kind of binary relation that holds between two things, defined as follows.

A **dependency relation** holds between two entities *D* and *I* when a change in *I* (the *independent entity*) may affect *D* (the *dependent entity*).

For example, suppose class *D* calls one or more operations of class *I*. A change to *I* may affect *D*, so *D* depends on *I*. There are many kinds of dependency relations. The following instances are common examples:

- Module *D* *uses* module *I* when a correct version of *I* must be present for *D* to work correctly.

- Module *D* depends for compilation on module *I* (in other words, *D* cannot be compiled without *I*).
- Class *D* imports elements from package *I*.

UML represents dependency relations with a *dependency arrow*. The dependency arrow points from the dependent to the independent entity. Dependency arrows may be stereotyped to indicate the precise nature of the dependency relation. The diagram in Figure 9-3-2 shows several dependencies between classes.

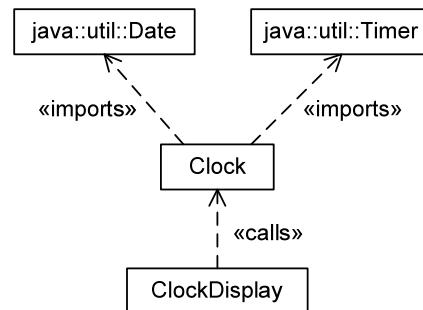


Figure 9-3-2 Some Class Dependencies

Dependencies represent links between individual model elements, not relations between sets of instances, so they do not have association adornments.

Packages A UML **package** is simply a collection of model elements, called *package members*. The UML package symbol is a tabbed rectangle, or a file folder symbol. Figure 9-3-3 illustrates the package symbol.

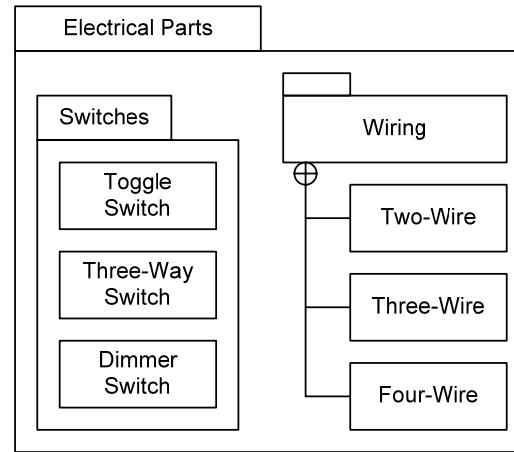


Figure 9-3-3 A Package Example

The members of a package can be shown either inside the rectangle or using a special containment symbol (a circled plus sign) attached to the containing package, with lines running to the members. The package name appears in the tab if the rectangle is occupied or in the rectangle if it is empty. Package symbols can be connected by dependency arrows to show that one package imports or exports members to another.

Package Diagrams

A UML **package diagram** is one whose primary symbols are package symbols. A package diagram may show groupings of use cases, classes, components, or nodes (discussed later in this chapter). A package diagram may also consist of only package symbols.

Modeling Architectures with Package Diagrams

UML package diagrams are useful for making static models of modules, their parts, and their relationships. UML packages are thus useful for architectural modeling. For example, Figure 9-3-4 shows the layers from Figure 9-2-6.

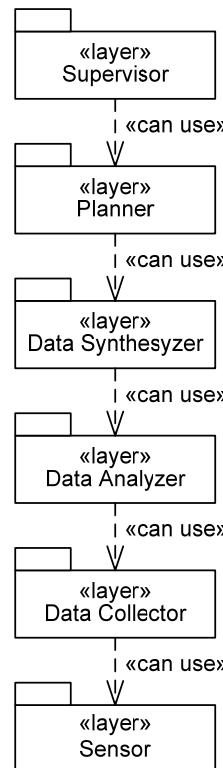


Figure 9-3-4 Layers Represented by Packages

The «layer» stereotype emphasizes that each package is an architectural layer. The *can use* dependency relation indicates that each layer is allowed to use the services of the layer beneath it.

UML packages are not as good as box-and-line diagrams for simply showing architectural layers, but they are very handy when the contents of the layer are shown as well.

Software Components

Reusable parts that can be bought from suppliers and included in software products are an important software development resource. For example, the Java class libraries are collections of such reusable parts, and database management systems, XML parsers, and transaction managers are examples of larger reusable parts. Another advantage of using such parts is that they can be replaced by comparable parts with different properties as product needs change. Developers may design programs with replaceable parts to increase the changeability of their designs.

Such reusable and replaceable parts are called **software components**. Products can be designed with them in mind and built in whole or in part with commercially available or custom-built software components, an approach called **component-based development**. UML reflects the importance of software components by including symbols and diagrams for modeling them.

UML Components and Component Diagrams

A UML **component** is a modular, replaceable unit with well-defined interfaces. Components, like packages, may include classes, but unlike packages they hide their internals to make them as replaceable and reusable as possible.

Components are represented by *component symbols* that resemble class symbols, except that they must either be stereotyped «component» or have a special component icon in their upper right-hand corners. Component names appear inside the component symbol. Figure 9-3-5 illustrates component symbols.



Figure 9-3-5 Component Symbols

A UML **component diagram** shows components, their relationships to their environment, and possibly their internal structure.

UML Interfaces

Interfaces are very important in component diagrams because they define the relationship between a component and its environment. In UML an **interface** is a named collection of public attributes and abstract operations. An **abstract operation** is an unimplemented operation. An abstract operation has a signature indicating its name, parameters, and return values, but no specification of the computation that it carries out when called.

Interfaces are like classes in that they have operations and attributes, but they are unlike classes in that they cannot be instantiated; instead, they must be realized by classes or components. A class or component **realizes** an interface when it includes the interface's attributes and implements its operations.

UML has two kinds of symbols for representing interfaces: One uses a rectangle with compartments for specifying interface details, and the other is an abbreviated form that shows only the interface name. We present the abbreviated form here and cover the other form in Chapter 11.

UML distinguishes two kinds of interfaces depending on an interface's relationship to a class or component:

Provided Interfaces—Interfaces realized by a class or component are **provided interfaces**.

Required Interfaces—Interfaces on which a class or component depends are **required interfaces**.

Each kind of interface is indicated by its own symbol in UML. Provided interfaces are represented by an unfilled circle connected to the class or component providing the interface by a solid line. This is a *ball* or *lollipop* symbol. Required interfaces are represented by half circles connected to the requiring class or component by a solid line. This is the *socket* symbol. Interface names are written beside the circle or half-circle. To illustrate, consider Figure 9-3-6.

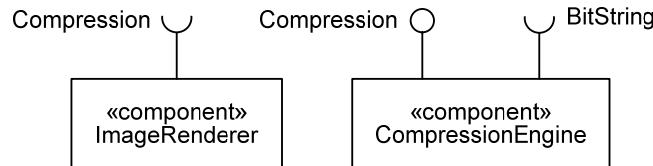


Figure 9-3-6 Provided and Required Interfaces

In this example, the **ImageRenderer** component displays images on a screen. The **CompressionEngine** converts between bit strings and various compression formats, such as ZIP, GIF, or JPG. The **CompressionEngine** provides a **Compression** interface containing operations such as `compressToJPG()` and `decompressJPG()`. The **ImageRenderer** requires this interface to handle compressed images. Furthermore, the **CompressionEngine** requires a **BitString** interface with operations for manipulating bit strings.

The **CompressionEngine** provides an interface the **ImageRenderer** requires. A designer can “wire” these components together in a design, specifying that one component uses the services of another, by combining the ball and

socket symbols to form an *assembly connector*. This is illustrated in Figure 9-3-7.

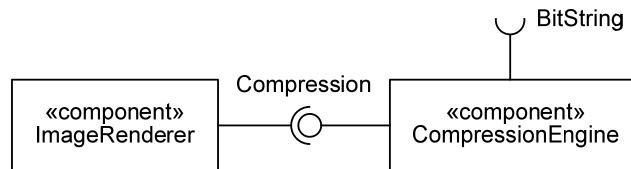
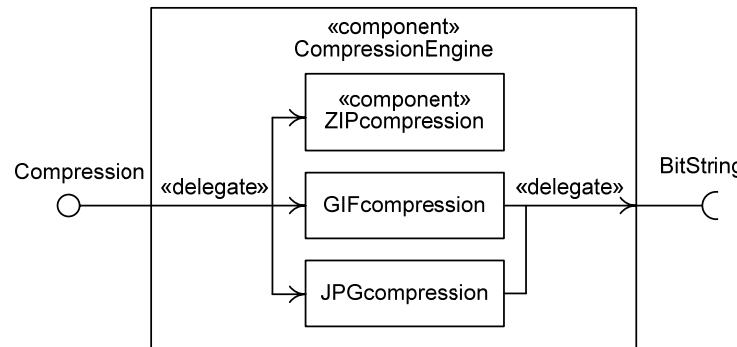


Figure 9-3-7 An Assembly Connector

It should now be clear why the interface symbols are called ball and socket symbols.

Component required and provided interfaces can also be listed inside the component symbol in a compartment below the component name. The interfaces can be listed under the stereotypes «required interfaces» and «provided interfaces». Other compartments can be added as well to document additional information about a component.

Component Internal Structure	<p>So far, we have concentrated on the UML notation for representing components and their relationships to their environment. However, components typically encapsulate classes and perhaps other components, and this structure may need to be designed as well. UML provides the means for modeling the constituents of components and their relationships to one another.</p> <p>Besides the component name, the main compartment of a component symbol can contain class and component diagram symbols. In addition, special symbols are provided to connect the nested symbols with the interfaces that the component uses to interact with its environment.</p> <p>A <i>delegation connector</i> ties a component interface to one or more internal classes or components that realize or use the interface. Delegation connectors are solid arrows stereotyped «delegate». A delegation connector may extend from an external interface lollipop to an internal class, component, or lollipop to indicate the constituent that realizes the provided interface. A delegation connector may also extend from an internal class, component, or interface socket to an external interface socket. For example, the component diagram shown in Figure 9-3-8 on page 276 shows the internal structure of the CompressionEngine component. This diagram shows that the CompressionEngine has three internal parts handling different kinds of compression. Services provided by the Compression interface are actually performed by the ZIPcompression component or one of the compression classes. The GIF and JPG compression classes require the BitString interface.</p>
------------------------------	--

**Figure 9-3-8 Component Internals**

External interface and delegation connectors may meet at a *port*, which is an interaction point between a component and its environment. Ports are drawn as rectangles on component borders. Their use is optional, however, so we have left them out of the component diagram in Figure 9-3-8.

Modeling Architectures with Components

Architects may decide that certain portions of a program should be reusable and replaceable sub-systems that are either purchased from software component vendors, obtained from previous projects, or developed as part of the product. In any case, such reusable and replaceable architectural constituents can be modeled in UML as components. It may be that an entire program is composed of components; in this case, architects can use UML component diagrams to show decomposition and component assembly relationships.

When designers must create a new component, they can use component diagrams to model the static internal structure of the component. The internal structures of components may be at too low a level of abstraction to be of interest during architectural design. If so, these diagrams can be used for component detailed design.

Heuristics Summary

Figure 9-3-9 summarizes the heuristics discussed in this section.

- Use notes, constraints, properties, and stereotypes to add information to UML models.
- Use stereotypes to name dependencies.
- Use packages to group elements in static models.
- Use package diagrams to model architectural modules, their parts, and their relationships.
- Use components to model reusable and replaceable program parts.
- Use component diagrams to represent the assembly relationships between components and their internal static structure.

Figure 9-3-9 UML Diagramming Heuristics

Section Summary

- Comments can be added to UML diagrams as **notes**.
- UML can be extended using **constraints**, **properties**, and **stereotypes**.
- A **dependency relation** can be shown between UML elements using a dependency arrow.
- UML **packages** are collections of items.
- UML **package diagrams** have packages as their primary symbols.
- In UML, a **component** is a modular, replaceable unit with well-defined interfaces.
- A UML **component diagram** shows components, their relationships to their environment, and possibly their internal structures.
- A UML **interface** is a named collection of public features and obligations.
- UML distinguishes between **provided interfaces** realized by a component or class and **required interfaces** used by a component or class.
- Components are defined by their provided and required interfaces.

Review Quiz 9.3

1. What can notes be attached to in UML?
2. Give two examples, different from those in the text, of dependency relations.
3. How are the contents of packages represented in UML?
4. Can UML components be nested?
5. How are required and provided interfaces represented?
6. What are assembly and delegation connectors?

9.4 UML Deployment Diagrams

Logical and Physical Architecture

So far we have been concerned with notations for modeling what might be called a product's **logical architecture**, which is the configuration of a product's major constituents and their relationships to one another in abstraction from the product's implementation as code and its execution on actual machines. Also important, especially for products deployed on several computers, is **physical architecture**, which is the realization of a product as code and data files residing and executing on computational resources.

Physical architecture can be modeled with box-and-line diagrams. UML also provides a notation for physical architecture modeling, called deployment diagrams.

Artifacts and Nodes

A UML **artifact** is any physical representation of data used or produced during software development or software product operation. Examples of artifacts are files, documents, messages, database tables, program code, and diagrams.

Artifacts have types and instances. For example, the source code file `prog.java` may exist in multiple copies on various storage media. Each of these copies is an instance of the single abstract file type `prog.java`.

Artifacts are represented in UML by rectangles containing the artifacts' names and marked with either the stereotype «artifact» or a special artifact icon in the upper right-hand corners. Artifact types have non-underlined names, while artifact instances have underlined names. The diagram in Figure 9-4-1 illustrates artifact symbols.

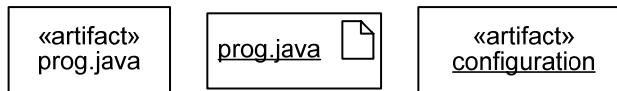


Figure 9-4-1 Artifact Symbols

The symbol on the left represents the artifact type `prog.java`, while the symbol in the middle represents an instance of this type. The dog-eared page is the artifact icon. The symbol on the right is an instance of an artifact called `configuration`.

Artifacts are the physical manifestations of the abstract software entities in our models. UML uses a dependency arrow stereotyped «manifest» to indicate the relationship between an abstract model element and the artifacts that realize it as a physical object.

A **node** is a computational resource. There are two kinds of nodes:

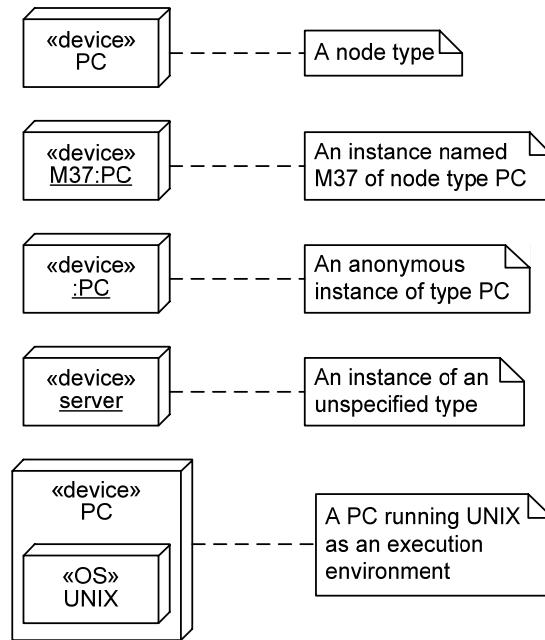
Device—A device is a physical processing unit, such as a computer, telephone, refrigerator controller, and so forth.

Execution Environment—An execution environment is a software system that implements a virtual machine. For example, an operating system, a database system, and the Java Virtual Machine are all execution environments.

A node is a real or virtual machine. Nodes are represented in UML by box or slab icons. Devices are stereotyped with «device» and execution environments either with the stereotype «execution environment» or, more often, with a stereotype describing the virtual machine, such as «OS» or «JVM».

Like artifacts, nodes have types and instances. For example, `WebServer` may be a type of machine, and `server1`, `server2`, and `server3` may be instances of that type. Node types are labeled with their names. Node instances are labeled with underlined identifiers of the form `name : type`. The `name` may be left off, indicating an unnamed instance of the type, or the `: type` may be left off, indicating a named instance with an unspecified type.

The diagram in Figure 9-4-2 illustrates node symbols.

**Figure 9-4-2 Node Symbols**

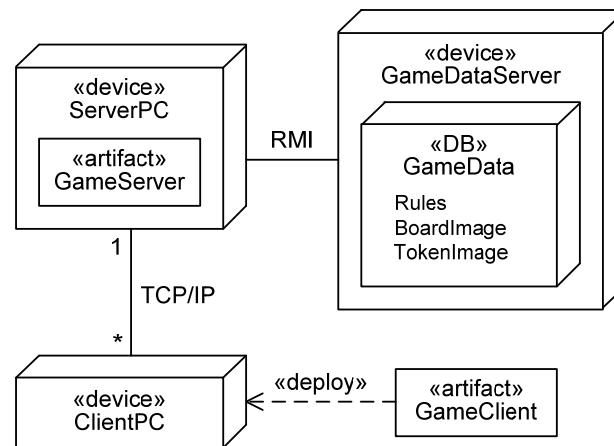
Nodes may be nested, as indicated by the last example. Usually, the outer node is a device and the inner nodes are various kinds of execution environments.

Deployment Diagrams

A **deployment diagram** models computational resources, the communication paths between them, and the artifacts that reside and execute on them. Deployment diagrams represent computational resources with node type or instance symbols. Communication connections between nodes are shown with *communication paths*, which are solid lines. Like an association line, a communication path can be labeled with the name of the communication link, and it can have multiplicities and role names at its ends.

Artifacts are deployed to nodes where they reside and may execute. The deployment relationships between artifacts and nodes can be shown in three ways: artifact symbols can be placed inside node symbols, artifacts names can be listed inside node symbols, or artifact symbols can be connected to node symbols with dependency arrows stereotyped «deploy». The deployment diagram in Figure 9-4-3 on page 280 illustrates these conventions.

This diagram depicts the physical architecture of a system for playing games over the Internet. A game player runs a **GameClient** program on a **ClientPC** that communicates with a **ServerPC** using TCP/IP. The **ServerPC** runs a **GameServer** program that directs play and mediates communication among the players. The **GameServer** consults a **GameData** database to

**Figure 9-4-3 A Deployment Diagram**

obtain the information it needs to oversee a game. The **GameData** database contains **Rules**, **BoardImage**, and **TokenImage** artifacts. The database is an execution environment that runs on its own **GameDataServer** computer. The **ServerPC** and **GameDataServer** communicate via Remote Method Invocation (RMI).

Deployment Diagram Uses	<p>Deployment diagrams show the real and virtual machines used in a system, the paths over which they communicate, the program and data files realizing the system, and where programs run and data reside. Deployment diagrams thus provide a rich notation for modeling physical architecture.</p> <p>Deployment diagrams are useful during architectural design, particularly for distributed systems. They can also be helpful in modeling physical deployment during detailed design.</p>
-------------------------	--

Section Summary

- **Artifacts** are physical representations—mainly files of some sort—of data used or produced in software development or operation.
- A **node** is a computational resource.
- UML **deployment diagrams** show nodes, communication paths between them, and the artifacts that reside and execute on them.
- Deployment diagrams are useful for representing **physical architecture**, which is the realization of a product as code and data files residing and executing on computational resources.

Review Quiz 9.4

1. What is the difference between artifact types and instances?
2. How does the UML notation for representing artifact instances differ from the notation for representing node instances?
3. How can the deployment of an artifact to a node be shown in UML deployment diagrams?

Chapter 9 Further Reading

- Section 9.1** Different views about the amount of detail appropriate for architectural design are evident in various software architecture texts, such as [Bass et al. 2003], [Bosch 2000], [Clements et al. 2003], and [Shaw and Garlan 1996]. Bass et al. [2003] discuss extensively the reciprocal influences between architecture and organizations. Bass et al. [2003] and Bosch [2000] discuss the architectural design process in detail. Clements et al. [2003] provide a much more complete SAD template appropriate for larger systems.
- Section 9.2** The best book on architectural design notations and documentation is [Clements et al. 2003], which goes far beyond our brief survey in this section. Box-and-line diagrams are discussed in [Shaw and Garlan 1996].
- Section 9.3-4** UML extension mechanisms, dependency relations, and package, component, and deployment diagrams are discussed in most UML books, including [Bennett et al. 2001], [Booch 2005], [OMG 2003], [OMG 2004], and [Rumbaugh 2004].

Chapter 9 Exercises

- Section 9.1**
1. *Fill in the blanks:* Architectural design must take account of both _____ and _____ requirements. Any number of architectural structures may allow a program to satisfy its functional requirements, but only _____ of these will allow it to also satisfy its non-functional requirements. Software architects must consider _____ to find those specifying a program that can satisfy both its functional and non-functional requirements.
 2. Suppose you are writing software for a radio station that manages its playlists. The program will generate candidate playlists from a record library automatically and station personnel can then check and modify them. Disc jockeys must also be able to change playlists when they are used because what is actually played is often different from what is planned. The playlists are then used to generate reports for paying royalties. You must decide what sort of data structure to use to store playlists. Make a choice and write a design rationale. Your rationale should explain the factors that went into your decision, the design alternatives you considered, your evaluation of design decisions, and the reasoning for your final choice.
 3. Classify the following specifications by quality attribute. Choose from the attributes maintainability, reusability, performance, availability, reliability, and security.
 - (a) The program must never lose data that has been saved to persistent store.
 - (b) Developers must be able to incorporate at least 60% of the StereoColor product code in the StereoColor Deluxe product.
 - (c) The program must be able to respond to queries between the hours of 10 A.M. and 7 P.M. Eastern Standard Time.

- (d) The program must be able to execute 9,000 transactions per minute and two million transactions per day.
- (e) The program must require passwords from all users.
- (f) When a transcript is requested, the program must scan the student record to detect tampering before producing the transcript.

Section 9.2

- 4. Classify the following specifications as syntactic, semantic, or pragmatic:
 - (a) A Java interface may contain only static final variable declarations and public method header declarations.
 - (b) A Java synchronized method can execute only when it has a lock on the object in which it resides.
 - (c) Only one class name can follow the `extends` keyword.
 - (d) In Java, a static method cannot access non-static attributes.
 - (e) Null values can be added to a Java Vector.
 - (f) The `length()` method of an array should be used to control loops that process each element of the array.
 - (g) Java has no pointers, so linked structures must be implemented using references.
 - (h) In Java, all `RuntimeExceptions` are unchecked.
- 5. Write an interface for a stack module using the interface specification template in Figure 9-2-2.
- 6. Write an interface for a hash table module using the interface specification template in Figure 9-2-2.
- 7. Compilers have a standard architecture consisting of a **Tokenizer** that converts incoming program text to tokens (or meaningful language units), a **Parser** that analyzes the syntax of the program and produces a syntax tree, a **Code Generator** that examines the syntax tree and produces object code, and an **Optimizer** that processes the object code to make it more efficient. Make a box-and-line diagram modeling this architecture.
- 8. A common way to design many applications that manipulate a database is to have a module for the user interface, a module for data processing, and a module to access the database. This is called a *three-tier architecture*. Draw a box-and-line diagram of a three-tier architecture.

Section 9.3

- 9. Make a UML package diagram showing a three-tier architecture (see the last exercise).
- 10. Make a UML class diagram with package symbols to model the `java.util.regex` package. You need only show class name compartments.
- 11. Draw a UML class diagram in which you have an attribute with a `{constant}` property, an operation with a `{synchronized}` property, and two associations with an `{xor}` (exclusive or) constraint.

12. Compare and contrast module interfaces, UML interfaces, and Java interfaces.
13. A software component vendor sells a product that provides text indexing and searching. The product provides a Boolean search interface and an SQL interface, and it requires file operations to store its index. Draw a component diagram showing this product and its provided and required interfaces.
14. *Find the errors:* Find at least four errors in the component diagram in Figure 9-E-1.

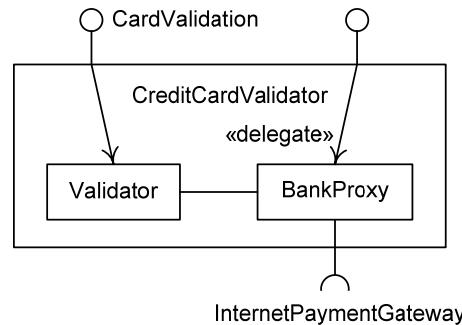
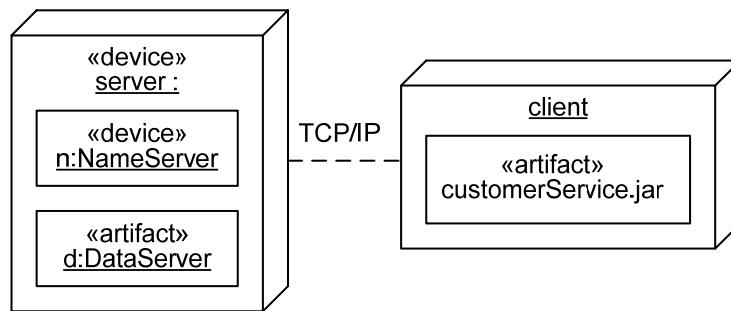


Figure 9-E-1 An Erroneous Component Diagram for Exercise 14

Section 9.4

15. An egg timer program is implemented in `EggTimer.java` and `Pulse.java` files. These files are compiled into class files. A manifest file is used by the `jar` program to create an executable `EggTimer.jar` file. Make a diagram to illustrate the artifacts involved in this process.
16. Make a deployment diagram showing the computers (as nodes) and programs (as artifacts) involved when you use your computer to access `http://java.sun.com`. You may assume that an instance of the Apache program is running on a Sun Web server computer.
17. Consider two architectures for an instant messaging system. Both architectures have a messaging server running somewhere on a TCP/IP network, and both have messaging clients running on computers somewhere on the network. In one architecture, the clients obtain connections to other clients through the server and also send all messages through the server. In the other architecture, the clients obtain connections through the server, but all messages are sent directly between the clients. Draw deployment diagrams depicting these alternative architectures.
18. *Find the errors:* Find at least four errors in the deployment diagram in Figure 9-E-2.

**Figure 9-E-2 An Erroneous Deployment Diagram for Exercise 18****Research Project**

19. Consult software engineering and software architecture texts to write a glossary of quality attributes. Each entry should include a definition of the attribute and an example illustrating it.

Chapter 9 Review Quiz Answers**Review Quiz 9.1**

- Development organizations influence architectural design in the following ways: the structure of the organization may be reflected in the structure of the architecture; strengths and expertise of the organization may be reflected in the architecture; the assets, standards, and practices of the organization may influence the architecture; and the experience, knowledge, and skills of the organization's architects influence the architecture.
A program's architecture may influence a development organization because groups may be formed to implement and support architectural constituents; people may be hired or trained to implement and maintain the architecture; assets may be created, practices altered, and standards created or modified to accommodate the architecture; and architects learn and change based on their design experiences.
- A software architecture document should have (or refer to) a product overview, present architectural models, provide mappings between the models, explain the design rationale, and include a glossary.
- Development quality attributes are program properties of interest to development stakeholders (developers and their managers), such as maintainability and portability. Operational quality attributes are program properties of interest to non-development stakeholders (clients, purchasers, users, and so forth), such as performance, reliability, and security.
- Availability is the readiness of a program for use. Maintainability is the ease with which a product can be changed. Performance is the ability of a program to do its job within resource limits. Reliability is the ability of a program to function according to its specification when used normally. Reusability is the degree to which artifacts created during product development can be used in developing other products. Security is the ability of a program to resist attack.

Review Quiz 9.2

- Collaborations between program parts can be modeled using UML sequence, communication, and activity diagrams; use case models; data flow diagrams; and box-and-line diagrams.

2. English syntax specifies that words must appear in a certain order. For example, “Snake a Peter is” is not syntactically correct, but “Peter is a snake” is correct. Semantics specify the meanings of syntactically correct expressions, so the sentence “Peter is a snake” means what it does by virtue of the semantics of English. Pragmatics specify how expressions can be used to accomplish tasks. For example, saying that Peter is a snake when discussing someone’s pet conveys the species of the pet. Saying that Peter is a snake when discussing someone’s personality states a metaphor.
3. A simple example of a protocol is the exchange that occurs at the start and end of a phone call. When people answer the phone, they say something to indicate that they are present, often giving their name or organization as well. The caller then proceeds with business. At the end of the conversation, one person says goodbye, the other person acknowledges with another goodbye, and both parties hang up.
4. A precondition is an assertion that must be true before an activity or operation begins. A postcondition is an assertion that must be true after an activity or operation finishes.
5. Box-and-line diagrams are composed of boxes (icons or symbols) and lines of various sorts.

**Review
Quiz 9.3**

1. In UML, notes can be attached to any model element.
2. Dependency relations that often show up in UML models include the *call* relation (when one entity invokes an operation of another), the *instantiate* relation (when an object is an instance of a class), the *manifest* relation (when code realizes a model element), the *deploy* relation (when a file is stored or executed on a computational resource), and the *extend* relation (when one entity augments the behavior of another).
3. UML package contents are represented in two ways: Either the contents are placed within the main rectangle of the package symbol, or they are connected by a line to a special symbol (a circled cross) attached to the package symbol.
4. UML components can be nested. Component symbols can contain components, classes, interfaces, and associations between them, as well as assembly connectors.
5. Required interfaces are represented by socket symbols, which are half circles connected to the requiring component or class by a solid line. Provided interfaces are represented by ball or lollipop symbols, which are unfilled circles attached to the providing class or component by a solid line.
6. An assembly connector is formed when an interface ball symbol is fitted into an interface socket symbol. This connector shows how the interface needs of one component or class, expressed by a required interface, are met by another component or class that provides the needed interface. A delegation connector is used inside component symbols to attach an external provided interface symbol to an internal entity that supplies it for the component, or to attach an internal component or class requiring an interface to an external required interface symbol. This connector shows how a component’s provided or required interfaces are related to the components or classes it encapsulates.

**Review
Quiz 9.4**

1. An artifact instance is a physical entity with a unique spatiotemporal location; for example, the copy of the Linux operating system presently residing on the disk drive of my computer. There is another copy of this operating system residing in the main memory (and the swap space) that is actually being executed. These two instances of Linux are not identical because they are in different places, but they are the same in some sense because they are both instances of the same artifact type. An artifact type is thus a kind of physical entity.
2. Artifact instances and types have the same names, but artifact instance names are underlined and artifact type names are not. For example, Key.java is an artifact type and Key.java is an artifact instance (of that type). Node type and instance names are different. Node types are labeled with the type name, with no underlining. Node instances are labeled with an identifier of the form name : type, where name may be omitted (indicating an unnamed instance of the type) or : type may be omitted (indicating a named instance of an unspecified type). For example, junior : SparcStation is a node called junior that is an instance of a SparcStation, magilla is the name of a node of unspecified type, and : IntelPC is an unnamed instance of type IntelPC.
3. The deployment of an artifact on a node can be shown in three ways in a deployment diagram: the artifact symbol can be placed within the node symbol, the artifact symbol can appear outside the node symbol but be attached to it by a dependency arrow from the artifact to the node stereotyped «deploy», and the artifact name can be listed inside the node symbol.

10 Architectural Design Resolution

Chapter Objectives

This chapter continues our survey of architectural design, focusing specifically on the steps in the architectural design resolution process, as illustrated in Figure 10-O-1.

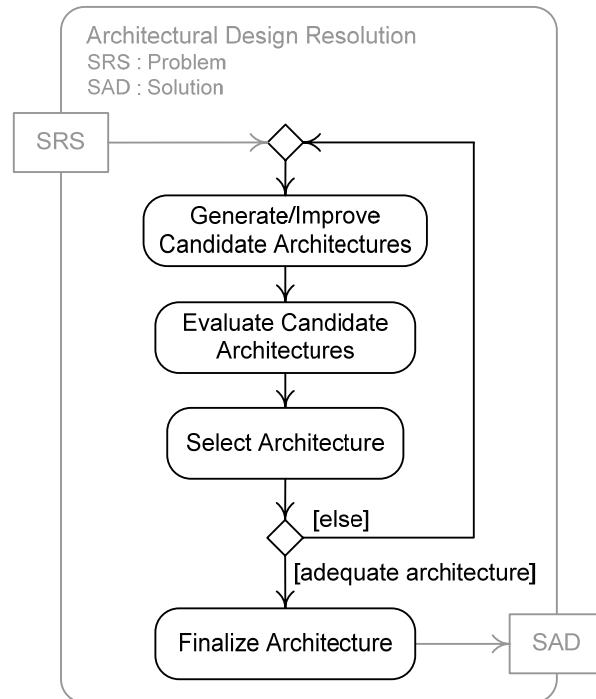


Figure 10-O-1 Architectural Design Resolution

The process is illustrated using the AquaLush architecture.

By the end of this chapter you will be able to

- List and explain several techniques for generating and improving a software architecture;
- Generate architectural alternatives by determining functional components or determining components based on quality attributes;
- Improve architectural designs by combining aspects of different design alternatives;
- Evaluate architectural design alternatives using scenarios and prototypes;

- Select architectural design alternatives by evaluating pros and cons or using scoring matrices;
- List software architecture document review techniques, including desk checks, walkthroughs, inspections, audits, and active design reviews; and
- Describe active design reviews and discuss their advantages.

**Chapter
Contents**

- 10.1 Generating and Improving Software Architectures
 - 10.2 Evaluating and Selecting Software Architectures
 - 10.3 Finalizing Software Architectures
-

10.1 Generating and Improving Software Architectures

**Generating
Design
Alternatives**

The first step in architectural design resolution is to generate design alternatives. Although coming up with a software architecture from thin air may seem difficult, there are several techniques for developing candidate architectures:

Determine Functional Components—This approach is based on studying the SRS and brainstorming candidate architectural constituents responsible for coherent collections of functional and data requirements. Designers revise the list of candidate components and add relationships to forge architectural alternatives.

Determine Components Based on Quality Attributes—This approach begins by forming constituents and constituent relationships to satisfy non-functional requirements. Designers then supply missing parts needed to satisfy functional and data requirements.

Modify an Existing Architecture—If the architecture for a similar program is available, it can be used as a starting point. Designers can add, remove, and alter components, responsibilities, and relationships to generate reasonable architectural alternatives.

Elaborate an Architectural Style—An **architectural style** is a paradigm of program or system constituent types and their interactions. There are about six standard styles suitable for a wide variety of programs. An architect can begin with an architectural style and elaborate it to produce candidate architectures.

Transform a Conceptual Model—A conceptual model describes the problem. If it is treated instead as a solution description, then its concepts can become architectural constituents and its relations can become architectural connections.

The third approach, modifying an existing architecture, requires an architecture for a similar system and hence is not as general an approach as the others; we do not consider this technique further. Architectural styles are treated in Chapter 15, so we also forego discussion of the fourth

approach, elaborating an architectural style. The last approach, transforming a conceptual model into a class model, is discussed in detail in the next chapter. In this section, we consider the first two approaches in more detail and use them to generate architectural alternatives for AquaLush.

In Chapter 5 we noted that studies show that individuals working alone generate more and better design alternatives than they do working in teams. Consequently, it is better for team members to spend time generating design alternatives on their own before and after a design team meets to generate alternatives. All these techniques work for both individuals and teams. If each architectural design team member applies two or three of these methods on their own and brings the results back to the team, there will be a rich set of design alternatives to consider.

Determining Functional Components

To begin an AquaLush functional decomposition, we consult the AquaLush SRS and other analysis models (such as the use case diagram) to make a list of the overarching functions or groups of functions that the program performs. Required AquaLush functions are

- Configuring the program at startup;
- Controlling irrigation (manually or automatically);
- Providing a user interface; and
- Allowing users to monitor and repair the system.

If we create a component to handle each function in this list, we get the first-cut architectural decomposition pictured in the box-and-line diagram in Figure 10-1-1.

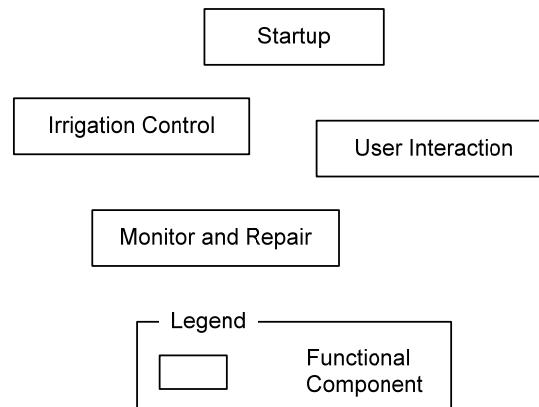


Figure 10-1-1 AquaLush Functional Decomposition, Version 1

At program startup, the Startup component must tell the Irrigation Control component about irrigation zones, valves, and sensors, so there must be an interaction connection between these two components. Users must be able

to set parameters for the **Irrigation Control** component, and during manual irrigation, irrigation information must be displayed to the user. This suggests that there must be an interaction connection between the **User Interaction** and **Irrigation Control** components. Similarly, users must be able to find out about failed parts and tell the system about repairs, so there must be an interaction connection between **User Interaction** and **Monitor and Repair**. These interaction relations are added to form the second draft shown in Figure 10-1-2.

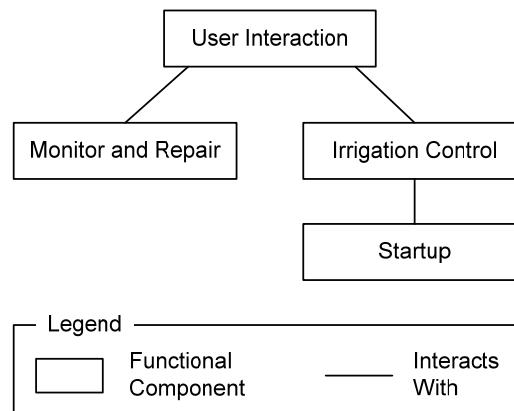
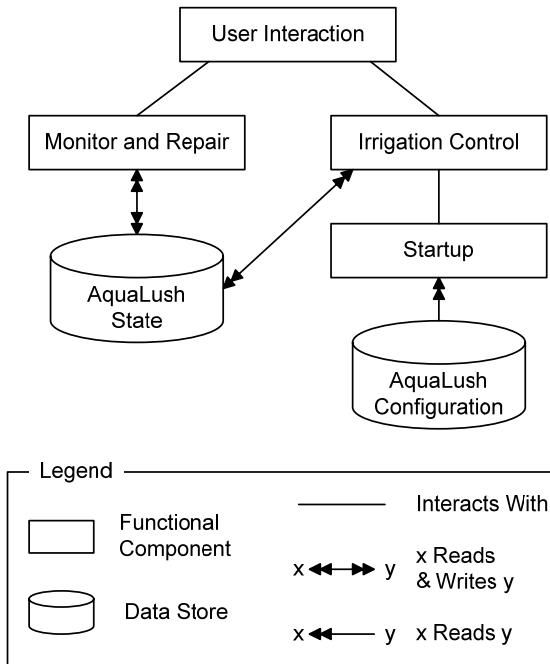
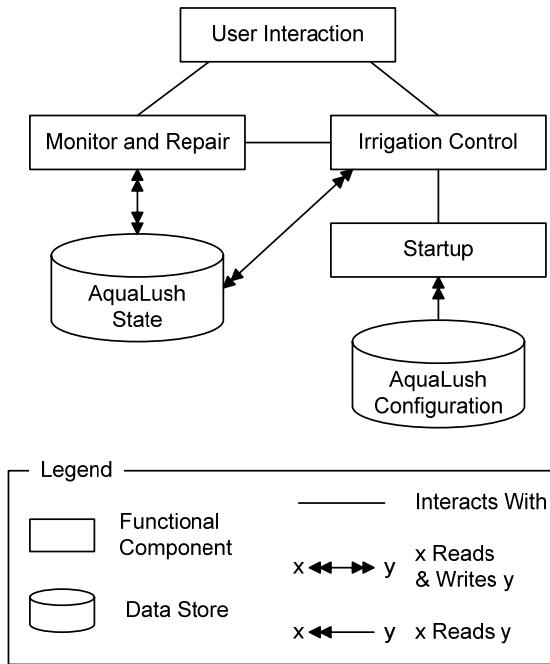


Figure 10-1-2 AquaLush Functional Decomposition, Version 2

The **Startup**, **Monitor and Repair**, and **Irrigation Control** components must have access to stored data. **Startup** must read a configuration data store. **Monitor and Repair** must maintain a persistent failed parts list. **Irrigation Control** must read and write the failed parts list and persistent irrigation control parameters. We need to add a persistent store holding the product configuration that **Startup** reads and a store holding irrigation parameters and the failed parts list read and written by both **Monitor and Repair** and **Irrigation Control**. A functional decomposition that includes the persistent data stores and their relationships with functional components is shown in Figure 10-1-3.

Should the **Monitor and Repair** component interact with the **Irrigation Control** component? The **Irrigation Control** component detects hardware failures, but it can write them to **AquaLush State** without notifying **Monitor and Repair**. Similarly, when a valve or sensor is repaired, the **Monitor and Repair** component can update the **AquaLush State** data store. However, **AquaLush** is required to begin using repaired components immediately during irrigation, so either **Monitor and Repair** must notify **Irrigation Control** whenever a repair is made, or **Irrigation Control** must check **AquaLush State** frequently during irrigation. We thus have two slightly different alternatives: the one shown in Figure 10-1-3 and the one shown in Figure 10-1-4.

**Figure 10-1-3** AquaLush Functional Decomposition, Version 3**Figure 10-1-4** AquaLush Functional Decomposition, Version 4

The box-and-line diagrams in the previous figures model functional decompositions, but they are short on details. It is too soon to elaborate any alternatives, but a bit more detail is needed to capture these alternatives more clearly. Table 10-1-5 lists the components and their responsibilities.

Component	Responsibilities
User Interaction	<ul style="list-style-type: none"> • Interacts with the AquaLush control panel hardware and implements the control panel user interface. • Obtains data from the Monitor and Repair component to display to users and transfers user repair data to Monitor and Repair. • During manual irrigation, obtains data from Irrigation Control for display and transfers user commands to Irrigation Control. • Interacts with Irrigation Control to obtain and set irrigation parameters.
Monitor and Repair	<ul style="list-style-type: none"> • Obtains the failed parts list from AquaLush State to pass on to User Interaction. • Modifies the failed parts list in AquaLush State when told that a part is repaired. • May notify Irrigation Control about repaired parts (depending on the version).
Irrigation Control	<ul style="list-style-type: none"> • Controls valves, reads sensors, and reads the clock. • Manages the irrigation parameters stored in AquaLush State. • In manual mode, implements user commands and returns irrigation data to User Interaction. • In automatic mode, implements irrigation cycles. • Records part failures in AquaLush State.
Startup	<ul style="list-style-type: none"> • Reads the AquaLush Configuration data store to obtain the configuration of irrigation zones, valves, and sensors. • Communicates this information to Irrigation Control.
AquaLush State	<ul style="list-style-type: none"> • Records all persistent data, including the failed parts list, and all the irrigation parameters.
AquaLush Configuration	<ul style="list-style-type: none"> • Records descriptions of the irrigation zones, sensors, and valves describing the installed product.

Table 10-1-5 AquaLush Component Responsibilities

Simulation Functional Components The functions considered so far are those needed in the fielded product. The Web-based simulator must have these functions, and it must also mimic the control panel, the irrigation site, and the sensors and valves. The Startup, Monitor and Repair, and Irrigation Control components, as well as the AquaLush State and AquaLush Configuration data stores, are still needed in the simulation. The User Interaction function is still present but must be very different in the simulator because it must simulate the control panel and interactions involving the simulated watering site. It might also simulate valves and sensors, or these might be broken out as a separate functional component, leading to several design alternatives. We present one alternative for the simulation program in Figure 10-1-6.

As with the components for the fielded product, component responsibilities should be recorded to help explain and document this design alternative.

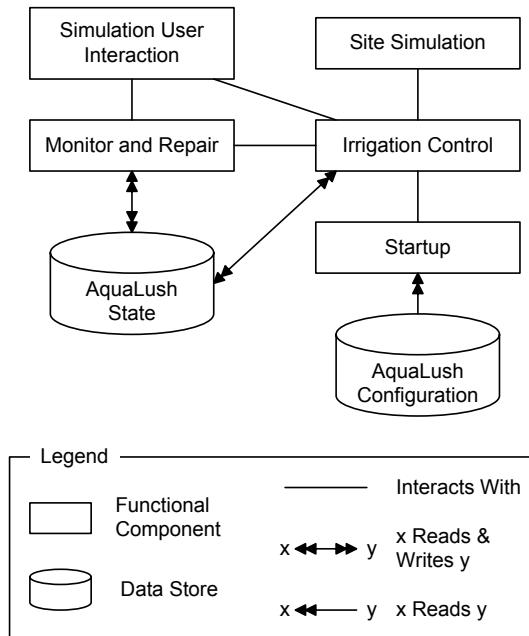


Figure 10-1-6 AquaLush Simulation Functional Decomposition

We have now generated two slightly different architectural decompositions for the fielded program and illustrated one candidate decomposition for the AquaLush simulation. We now turn to a different approach to generate other alternatives.

Determining Components Based on Quality Attributes

To begin an AquaLush program decomposition based on quality attributes, we consider the program's main non-functional requirements. Besides those having to do with ease of use (which are mainly satisfied in the user interface design), in broad terms AquaLush has the following non-functional requirements:

Reusability—The main irrigation software components must be used in the fielded product, the simulation, and future versions of the product.

Hardware Adaptability—The program must work with various valve types and probably various types of sensors, displays, keypads, and screen buttons.

Reliability—The program must not fail often in normal use.

Modifiability—The program must accommodate changes in irrigation strategy.

Reusability requirements can be met by architectural parts that are highly modular: small, cohesive parts that hide information and are loosely coupled to other parts. Thus, the main irrigation components should have these qualities, which immediately suggest a separate **Irrigation Control** module.

The Irrigation Control module must also be modifiable, which re-emphasizes the need for it to be modular and further suggests that it should be broken into parts that can be replaced or modified separately. Hence, the Irrigation Control module can contain a Manual Irrigation Control module and an Automatic Irrigation Control module. This is illustrated in the box-and-line diagram in Figure 10-1-7.

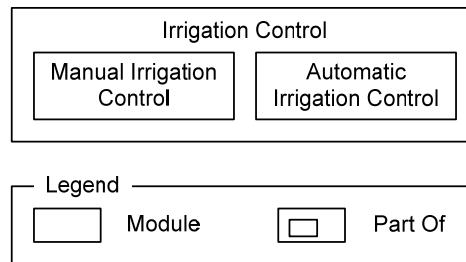


Figure 10-1-7 AquaLush Modular Decomposition, Version 1

AquaLush must be able to work with a variety of valves and be able to handle different kinds of control panel hardware and moisture sensors. The details of the device drivers for all this hardware should be hidden from most of the program. Otherwise, it will be difficult to change hardware later.

Device Interface Modules and Virtual Devices

A standard way to design a program with complex interfaces to devices or other systems is to segregate the code that deals with such things in a **device interface module**. The program units in the device interface module hide all details of interactions with hardware devices or other systems. Their only job is to communicate with entities outside the program, and they are changed whenever the device or system that they communicate with is changed. Therefore, the units in the device interface module must contain all device-dependent communication details and nothing else.

Units in the device interface module should present a stable interface to the rest of the program. One way to do this is to make these units into virtual devices.

A **virtual device** is a software simulation of, or interface to, a real hardware device or system.

A virtual device's main job is to provide a stable interface to a real device or system while hiding all details of the actual interface to the real device or system.

Actual hardware devices often have odd idiosyncrasies. A real device may do much more or much less than what one might expect. For example, a barometric pressure device may provide both pressure and temperature, or

it may provide only a voltage level that must be converted to a pressure. A real device may have an awkward interface. For example, a real device may require some unusual combination of input signals or require that input is provided only at certain times. Different versions of the same device or similar devices from different manufacturers may have unique peculiarities.

Virtual devices should hide all this and be carefully designed to simulate “ideal” devices. Such an ideal device would

- Do exactly one job completely (cohesion);
- Have a simple and consistent interface meeting the needs of the rest of the program (simplicity);
- Be loosely coupled to the rest of the program (coupling);
- Hide its implementation (information hiding); and
- Never change its interface.

A virtual device may be relatively simple or quite complicated, depending on the characteristics of the real hardware device or system with which it communicates. It must also be reimplemented whenever the actual device or system with which it communicates changes. Thus, a virtual device may be realized as a family of replaceable components implementing the same virtual device interface but using various real hardware devices or systems.

More Alternatives We can use a device interface module to achieve AquaLush hardware adaptability requirements. A device interface module for AquaLush can contain virtual devices for valves, sensors, and the control panel hardware, as illustrated in Figure 10-1-8.

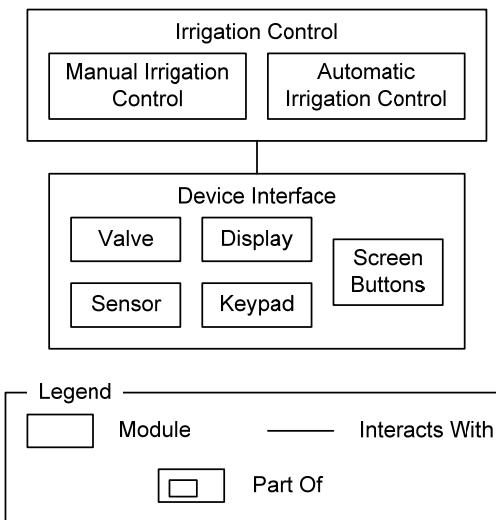


Figure 10-1-8 AquaLush Modular Decomposition, Version 2

An interaction line connects the **Irrigation Control** and **Device Interface** modules to indicate that the **Irrigation Control** module will use the virtual devices in the **Device Interface** module to communicate with real sensors and valves.

The only quality attribute we have not yet considered is reliability. There are several strategies for improving reliability, but the one that is perhaps most applicable in this case is to make components as small and simple as possible and to make their interactions as simple as possible. AquaLush is already a small and simple product with relatively few functions and capabilities. It has no requirements that demand elaborate data structures, difficult algorithms, or concurrency, which tend to make programs more complicated. The decompositions already made to achieve reusability, configurability, and modifiability have incidentally resulted in small, simple, cohesive, decoupled modules. It should be possible to make these modules very reliable.

Having considered non-functional requirements, we must now consider functional requirements and modify the architecture to include missing functionality. A glance at Figure 10-1-8 reveals that there is no user interface module. We can add this, along with the necessary interaction connections, to reach the final version of this design alternative, which appears in Figure 10-1-9.

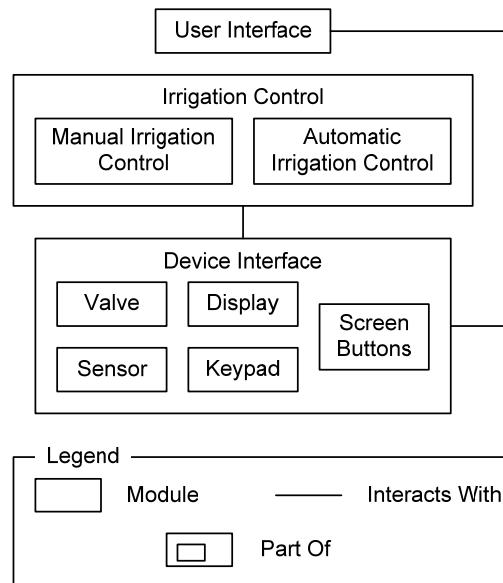


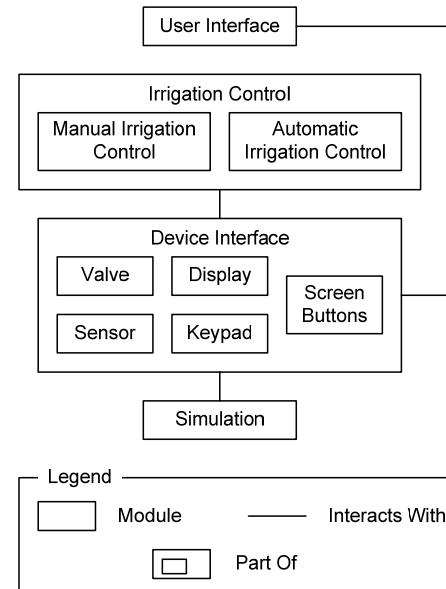
Figure 10-1-9 AquaLush Modular Decomposition, Version 3

Table 10-1-10 augments the box-and-line diagram by detailing module responsibilities.

Module	Responsibilities
User Interface	<ul style="list-style-type: none"> • Interacts with the user through virtual control panel devices in the Device Interface module. • Displays data about the program state, irrigation parameters, failed devices, and so forth. • Accepts user commands to set irrigation parameters, registers devices as repaired, controls manual irrigation, and passes user commands on to the Irrigation Control module.
Irrigation Control	<ul style="list-style-type: none"> • Controls irrigation parameters and hosts modules for manual and automatic irrigation control. • Contained modules control virtual valves, read virtual sensors, and maintain irrigation cycle states.
Device Interfaces	<ul style="list-style-type: none"> • Hosts virtual devices for valves, sensors, and the control panel display, keypad, and screen buttons. • Hides details of communicating with real or simulated hardware devices.

Table 10-1-10: AquaLush Module Responsibilities

Simulation Module The Device Interface module simplifies adding most items needed in the Web-based simulation: The simulated valves, sensors, and control panel can all be connected to the program through the device interface module. However, we still need one or more modules to realize the simulated irrigation site and the user interface to control it. These responsibilities can be taken on by a **Simulation** module. The **Simulation** module interacts with the simulation user and simulates the irrigation site. It is also a container for all simulated devices, so it needs to interact with the virtual devices in the device interface module. Hence, we have the augmented architecture in Figure 10-1-11 for the Web-based simulation program.

**Figure 10-1-11 AquaLush Simulation Modular Decomposition**

Comparing Alternatives

The design alternatives generated using functional decomposition and quality-attribute-based techniques are quite different. This is good because each alternative may have strong points that can be used in the ultimate design.

One difference between the alternatives generated by these two techniques was previously alluded to by calling the parts of the functional decomposition “functional components” and the parts of the quality attribute decomposition “modules.” Functional decomposition naturally results in runtime components; the AquaLush quality attributes are mainly development attributes and hence are characteristics of design-time code units. Thus, the design alternatives, besides being different architectural solutions, also model different architectural views.

Nevertheless, there is a correspondence between a program’s runtime functional components and its design-time module structure, so we can still compare these alternatives, either in their present forms or by generating comparable views and comparing those.

Improving Alternatives

After architectural alternatives have been evaluated and the best ones have been selected for improvement, they must be improved. Evaluation and selection are discussed in the next section; improvement is akin to alternative generation, so we will discuss it now.

Several methods can be used to improve a software architecture:

Combine Alternatives—The best features of two or more design alternatives can be combined into an improved design. This is one advantage of generating many alternatives: It is likely they solve different problems, but most problems are solved by some member of the set. For example, the AquaLush design alternative in Figure 10-1-4 has data stores for the product configuration and the product state. These are both clearly needed. The alternative in Figure 10-1-9 has a device interface module, which is the best way to handle hardware adaptability requirements. Combining these features can produce a new design better than either of the current alternatives.

Impose an Architectural Style—Architectural structures that approximate a particular style may be improved by modifying them to fit the style exactly. For example, the modular decomposition shown in Figure 10-1-11 nearly fits the Layered architectural style discussed in Chapter 15. In a Layered style, modules are layers, and the layers are arranged into a stack such that each layer can use only the layers below it (though sometimes with some exceptions). In the design pictured in Figure 10-1-11, the interaction relationships approximate this pattern. If the interactions between modules can be constrained to uses of the lower modules by the modules above them, we can impose the Layered style on this alternative. Layered style constraints decrease coupling, increase cohesion and information hiding, and make the layers more reusable.

Apply a Mid-Level Design Pattern—Mid-level design patterns are like architectural styles but apply at lower levels of abstraction, typically involving a few interacting classes. Mid-level design patterns are discussed extensively in Chapters 16 through 19. These patterns can be applied during architectural design to improve parts of the design. For example, the AquaLush module decomposition separates the Irrigation Control module to try to make it more reusable. Components must be weakly coupled to other components to be highly reusable. The Irrigation Control module must provide a lot of data to the User Interface module, so how can they be weakly coupled? The Observer design pattern provides a means to solve this problem. In the Observer pattern, observer modules register with a subject module. Thereafter, the subject notifies its observers of any changes in the subject that may be of interest to the observers. In response to a notification, an observer can query the subject to learn about changes. This pattern helps decouple subjects from their observers. The User Interface can be made an observer of the Irrigation Control module, greatly reducing their coupling. This relationship can be shown by a special connector in a box-and-line diagram, illustrated in Figure 10-1-10.

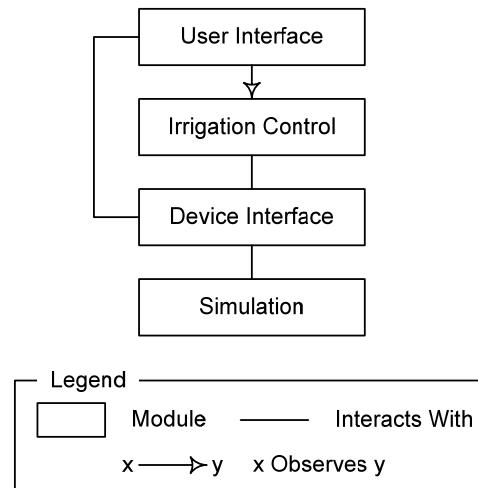


Figure 10-1-12 Decoupling with the Observer Pattern

Section Summary

- Software architects can generate design alternatives by determining functional components, determining components based on quality attributes, modifying an existing architecture, elaborating an architectural style, or transforming a conceptual model.
- Designers should use these techniques both alone and in teams to generate many architectural design alternatives.
- A **device interface module** contains **virtual devices** that hide the details of interactions with real or simulated hardware devices or systems.

- Device interface modules make programs more configurable and more changeable.
- Design alternatives can be improved by combining the best aspects of several alternatives, imposing an architectural style, or applying mid-level design patterns.

**Review
Quiz 10.1**

1. What is an architectural style?
 2. What parts of an SRS are most relevant when determining architectural components based on quality attributes?
 3. What characteristics should an ideal virtual device have?
-

10.2 Evaluating and Selecting Software Architectures

Evaluation Challenges

A software architecture is satisfactory if it specifies the high-level structure, behavior, and characteristics of a program that satisfies software product requirements. Once a product is built, it can be examined to determine whether it meets its requirements, and hence whether its architecture is adequate. But after a product is built, it is too late for an architecture evaluation to do much good. How can designers evaluate a software architecture during architectural design before there is a working program? This is the main software architecture evaluation challenge.

No one knows how to guarantee that a program built according to an architectural specification will meet its requirements. Instead, the goal of software architecture evaluation is to determine whether a program built to an architectural specification is *likely* to satisfy its requirements. In this section we consider two approaches to establishing this likelihood: scenarios and prototyping.

Scenarios

In Chapter 6, a **scenario** was defined as an interaction between a product and particular individuals and was illustrated with use case instances. Use case instances are indeed scenarios, but there are other kinds of scenarios as well. Scenarios that instantiate use cases are interactions between *actors* and the product, but in general a scenario can be an interaction between *any individuals* and the product. Scenarios involving individuals who are not actors, especially those involving developers, are important for evaluating whether an architecture specifies a program that can meet its non-functional requirements.

Scenarios are stated in **scenario descriptions**. There is no accepted format for scenario descriptions; they are usually short narratives. For example, the paragraph in Figure 10-2-1 describes a scenario in which a developer adds support for a new valve to AquaLush.

Add a Valve Type—A developer modifies the AquaLush source code and design documents so that AquaLush supports a new type of valve. The developer completes the process in two days.

Figure 10-2-1 An AquaLush Maintenance Scenario

Although there are no established formats for writing scenarios, the following heuristics help make clear and useful scenario descriptions:

Label each scenario with a descriptive phrase. It is easier to work with scenarios when they have short and meaningful identifiers. For use case scenarios, the identifier can be based on the use case name.

Write simple declarative sentences in the active voice. This standard rule of good technical writing makes scenario descriptions easier to read.

Write scenario descriptions in three parts. Scenarios begin when the product and the environment are in an initial state, consist of an activity flow involving the product and some individuals, and conclude with the product and the environment in a final state. Scenario descriptions can be written by describing the initial state, describing the activity flow, and describing the final state.

Make a principal in the interaction the subject of each sentence describing the activity flow. When writing use case scenario activity flows, the subjects of the sentences are either the product or actors, which may be people, devices, or other systems. When writing other kinds of scenario activity flows, the subjects of the sentences are usually people doing something to the product, such as changing it or testing it.

Describe the initial and final states of both the product and its environment. Interesting scenarios often revolve around how the product behaves in special circumstances, such as when recovering from a failure or when presented with erroneous input. The starting and ending points of such scenarios must be described to make them clear. This requires describing both the state of the product and the state of its environment before and after the scenario activity flow.

State target outcome measures whenever possible. Scenario outcome measures are often important for determining whether requirements are met. For example, the maintenance scenario in Figure 10-2-1 includes a measure of the interaction elapsed time, which is the criterion used to determine whether AquaLush satisfies the requirement that it must be easy to modify to control a variety of irrigation valves.

Proofread the description. Proofreading is a proven error-detection technique and should be applied to all documents.

Profiles Usually several scenarios are needed to determine whether an architecture specifies a product likely to meet its requirements. A **profile** is a set of scenarios used to evaluate whether a product is likely to meet a set of requirements. For example, a *usage profile* is a set of scenarios used to

determine whether a product is likely to meet its functional requirements. It generally features scenarios characterizing typical program uses. A *maintenance profile* includes scenarios for typical maintenance activities used to determine whether the product meets maintenance requirements.

The scenarios in a profile often have importance weights. These weights are used during alternative selection, discussed in the next section.

Except for the simplest products, the set of all scenarios for a product will be very large. For example, most AquaLush use cases can be instantiated in many ways (because of variations in activity flows), and there are eight use cases, so there are perhaps several hundred usage scenarios. Similarly, there are all sorts of changes that might be made to the AquaLush product itself, so there are dozens of possible maintenance scenarios.

It is not practical to consider every possible scenario when evaluating software architecture design alternatives. Instead, profiles are formed by choosing a few that are most important or most likely to occur from all possible scenarios. Profiles usually contain between 3 and 10 scenarios. In summary, profiles are small representative samples of all possible scenarios.

Scenario and Profile Creation

A useful tool for generating profiles and scenarios is a utility tree. A **utility tree** is a tree whose sub-trees are profiles and whose leaves are scenarios. A utility tree is constructed starting at the root, which is labeled “utility” (*utility* is overall product value).

The next step is to add profile names as children of the root. The profiles are named after quality attributes, such as *reusability* or *Maintainability*. One or more usage profiles may also be added to the tree. The profiles are chosen to reflect product requirements. For example, AquaLush has important usage, modifiability, configurability, reusability, and reliability requirements. However, AquaLush does not have demanding performance, availability, fault tolerance, or security requirements. Hence, the AquaLush utility tree includes the five profiles in the former group but not the four in the latter group. Depending on product size and the effort developers can afford for architectural evaluation, there could be many or relatively few profiles.

Next, scenarios are filled in below each profile name. This is best done as a four-step process applied to each profile:

1. Brainstorm scenarios for the profile, recording each suggestion using a brief descriptive phrase. The SRS and various models (especially the use case model) are valuable resources for this exercise.
2. Rationalize the list, combining similar scenarios, clarifying vague scenarios, and perhaps arranging them into groups.
3. Weight each scenario according to its importance or likelihood. Weightings can be numeric (a five-point scale) or qualitative (Low, Medium, and High).

4. Eliminate the lowest-rated scenarios until the profile contains from 3 to 10 scenarios. More important profile areas should contain more scenarios and less important areas fewer.

This completes the utility tree construction process. A finished AquaLush utility tree appears in Figure 10-2-2. Scenario weights (on a qualitative scale) appear in parentheses after the scenario names.

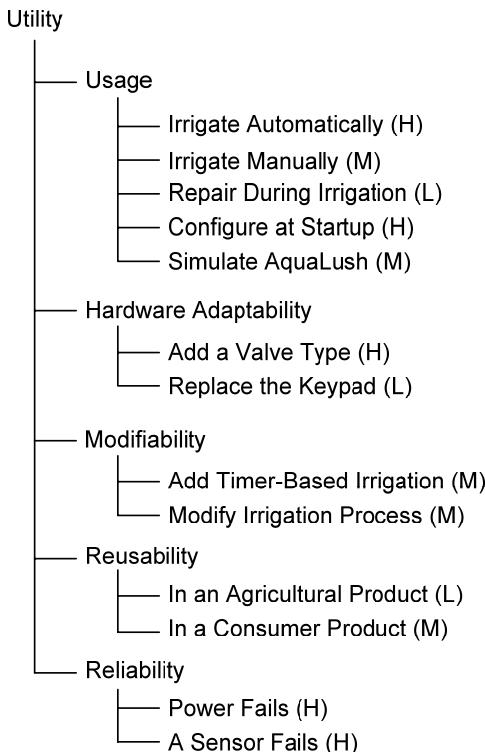


Figure 10-2-2 AquaLush Utility Tree

Scenario generation is completed by writing scenario descriptions for the scenarios at the leaves of the utility tree. The AquaLush utility tree and scenario descriptions appear in Appendix B.

Evaluating with Scenarios

Architectural alternatives are evaluated by “walking through” each scenario. In other words, the interaction in a scenario is applied to a design alternative. The goal is to see how well the design alternative supports the scenario. To illustrate, we will walk through the *Add a Valve Type* scenario from the *Hardware Adaptability* profile on the two design alternatives generated in the last section. This scenario appears in Figure 10-2-1, and the design alternatives are presented in Figure 10-1-4 and Table 10-1-5, and Figure 10-1-9 and Table 10-1-10.

The design alternative generated by determining functional components (in Figure 10-1-4 and Table 10-1-5) controls valves from the **Irrigation Control** component, so it must be changed to accommodate a new valve. The **Startup** component must also be changed to recognize a new valve type in the **AquaLush Configuration** data store and to communicate this information to the **Irrigation Control** component. When the developer modifies the design and code to add a new valve type, he or she will have to change two architectural components, and the changes could be quite extensive (particularly in the **Irrigation Control** component, where new valve control code must be added). It seems unlikely that this change could be made in two days as the scenario requires, so this design does not support this scenario well.

The design alternative generated by determining components based on quality attributes has a **Device Interface** module that contains virtual valve devices. Our developer must write a new virtual valve module conforming to the valve device interface, which should not be a very hard job. This architecture does not specify how the **Irrigation Control** module is configured to use devices (a missing element in the architecture), but presumably it is a fairly straightforward job, so the developer ought to be able to complete this work in two days, as envisioned in the scenario. Hence, this architecture supports this scenario well.

All scenarios can be considered for each design alternative, with results recorded in a list. This data is the input to an architectural alternative selection activity, discussed below.

Evaluating with Prototypes

Prototypes are working models of part or all of a product, as discussed in Chapter 5. Prototypes may be used to evaluate architectural alternatives by writing programs that explore some aspect of one or more alternatives. This is expensive, so it is done only to explore areas that the architects feel unable to judge in any other way. For example, suppose that architects must decide whether to replicate a component to achieve performance goals. If a single copy of the component can do the work fast enough, there is no need for replication and the result will be a simpler design with fewer and simpler components. However, if the single component is not fast enough, then two or more copies may be required to complete the job, even though that would require more complicated components to distribute the work and perhaps a coordination component to mediate interaction. In this case, the architects may build a prototype to see how fast the computation can be performed by a single component to get data on which to base this crucial design decision.

Scenario walkthroughs may provide the impetus for making prototypes: Designers may decide when walking through scenarios that they are unable to determine how well an alternative supports a scenario without building a prototype to try it.

Using Design Principles in Architectural Evaluation

Scenarios and prototypes are mainly aimed at addressing the basic design principles of feasibility, adequacy, economy, and changeability. How do the constructive design principles come into the architectural evaluation process?

The constructive principles are implicitly at work when scenarios and prototypes are used to evaluate software architectures. For example, scenarios in maintainability profiles can be supported only by architectures whose components are modular (in other words, are small, cohesive, loosely coupled; hide information; and have restricted privileges). Other constructive principles come into play when other quality attributes are considered. Thus, all the engineering design principles discussed in Chapter 8 are involved in architectural design evaluation.

Alternative Selection Techniques

There are two main techniques for selecting among architectural design alternatives:

Pros and Cons—Once designers have applied scenarios to architectural alternatives, and perhaps developed prototypes, each alternative's strengths and weaknesses may be apparent. It may be obvious which alternative is best, or that no alternative is acceptable and new alternatives must be generated. Thus, one selection technique is to weigh each alternative's pros and cons and decide how to proceed. In AquaLush, it is clear after running through the scenarios that both alternatives are seriously deficient and that their best features must be combined into a better design.

Multi-Dimensional Ranking—It may happen that several design alternatives are quite closely matched and that there are no obvious ways to improve them. Designers may have a hard time combining the strengths and weaknesses found walking through many scenarios into an overall rating for each design alternative. In this case, a scoring matrix (introduced in Chapter 5) can be used to make a selection.

A **scoring matrix** is a table showing design alternatives in the columns and weighted selection criteria in the rows that produces an overall score for each design alternative. As an illustration, we will construct a scoring matrix for the AquaLush architectural alternatives generated in the last section using the scenarios in the utility tree in Figure 10-2-2.

The first step in creating the scoring matrix is to set up a table (preferably in a spreadsheet) with the design alternatives in the columns and the scenarios in the rows. We can make scoring matrices for all scenarios or just those in a profile. For this example we include all scenarios.

Next, the scenario weights must be quantified and normalized. *Quantifying* the weights means making them into numbers. In this case, we convert High to 3, Medium to 2, and Low to 1. *Normalizing* them means converting each numeric ranking to a value between 0 and 1, such that all weights sum to 1. This is done by summing the un-normalized weights and then dividing each un-normalized weight by this sum. In this example the

weights assigned to the scenarios sum to 28, so each weight is normalized by dividing it by 28. This results in normalized weights of 0.036 (1/28), 0.071 (2/28), and 0.107 (3/28). The normalized weights are shown in the second column of Table 10-2-3.

Next, each alternative is given a rating. We used a six-point scale to rate how well an alternative supports a scenario, with 0 meaning no support and 5 meaning excellent or complete support. Once these ratings are entered, scores for each alternative under a scenario are computed by multiplying the scenario weight by the rating for that alternative. The overall score for each alternative is the sum of the scores for each scenario. The finished scoring matrix is shown in Table 10-2-3.

Scenarios		Functional Decomposition		Modular Decomposition	
Description	Weight	Rating	Score	Rating	Score
Usage					
Irrigate Automatically	0.107	5	0.54	5	0.54
Irrigate Manually	0.071	5	0.36	5	0.36
Repair During Irrigation	0.036	5	0.18	3	0.11
Configure at Startup	0.107	5	0.54	0	0.00
Simulate AquaLush	0.071	2	0.14	5	0.36
Hardware Adaptability					
Add a Valve Type	0.107	2	0.21	5	0.54
Replace the Keypad	0.036	1	0.04	5	0.18
Modifiability					
Add Timer-Based Irrigation	0.071	2	0.14	5	0.36
Modify Irrigation Process	0.071	3	0.21	4	0.29
Reusability					
In an Agricultural Product	0.036	1	0.04	4	0.14
In a Consumer Product	0.071	2	0.14	5	0.36
Reliability					
Power Fails	0.107	5	0.54	0	0.00
A Sensor Fails	0.107	4	0.43	4	0.43
Total Score			3.50		3.64

Table 10-2-3 AquaLush Architectural Alternative Scoring Matrix

The modular decomposition alternative has a higher score. This is expected because it generally does better in the quality attribute profiles containing most of the scenarios. We might decide based on this scoring matrix to add features to the modular decomposition alternative to create a better design. This can be done by incorporating features from the functional decomposition alternative that better support the usage scenarios.

Heuristics Summary

Figure 10-2-4 summarizes the scenario description heuristics discussed in this section.

- Label each scenario with a descriptive phrase.
- Write simple declarative sentences in the active voice.
- Write scenario descriptions in three parts.
- Make a principal in the interaction the subject of each sentence describing the activity flow.
- Describe the initial and final states of both the product and its environment.
- State target outcome measures whenever possible.
- Proofread the description.

Figure 10-2-4 Scenario Description Heuristics

Section Summary

- Software architectures are evaluated based on the likelihood that the software product they specify will satisfy its requirements when implemented.
- A **scenario** is an interaction between a product and particular individuals. Scenarios are stated in **scenario descriptions**.
- A **profile** is a set of scenarios used to evaluate whether a product is likely to meet a set of requirements.
- An architecture can be evaluated in some area by walking through the scenarios in a profile for that area to see how well the architecture supports the scenarios.
- Architectural alternatives can be evaluated by constructing prototypes to explore their characteristics.
- Alternatives can be selected by considering their pros and cons or by making a scoring matrix.

Review Quiz 10.2

1. Why are scenarios best described in three parts?
2. What is a utility tree?
3. What role do scenarios play in a scoring matrix for architectural design alternatives?

10.3 Finalizing Software Architectures

SAD Quality Characteristics

The last step in the architectural design process is to finalize the software architecture. This activity is a final check to ensure that the architecture is of high quality and that it is properly recorded in the software architecture document. A good SAD has the following characteristics:

Feasibility—A design is **feasible** if it can be built. The software architects must investigate their design thoroughly to ensure that it can be implemented.

Adequacy—An **adequate** software architecture is one that specifies a program that, when built, can meet its requirements subject to constraints. Adequacy should be established during the architectural design process but checked again during design finalization.

Well-Formedness—A SAD is **well formed** if its notations are used properly.

Completeness—A SAD is **complete** if it includes all required sections (assuming it conforms to some template), contains models needed to explain the design, specifies all important architectural component characteristics, and specifies all important relationships and interactions between architectural components.

Clarity—A SAD can be well formed and complete but still very hard to understand. Designers should check that the specifications in the SAD are **clear**; that is, understandable to someone familiar with the problem and the modeling notations.

Consistency—A set of design specifications is **consistent** if a single program can satisfy them all. The SAD should not specify that architectural components have incompatible properties or impossible relationships.

Design Reviews

Product design reviews were discussed in Chapter 5. Software architecture reviews are very similar. In this section we summarize reviews and then consider the active design review, a kind of review especially for engineering design.

A review is defined as follows.

A **review** is an examination and evaluation of a work product or process by qualified individuals or teams.

A wide range of activities qualify as design reviews:

Desk Check—A **desk check** is an assessment of a design by the designer. Designers should always look back over their work. Desk checks have been shown to be more effective when guided by a checklist of mistakes the designer is prone to making.

Walkthrough—A **walkthrough** is an informal presentation to a team of reviewers. Typically, designers show viewgraphs or read through the SAD, and reviewers ask questions and make comments and suggestions for improvement.

Inspection—An **inspection** is a formal review by a trained inspection team. Inspections follow a strict process, with each participant playing a particular role. Inspections are always driven by checklists. Studies show that inspections are the most effective review technique. Inspections are discussed in detail in Chapter 5, and a sample design inspection checklist is provided in Figure 10-3-1.

Audit—An **audit** is a review conducted by experts who are not members of the design team. Audits bring a fresh perspective to the project and are especially good at finding faults that the design team is too close to the work to see, such as unclear or incomplete specifications. Software architecture audits are done routinely in many large development organizations.

Active Reviews—An **active review** is an examination by experts who answer questions about specific aspects of the design. Active design reviews have been proposed as especially appropriate for engineering design.

A Design Inspection Checklist

Before taking a closer look at active design reviews, we present a sample design inspection checklist. The checklist in Figure 10-3-1 includes items based on the DeSCRIPTR aspects of architectural specification discussed in Chapter 8.

- The notations used for each model are correct.
- Every required section of the SAD is present.
- The SAD specifies the program's main components.
- The SAD specifies the states and state transitions for all components with important states.
- The SAD specifies important or complex component collaborations.
- The SAD specifies each component's responsibilities.
- The SAD specifies each component's interface.
- The SAD specifies each component's important properties.
- The SAD specifies each component's important relationships to other components.
- The SAD clearly states the connections between different architectural models.
- The SAD states the rationale for all important design decisions.
- Each design rationale states the problem to be solved and the constraints on the designer.
- Each design rationale summarizes the major design alternatives and their evaluations.
- Each design rationale explains why the final design was selected.
- All specifications are clear.
- No specification contradicts any other specification in the SAD.

Figure 10-3-1 An Architecture Inspection Checklist

As discussed in Chapter 5, inspection checklists should be tailored for each organization that uses them, and they should be modified over time so that they list the defects inspectors are most likely to find.

Active Design Reviews

Active design reviews were developed to remedy problems with traditional reviews. For example, reviewers are often asked to examine an entire document even though they are not competent to review everything in the

document. This wastes time because reviewers are not likely to find defects or think of improvements outside their areas of expertise. Another problem with traditional reviews is that reviewers are simply asked to check a document. This makes it too easy for reviewers to simply skim through the material without engaging it. Active design reviews remedy these and other problems.

The active design review process is shown in Figure 10-3-2.

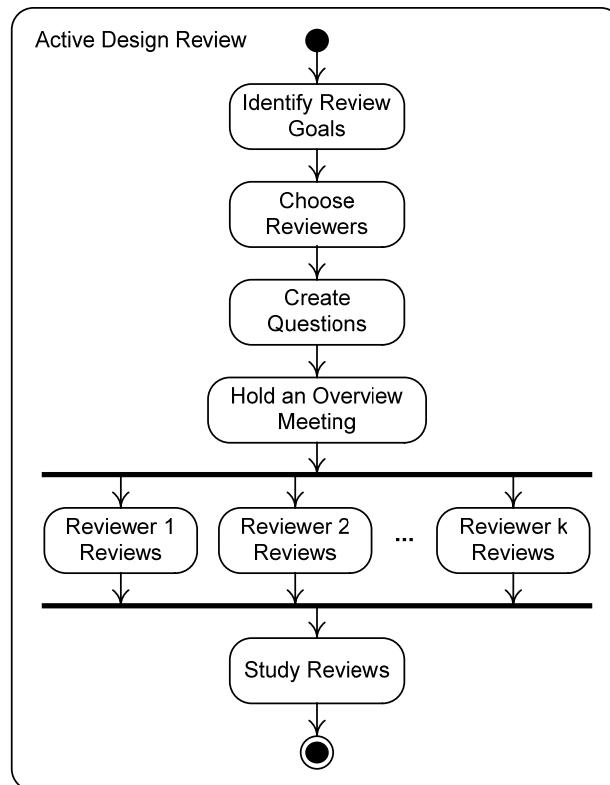


Figure 10-3-2 Active Design Review Process

The activities in this process comprise three phases. Review preparation phase activities are aimed at setting up the review; they are carried out by the design team. During the review performance phase, the reviewers work on their reviews concurrently. During the review completion phase, the design team digests the reviewers' reports and incorporates what they have learned into the design.

- | | |
|--------------------|--|
| Review Preparation | The review preparation phase begins with the activity Identify Review Goals , during which the designers choose a specific aspect of the software architecture that they would like to have reviewed. For example, the AquaLush designers might want reviewers to check whether the virtual device interfaces are reasonable given actual hardware characteristics, |
|--------------------|--|

whether the design is understandable by someone outside the design team, or whether the irrigation requirements are all accounted for in the design. Restricting review scope keeps reviewers from being overwhelmed.

The next activity is **Choose Reviewers**. The designers select two to four individuals with the knowledge and skills needed for the review and obtain their agreement to do a review.

Next, the designers must **Create Questions**. Rather than simply being asked to look over the architecture, reviewers are given a set of questions to answer based on the SAD. This is the sense in which these reviews are *active*: Reviewers must do something in addition to passively reading the document. The questions should force reviewers to understand the aspect of the design under review and make sure it is satisfactory. For example, in a review of whether the AquaLush virtual device interfaces are adequate, reviewers might be asked to outline how they would write a virtual device for a particular valve or sensor. In a review of whether the SAD is clear, reviewers might be asked to write an account of how a few scenarios are accommodated in the architecture. In a review of whether the irrigation requirements are all accounted for, each reviewer might be given several requirements and asked to explain how each one is satisfied by the software architecture. Creating the questions completes the review preparation phase.

Review Performance	The review performance phase begins when the designers and reviewers Hold an Overview Meeting . The designers sketch the architecture, explain how the part the reviewers are considering fits in, explain the ground rules for the review, and set deadlines for review completion.
Review Completion	The reviewers then work up their reviews on their own. They may meet with the designers to ask questions about the design, or call or email questions. Eventually, they complete their work and deliver it to the designers. The designers Study Reviews during the review completion phase. The designers may meet with the reviewers to ask them questions about their reviews, or call or email questions. The designers may meet with reviewers individually or in groups, whichever seems most efficient. Usually, group meetings are not needed. When the process is done, designers have the information they need to improve the software architecture or the SAD. In summary, active design reviews are a technique that software architects can use to obtain focused and thoughtful feedback about their design and its documentation. Active design reviews avoid the inefficiency of traditional reviews by selecting reviewers with the right knowledge and skills, having them focus on specific parts of the design, asking them probing questions, and streamlining communications between reviewers and designers.

Continuous Review Although we have discussed reviews in the context of architectural design finalization, reviews can and should be used during the entire architectural design process to help catch defects as soon as possible. For example, walkthroughs with the design team are useful when design alternatives are being evaluated. Audits are useful to check the feasibility and adequacy of an architectural decomposition. Active design reviews can be conducted when a portion of the architecture has been completed.

Section Summary

- A SAD should specify a **feasible** and **adequate** software architecture using **well-formed** models, and it should be **complete**, **clear**, and **consistent**.
- Many review techniques can be applied to a SAD, including **desk checks**, **walkthroughs**, **inspections**, **audits**, and **active design reviews**.
- An **active design review** is an examination of a design by experts who answer questions about it.
- A SAD should be reviewed during the design process (to catch defects early), and during design finalization.

Review Quiz 10.3

1. Compare and contrast active design reviews and design inspections.
2. Name three checklist items that might be used in a SAD desk check or inspection.
3. What activity requires the most effort in preparing for an active design review?

Chapter 10 Further Reading

Section 10.1

Bosch [2000] discusses generating architectural alternatives from functional requirements and transforming architectures to improve them. Bass, Clements, and Kazman [2003] suggest starting with quality attributes and using them to generate architectural alternatives. Britton et al. [1981] discuss device interface modules and virtual devices. McGrath [1984] discusses the relative effectiveness of individuals and teams in generating new ideas.

Section 10.2

Bosch [2000] and Bass, Clements, and Kazman [2003] discuss evaluating and selecting software architectures. [Clements, Kazman, and Klein 2002] is entirely about evaluating software architectures; it provides detailed steps for conducting several kinds of highly structured evaluations.

Section 10.3

Parnas and Weiss [1985] proposed active design reviews. See [Gilb and Graham 1993] for a thorough discussion of inspections.

Chapter 10 Exercises

The following product specifications are used in the exercises.

Computer Assignment System (CAS)

A group of system administrators must keep track of which computers are assigned to computer users in the community they support. The Computer Assignment System (CAS) must aid system administrators by keeping track

of computers, computer users, and assignments of computers to users. Developers in the same enterprise will implement CAS.

CAS has the following functional and data requirements:

- CAS must maintain the location, components, operational status, purchase date, purchase price, and assignment of every computer in the organization.
- CAS must maintain the name, location, and title of every computer user in the organization.
- CAS must support queries about individual computers, users, and assignments.
- CAS must generate summary reports about all users, computers, and assignments.
- CAS must generate reports about computers, their costs, and their purchase dates so that accountants can compute capital expenditures, depreciation, and so forth.

CAS has the following non-functional requirements:

- Three people must develop CAS in three months or fewer.
- CAS must require no more than one person-week per year for maintenance.
- CAS users must take no more than one minute per transaction, on average, to maintain this information.
- CAS must maintain accurate data.

Fingerprint Access System (FAS)

The Fingerprint Access System (FAS) must control access to secure facilities using fingerprints. FAS reads fingerprint scanners and controls entry and exit gates. Personnel wishing to enter or leave a facility must place their fingers on the fingerprint reader at a gate, and the gate will be unlocked to let them through.

FAS has the following functional and data requirements:

- FAS must match scanned fingerprints against a fingerprint database and unlock gates when it identifies legitimate commuters.
- Authorized personnel must be able to unlock individual gates.
- Authorized personnel must be able to record that individuals unknown to FAS (visitors) have entered or left the building.
- FAS must log all entries, exits, and failed or aborted attempts at entry or exit.
- FAS must support queries by authorized personnel about particular individuals' activities.
- FAS must provide authorized personnel with reports about all current facility occupants.

- Authorized personnel must be able to unlock all gates during an emergency.

FAS has the following non-functional requirements:

- FAS must unlock gates in no more than two seconds, on average, after a successful fingerprint scan.
- The fingerprint recognition error rate (false positives versus false negatives) must be adjustable.
- FAS must be configurable at installation and easily reconfigurable in operation.
- FAS must support a variety of brands of fingerprint scanners and gates.
- FAS must be available at least 23.5 hours per day.
- FAS must deny unauthorized personnel access to its database and control functions.

Section 10.1

1. *Fill in the blanks:* Architectural decomposition can be represented in several ways. One way is to show the _____ active during program execution. Another way is to show the _____ into which the source code is divided at design time. There is a _____ between these two representations, but they need not be identical.
- AquaLush 2. Figure 10-1-6 shows a functional decomposition of the Web-based AquaLush simulator. Augment this diagram with a table listing component responsibilities.
3. Figure 10-1-6 shows a functional decomposition of the Web-based AquaLush simulator. This decomposition lacks any mechanism to configure AquaLush at program startup. Add components and connectors to this diagram to include this missing function in the architecture.
4. Augment the diagram you made in the last exercise with a table listing component responsibilities.
5. It is noted in the text that Figure 10-1-4 shows functional components while Figure 10-1-9 shows program modules. Draw a diagram to model program modules for the architecture depicted in Figure 10-1-4 and a diagram to show functional components for the architecture depicted in Figure 10-1-9.
- CAS 6. Draft an architectural decomposition of the Computer Assignment System by determining functional components. Document your design using a box-and-line diagram and brief component responsibility descriptions.
7. Draft an architectural decomposition of the Computer Assignment System by determining components based on quality attributes.

Document your design using a box-and-line diagram and brief component responsibility descriptions.

8. Compare and contrast the architectural alternatives you generated in the last two exercises. Combine the best features of each of your alternatives in another design.
- FAS 9. Draft an architectural decomposition of the Fingerprint Access System by determining functional components. Document your design using a box-and-line diagram and brief component responsibility descriptions.
10. Draft an architectural decomposition of the Fingerprint Access System by determining components based on quality attributes. Document your design using a box-and-line diagram and brief component responsibility descriptions.
11. Compare and contrast the architectural alternatives you generated in the last two exercises. Combine the best feature of each of your alternatives in another design.

- Section 10.2** 12. *Fill in the blanks:* In a _____, the root of the tree is labeled _____, children of the root are the names of _____, and the leaves designate _____. These trees help generate _____ used to evaluate architectural alternatives.
- AquaLush 13. Propose a process that a team might use to generate scenarios, taking advantage of the fact that designers can work both alone and in a team.
14. Propose three additional AquaLush scenarios, add them to the appropriate spots in the utility tree in Figure 10-2-2, and write scenario descriptions for them.
15. Use a spreadsheet and modify the scoring matrix in Table 10-2-3 to include the scenarios you concocted in the previous exercise. Do the relative overall design alternative scores change?
- CAS 16. Construct a utility tree for the Computer Assignment System.
17. Write descriptions for five scenarios from the utility tree you constructed in the previous exercise.
18. Use a spreadsheet to construct a scoring matrix to rank the design alternatives you compiled in exercises 6 and 7. Use it to select the best design alternative.
- FAS 19. Construct a utility tree for the Fingerprint Access System.
20. Write descriptions for five scenarios from the utility tree you constructed in the previous exercise.
21. Use a spreadsheet to construct a scoring matrix to rank the design alternatives you compiled in exercises 9 and 10. Use it to select the best design alternative.
- Section 10.3** 22. *Fill in the blanks:* A SAD that correctly uses notation is _____. A set of design specifications that can all be satisfied by a single program is _____. A software

architecture that specifies a program that can meet all its requirements is said to be _____ . If the SAD is easy to understand, it is _____ .

- CAS** 23. Identify a goal for an active design review of the draft architectural decomposition you created for the Computer Assignment System in exercise 8. Write at least one question for reviewers to achieve this review goal.
- FAS** 24. Identify a goal for an active design review of the draft architectural decomposition you created for the Fingerprint Access System in exercise 11. Write at least one question for reviewers to achieve this review goal.
- Research Project** 25. Research architectural evaluation techniques and write a report detailing the one that you find the most useful or interesting.
- Team Projects** 26. Form a team of three. Individually generate at least two alternative architectural decompositions for the Fingerprint Access System. Then meet as a team, consider your alternatives, and try to come up with a few more.
- 27. Form a team of three. Individually generate a utility tree for the Fingerprint Access System. Then meet as a team and revise your utility tree to include the best profiles and scenarios from all team members. Make sure that the tree does not get too big.
- AquaLush** 28. Form a team of five. Inspect the AquaLush software architecture document in Appendix B. Follow the process discussed in Chapter 5 and use the inspection checklist items listed in Figure 10-3-1.

Chapter 10 Review Quiz Answers

Review Quiz 10.1

- An architectural style is a general example of program or system component types and their interactions that can be emulated in particular program architectures.
- When determining architectural components based on quality attributes, designers begin by considering non-functional requirements. After a decomposition is generated, functional and data requirements are used to adjust the decomposition to account for missing features or capabilities.
- An ideal virtual device should do exactly one job, have a simple and consistent interface meeting the needs of the rest of the program, be loosely coupled to the rest of the program, hide its implementation, and never change its interface.

Review Quiz 10.2

- Scenarios are best described in three parts because readers must understand the state of the program and its environment when the scenario begins, the interaction comprising the scenario, and the state of the program and its environment when the scenario is done. Scenario descriptions thus divide nicely into three parts: an initial state description, the interaction description, and a final state description.

2. A utility tree is a tree whose root is labeled “Utility,” whose first-level nodes (the children of the root) are labeled with profile names, and whose leaves are scenarios. Utility trees are a tool to help generate and organize scenarios.
3. Scenarios play the role of evaluative criteria in scoring matrices for architectural design alternatives. More specifically, scenario identifiers and weights head scoring matrix rows, and the cells in each row represent the ratings and weighted scores for how well each design alternative supports the scenario in that row.

**Review
Quiz 10.3**

1. In an active design review, the reviewers use the design document to answer specific questions. In a design inspection, inspectors check the design document against a checklist. In both cases reviewers use written documents to guide their work. However, the active design review questions are quite different from inspection checklist items. Active design review questions are formulated specifically to explore a particular aspect of the design at hand, while inspection checklists contain general items applicable to any design. The active design review and design inspection processes are also quite different. Active design reviewers work alone and usually deliver their results individually to the designers. Inspectors prepare alone but work as a team to conduct the review. Also, the inspection process is more formally defined than the active review process.
2. SAD design desk checks or inspection checklists can include items having to do with the quality of the design itself, such as checking whether functional, data, and non-functional requirements are satisfied by some aspect of the design, or whether the design is feasible. They may also contain design documentation quality checks, such as whether the design is clear, consistent, complete, and well formed. Examples in all these categories are shown in Figure 10-3-1.
3. The three active design review preparation activities are identifying review goals, choosing reviewers, and creating questions. Identifying review goals takes some thought, but generally designers are aware of design weak points that can benefit from review. Choosing reviewers is straightforward once review goals are identified: The reviewers should have adequate skills and knowledge in the area to be reviewed. Creating good questions is very difficult. They must be detailed and specific enough to force reviewers to examine the software architecture carefully but not require so much work that reviewers are unwilling or unable to answer them. Ideally, questions should be challenging and thought provoking so the reviewers are interested but still provide designers with useful feedback. Writing such questions is arduous and requires considerable effort.

11 Static Mid-Level Object-Oriented Design: Class Models

Chapter Objectives

This chapter is the first of several that discuss detailed design. It introduces the detailed design phase and its sub-phases, considers the relationship of detailed design to architectural design, and specifies the contents of the design document. The context of this discussion is indicated in the diagram in Figure 11-O-1.

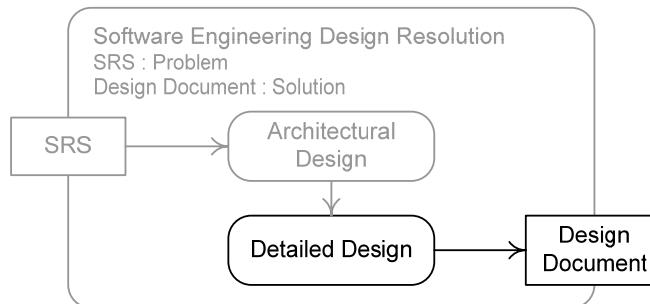


Figure 11-O-1 Detailed Design and Its Output

In this chapter we also cover advanced features of class diagrams, completing the discussion of UML class diagrams begun in Chapter 7. Finally, we consider techniques and heuristics for generating static design models.

By the end of this chapter you will be able to

- Describe and explain the detailed design process, including the distinction between mid-level and low-level design;
- List the contents of a detailed design document;
- Read and write UML class diagrams incorporating advanced features;
- Produce draft design class models using a generational technique and a transformational technique;
- Explain what component responsibilities are and how they can be used to make better designs; and
- Explain what inheritance and delegation are and how they can be used to make better designs.

Chapter Contents

- 11.1 Introduction to Detailed Design
- 11.2 Advanced UML Class Diagrams
- 11.3 Drafting a Class Model
- 11.4 Static Modeling Heuristics

11.1 Introduction to Detailed Design

The Context of Detailed Design

The boundary between architectural design and detailed design is fuzzy. Every program has a software architecture, but its level of abstraction is highly variable, depending mostly on program size. Very large programs have many large sub-systems described during architectural design at high levels of abstraction. Small programs, on the other hand, have a few small components described in some detail during architectural design. There is no fixed boundary dividing architectural from detailed design.

The boundary between detailed design and implementation is blurry as well. During detailed design, designers specify class responsibilities, class attributes, class operations, object interactions, object states, state changes, processes, and process synchronization. Programmers choose control structures, program entity names, primitive types, parameter passing mechanisms, and programming idioms. But either party can make (or negotiate) a variety of low-level design decisions involving packaging, visibility, algorithms, and data structures.

There is no reason for strict boundaries between architectural design, detailed design, and programming. Design decisions at various levels of abstraction are deeply interconnected, so there must be interplay among these three design activities. In practice, there is so much iteration between these activities that it is impossible to separate them cleanly. We make the separation as a logical and pedagogical device to help you understand software design and to emphasize the top-down trend of design decision-making and specification from more to less abstract.

The Scope of Detailed Design

Detailed design covers a tremendous range of levels of abstraction and system representations, and it encompasses a great mass of specifications. At the highest levels of abstraction, detailed design may be like architectural design in specifying the main parts of major sub-systems, including their states and transitions, collaborations, responsibilities, interfaces, properties, and relationships with other components. At the lowest levels of abstraction, detailed design may specify implementation details down to pseudocode and data formats.

Detailed design representations include static models, such as class diagrams, operation specifications, and data structure diagrams, as well as dynamic models, such as interaction diagrams, state diagrams, and pseudocode. We present these notations in the next several chapters.

In terms of sheer bulk, most of a software design specification is created during the detailed design phase. Detailed design is the largest, most complex, and longest software engineering design phase.

Because of its size, complexity, and range of abstraction and representation, it is helpful to further divide detailed design into two

stages: *mid-level design* and *low-level design* (*high-level design* is architectural design). We sketch these in turn below.

- Mid-Level Design** Detailed design begins with a software architecture and fills in details of the architectural components. Decomposing the major components of software architecture yields medium-sized components, such as compilation units or classes. This activity we call mid-level design.

Mid-level design is the activity of specifying software at the level of medium-sized components, such as compilation units or classes, and their properties, relationships, and interactions.

Mid-level design must specify both static and dynamic aspects of components. Specifications include the DeSCRIPTR aspects discussed in Chapter 8:

Decomposition—Mid-level components are the parts comprising architectural components. These parts must be identified.

States and State Transitions—Some components have important states and state-based behavior. Designers must specify the states and state transitions of such components.

Collaborations—Designers must specify the dynamic flow of control and data among mid-level components enabling them to solve problems collaboratively. Component interactions include object message passing behavior, calling relationships between operations, data flow, and processes and process synchronization.

Responsibilities—Designers must specify mid-level component obligations to carry out tasks or maintain data.

Interfaces—Designers must describe the communication mechanisms mid-level components use when interacting. This includes interface syntax, semantics, and pragmatics.

Properties—Designers must state important mid-level component properties, usually having to do with quality attributes, such as security, reliability, and modifiability.

Relationships—Some component relationships are established during architectural design, but many are not. The most important of these established during mid-level design are inheritance relationships, interface realization and dependency relationships, and visibility and accessibility associations.

Design patterns (general solutions to common problems) are important guides during mid-level design. Part IV of this book discusses mid-level design patterns in detail.

- Low-Level Design** Low-level design fills the gap between mid-level design and programming, though it blends into the design activity that most often occurs during

programming. It thus comprises the area of overlap or fuzziness between detailed design and programming.

Low-level design is the activity of filling in small details at the lowest levels of abstraction.

Because it blends into programming, low-level design involves issues beyond DeSCRIPTR specifications involving coding and programming languages, the PAID specifications:

Packaging—Decisions must be made and documented about how to bundle code into various compilation units, libraries, packages, and so forth.

Algorithms—Certain algorithms may be chosen depending on consideration of time and space efficiency, ease of implementation and maintenance, and programming language support. Sometimes designers may specify the processing steps of individual operations.

Implementation—Designers must resolve implementation issues, notably the visibility and accessibility of various entities, and how to realize associations.

Data Structures and Types—Designers may decide how to store data to meet product requirements, which involves selecting data structures and choosing variable data types.

Patterns are also important in low-level design, where they are known as *programming idioms* and *data structures and algorithms*. These topics are out of this book’s scope, but other low-level design topics are not—they are discussed in Chapter 14.

The Detailed Design Process

The detailed design process is part of the software engineering design process discussed in Chapters 2 and 8. Figure 11-1-1 on page 322 shows this process.

Although designers do not follow rigid paths in problem solving, the general trend in detailed design is top down; that is, from mid-level design to low-level design. During the detailed design analysis step, designers study the SRS and the SAD as a starting point for refining the architectural components specified in the SAD. Various models may be used for this task, including those discussed in Chapters 2, 5, 6, 7, and 9. We study detailed design modeling notations in this and the next three chapters.

Generating and improving detailed design alternatives proceeds much as discussed in Chapter 10 for architectural design: Designers can work out decompositions based on program functions or quality attributes, a conceptual model, the design of a similar program, or design patterns. The last section of this chapter considers how to generate a mid-level class model from a conceptual model, and Part IV of the book covers design patterns.

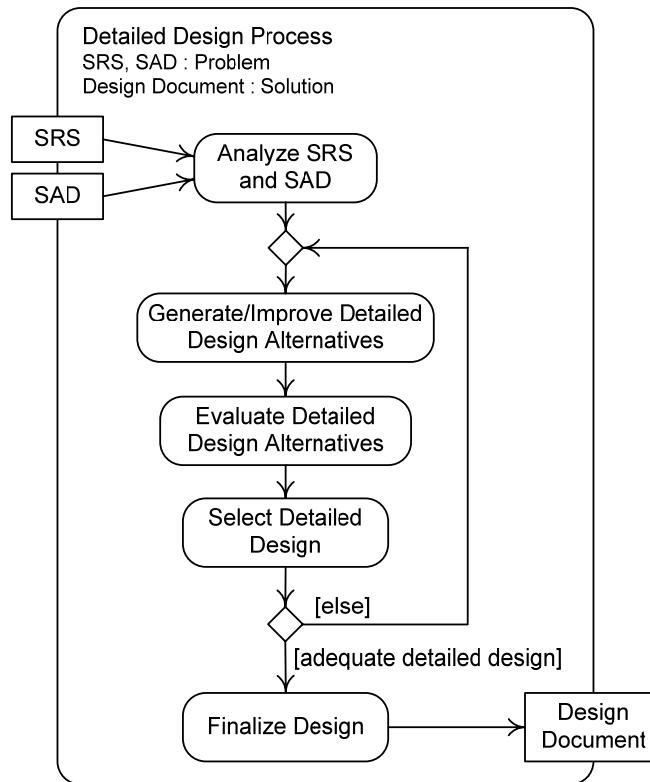


Figure 11-1-1 Detailed Design Process

Detailed design alternatives are evaluated by considering constructive design principles (discussed in Chapter 8), by modeling collaborations (discussed in Chapter 12), and by prototyping alternatives to gain information. Designers select alternatives by considering their pros and cons or, in complicated cases, by using scoring matrices, as discussed in Chapters 5 and 10. Detailed designs are finalized through design reviews, such as design inspections and active reviews, also discussed in Chapters 5 and 10.

Design Document Contents

The output of the detailed design process is a design document. A **design document** is a complete engineering design specification. It has two parts: a **software architecture document (SAD)** and a **detailed design document (DDD)**. The SAD specifies a program's software architecture (discussed in Chapter 9) and the DDD specifies a program's detailed design.

There is no standard template for a DDD. Different design methods advocate various sorts of detailed design documents. The template shown in Figure 11-1-2 is quite general and is adequate for most detailed design projects.

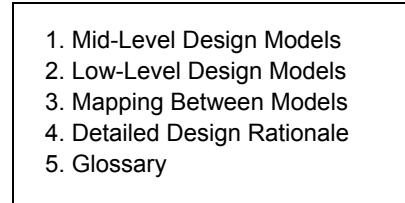


Figure 11-1-2 Detailed Design Document Template

This template has the following sections:

Mid-Level Design Models—This section presents the mid-level design using a variety of models to represent different aspects or views.

Low-Level Design Models—This section documents low-level design specifications such as algorithms, data structures and types, operation specifications, and so forth.

Mapping Between Models—This section uses tables and textual explanations to connect the elements of various design models used in the DDD and the SAD.

Detailed Design Rationale—This section justifies important detailed design decisions. Only decisions that are central to satisfying requirements, are puzzling or unusual, or were difficult to make should be discussed. As with any design rationale, designers should explain the factors affecting the decision, the design alternatives considered, design alternative evaluations, and the argument supporting the final decision.

Glossary—This section catalogs terms and acronyms used in the document that readers may not know.

The DeSCRIPTR-PAID specifications appear in sections one and two of this template.

This template is designed to complement the SAD template presented in Chapter 9 so that when the two are combined they form a coherent and complete design document. Appendix B contains the AquaLush design document, illustrating the SAD and DDD templates.

Section Summary

- The boundaries between architectural design, detailed design, and programming are fuzzy.
- Detailed design covers such a range of levels of abstraction and means of representation and is so large that it helps to divide it into two stages: mid-level design and low-level design.
- **Mid-level design** is the activity of specifying medium-sized program components, such as compilation units and classes, and their properties, relationships, and interactions.
- **Low-level design** is the activity of specifying details at the lowest levels of abstraction; low-level design blends into programming.

- A **design document** is a complete engineering design specification. It contains a **detailed design document (DDD)** that specifies a program's detailed design.

Review
Quiz 11.1

1. What is the difference between mid-level and low-level design?
 2. What does DeSCRIPTR-PAID stand for and what does it describe?
 3. What should a detailed design document (DDD) contain?
-

11.2 Advanced UML Class Diagrams

Generalization

An essential concept in the object-oriented paradigm is **inheritance**, the ability of one class to assume all the attributes and operations of another, its *super-class*, along with its own additional attributes and operations. UML broadens the notion of inheritance to include other entities besides classes and calls this relationship **generalization**. We will focus on the generalization relation between classes; that is, inheritance.

The UML *generalization connector* is a solid line from the specialized element (the child) to a hollow triangle attached at its apex to the general element (the parent). Lines from child classes can share a triangle or have their own triangles. The diagram in Figure 11-2-1 illustrates this notation.

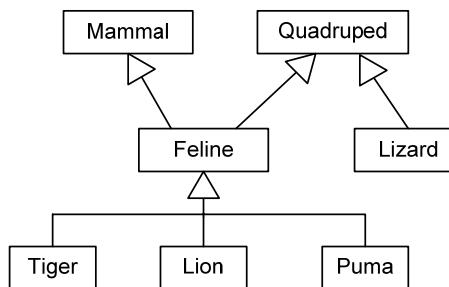


Figure 11-2-1 Generalization Relationships

UML does not rule out multiple inheritance, so a child may have several parents, as does **Feline** in Figure 11-2-1.

Note that generalization is a relation between *classes*, and a particular use of the generalization connector shows that two classes participate in this relation. This contrasts with *associations*, which do not relate the classes they connect, but rather represent relations on sets of connected class *instances*. A generalization connector is more like a link line between objects than an association line between classes.

Relations on sets of instances may contain various ordered pairs. For example, a single instance in one set may be related to zero, one, several, or many instances in the other set. Thus, associations have *multiplicity*.

However, the generalization connector always indicates that two particular classes participate in the generalization relation, as a link line shows that two objects participate in a particular relation. Hence, there is never a question of multiplicity with generalizations.

Because there are many different relations on sets of instances that might be modeled, there are many different kinds of associations. Consequently, associations may have names and possibly rolenames. But because there is only one generalization relation, there is no need to name it on the diagram. Furthermore, the roles are clear (parent and child), and the triangle indicates which is which, so no rolenames are needed. The upshot is the following heuristic:

Never place a name, rolenames, or multiplicities on a generalization connector.

Abstract Classes

An **abstract operation** is an operation without a body; that is, an operation with a signature but no procedural specification of the computation that the operation carries out when it is called. In contrast, a **concrete operation** has a body. A concrete operation can be called, but an abstract operation cannot because it has no body to execute.

An **abstract class** is a class that cannot be instantiated. Any class with at least one abstract operation *must* be abstract because if it were instantiated, then a client could call an abstract operation. A class that is not abstract is a **concrete class**.

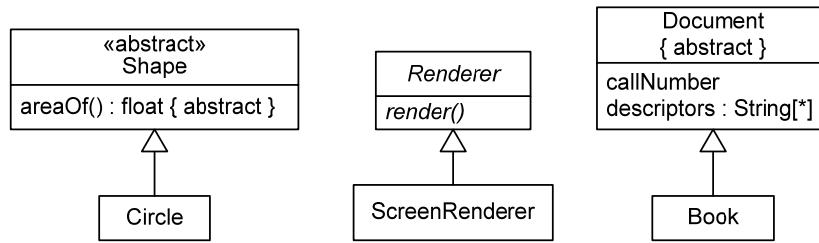
Abstract classes are useful only as super-classes specialized by concrete classes that implement their inherited abstract operations. But they are very useful in this respect because they force all their concrete descendants to implement certain operations. For example, an abstract **Shape** class might have an abstract **areaOf()** operation. Area computations differ too much for the **Shape** class to have a reasonable implementation of the **areaOf()** operation, but by having an abstract **areaOf()** operation, the **Shape** class ensures that all its descendent classes (such as **Circle**, **Square**, or **Triangle**) have an **areaOf()** operation.

Abstract classes may have attributes and both concrete and abstract operations. A class may still be abstract even if it has no abstract operations when, for some reason, it is not supposed to be instantiated.

In UML, operations and classes with italicized names are abstract. Alternatively, abstract classes can be denoted with an **«abstract»** stereotype, or abstract classes and operations can have an **{abstract}** property.

Stereotypes and properties are especially useful for marking operations and classes in handwritten diagrams because it is hard to recognize handwritten italics. The diagram in Figure 11-2-2 on page 326 shows several ways to represent abstract classes and operations in UML class diagrams.

It is best to use italics rather than properties and stereotypes in typed diagrams because it takes less space. It is also a good idea not to mix conventions for marking abstract entities in the same diagram.

**Figure 11-2-2 Showing Abstract Operations and Classes****Interfaces**

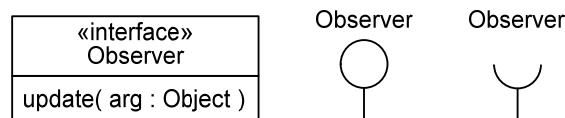
Recall from our discussion of component diagrams in Chapter 9 that a UML **interface** is a named collection of public attributes and abstract operations. **Features** are operations and attributes. Interfaces are important because they specify a bundle of features without specifying its implementation. Separating the feature specification from its implementation means that classes can be connected to features they may supply or need while remaining decoupled from specific classes.

UML distinguishes two kinds of interfaces depending on the connection between the interface and a class or component:

Provided—**Provided interfaces** are interfaces realized by a class or component. A class or component *realizes* an interface if it includes the interface's attributes and implements its operations. Interfaces provided by a class or component specify the features that it supplies.

Required—**Required interfaces** are interfaces upon which a class or component depends. A class or component *depends* on an interface when a change in the interface may affect the class or component. The interfaces required by a class or component specify the features it needs.

Interfaces can be represented in UML in three ways: stereotyped class symbols, interface ball or lollipop symbols, and interface socket symbols. These are shown in Figure 11-2-3.

**Figure 11-2-3 Three Ways to Represent Interfaces**

The stereotyped class symbol can show interface features explicitly in attributes and operations compartments. For example, the class symbol shows that the **Observer** interface has an `update()` operation. The ball and socket symbols show only the interface's name.

All the operations in an interface must be abstract, but we do not need to show them in italics; doing so is redundant, though allowed. The interface name should not be italicized using any of these symbols.

A provided interface can be shown in two ways. The first is to attach the stick of an interface lollipop symbol to a class or component. The second is to connect a stereotyped class symbol representing the interface to the providing class or component using a special *realization connector*. The realization connector combines the generalization connector and the dependency arrow: It has a hollow triangle attached to the interface with a dashed line leading to the realizing class. Figure 11-2-4 shows these conventions.

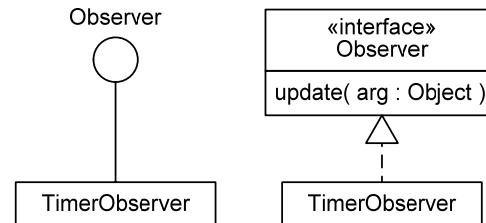


Figure 11-2-4 Two Provided Interface Notations

The main difference between these notations is that the version with the class symbol can provide more information about the interface.

A required interface can be shown in three ways. The first is to attach the stick of an interface socket symbol to a class or component. The second is to connect the class or component requiring the interface to an interface ball with a dependency arrow. The third way is to connect the class or component to a stereotyped class symbol with a dependency arrow. These alternatives are illustrated in Figure 11-2-5.

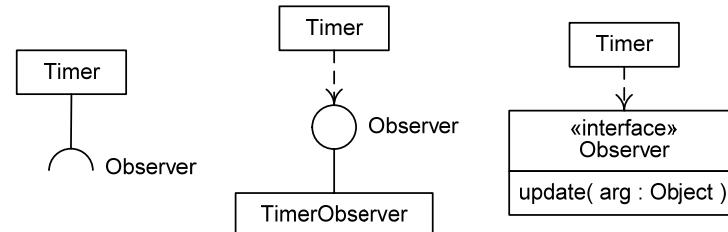
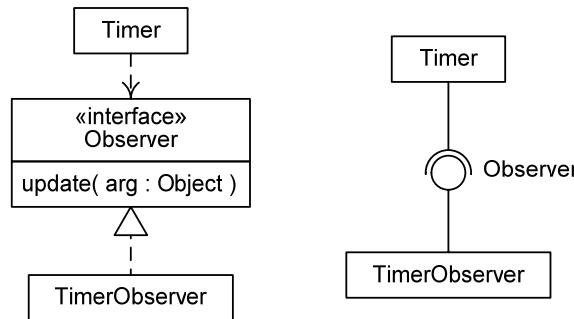


Figure 11-2-5 Three Required Interface Notations

All three models in the diagram specify that the **Timer** class requires the **Observer** interface. In addition, the middle model shows that the **TimerObserver** provides this interface. The relationship between classes or components that require and provide interfaces to one another can be shown in other ways by combining these notations, as exemplified in Figure 11-2-6.

**Figure 11-2-6 Two Module Assembly Notations**

These models show various ways to assemble modules using interfaces. In particular, the interlocking interface ball and socket symbols form an *assembly connector* between the Timer and TimerObserver classes.

Feature Properties and Visibility

We noted when introducing UML class diagrams in Chapter 7 that more could be specified about attributes and operations than was presented there. In Chapter 9 we mentioned that properties can be added to UML diagrams. In particular, an attribute or operation specification can include a property specification at its end. To illustrate, consider the example in Figure 11-2-7.

Order
<code>id : String { constant }</code>
<code>itemList : Item[*] { ordered }</code>

<code>addItem(item : LineItem) { synchronized }</code>
--

Figure 11-2-7 Attribute and Operation Properties

The `id` attribute has the `constant` property to specify that it cannot be changed, and the `itemList` attribute is `ordered`, indicating that it is an ordered collection. The `addItem()` operation is `synchronized`, meaning that it cannot execute without obtaining exclusive access to its host object.

Another item that can be included in UML feature specifications is attribute and operation visibility. The portion of a program text over which an entity can be accessed by name is its **visibility**. For example, a local variable declared in a method can be accessed by name inside the method, so the visibility of a local method variable is the body of the method.

UML provides four levels of visibility that can be applied to attributes and operations:

Public—A feature with **public visibility** is visible anywhere in the program that the class in which it appears is visible; it is denoted by the plus sign (+) in UML.

Package—A feature with **package visibility** is visible anywhere in the package containing the class in which it appears; denoted by the tilde (~).

Protected—A feature with **protected visibility** is visible only in the class in which it appears and all its sub-classes; denoted by the pound sign (#).

Private—A feature with **private visibility** is visible only in the class in which it appears; denoted by the minus sign (-).

Visibility markers appear before the attribute or operation name and may be suppressed. To illustrate, consider the class diagram in Figure 11-2-8.

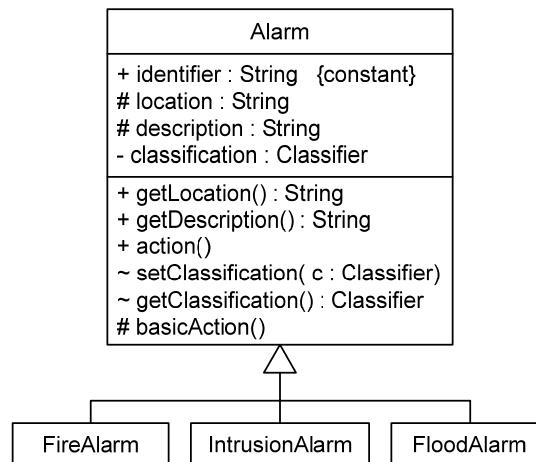


Figure 11-2-8 Feature Visibility

In this example the **Alarm** class has a public constant **identifier** attribute visible wherever the class is visible. Because this attribute is constant, there is no need to hide it inside the class. **Alarm** has protected **location** and **description** attributes visible only in **Alarm** and its sub-classes. These need to be modifiable in sub-classes but not accessible to clients. Finally, **Alarm** has a private **classification** attribute visible only in the **Alarm** class itself. This attribute may only be accessed through the **setClassification()** and **getClassification()** operations, even by sub-classes of the **Alarm** class.

The **Alarm** class has several public operations that can be called wherever the class is visible. The classification manipulation operations have package visibility so that they can be accessed only within the package (which we might suppose is an alarm handling package). The **basicAction()** operation has protected visibility so it can only be called by the sub-classes of **Alarm**.

Instance and Class Variables and Operations

An attribute is an **instance variable** when each object stores its own value for the attribute. This is the usual sort of attribute. An attribute can also be a **class variable**, which means that there is only one value stored for the attribute that is shared by all class instances. You might think of a class variable's value as being stored in the class rather than in instances. In Java,

class variables (called *class fields*) are designated by the keyword `static`. Class variables are useful for storing constants common to all instances because there is no point in storing multiple copies of constants. In addition, class variables can hold class-wide values, such as the number of class instances or the value of a counter used to generate unique instance identifiers.

Operations can also be associated with instances or with classes. An **instance operation** can only be called using an instance. This is the usual way of calling an operation. In contrast, a **class operation** is encapsulated in a class and can be called through the class. Class operations can still be called even if no instances of the class exist. Java class operations are designated by the keyword `static`. For example, all operations in the Java `Math` class are static operations.

In UML, class variables and operations are called *static*, and they are indicated by underlining an attribute's or operation's specification. An attribute or operation specification that is not underlined is assumed to be an instance variable or operation.

The class diagrams in Figure 11-2-9 illustrate instance and class features.

Tokenizer	TokenType
<pre> - theInstance : Tokenizer + token : TokenType = NONE + tokenText : String - buffer : String - bufferPosition : integer - delimiters : String + getInstance() : Tokenizer + nextToken() + setBuffer(buffer : String) + setDelimiter(delim : String) </pre>	<pre> + NONE : TokenType { constant } + NUMBER : TokenType { constant } + IDENTIFIER : TokenType { constant } + EQUAL : TokenType { constant } + MINUS : TokenType { constant } + PLUS : TokenType { constant } - name : String - TokenType(name : String) + toString() : String </pre>

Figure 11-2-9 Attribute and Operation Specifications

These classes display a variety of feature specifications. The `Tokenizer` class has two public and three private instance variable attributes and one private class variable attribute. It also has a single public class operation and three public instance operations. The `TokenType` class has several public constant class variable attributes and one private instance variable attribute. Its constructor is private and it has one public instance operation.

Aggregation and Composition

UML has an association with a special symbol: the aggregation association. The **aggregation association** represents the part-whole relation between the instances of the associated classes. A strong form of aggregation is composition. In a **composition association**, each part can be related to only a single whole at one time.

An association line with a hollow diamond represents aggregation in UML. The diamond is placed at the end of the line with the class whose instances are the wholes. A filled diamond represents composition. Figure 11-2-10 illustrates aggregation and composition.

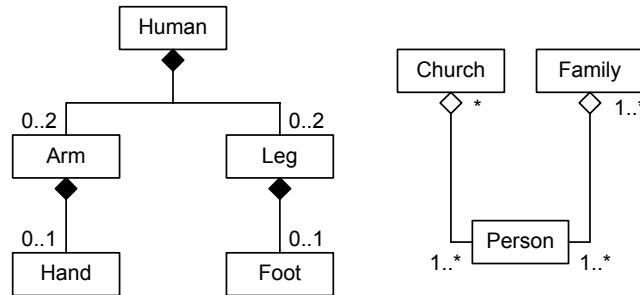


Figure 11-2-10 Aggregation and Composition

A **Human** has parts that cannot be shared with others, so the composition association is used to depict some **Human** parts. Note that parts (such as **Arms**) can have parts of their own (such as **Hands**), which are in turn parts of the larger whole (**Human**). This is an essential feature of the part-whole relation: It is transitive. If we regard organizations such as a **Church** or a **Family** as wholes with parts, then a single part (a **Person**) can be in more than one whole at a time, and we have aggregation but not composition.

This diagram also illustrates that aggregation is a form of association and, as such, can have association adornments, such as multiplicity. However, since composition requires that a part be a member of only one whole at a time, the multiplicity on the filled-diamond end of a composition association must be 1.

Aggregation is unfortunately one of the most abused notations in UML. The problem stems from metaphorical use of the part-whole relation to describe other relations. For example, we commonly say that items held in a container are part of the container. For example, we may say (metaphorically) that 1 is *part* of the set {1, 2, 3} when we mean that 1 is a *member* of the set. This may lead to improper use of aggregation to represent containment. This is incorrect because containment relations are usually not transitive, which, we noted, is an essential feature of aggregation. For example, 1 is an element of the set {1}, and the set {1} is an element of the set {{1},{2},{3}}, but 1 is *not* an element of the set {{1},{2},{3}}, so set membership is not transitive.

Correct uses of aggregation are relatively rare, and incorrect uses are quite common. Furthermore, part-whole relations can be represented in UML just as well with plain associations. Consequently, designers should avoid the UML aggregation notation. If it is used, designers should check that the relation represented by the association is really transitive and hence truly a part-whole relation.

Association Classes

Sometimes associations themselves have characteristics that don't belong to the classes connected by the association. Consider, for example, a patient record program that tracks patients and the medications prescribed to them. An obvious way to model this situation is as pictured in Figure 11-2-11.

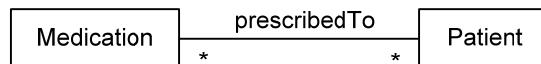


Figure 11-2-11 Medications Prescribed to Patients

Suppose that one of the pieces of data tracked by the program is the date a medication is prescribed. This date is neither a patient property nor a medication property, but a property of the association between them. Where should it go?

Associations need attributes (and even operations) often enough that UML supports association classes. An **association class** represents a relation on the sets of instances of the classes it connects, and it also holds data and behavior pertinent to the relation. An association class symbol is a regular UML class symbol connected to an association line by an *association class connector*, which is a dashed line. The dashed line should join the association line near its center. The association line is a regular UML association line and can have the usual adornments, except for an association name. The association class name is the association name.

Classes are usually named by noun phrases and associations by verb phrases, so how should an association class be named? Either a noun or a verb phrase will do, but a noun phrase is preferable. The best situation is when there is a noun phrase that nicely names the relationship between the instances of the associated classes.

The solution to our prescription problem using an association class is shown in Figure 11-2-12.

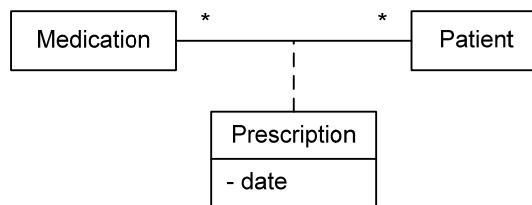


Figure 11-2-12 Prescription Association Class

The Prescription class is both an association and a class. As an association, it has instances (links) that relate a single Patient object to a single Medication object. Thus, there is only one occurrence of each link, and there can be only a *single* Prescription instance (and hence a single prescription date) for each medication-patient pair.

Contrast this situation with the one that results when the prescription is modeled as in Figure 11-2-13.

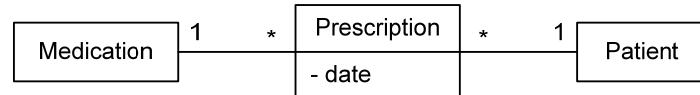


Figure 11-2-13 Prescription Class

This sort of class is called a *reified association* because a class is used to capture association characteristics, but the class is not an association class. In this case, there can be many instances of the Prescription class linked to the same Patient and Medication instances. Hence, there may be many prescription dates for each medication-patient pair.

The right way to model this situation depends on whether authorizing the same medication for the same patient on different occasions should be regarded as different prescriptions or as the same prescription refilled. If the latter, an association class must be used.

Association Qualifiers

Consider a Catalog class and its associated Course class. Each Course has a unique identifier consisting of its department descriptor and course number. Given a particular Catalog instance, a department descriptor, and a course number, a particular Course instance is selected.

UML provides a notation to model such situations in the form of association qualifiers. An **association qualifier** consists of one or more association attributes that, together with an instance of the qualified class, pick out instances of another class participating in a binary association with the qualified class. In the previous example, the department descriptor and course number together form an association qualifier.

An association qualifier is drawn as a box at the end of an association line connected to the qualified class. The box contains the association attributes comprising the qualifier. The attributes are listed one per line using the same syntax as class attributes, except that there can be no initial value specification. Usually all that is needed is the name and type of the attribute. Qualifiers are optional but cannot be suppressed. Figure 11-2-14 illustrates association qualifiers.

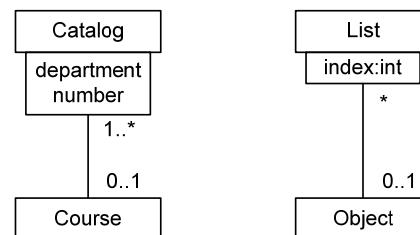


Figure 11-2-14 Association Qualifiers

The class attached to the qualifier is the *qualified class*, and the one opposite the qualifier is the *target class*. Target multiplicities are determined using the qualifier. To set the multiplicity on the target class end of the association, determine how many instances of the target class are related to a single instance of the qualified class given a particular combination of qualifier attribute values. For example, given a particular **Catalog** object, plus a particular **department** descriptor and course **number**, zero or one **Course** instances are selected (there may be no course with that **department** and **number**).

To determine the multiplicity on the qualified class end of the line, figure out how many qualified class instances with particular qualifier attribute value combinations can be related to a single instance of the target class. For example, a particular **Course** has a single **department** descriptor and course **number** in a **Catalog**, but it may appear in several **Catalogs**, so a **Course** instance is related to one or more **Catalog** instances with a particular **department** descriptor and course **number**.

More Association Adornments

UML provides two more association adornments that are sometimes useful: rolename visibility and navigability.

Associations may be implemented using references or pointers to establish the connections between instances. If so, a rolename can be interpreted as the name of an attribute in the class at the opposite end of its association line. The type and multiplicity of this attribute is already evident in the diagram, but its visibility is not. Visibility is specified by prefixing the rolename with one of the standard attribute visibility markers (+, #, ~, -). Rolename visibility may be suppressed.

An association between classes *A* and *B* is **navigable** from class *A* to class *B* if an instance of *A* can access an instance of *B*; otherwise it is **non-navigable**. It is often useful to know which directions of an association are navigable. Navigability from *A* to *B* can be marked in UML by placing an arrowhead on the association line pointing to *B*. Non-navigability is marked by placing a small X on the end of the association line near *B*.

Navigability markers can be suppressed. There is no default navigability, so no conclusions about association navigability can be drawn when there are no navigability markers.

Figure 11-2-15 illustrates these additional adornments.

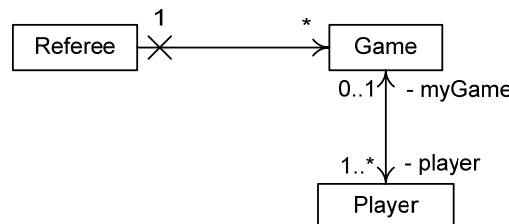


Figure 11-2-15 Rolename Visibility and Navigability

In this example, the **Referee** can access the collection of **Games** for which it is responsible, but the **Games** cannot access their **Referees**. **Games** and **Players** can access each other using the private rolenames **player** and **myGame**, respectively.

Heuristics Summary

Figure 11-2-16 summarizes class diagram heuristics distilled from this section.

- Never place a name, rolenames, or multiplicities on a generalization connector.
- Use the «abstract» stereotype and {abstract} property to indicate abstract classes and operations when drawing diagrams by hand; use italics for this purpose when drawing diagrams on the computer.
- Use the interface ball and socket symbols to abstract interface details and a stereotyped class symbol to show details.
- Don't italicize interface or operation names.
- Show provided interfaces with the interface ball symbol or the stereotyped class symbol and a realization connector.
- Show required interfaces with the interface socket symbol or dependency arrows to stereotyped class symbols or interface ball symbols.
- Avoid aggregation and composition.
- If aggregation or composition is used, check that the represented relation is transitive.
- Use association classes when only a single association class instance is associated with each pair of instances of the associated classes; otherwise use a reified association.
- Determine multiplicities involving a qualifier by considering qualified class instances and particular combinations of qualifier attribute values.

Figure 11-2-16 Class Diagram Heuristics

Section Summary

- UML has different connector symbols to represent association, generalization, interface realization, and dependency.
- Abstract operations are marked in UML using italics or properties, and abstract classes are marked using italics or stereotypes.
- A UML **interface** is a named collection of public attributes and abstract operations.
- A **provided interface** is realized by a class or component; a **required interface** is an interface upon which a class or component depends.
- UML provides several notations for specifying interfaces.
- UML allows designers to specify feature properties and visibilities, and whether they are static.
- UML has special symbols to represent **aggregation** and **composition associations**.

- An **association class** represents a relation on the sets of instances of the classes it connects; UML has an association class connector to indicate association classes.
- An **association qualifier** maps qualified class instances and values of one or more association attributes to instances of an associated class; UML has an association qualifier symbol to represent them.
- UML provides association adornments for rolename visibility and **navigability**.

Review
Quiz 11.2

1. What are abstract operations and classes?
2. Is there a difference between an abstract class with no concrete operations and a UML interface?
3. What are the four kinds of visibility supported in UML?
4. What is the difference between a class variable and an instance variable?
5. Give an example, different from those in the text, of a relation that is not transitive but might be mistaken for a part-whole relation.
6. Give an example, different from those in the text, of an association qualifier.
7. What is association navigability?

11.3 Drafting a Class Model

**Mid-Level
Design
Generation
Techniques**

Designers can make mid-level static design models by refining architectural models or by creating them independently and then modifying them in light of the architecture. In either case, designers can use several techniques from two broad categories:

Creational Techniques—Make a mid-level design model from scratch. This can be done through decomposition based on function, quality attributes, or both.

Transformational Techniques—Change another model into a mid-level design model. Specifically, designers can transform a design from a similar system, customize design patterns, or start with an analysis model and change it into a design model.

In Chapter 10, two creational techniques were used to illustrate generation of architectural design alternatives. In this section, one creational and one transformational technique are used to create alternative mid-level static designs for AquaLush. AquaLush is small enough that we can bypass the architectural design and modify the alternatives in light of the architecture. We might instead have applied these techniques to refine individual architectural components: This would probably be necessary for a larger product.

Although creational and transformational techniques can produce various sorts of components, AquaLush has an object-oriented design, so we produce alternative class models. Static and dynamic modeling should go hand-in-hand during mid-level design, a point elaborated in Chapter 12,

but a good starting point is static modeling. In this section we draft a class model that we expect to modify during dynamic modeling.

A Creational Technique

In Chapter 10, architectural design alternatives were generated by decomposing AquaLush functionally or with respect to quality attributes. Here we present a decomposition technique based on **design themes**, which are important problems, concerns, and issues that must be addressed in a design. The process of generating a class model based on design themes is depicted in the activity diagram in Figure 11-3-1.

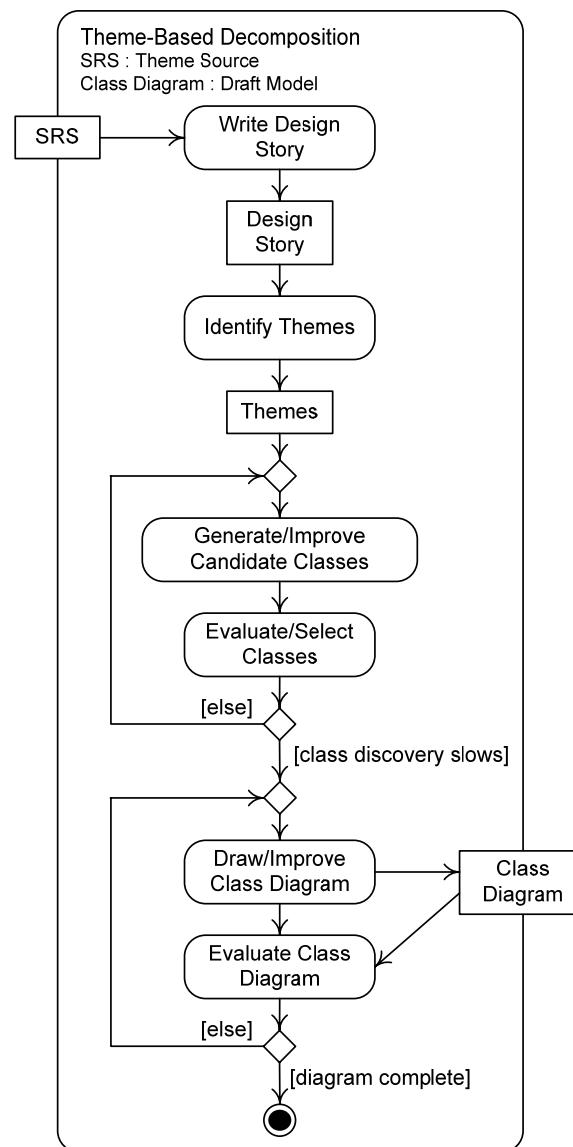


Figure 11-3-1 Theme-Based Decomposition Process

The first step of this process is to write a design story. A **design story** is a very brief (several paragraph) description of an application stressing its most important aspects. Figure 11-3-2 is a design story for AquaLush.

AquaLush is an automated irrigation system that controls irrigation by monitoring moisture sensors. As in a traditional irrigation system, irrigation occurs at a set time chosen by users, but the amount of water delivered to an irrigation zone is based on moisture sensors rather than a timer, providing more efficient water usage.

AquaLush also provides means to limit the water used, in case the soil is extremely dry or a water sensor provides bad readings. This mechanism is an overall water allocation for an irrigation cycle. The water allocation is divided between the irrigation zones, with larger zones (those with more valves) getting more and smaller zones getting less. If an irrigation zone uses up its water, then irrigation in that zone stops, despite what the moisture sensor may say. If zone irrigation stops because the moisture sensor indicates sufficient moisture, then the unused portion of the zone's water allocation is redistributed to the zones that have not yet been irrigated.

AquaLush has a manual mode that allows users to open and close individual valves. The moisture sensors are used even here, though, because moisture level readings, along with water usage levels, are provided to help users decide when an irrigation zone has had enough water.

AquaLush must eventually work with different irrigation valves and perhaps different sensors and control panel hardware.

Finally, AquaLush must provide a Web simulation that shows how it works in the field. This will involve simulating a customer site with irrigation zones, sensors, valves, and the AquaLush control panel.

Figure 11-3-2 AquaLush Design Story

Designers study the design story to identify important design problems, concerns, and issues. The goal is to find elements to focus on when first identifying candidate classes. Some design themes that emerge from the AquaLush design story are

- Providing manual and automatic modes of operation;
- Monitoring sensors, monitoring water usage, and controlling valves in irrigation zones;
- Combining water usage data and sensor readings to decide when to stop irrigation in a sensor zone;
- Supporting different kinds of valves, sensors, and control panel hardware; and
- Simulating the user interface as well as a customer site on a Web page.

Note that themes involve both functions and quality attributes.

The next step in the process is to generate and refine candidate classes. Themes are used as a basis for brainstorming. In light of the themes, designers look for candidate classes to model the following sorts of things:

- Entities in charge of or involved in program tasks;
- Things in the world that interact directly with the program (actors);
- Things about which the program stores data; and
- Structures and collections of objects.

Brainstorming AquaLush themes might have the results in Table 11-3-3.

Theme	Type of Thing	Candidate Class	Responsibilities
Manual and automatic modes	Entities in charge or involved	IrrigationController	Track mode, initiate irrigation
	Clock	Clock	Track irrigation time
	Actors	User	Interface with user
	Things whose data is stored	IrrigationCycle	Hold data about an irrigation cycle
Monitor sensors, monitor water usage, control valves	Actors	Sensor, Valve	Represent devices
	Things whose data is stored	IrrigationZone	Keep track of sensors, valves, and water allocation
	Structures and collections	ZoneList	Hold zones yet to be irrigated
		ValveList	Hold valves in a zone
Control irrigation	Entities in charge or involved	ManualIrrigator	Control manual irrigation
	Actors	AutoIrrigator	Control automatic irrigation
	Things whose data is stored	WaterUsage	Hold water allocation and how much is used so far
Hardware flexibility	Actors	VirtualValve, VirtualSensor, VirtualControlPanel	Specify virtual devices
	Actors	RealValve, RealSensor, RealControlPanel	Implement actual hardware interfaces
	Things whose data is stored	Valve, Sensor	Hold valve and sensor data
Web-based simulation	Entities in charge or involved	AquaLushApplet	Execute on a Web page
	Actors	SimValve, SimSensor, SimClock, SimControlPanel	Simulate real devices
	Things whose data is stored	SimZone	Simulate moisture in a zone, hold SimSensors and SimValves
	Structures and collections	SimSite	Hold SimZones

Table 11-3-3 Candidate Classes Generated from Themes

After generating candidate classes, designers evaluate them and choose the best ones to include in the model using the following heuristics:

Discard candidates with vague names or murky responsibilities. Candidates should have precise names and clear responsibilities.

Rework candidates with overlapping responsibilities to divide their responsibilities cleanly. Removing overlapping responsibilities increases cohesion and information hiding and decreases coupling.

Discard candidates that do something out of scope. Some external entities identified from the design story, for example, may not be needed in the design class model.

The process of generating, refining, evaluating, and selecting an initial list of candidate classes is repeated until no new classes come to mind.

Once designers have produced a solid collection of candidate classes, they draw a draft class diagram. Class attributes, operations, associations, relations, and so forth should reflect class descriptions.

The next step is to evaluate the class diagram to check that all candidate classes are present and that the diagram reflects their descriptions.

Designers can apply the following heuristics to this activity:

Check each class for important but overlooked attributes, operations, or associations. For example, `SimClock` is given a `speed` attribute in the design class model as a means to control the simulation rate.

Combine common attributes and operations in similar classes into a common superclass. For example, the `AquaLush Manuallrrigator` and `Autolrrigator` classes generated from design themes are made sub-classes of an `Irrigator` super-class. Identifying common operations may also lead to the introduction of interfaces in the model. For example, the common operations of the real and simulated clocks in `AquaLush` led to the introduction of a `Clock` interface.

Apply design patterns where appropriate. We will consider design patterns in detail in Part IV of this book.

When designers detect opportunities for improvement or correction, they return to the drawing and improvement step of the theme-based decomposition process.

A diagram that passes muster can be considered complete. However, it is only complete as a *draft*. It will surely change in light of the architecture and in conjunction with dynamic modeling.

Figure 11-3-4 shows the draft `AquaLush` class model resulting from this process.

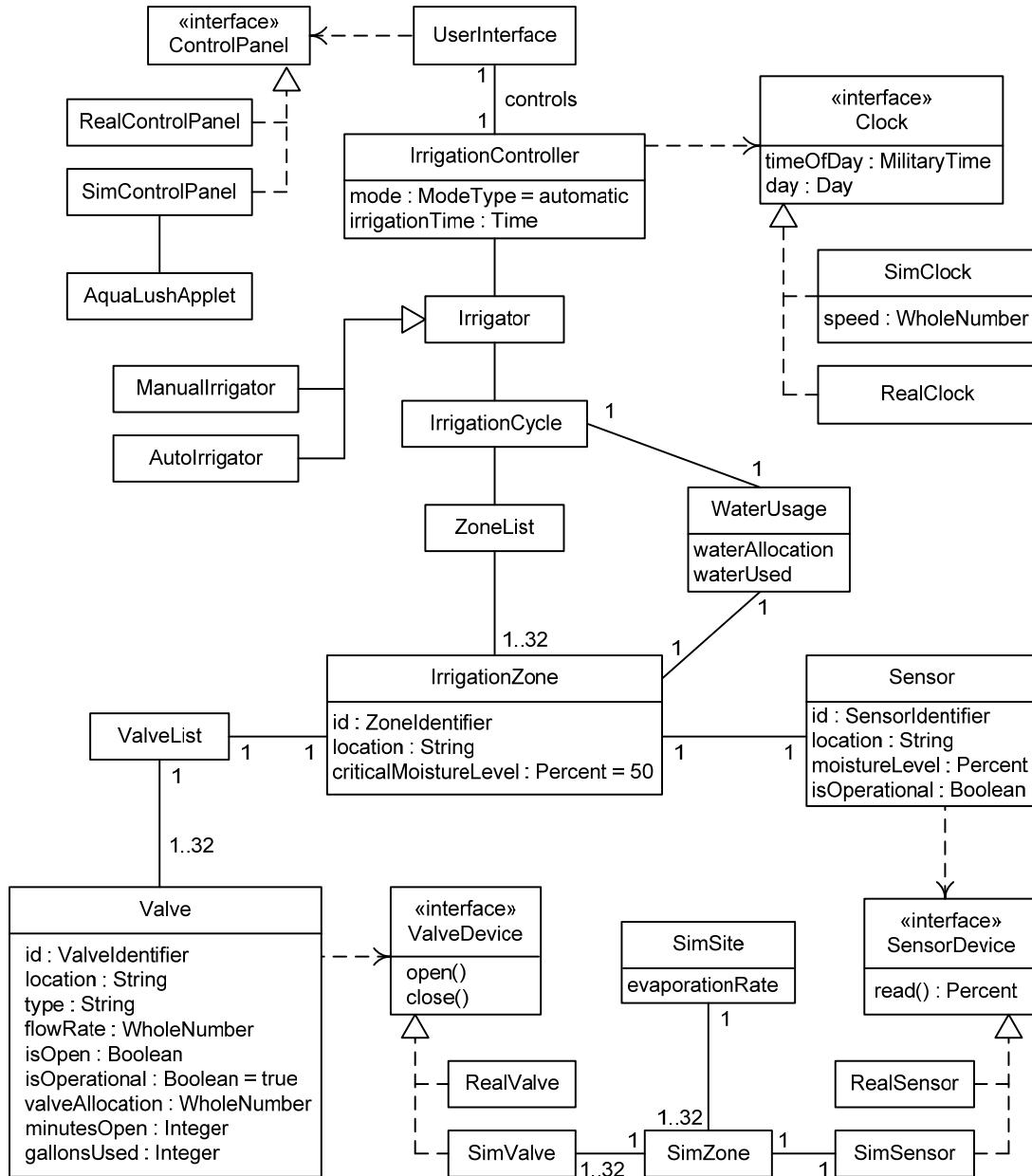


Figure 11-3-4 Draft AquaLush Design Class Diagram (Creational)

A Transformational Technique An important feature of the object-oriented paradigm is that it provides solutions closely resembling problems. This has several advantages. First, it makes the connection between the problem and the solution easy to understand, which makes both development and maintenance easier. Second, important entities in the problem tend not to change very much over time, though their behaviors and characteristics may. If these stable

entities are reflected in the program, then program entities will also tend not to change over time; only their attributes and operations may change. This makes it easier to modify the program in response to new or changed requirements. Thus, it is advisable to have program entities that mimic problem entities.

A conceptual model describes the important concepts in the engineering design problem. If it is taken as a starting point for problem solution, then the resulting design will contain many entities reflecting the problem domain. Transformational techniques take advantage of this insight by starting with a conceptual class model and modifying it to make a draft design class model.

Our transformation technique uses the following heuristics to convert a conceptual class model into a design class model. These are illustrated by the AquaLush draft design model in Figure 11-3-5 on page 343, which is the result of transforming the AquaLush conceptual model in Appendix B.

Change actors to interface classes. Conceptual models contain classes representing actors. Actors are not part of the design solution, but interfaces to actors are. Hence, each actor class should be converted into an interface class for that actor. For example, the AquaLush conceptual model actors `User`, `Clock`, `Control Panel`, `Valve`, and `Sensor` are converted into `UserInterface`, `Clock`, `ControlPanel`, `ValveDevice`, and `SensorDevice` in the design class model.

Add actor domain classes. A program may also record data about actors. Any conceptual model actor class about which the program needs to record data should also be made into an application domain class with the appropriate attributes. AquaLush needs to keep track of `Valves` and `Sensors`, so these are included as domain classes in the model.

Add a startup class. Many programs need a place to begin, so a startup class may need to be added to the design class model. AquaLush needs to configure itself and restore settings from persistent store, so it certainly needs a startup class to handle this chore. The `Configurator` class plays this role.

Convert or add controllers and coordinators. Classes that control and coordinate program activities may already be present as conceptual classes—these are simply converted to design classes. Some controllers and coordinators may need to be added; some may become evident only when interactions are modeled. AquaLush needs irrigation controllers and a simulation controller, but these are already present in the conceptual model and are simply carried over into the design model.

Add classes for data types. Conceptual models often include types with a great deal of structure and behavior, and they must be implemented as data types in the program. Classes to implement data types should be added to the design class model. For example, AquaLush has two irrigation modes. These are modeled by an enumeration class in the design class model.

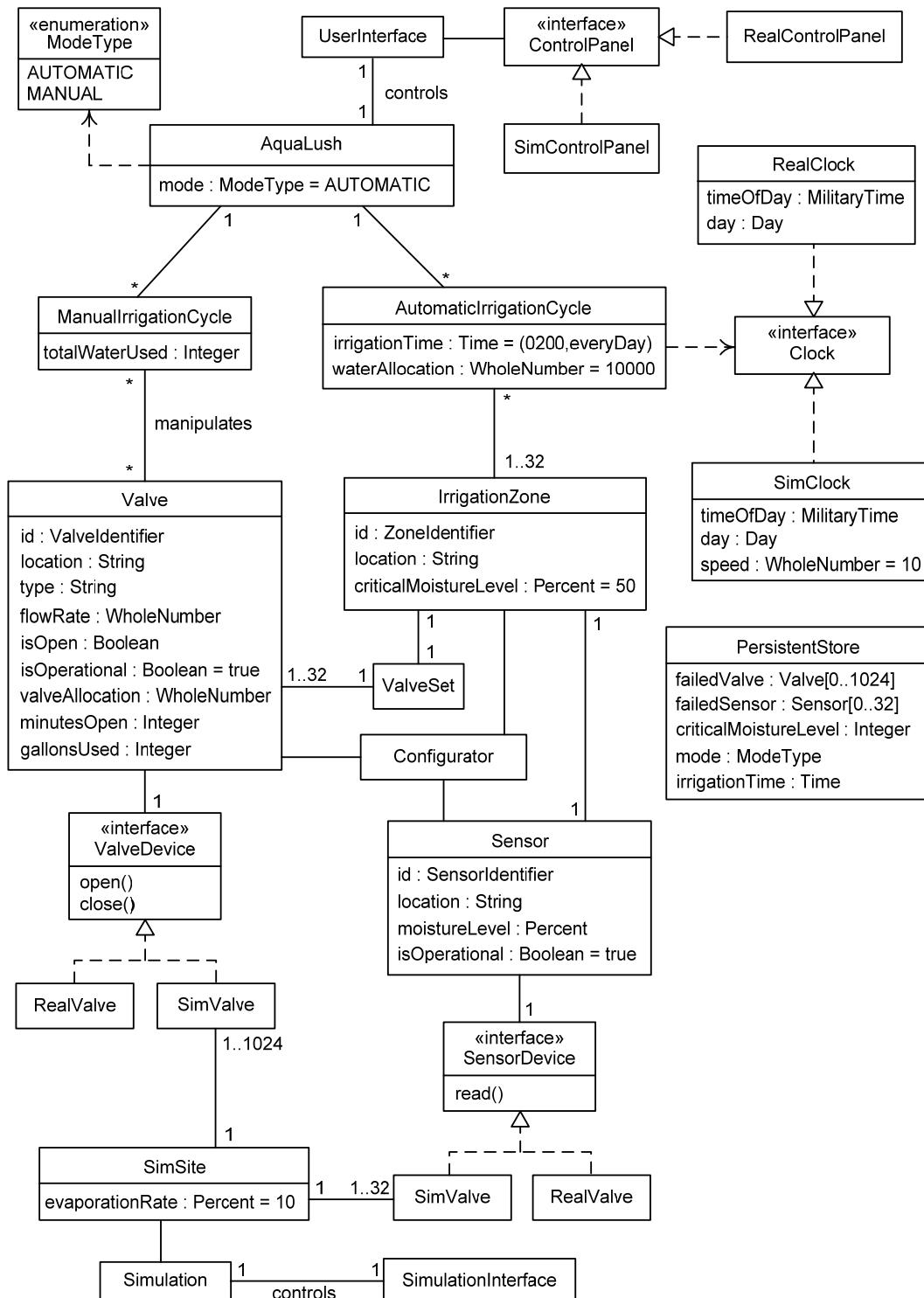


Figure 11-3-5 Draft AquaLush Design Class Diagram (Transformational)

Convert or add container classes. A **container class** has instances that hold a collection of objects. Containers may already be present in the conceptual model, or they may be added to the design class model. This is a somewhat low-level design detail, so often decisions about particular containers can be skipped in mid-level design. As an example, AquaLush irrigation zones have between 1 and 32 valves. The design class model has a `ValveSet` class to contain the valves for each irrigation zone.

Convert or add engineering design relationships. Generalization relationships, interfaces and interface realization relationships, dependencies, and so forth may already be present in the conceptual model or can be added as appropriate. Many of these will be added as the design progresses to dynamic modeling. In our example, the realization relationships between virtual device interfaces and the device drivers that realize them for simulated and actual devices are shown in the model.

Comparing Creational and Transformational Techniques

The AquaLush models produced using the generational and transformational techniques have much in common, but they have some differences as well. Many core classes, such as `Sensor`, `Valve`, `IrrigationZone`, and `Clock`, are identical. The classes for managing irrigation, however, are different. In fact, either of these design alternatives could have been produced by either a creational or a transformational technique: There is nothing about them that is a consequence of the way that they were created. There is no particular advantage to using one technique over the other in terms of results.

One technique does have an advantage in terms of effort, however. It is much easier to transform a conceptual model into a draft design class model than it is to generate one from scratch. Consequently, a transformational technique should be used if a conceptual model is available. If there is no conceptual model, then one can be created and then transformed into a design class model, or a generational technique can be used to create a design class model from scratch.

Heuristics Summary

Figures 11-3-6 and 11-3-7 summarize the class diagram generation heuristics discussed in this section.

- Change actors to interface classes.
- Add actor domain classes.
- Add a startup class.
- Convert or add controllers and coordinators.
- Add classes for data types.
- Convert or add container classes.
- Convert or add engineering design relationships.

Figure 11-3-6 Conceptual Model Transformation Heuristics

Generating Classes from Themes

- Look for entities in charge of program tasks.
- Look for actors.
- Look for things about which the program stores data.
- Look for structures and collections of objects.

Evaluating and Selecting Candidate Classes

- Discard classes with vague names or murky responsibilities.
- Rework candidate classes with overlapping responsibilities to divide their responsibilities cleanly.
- Discard classes that do something out of scope.

Evaluating and Improving the Class Diagram

- Check each class for important but overlooked attributes, operations, or associations.
- Combine common attributes and operations from similar classes into a common super-class.
- Apply design patterns where appropriate.

Figure 11-3-7 Theme-Based Decomposition Heuristics

- Section Summary**
- Drafting a design class model is a starting point for mid-level design.
 - Various creational or transformational techniques can produce a draft design class model by either refining the architecture or working directly from the SRS and analysis models.
 - A theme-based creational technique works by writing a design story, extracting its themes, using the themes to produce candidate classes with clear responsibilities, and drawing and improving a class diagram.
 - A transformational technique works by applying rules to produce a design class model from a conceptual class model.

- Review Quiz 11.3**
1. Describe two creational and two transformational techniques for generating a mid-level design model.
 2. What is a design story and what is it used for?
 3. Explain what a container class is and list two examples.

11.4 Static Modeling Heuristics

- Responsibilities** We have often mentioned responsibilities without talking much about what they are. A responsibility is an obligation; that is, something that one is bound to do.

In software engineering design, there are two kinds of responsibilities:

- An obligation to *do something*; that is, to carry out some task.
- An obligation to *know something*; that is, to maintain some data.

We can thus define a software component responsibility as follows.

A **responsibility** is an obligation to perform a task (an **operational responsibility**) or to maintain some data (a **data responsibility**).

Operational responsibilities are usually fulfilled by one or more software component operations. Operations typically need data to work with, and they may call other operations. Data responsibilities are fulfilled by storing data in variables such as attributes or computing values from stored data. In both cases, collaborations may be involved in fulfilling responsibilities. For example, in AquaLush **Valve** objects are responsible for opening and closing particular hardware valves (an operational responsibility) and keeping track of the valves' properties (a data responsibility). **Valve** objects collaborate with **ValveDevice** objects to fulfill both these responsibilities. The **Valve** calls **ValveDevice** operations to open and close the hardware valve. If the hardware valve fails, then the **ValveDevice** object throws an exception and the **Valve** object records that the hardware valve has failed.

When stating software component responsibilities, designers tend to focus on operational rather than data responsibilities. However, the latter are also important. Hence, when documenting the responsibilities of a software component, designers should apply the following heuristic:

State both operational and data responsibilities.

Responsibility-Driven Decomposition	<p>Responsibilities may be stated at different levels of abstraction. Responsibilities at high levels of abstraction may be decomposed into lower-level responsibilities. For example, the AquaLush AutoCycle is responsible for controlling automatic irrigation cycles. This high-level responsibility can be divided into lower-level responsibilities of starting automatic irrigation at the right time, controlling irrigation in each irrigation zone, and maintaining the data that controls automatic irrigation. Each of these responsibilities can then be further decomposed.</p> <p>A large software component has high-level responsibilities. These can be decomposed into lower-level responsibilities and assigned to sub-components. Furthermore, the lower-level responsibilities can be the basis for generating sub-components. Responsibility-driven decomposition is a technique for program decomposition in which component responsibilities are decomposed and used to generate sub-components. The decomposed responsibilities are assigned to the sub-components, which can then be further decomposed.</p> <p>Responsibilities are more general than functions, and they can reflect data and non-functional requirements. Consequently, a responsibility-driven</p>
-------------------------------------	---

decomposition may be different from a functional decomposition, a quality-attribute-based decomposition, or a theme-based decomposition. We can thus add responsibility-driven decomposition to the other decomposition techniques for design alternative generation.

Responsibilities, Coupling, and Cohesion

Recall from Chapter 8 that two important constructive design principles are the Principles of Cohesion and Coupling. The Principle of Cohesion states that module cohesion should be maximized, where *cohesion* is the degree to which a module's parts are related to one another. The Principle of Coupling says that module coupling should be minimized, where coupling is the degree of connection between pairs of modules.

Designers can use responsibilities to make modules more cohesive and less tightly coupled in several ways:

Assign modules at most one operational and one data responsibility. Modules with conflicting responsibilities lack cohesion. The best way to avoid conflicting responsibilities is to have only one of each kind. Remember that responsibilities can be at various levels of abstraction, so having only one operational responsibility and one data responsibility does not mean that the module has only one operation and stores only one data value.

Assign complementary data and operational responsibilities. Data and operational responsibilities are complementary when the data stored to fulfill data responsibilities is needed by and maintained by the code fulfilling operational responsibilities. In other words, a module should store the data its operations need, and the operations should be in charge of maintaining this data. For example, consider a bank **Account** module responsible for keeping track of the money in the account. This data responsibility is complemented by the operational responsibility to handle deposits, withdrawals, and interest payments, which all have to do with altering the money in the account. This data responsibility is not complemented by operational responsibilities, such as changing the owner of the account or changing the account type, which may be data responsibilities of other modules.

Make sure module responsibilities do not overlap. Modules with overlapping responsibilities will have to either share or duplicate module elements. For example, suppose a program for managing a warehouse has a **Storage and Retrieval** module responsible for keeping track of empty storage slots, putting items into empty storage slots, and removing items from storage slots, which empties them. It also has an **Optimization** module that keeps track of empty storage slots so that it can recommend where items should be stored to speed up warehouse operations. These module responsibilities overlap because both modules must keep track of the empty storage slots. They will be strongly coupled because the **Optimization** module must be told of every change that the **Storage and**

Retrieval module makes. These responsibilities should be reworked so they do not overlap.

Place operations and data in a module only if they help fulfill the module's responsibilities. Module elements that have nothing to do with fulfilling a module's responsibilities are not strongly related to the other elements in the module, decreasing cohesion. Also, these foreign elements probably help fulfill a responsibility of some other module that will then be strongly coupled to the module where the needed elements reside.

Place all operations and data needed to fulfill a module responsibility in that module. This heuristic is the counterpart of the last. If a module does not contain all the elements it needs to fulfill its responsibilities, then it must collaborate with the module (or modules) where the missing elements reside. The modules will be strongly coupled, and the misplaced elements will lower the cohesion of the module(s) where they reside.

Assigning responsibilities with cohesion and coupling in mind and decomposing modules in accord with module responsibilities makes for more cohesive and loosely coupled modules and thus better designs.

Inheritance **Inheritance** is a declared relation between a class and one or more super-classes that causes the sub-class to have every attribute and operation of the super-class or super-classes. Inheritance is a powerful design and implementation mechanism that offers two advantages:

- Inheritance captures a generalization-specialization relation between the super-class(es) and the sub-class. In other words, it reflects that fact that the sub-class is a *kind of* the super-class or super-classes.
- Inheritance provides reuse of the implementations of attributes and operations in the super-class or super-classes.

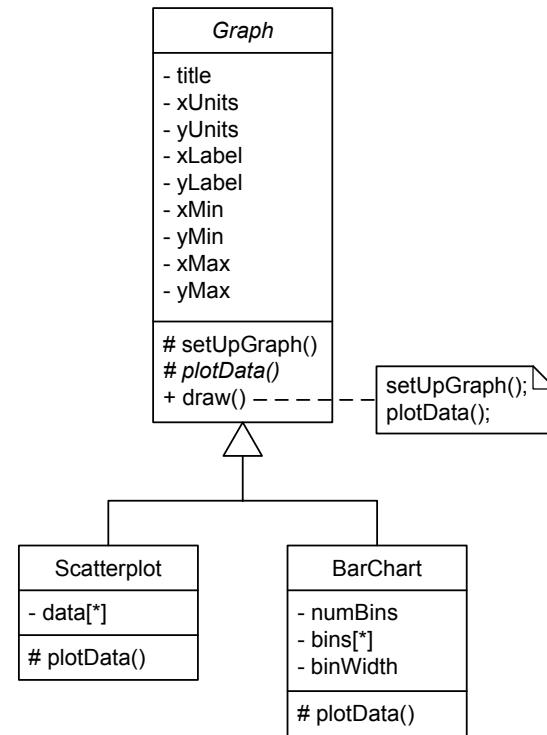
Unfortunately, designers may be tempted to use inheritance for reuse even when the sub-class is not a specialization of its super-class or super-classes. For example, suppose that a data analysis program has a `Scatterplot` class like the one shown in Figure 11-4-1 on page 349.

Suppose further that the program also needs a `BarChart` class. Bar charts and scatter plots have much in common; in particular, their axes are almost the same. A designer might be tempted to make `BarChart` a sub-class of `Scatterplot`, to inherit the attributes describing the title and axes as well as the `setUpGraph()` operation that draws them. This does save programmers some work, but it is an abuse of inheritance. The problem is that a `BarChart` is not a kind of `Scatterplot`, so using inheritance like this falsely asserts that there is a generalization relationship between `BarChart` and `Scatterplot`. Consequently, we state the following heuristic:

Use inheritance only when there is a generalization relationship between the sub-class and its super-class(es).

**Figure 11-4-1 A Scatterplot Class**

There are ways to use inheritance properly and still get the benefits of reuse in cases like the previous one. Classes can be rearranged so that common attributes and operations are placed in a super-class that really does generalize its sub-classes. Figure 11-4-2 illustrates one way to do this.

**Figure 11-4-2 Proper Use of Inheritance**

In this design, the `Graph` abstract super-class contains the common axis attributes and graph-plotting operations. It implements the axis-plotting operation `setUpGraph()` as well as the public `draw()` operation, and it declares an abstract `plotData()` operation that sub-classes must implement to plot the data in the body of the graph. Both `Scatterplot` and `BarChart` inherit and hence reuse all this common implementation, and each has its own specialized `plotData()` implementation. We see here an application of a heuristic mentioned in the previous section as a rule of thumb for improving class diagrams:

Combine common attributes and operations in similar classes into a common super-class.

Delegation

Another way to solve the problem of reusing code without abusing inheritance is to decompose responsibilities and assign some to separate classes. The tactic wherein a module entrusts another module with a responsibility is called **delegation**. Delegation not only allows reuse without inheritance, but also provides a mechanism to make software much more flexible and configurable. We see many applications of delegation in design patterns, covered in Part IV. For now, we illustrate delegation by showing its application to the previous example.

The `Scatterplot` class is responsible for maintaining scatter plot data and drawing scatter plots. Similarly, the `BarChart` class is responsible for maintaining bar chart data and drawing bar charts. These responsibilities can each be divided in two: The `Scatterplot` and `BarChart` classes must each (a) maintain graph axis data and draw the axes, and (b) maintain plotted data and draw the plotted data. The first of these responsibilities can be delegated to an `Axes` class. The `Scatterplot` and `BarChart` classes can then delegate responsibility for maintaining axis data and drawing axes to the `Axes` class, and they can handle the other responsibilities on their own. Figure 11-4-3 shows this design.

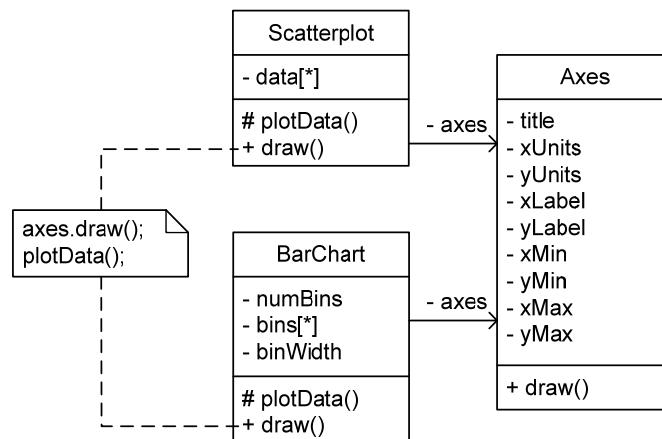


Figure 11-4-3 Delegation for Reuse

We derive the following heuristic from this example:

Use delegation to increase reuse, flexibility, and configurability.

Heuristics Summary

Figure 11-4-4 summarizes the static modeling heuristics discussed in this section.

- State both operational and data responsibilities.
- Assign modules at most one operational and one data responsibility.
- Assign complementary data and operational responsibilities.
- Make sure module responsibilities do not overlap.
- Place operations and data in a module *only* if they help fulfill the module's responsibilities.
- Place *all* operations and data needed to fulfill a module responsibility in that module.
- Use inheritance only when there is a generalization relationship between the sub-class and its super-class(es).
- Combine common attributes and operations in similar classes into a common super-class.
- Use delegation to increase reuse, flexibility, and configurability.

Figure 11-4-4 Static Modeling Heuristics

Section Summary

- A **responsibility** is an obligation to perform a task or to maintain some data.
- **Responsibility-driven decomposition** can be used to generate static design models based on decomposing responsibilities.
- Responsibilities can guide module formation to increase cohesion and decrease coupling.
- **Inheritance** states a generalization relation and provides reuse of super-class implementations in sub-classes.
- Inheritance should never be used for reuse alone; it should always reflect a generalization relationship.
- **Delegation** is the tactic of entrusting a responsibility to another module (a delegate).

Review Quiz 11.4

1. How are operational and data responsibilities related?
2. How can responsibilities help guide decisions about what operations and data should be put in a module?
3. What is a generalization-specialization relationship? Give an example to illustrate it.
4. Name an advantage of delegation.

Chapter 11 Further Reading

- Section 11.2** Although many UML books cover the advanced features discussed in this section, the many subtle details of these features are discussed most thoroughly in [Bennett et al. 2001], [Rumbaugh et al. 2004], and [OMG 2003].
- Section 11.3** The creational technique based on design themes is drawn from [Wirfs-Brock and McKean 2003]. Larman [2005] discusses a transformational technique.
- Section 11.4** Responsibility-driven decomposition is based on a design method called *responsibility-driven design* proposed by Rebecca Wirfs-Brock and others, documented most recently in [Wirfs-Brock and McKean 2003]. Riel [1996] discusses object-oriented design heuristics in depth. Delegation is a well-known tactic often discussed in books about object-oriented design, including [Larman 2005] and [Gamma et al. 1995].

Chapter 11 Exercises

The following product specifications are used in the exercises.

Prescription Archive Program (PAP)

The Prescription Archive Program (PAP) assists pharmacists by recording data about patients and their prescriptions.

Patient data includes the customer's first name, middle initial, last name, address, phone numbers, birth date, known allergies, insurance company, and insurance card identifier.

Prescription data includes the prescription identifier, name of the drug, dosing unit (such as 100-milligram capsules), number of dosing units, instructions for taking the drug, drug expiration date, number of refills, date when refills expire, and dates when the prescription was filled or refilled. Prescription identifiers are unique.

Doctor data includes the doctor's first name, middle initial, last name, address, and phone numbers.

Pharmacist data includes the pharmacist's first name, middle initial, last name, address, and phone numbers.

Insurance data includes the insurance company's name, address, and phone numbers. There are also insurance plans. Each insurance plan is derivable from the insurance company and the patient's insurance card identifier. Insurance plans contain lists of the drugs they cover. Insurance plan identifiers are unique for each company.

Besides this information about patients, prescriptions, doctors, pharmacists, insurance companies, and insurance plans, the PAP records all prescriptions for each patient, the doctor who wrote the prescription, the pharmacist that filled the prescription, and the insurance plan that covered the prescription, if any.

Pharmacists can record information in the system and make the following queries:

- Find patient data by name, address, or phone number.
- Find a patient's prescriptions for a drug given the patient name and the drug name.
- Find a prescription given its prescription number.
- Find whether a drug is covered given a company and an insurance card identifier.
- Find a patient's prescriptions written by a certain doctor given the patient identifier and the doctor's name.

Grain Elevator Tracking System

Harvested grain is trucked from farms to a central storage elevator, where it is placed in silos. The grain is eventually moved from the silos to railroad cars that take it to processing plants. The depot relies on a software system to keep track of the contents of each silo.

The depot accepts shipments of high-grade corn, low-grade corn, long-grain rice, short-grain rice, wheat, and barley. An empty silo may store any kind of grain, but a silo with grain in it can accept only grain of the same type.

There are 12 silos in the depot: 6 silos hold 8,000 bushels and 6 silos hold 20,000 bushels.

When a truck arrives at the depot with a load consisting of some number of bushels of grain of some type, the depot manager informs the system of the type of grain and its quantity. The system must find silos to store the grain in and tell the depot manager which silos it has chosen and how much grain is to go in each one. The system may accept only part of a load if there is not room for the entire load. If there is no silo that will hold the grain, the system must inform the depot manager.

The depot manager may accept or overrule the system's choice of silos for an arriving load of grain. If the system is overruled, the depot manager must inform the system that it is being overruled and how much grain is actually deposited in the silos being used. If the system is not overruled, the depot manager must verify that the grain is deposited as suggested by the system.

As a train is loaded, the depot manager must tell the system how much grain has been removed from each silo. The system should acknowledge receipt of this information.

Upon request from the depot manager, the system must produce a complete report of the state of the depot. This report should list, for each silo, the type of grain stored, the amount stored, and the remaining capacity of the silo. The report should also list the total remaining capacity of the depot for each type of grain currently stored and the total capacity of the depot not currently committed to any type of grain.

Section 11.1

1. *Fill in the blanks:* Detailed design resolution closely resembles _____ design resolution. Among the techniques useful in both for generating and improving design alternatives are decomposition based on _____, decomposition based on _____, copying a design from _____, creating a design from a _____ model, and basing a design on a _____.
2. Indicate which of the following activities are part of mid-level design (M), low-level design (L), or programming (P):
 - (a) Choosing a sorting algorithm
 - (b) Deciding whether a class should be a sub-class of another class
 - (c) Naming the local variables in a method
 - (d) Deciding how to implement an enumeration type
 - (e) Deciding which interfaces a class should implement
 - (f) Figuring out how two threads should synchronize when using the same object
 - (g) Breaking a large class into several collaborating classes
3. Write a brief essay in which you consider how the notations presented to this point in the book might be used in detailed design.

Section 11.2

4. *Find the errors:* Find at least five errors in Figure 11-E-1.

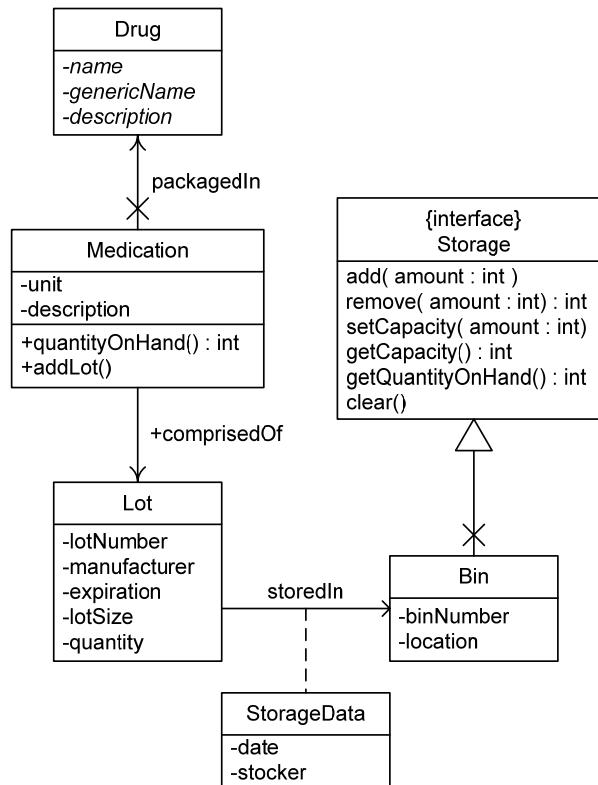
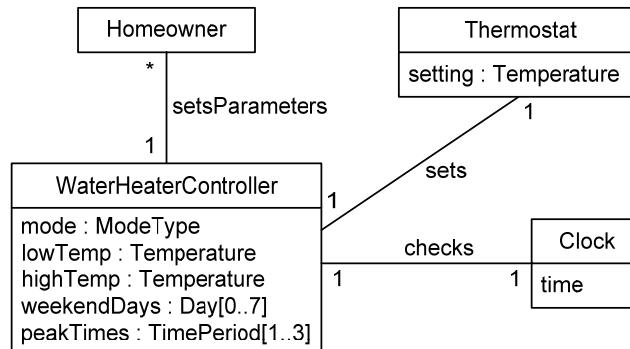
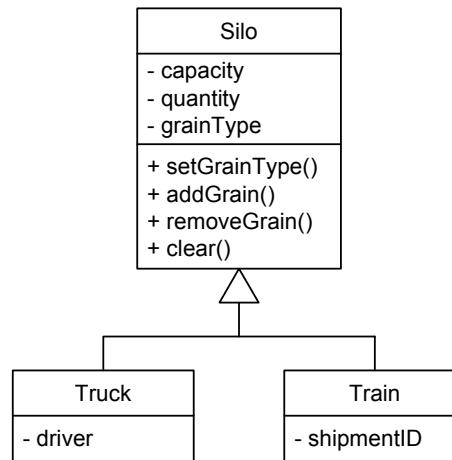


Figure 11-E-1 An Erroneous Class Diagram for Exercise 4

5. *Find the errors:* Which of the following expressions are valid UML operation specifications?
 - (a) `addListener(EventListener e)`
 - (b) `register(o : Object) : void`
 - (c) `#getAdapter() : Adapter`
 - (d) `+setCapacity(in newCapacity : int = 10) : Boolean`
 - (e) `~createIterator(flag : FlagType) : Boolean, Iterator`
 - (f) `+toString(+arg : Object) : String`
 - (g) `-computeArea() {abstract}`
 - (h) `#configureInstance(o : BulkStore; p : Set)`
6. Suppose that you are writing the Prescription Archive Program. Make a class model for this program that uses at least one super-class with at least one sub-class, at least one association qualifier, at least one association class, and at least one navigation marker.
7. One idea for modeling the Prescription Archive Program is to have a Prescription association class. The Prescription class would associate the classes Patient, Doctor, Pharmacist, and Insurance Company. Evaluate this design alternative.
8. Although the UML generalization connector marks classes that participate in the generalization relation and thus have no multiplicity, the generalization relation itself does have a multiplicity. Let C be the set of all classes. Then “generalizes” is a relation on C . What is the multiplicity of the “generalizes” relation?
9. Is there such a thing as an abstract attribute? If so, explain what it is, and if not, explain why not.
10. How do the four kinds of attribute and operation visibility provided by UML correspond to the kinds of attribute and method visibility available in Java programs?
11. What is the difference between inheritance and realizing an interface?
12. Can a class operation be abstract? Explain why or why not.
- Section 11.3** 13. Compare and contrast the theme-based decomposition process pictured in Figure 11-3-1 to the generic problem solving process discussed in Chapter 2.
14. Use the theme-based decomposition technique to create a design class model for the Prescription Archive Program. Turn in your design story, themes, and class model.
15. Use the transformational technique presented in the text to draft a design class model for the Caldera program whose conceptual model is shown in Figure 11-E-2.

**Figure 11-E-2 Caldera Conceptual Model for Exercise 15****Section 11.4**

16. Make a table listing the data and operational responsibilities of the classes in the model you created in the previous exercise.
17. Use responsibility-driven decomposition techniques to create a design class model for the Prescription Archive Program. Include a table listing class responsibilities with your class model.
18. Reorganize the class model in Figure 11-E-3 to improve the design.

**Figure 11-E-3 A Poor Class Model for Exercise 18**

19. Compare and contrast inheritance and delegation.
20. Suppose that a language (such as Java) supports only single inheritance, but there are two classes that could reasonably be super-classes of a class under design. How could the class under design receive at least some of the benefits of multiple inheritance using only single inheritance?
21. Compare and contrast the advantages of inheritance and interface realization.
- Research Projects** 22. Find a book about responsibility-driven design. Write a summary of this method.

23. Find a book about object-oriented design. List at least 10 static modeling heuristics discussed in the book.

Team Project

24. Form a team of three. Have two people use different creational techniques and the third a transformational technique to produce a draft design class diagram for the Grain Elevator Tracking System. Write a brief report in which you compare and contrast the results. Turn in your models and your report.

Chapter 11 Review Quiz Answers**Review Quiz 11.1**

1. Mid-level design focuses on the design of medium-sized components, such as classes and compilation units. Mid-level design decomposes architectural components. Low-level design is about the details at the lowest levels of design abstraction. Low-level design is the middle ground between design and programming.
2. DeSCRIPTR-PAID stands for Decomposition, States, Collaborations, Responsibilities, Interfaces, Properties, Transitions, Relationships, Packaging, Algorithms, Implementation, and Data structures and types. It is an acronym for the kinds of specifications that should be found in a low-level design.
3. A detailed design document contains mid-level design models, low-level design models, a section showing the relationships between the models, a design rationale, and a glossary.

Review Quiz 11.2

1. An abstract operation is an operation with a signature but no body. An abstract class is a class that cannot be instantiated. A class with an abstract operation must be abstract.
2. UML interfaces can contain only public features, while abstract classes can contain features with arbitrary visibility.
3. UML supports public, package, protected, and private visibility.
4. A class variable is encapsulated in a class and shared across all instances of the class. An instance variable is encapsulated in instances, so each instance stores its own value(s) for each instance variable.
5. Some examples of relations that are not transitive but might be mistaken for a part-whole relation are the list membership relation and various organizational membership relations.
6. Consider a student record system with a `StudentBody` class connected to a `Student` class by a `memberOf` association. The `StudentBody` can be a qualified class with a `studentNumber` qualifier that yields zero or one `Student` instances.
7. Association navigability is the ability to access an instance of one class from another class. More specifically, an association between classes *A* and *B* is *navigable* from class *A* to class *B* if an instance of *A* can access an instance of *B*, and *non-navigable* otherwise.

Review Quiz 11.3

1. Two creational techniques are functional decomposition and theme-based decomposition. In functional decomposition, the functions of higher-level components are decomposed to yield lower-level components. In theme-based

decomposition, design themes are extracted from a design story and used to generate modules to account for each theme.

Two transformational techniques are transforming a conceptual model into a design model and transforming the design of a similar program. Transforming a conceptual model begins with a conceptual model and changes its elements from problem entities to design components, altering them and adding new components as necessary. Transforming a similar design involves modifying the original design to satisfy the requirements of the new product.

2. A design story is a very brief (several paragraph) description of an application stressing its most important aspects. It is used as the basis for themes that help find candidate classes in generating design class diagrams.
3. A container class is a class whose instances hold a collection of objects. Examples from the Java class library include any class that implements the `Collection` interface, such as `ArrayList`, `Hashtable`, `Stack`, or `Vector`.

**Review
Quiz 11.4**

1. Operational responsibilities are obligations to carry out some task. Completing a task usually involves data, so a component should have data responsibilities related to its operational responsibilities.
2. The data and operations placed in a module should be all and only those needed to carry out the responsibilities of the module. This will increase module cohesion and decrease module coupling.
3. A generalization-specialization relationship is a *kind of* relationship. For example, a book is a kind of document. A book is a special kind of document (specialization), and a document is the general kind of thing that a book is (generalization).
4. Delegation provides a means of code reuse that does not involve inheritance. It also allows modules to be more flexible and more configurable.

12 Dynamic Mid-Level Object-Oriented Design: Interaction Models

Chapter Objectives This chapter continues our discussion of mid-level design by introducing UML sequence diagrams as a mid-level dynamic modeling tool and discussing heuristics for designing object interactions.

By the end of this chapter you will be able to

- Read UML sequence diagrams and use rules and heuristics to write good sequence diagrams for interaction design;
- Design object interactions and develop mid-level static and dynamic models together;
- Describe several interaction control styles, recognize them, and use good control style in interaction design; and
- Describe and use various heuristics to make good object interaction designs.

Chapter Contents 12.1 UML Sequence Diagrams
12.2 Interaction Design Process
12.3 Interaction Modeling Heuristics

12.1 UML Sequence Diagrams

Interaction Diagrams In UML, an **interaction** is the communication behavior of individuals exchanging information to accomplish a task. **Interaction diagrams** are the UML notations for dynamic modeling of collaborations, a central focus of engineering design (especially mid-level design). UML has four interaction diagram notations:

- A *sequence diagram* shows interacting individuals along the top of the diagram and messages passed among them arranged in temporal order down the page.
- A *communication diagram* shows messages superimposed on a diagram depicting collaborating individuals and the links among them.
- An *interaction overview diagram* is an activity diagram whose activity nodes are replaced with fragments of sequence diagrams.
- A *timing diagram* shows the change of an individual's state over time.

Each of these diagrams emphasizes some aspect of collaboration. Sequence diagrams emphasize the way that individuals exchange messages over time. Communication diagrams emphasize the links among individuals accommodating communication. Interaction overview diagrams emphasize

the flow of control among several interactions. Timing diagrams emphasize timing constraints on individuals during interaction.

Each type of diagram has advantages, but by far the most useful and popular interaction diagram is the sequence diagram. This is because the message flow between individuals is the essential aspect of collaborations. We use sequence diagrams for interaction modeling in this book and present them in this section. We do not cover communication, interaction overview, and timing diagrams.

Sequence Diagram Frames

A sequence diagram has a *frame* consisting of a rectangle with a pentagon in its upper left-hand corner. The pentagon is its *name compartment*; the interaction is represented inside the rectangle. The string in the name compartment has the form

sd interactionIdentifier

where *interactionIdentifier* is either a simple name or an operation specification with the same format as in a class diagram. To illustrate, Figure 12-1-1 shows two sequence diagram frames.

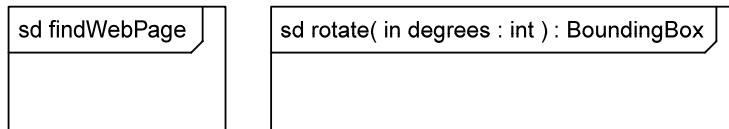


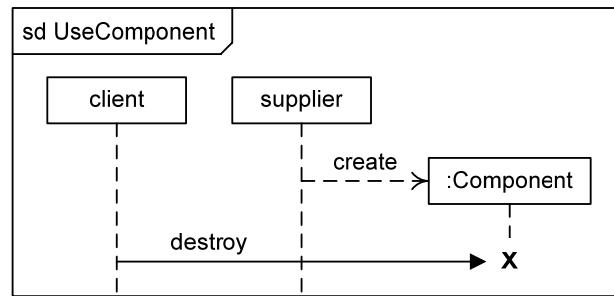
Figure 12-1-1 Sequence Diagram Frames

Lifelines

Sequence diagrams show individuals participating in an interaction across the top of the framed region. These are depicted by lifelines. A *lifeline* is a rectangle containing an identifier with a dashed line extending below the rectangle. The vertical dimension of a sequence diagram represents time, and the dashed line in a lifeline represents the time during the interaction when the represented individual exists. The horizontal dimension represents individuals, but their order is not significant.

If an object is created in the course of the interaction, then it appears vertically at the point of its creation. An object destroyed during an interaction has a truncated lifeline marked with a large **X** at the point of destruction. Objects that persist throughout an interaction have lifelines running from the top to the bottom of the diagram. Figure 12-1-2 illustrates object creation and destruction.

In this interaction, a **supplier** object creates a new **Component** instance, as indicated by the dashed arrow labeled **create**. The **client** object later destroys the **Component** object, as indicated by the arrow labeled **destroy**. Note that the **Component** instance appears vertically at the point of its creation, and its lifeline ends in an **X** when it is destroyed. The **client** and **supplier** objects exist for the entire interaction, so their lifelines run to the bottom of the diagram.

**Figure 12-1-2 Lifelines Indicating Individual Existence**

The identifiers written inside lifeline rectangles may show the individual's name, type, or both. Lifeline identifiers have the form shown in Figure 12-1-3.

Lifeline Identifier Format

name[selector] : typeName

Where:

name—The name of the individual, which must be a simple name or the word **self**. The *name* is optional.

selector—An expression in a format not specified by UML picking out an individual from a collection (such as a set or list). The *selector* is optional; if it does not appear, then neither do the square brackets. A *selector* can only be used with a *name*.

typeName—The name of the type of individual represented by the lifeline. The *typeName* is optional; if it does not appear, then neither does the colon preceding it.

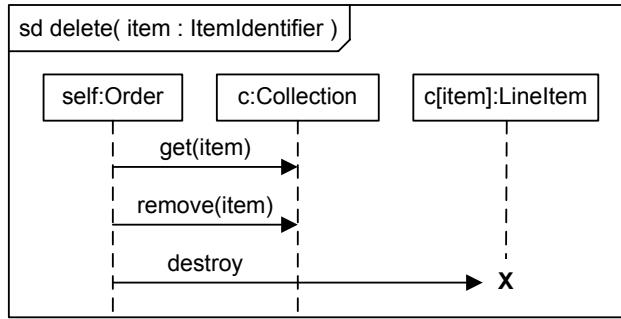
Although all elements of a lifeline identifier are optional, the identifier cannot be empty, so either the *name* (with or without a *selector*), the *typeName*, or both must appear.

Figure 12-1-3 Lifeline Identifier Format

A lifeline whose identifier indicates its type but not its name represents an *anonymous individual*.

The special name **self** is used when a sequence diagram depicts an interaction “owned by” one of the interacting individuals, such as the interaction that occurs when an object’s method is invoked. To illustrate, suppose that an **Order** object has a **delete()** operation for removing a **LineItem** from the **Order** given its **itemIdentifier**. Figure 12-1-4 depicts the interaction that takes place between the **Order** object and a collection it uses to hold **LineItems**.

This diagram models the interaction realized by the **Order.delete()** operation. The object whose **delete()** method is called is an anonymous **Order** instance

**Figure 12-1-4 Using a self Object**

that must somehow be associated with the operation indicated in the sequence diagram name compartment. The name `self` is how this association is made. When the `delete()` operation executes, it calls the `Collection`'s `get()` method to obtain a reference to the deleted item. It then tells the `Collection` to remove the item. Finally, the `self` object destroys the deleted `LineItem`. Note that the `LineItem` name shows that it is an element of the `Collection` `c` whose selector is the `item` value passed to the `delete()` operation as its argument.

Lifelines represent individuals participating in an interaction. These individuals may be actors and a software product, architectural components, packages, sub-systems, modules, or objects. In other words, sequence diagrams can model interactions at all levels of abstraction in software design. In this and later chapters we will concentrate on using them for mid-level design, and the individuals in our examples will be class instances.

Messages and Message Arrows

An object can call another's operations, create or destroy another, raise an exception that another catches, or send a signal to another object. All communications from one object to another are called **messages** and are represented by *message arrows* in sequence diagrams. Message arrows go from the sending object's lifeline to the receiving object's lifeline. An object may send a message to itself, in which case the message arrow curves down and back to the same lifeline.

There are three kinds of message arrows, shown in Figure 12-1-5.

- Synchronous message
- Asynchronous message
- - - - - → Synchronous message return
or instance creation

Figure 12-1-5 Three Kinds of Message Arrows

The *synchronous* message arrow is used when a sending individual suspends execution after sending the message. This is what normally occurs when an operation is called. The suspended individual resumes execution when the called operation returns. The *asynchronous* message arrow is used when a sending individual continues execution after sending the message. This can occur when one individual signals another running in a different thread or process. The dashed arrow is used either to show the *return* of control from a synchronous message or to create a new entity. It is easy to distinguish these two uses because creation arrows always point at a lifeline rectangle while return arrows always point at lifeline dashed lines. Returns from synchronous messages need not be shown, so return arrows are optional. Figure 12-1-6 illustrates the use of different message arrows.

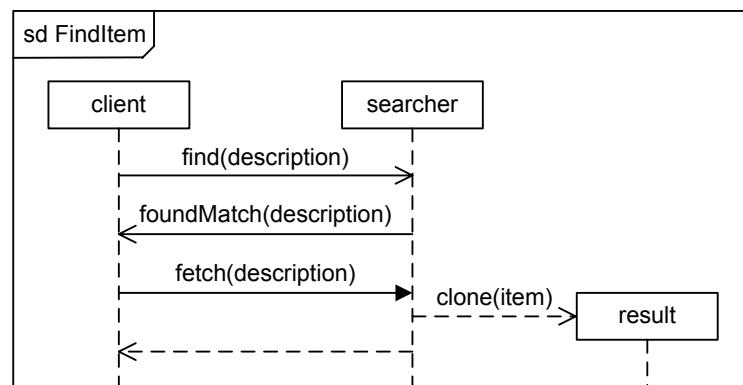


Figure 12-1-6 Using Message Arrows

In this diagram, the **client** object asks the **searcher** to find something matching a **description**. This request is sent asynchronously, so the **client** can continue its processing while the **searcher** looks for a matching item. When the **searcher** locates a result, it asynchronously notifies the **client**, allowing the **searcher** to go on with searches for other clients. When it is ready, the **client** calls the **searcher's** **fetch()** operation. This is a synchronous call, so the **client** suspends execution until the operation returns. The **searcher** creates a copy of the desired item to return to the **client** by cloning the item, indicated by the dashed arrow. The final dashed arrow represents return of control from the **searcher** to the **client** when the **fetch()** operation completes execution.

Message arrows are generally drawn horizontally to model messages as instantaneous communications between objects. Usually this is satisfactory, but if the time that it takes a message to get from one object to another must be modeled, then message arrows can be drawn obliquely down the page between lifelines.

Message Specifications

Every synchronous and asynchronous message arrow must be labeled with a *message specification*; return arrows need not be labeled, and we adopt the

convention that they never are. A message specification represents an operation call, the raising of an exception, or the sending of a signal. Consequently, the message specification's syntax is much like that of a function or procedure call in a programming language.

Message specification syntax rules are summarized in Figure 12-1-7.

Sequence Diagram Message Specification Format

variable = *name argumentList*

Where:

variable—A simple name standing for a variable assigned the return value of the message. The name designates a variable accessible to the object represented by the lifeline sending the message. Variable assignments may be suppressed. If a message does not return a value, the value is not saved, or the variable assigned to is suppressed, then the *variable* and the assignment operator (=) are both omitted.

name—A simple name designating the message. It cannot be suppressed.

argumentList—A comma-separated list of message arguments enclosed in parentheses. Each argument has one of the following formats:

- (a) *varName* = *paramName*
- (b) *paramName* = *argumentValue*
- (c) –

Format (a) represents the assignment of a value returned through an out, inout, or return parameter, designated by the *paramName*, to a variable accessible to the sending object, designated by the *varName*. Both the *varName* and the *paramName* must be simple names. The *paramName* and the assignment operator (=) may be omitted if the *varName* occupies a position in the *argumentList* corresponding to an out, inout, or return parameter.

Format (b) represents passing a value, designated by *argumentValue*, through a parameter, designated by *paramName*. The *paramName* must be a simple name. It may be omitted along with the assignment operator (=), in which case the position of the *argumentValue* in the *argumentList* indicates which parameter gets the value. The *argumentValue* is an expression in a format not specified by UML. It represents a value, which may be constant, a variable accessible to the object sending the message, or an expression.

Format (c) is a placeholder for an undefined or suppressed argument. The parentheses may appear even if the message has no arguments. The *argumentList* may be suppressed.

A message may also be labeled with only an asterisk (*). The asterisk specifies that any message may be sent.

Figure 12-1-7 Sequence Diagram Message Specification Format

The sequence diagrams in Figures 12-1-2, 12-1-4, and 12-1-6 illustrate message specifications consisting of only names and names with simple arguments. More message specification examples appear below.

`total = sum` specifies that the variable `total` is assigned the value returned by the `sum` operation.

`find(flag = isFound, where = location, x)` specifies that the local variables `flag` and `where` are assigned the values of the out parameters `isFound` and `location`. The argument `x` is passed into the operation.

`getTokenProperties(word, type = theType, -)` specifies that the `getTokenProperties()` operation is called with argument `word` passed in, the type of the word passed out and assigned to the `type` variable, and the third argument not specified.

More examples of message specifications and their uses appear in the remainder of this chapter.

Execution Occurrences

An operation is **executing** when some process is actually running its code. An operation is **suspended** when it sends a synchronous message and it is waiting for the message to return. An operation is **active** when it is either executing or suspended.

An object is **active** if one or more of its operations is active. Objects are typically not active during an entire interaction. The period when an object is active can be shown using an *execution occurrence*, which is a thin rectangle covering the dashed line of an individual's lifeline during the periods it is active.

A synchronous message (an operation call) always initiates a new execution occurrence, and return from the operation marks its end. An asynchronous message may or may not start a new execution occurrence. An object that calls one of its own operations starts a new execution occurrence for execution of that call. Figure 12-1-8 illustrates execution occurrences.

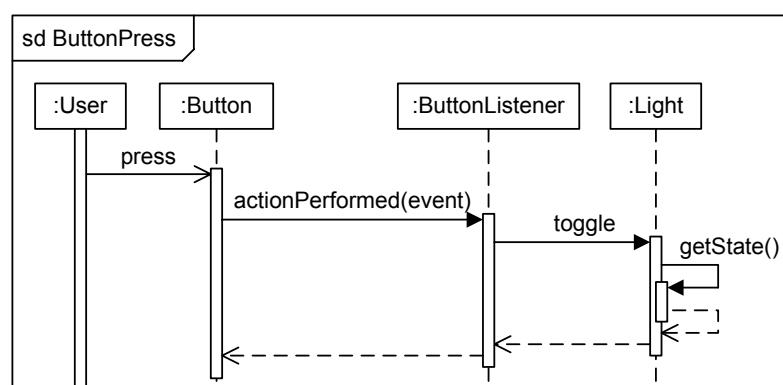


Figure 12-1-8 Execution Occurrences

This diagram shows that the **Button** object is activated when pressed by a **User**. The **User** continues operation, so the message is asynchronous. This message starts a **Button** execution occurrence. The **Button** calls the `actionPerformed()` operation of the **ButtonListener**, which initiates an execution occurrence of the **ButtonListener** object. The **Button** class is still active, though it is suspended until the call of `actionPerformed()` returns. The `actionPerformed()` operation calls `Light.toggle()`, starting an execution occurrence of the **Light** object. The **Light** object executes its `toggle()` computation, during which it calls one of its own operations, `getState()`. This call initiates another execution occurrence on top of the execution occurrence already present on the **Light** object. When `getState()` returns, its execution occurrence ends. Then `toggle()` returns, ending the first **Light** execution occurrence. The **ButtonListener** then returns control to the **Button**, ending the **ButtonListener** execution occurrence. The **Button** then deactivates itself and its execution occurrence ends. Notice that the **User** execution occurrence spans the entire diagram because the **User** is active during the entire period of the interaction.

Interaction Fragments

Most interactions proceed differently depending on circumstances. Sequence diagrams are able to show various message flows using interaction fragments. An *interaction fragment* is a marked part of an interaction specification. Interaction fragments can be combined in a sequence diagram to show different message flows, including branching, iteration, parallel execution, and so forth. There are about a dozen kinds of interaction fragments; we focus on four fragments sufficient to specify non-parallel message flows.

An interaction fragment has a rectangular frame with a pentagonal *operation compartment* in the upper left-hand corner. The interaction fragment *operation* is named in the operation compartment. The area inside the frame may be divided into stacked rectangles by dashed horizontal lines. The regions resulting from these divisions hold the interaction fragment *operands*. The frame is superimposed on lifelines in a sequence diagram to enclose portions of the interaction in the operand regions. Figure 12-1-9 illustrates the form of an interaction fragment.

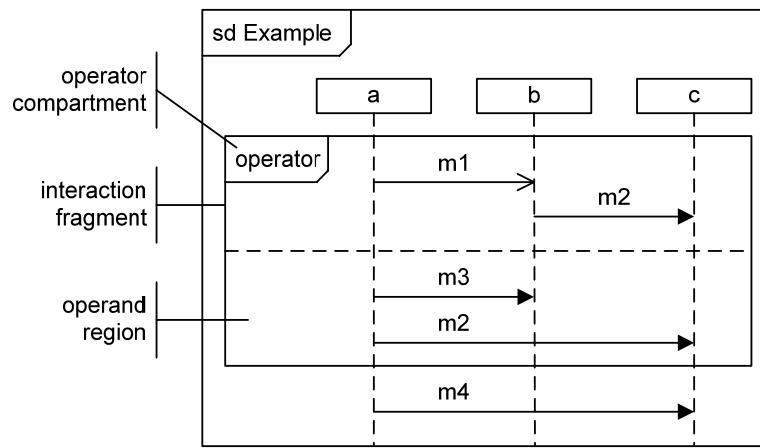


Figure 12-1-9 Interaction Fragment Layout

Optional Fragments

The first kind of interaction fragment we consider is the optional fragment. An *optional fragment* is a portion of an interaction that may be done. The fragment operator name is *opt*. Optional fragments have only a single operand, which must contain a guard. As in other UML diagrams, a *guard* is a Boolean expression enclosed in square brackets. The format of the Boolean expression is not specified by UML, except for the special guard *[else]*, which is true just in case the guards on every other operand are false. Operand guards must be placed inside the operand region at the top of the lifeline representing the individual that evaluates the guard.

The interaction in the optional fragment is performed if the operand guard is true at the point during message exchange where the optional fragment occurs. For example, consider the sequence diagram in Figure 12-1-10.

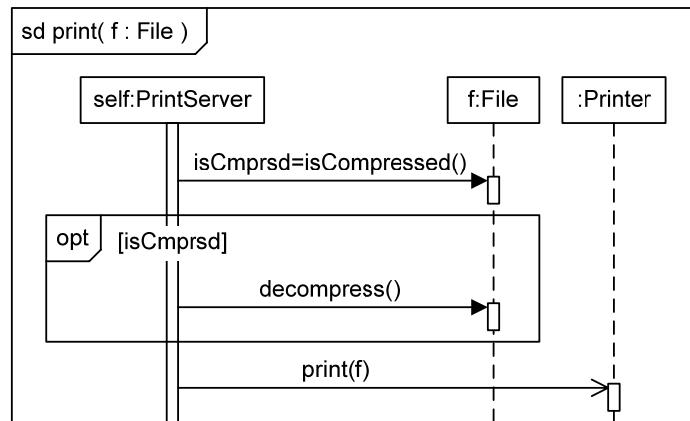


Figure 12-1-10 An Optional Fragment

This sequence diagram models how a `PrintServer` handles print requests. It first determines whether the `File` it is supposed to print is compressed. Then an optional fragment is encountered. The operand guard is evaluated at this point; it is true only if the file is compressed. Thus, the `decompress()` message in the optional fragment is sent only if the file is compressed. After that, the `PrintServer` sends an asynchronous message to a `Printer` to print the file.

The optional fragment works like an `if-then` statement in a procedural programming language. Next we consider an interaction fragment that works like an `if-then-else-if` statement.

Alternative Fragments

An *alternative fragment* has one or more guarded operands whose guards are mutually exclusive—that is, at most one of them can be true at any time. The operand whose guard is true (if any) is performed. If no guard is true, then none of the operands is performed. If more than one is true, then the behavior of the fragment is undefined. The alternative fragment operator name is `alt`.

Figure 12-1-11 illustrates the alternative fragment.

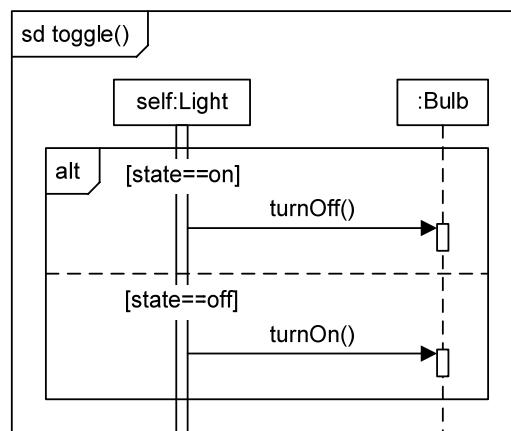


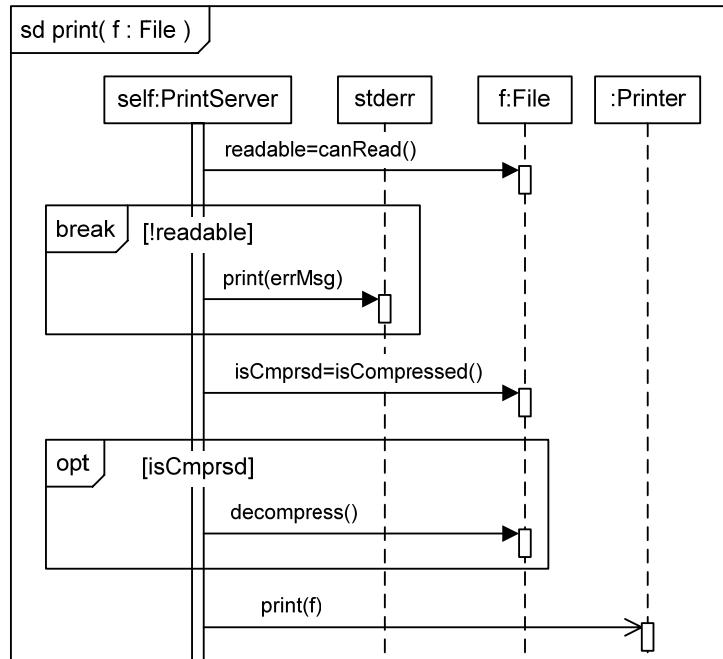
Figure 12-1-11 An Alternative Fragment

According to this sequence diagram, when a `Light` object receives a `toggle()` message, it takes one of two actions depending on the value of its `state` variable. If `state` is on, it sends a `turnOff()` message to the `Bulb` object. If `state` is off, it sends a `turnOn()` message to the `Bulb` object.

Break Fragments

A *break fragment* has a single operand that is performed instead of the remainder of the enclosing fragment or diagram if the operand guard is true. The break fragment thus works like the `break` statement in Java. The fragment operator name is `break`.

Figure 12-1-12 elaborates a previous example to illustrate the break fragment.

**Figure 12-1-12 A Break Fragment**

In this model of the PrintServer printing interaction, the PrintServer first checks whether it can read the file that has been sent to it. If not, the break fragment prints an error message and the rest of the enclosing fragment or diagram (in this case, the diagram) is skipped. If the break fragment guard is false, then it is skipped and the rest of the interaction proceeds as before.

Loop Fragments Sequence diagrams can also show iteration. A *loop fragment* has a single operand that may or may not have a guard. The operand is the loop body. The loop operator and the guard together control the loop.

The loop operator has the following format:

loop (min, max)

The parenthesized expression contains the *loop parameters* *min* and *max*. The parameters are optional; if there are none, then the parentheses are omitted as well. The *min* parameter is a non-negative integer; *max* is a non-negative integer at least as large as *min*, or *. The *max* value is optional; if it does not appear, then neither does the comma separating it from *min*.

The loop is controlled according to the following rules:

- The loop body is performed at least *min* times and at most *max* times.
- If the loop body has been performed at least *min* times but less than *max* times, then it is performed only if the operand guard is true.
- If *max* is *, then the upper iteration bound is unlimited.

- If *max* is not specified but *min* is, then *max*=*min*.
- If the loop has no parameters, then *min*=0 and *max* is unlimited.
- The default value of the guard is true.

These rules allow for almost any sort of loop control. For example, a loop that iterates 10 times (such as a Java `for` loop) is specified by the operator `loop (10)` and no guard. A loop that iterates as long as some Boolean test is true (such as a Java `while` loop) is specified by a loop operator with no parameters and a guard containing the Boolean test.

Figure 12-1-13 illustrates loop fragments.

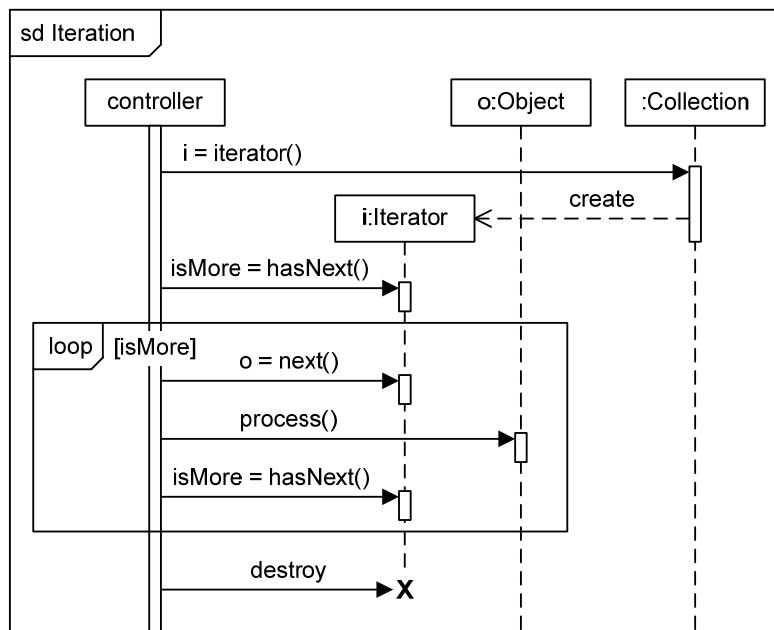


Figure 12-1-13 A Loop Fragment

This sequence diagram shows how iteration over a collection works using an `Iterator` object (a subject explored in depth in Chapter 16). The `controller` object requests an iterator from the `Collection`, which creates and returns one. The `controller` then asks the `Iterator` whether it has any more `Collection` objects to provide, storing the result in the variable `isMore`. This variable is then tested in a loop fragment. If the loop fragment body is executed, the next object is retrieved from the `Iterator` and processed, and the `Iterator` object is again asked whether it has more objects. The result is stored in `isMore`, and the guard is tested again to determine whether the loop should continue. Eventually, the `Iterator` traverses the entire collection, the loop fragment guard fails, and iteration ends. The `Iterator` object is then destroyed.

**Sequence
Diagram
Heuristics**

The horizontal order of lifelines in a sequence diagram has no significance, but the order does make the diagram more or less readable. The lifeline representing the individual that sends the first message should be leftmost. The following heuristics help order the remaining objects:

Put pairs of lifelines that interact heavily next to one another.

Position lifelines to make message arrows as short as possible.

Position lifelines to make message arrows go from left to right.

It is usually impossible to do all this at once, and common sense must come into play in trading off these heuristics against one another.

A sequence diagram may model an interaction carried out in implementing an operation. When this is the case, the operation signature appears in the sequence diagram's name compartment and the "host" lifeline is identified in the interaction area by being named `self`. Usually the `self` object sends the first message, and it may send most of the messages in the diagram. Generally, then, designers should adhere to the following heuristic:

Put the `self` lifeline leftmost.

Also, in a sequence diagram modeling an operation's interactions, the `self` object must be active throughout the interaction—it is only deactivated when the operation returns, which occurs when the diagram ends. Hence, we have the following heuristic:

In a sequence diagram modeling an operation interaction, draw the `self` execution occurrence from the top to the bottom of the diagram.

Individuals should be named when they are message arguments or are used in expressions. An individual might also be named if doing so helps make a sequence diagram more understandable. Otherwise, individuals should be anonymous—irrelevant names only clutter up the diagram.

Name individuals only if they are message arguments or are used in expressions.

Pressing a button in the user interface of a complicated program might invoke an operation that ultimately involves the interaction of dozens of individuals exchanging hundreds of messages. A sequence diagram recording so much detail would be almost impossible to draw and very hard to read. It is important to suppress details when making sequence diagrams. Ways to do this are suggested in the following heuristics:

Choose a level of abstraction for the sequence diagram. The level of abstraction can be based on how far removed individuals are from the primary individual in the interaction. The *primary individual* in an interaction is the one sending or receiving the first message. The primary individual interacts directly with several individuals—these are at one remove from the primary individual. The individuals at one remove may interact with several other individuals that the primary individual does not interact with—these individuals are at two removes from the primary individual. Other individuals may be at three, four, or more removes from the primary individual. Choose a level of remove k for a sequence diagram so that it has no more than about seven individuals. Usually, k will be

three or less. Messages in a sequence diagram that are themselves implemented by important interactions not shown in the diagram can have their own diagrams.

Suppress messages that individuals send themselves that do not generate messages to other individuals. Although such messages are important for making complex computations, testing conditions, and so forth, they do not have to be shown to understand interactions, which is the goal of the sequence diagram.

Suppress return arrows when using execution occurrences. Return arrows are redundant when execution occurrences are present, so a sequence diagram can be simplified somewhat by removing them.

Finally, you may have noticed when considering message examples that there is no way to distinguish an argument specification in which a variable is assigned a value from an out, inout, or return parameter *y*, and a specification in which an argument value is assigned to a parameter by name. For example, the message *m(x=y)* may be assigning the variable *x* a value returned by the message through parameter *y*, or it may be assigning parameter *x* the value *y*.

This ambiguity can be avoided if parameters are assigned values based on their positions in the argument list rather than their names. In other words, we can clear up this confusion if we never assign parameters argument values by name. Following this convention, we know that in the previous example, *x* is a variable assigned a value returned from the message through parameter *y*. Consequently, we state the following heuristic:

Don't assign values to message parameters by name.

Sequence Diagram Uses

Sequence diagrams are useful tools in mid-level design, as the examples throughout this and later chapters show. But sequence diagrams are also useful in product and architectural design.

In product design, a sequence diagram can be made with the product as one individual and actors as the others. Such *system sequence diagrams* can show the interactions that occur in scenarios or use cases. System sequence diagrams can help develop use case models, or they can serve as (partial) use case descriptions.

In architectural design, sequence diagrams can show major program constituents as individuals, thus modeling architectural constituent interactions.

Heuristics Summary

Figure 12-1-14 summarizes the sequence diagram heuristics discussed in this section.

- Put the sender of the first message leftmost.
- Put pairs of individuals that interact heavily next to one another.
- Position individuals to make message arrows as short as possible.
- Position individuals to make message arrows go from left to right.
- Put the self lifeline leftmost.
- In a sequence diagram modeling an operation interaction, draw the self execution occurrence from the top to the bottom of the diagram.
- Name individuals only if they are message arguments or are used in expressions.
- Choose a level of abstraction for the sequence diagram.
- Suppress messages individuals send to themselves unless they generate messages to other individuals.
- Suppress return arrows when using execution occurrences.
- Don't assign values to message parameters by name.

Figure 12-1-14 Sequence Diagram Heuristics

Section Summary

- In UML, an **interaction** is the communication behavior of individuals exchanging information to accomplish a task.
- UML has four kinds of **interaction diagrams**: sequence, communication, interaction overview, and timing diagrams.
- UML sequence diagrams describe interactions by showing the sequence of messages exchanged by individuals over time.
- Individual existence during interactions is indicated by the placement of the lifeline rectangle and the extent of the dashed line that extends below it.
- Message arrows show synchronous and asynchronous messages, returns from synchronous messages, and individual creation.
- Message specifications include the message name, arguments, and assignments of message return values to variables.
- Execution occurrences show when an object is **active**.
- An **interaction fragment** is a marked part of an interaction specification.
- Interaction fragments have operators and operands that together determine which portions of an interaction are performed under various circumstances.
- Among the various kinds of interaction fragments are optional, alternative, break, and loop fragments.
- Sequence diagrams can model interactions in mid-level design, architectural design, and product design.

Review Quiz 12.1

1. What two things are dashed lines used for in sequence diagrams?
2. What is a selector in a lifeline identifier, and what is it used for?

3. What is the difference between a synchronous and an asynchronous message?
 4. Explain when an operation is active, suspended, or executing. Can an object be suspended?
 5. How many operands can optional, alternative, break, and loop fragments have?
-

12.2 Interaction Design Process

Component and Interaction Co-Design

As noted in Chapter 11, components and interactions are refined together during mid-level design. When designing an interaction, computational services will be needed. Designers can check static models to see whether any components already provide the needed service. If not, the designers can check the static model again for components that might take it on as a new responsibility. Good candidates may be found. If not, the designers can add a new component to provide the needed service. Even if no new components are introduced, designers have to change components to store additional data and perform new operations, so components' features change during interaction design. When designers add components, responsibilities tend to shift. This may cause changes in interactions. Interaction design thus leads to changes in component models, which in turn may alter already designed interactions. We refer to the process of generating, evaluating, and improving designs of components and interactions together as **component and interaction co-design**.

Besides the instability resulting from modifying and combining component and interaction designs together, mid-level design is further complicated by the need to consider design alternatives. As noted so many times before, the first design idea is rarely the best—designers need to generate and consider alternatives. But alternative interaction models usually go with alternative component models. Keeping track of the alternatives and what models belong together can be a burden.

An interaction realizing a product function or operation may be specified in a sequence diagram that ignores individuals at several removes from the primary individual in the interaction (as discussed in the last section). Also, some operations in sequence diagrams may be complicated enough to require their own interaction models. Designers may thus need to model product functions and operations at several levels of abstraction in several sequence diagrams.

Outside-In Design

As with any sort of design process, the interaction design process should be mainly top down, moving from higher to lower levels of abstraction. Interactions specify how program components work together to achieve computational goals, so a top-down approach means starting with the highest-level computational goals, which are the computational goals of the program as a whole. These goals are specified in the SRS. Thus, interaction design works mainly from the computational goals mandated in

requirements specifications (the outside of the program), to their realization as interactions between cooperating program units (the inside of the program). This is sometimes called **outside-in design**. This outside-in movement spans architectural and detailed design.

In outside-in design, designers consider each operation or interaction that a program has with its environment in turn and formulate an interaction between program components to implement it. Many program interactions do not give rise to interesting or complicated program component interactions. For example, when a user sets a program parameter that happens to be an attribute of an application domain object, the interaction usually consists of a single operation call from a user interface object to the application domain object. It would be a waste of time for designers to specify trivial interactions such as these or to document them in sequence diagrams. Designers should concentrate instead on operations involving several components, especially when it is not obvious how the interactions should go.

In this section we illustrate mid-level interaction design and component and interaction co-design with a simple example from the Caldera product introduced in Chapter 7.

Caldera Example

Recall that Caldera controls a water heater's thermostat, setting it so that water is kept hot only when it is likely to be needed. The Caldera program description is reproduced below.

Caldera Description

Caldera is a smart water heater controller that attaches to the thermostat of a water heater and provides more efficient control of the water temperature to save money and protect the environment. Caldera sets the water heater thermostat high when hot water is much in demand and sets it low when there is not much demand. For example, Caldera can be told to set the thermostat high on weekday mornings and evenings and all day on weekends, and low during the middle of the day and at night. Furthermore, Caldera can be told to set the thermostat high all the time in case of illness or other need or be told to set the thermostat low all the time in case of vacation or some other prolonged absence from the house.

The homeowner can specify values for the following Caldera parameters:

Low Temp—Temperature when little or no hot water is needed.

High Temp—Temperature when much hot water is needed.

Weekend Days—Days when the thermostat will be set to *High Temp* all day long; on all other days it will vary between *Low Temp* and *High Temp*.

Peak Times—From one to three time periods on a 24-hour clock during which the thermostat will be set to *High Temp* on non-*Weekend Days*. On *Weekend Days*, the thermostat will be set to *High Temp* during the entire period between the earliest time and the latest time set in *Peak Times*.

Mode—One of the following Caldera states:

Stay Low Mode—Thermostat is set to stay at *Low Temp*.

Stay High Mode—Thermostat is set to stay at *High Temp*.

Normal Mode—Thermostat is changed between *Low Temp* and *High Temp* on a regular schedule, as explained above.

Caldera has its own internal clock that it checks every second to determine how to set the water heater thermostat.

Figure 12-2-1 is a draft design class model produced by transforming the conceptual model from Chapter 7.

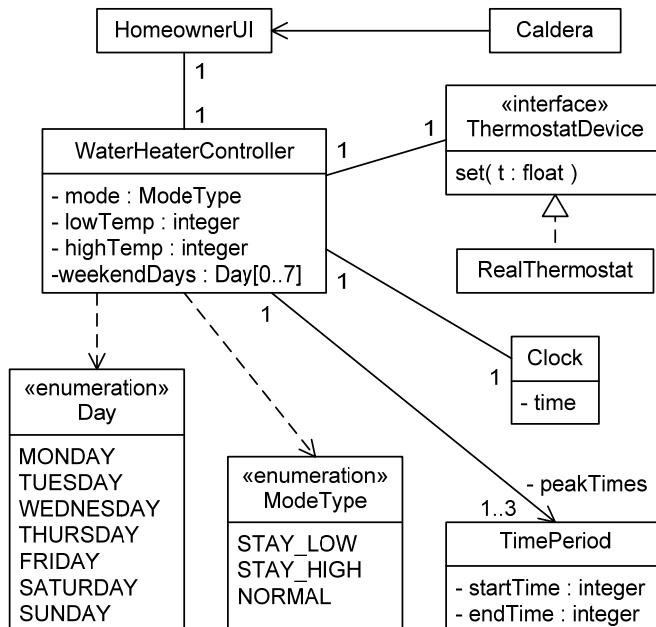


Figure 12-2-1 Draft Caldera Design Class Model

Caldera must provide several functions to homeowners, such as displaying and setting program parameters. The interactions supporting these functions are all quite straightforward (though there may be some complexities within the `HomeownerUI`). One product function that is more complex is what happens when time passes and the `WaterHeaterController` must adjust the thermostat setting. Let's consider this interaction.

Adjusting the Thermostat

The first thing to consider is how the `WaterHeaterController` and the `Clock` interact so that the `WaterHeaterController` can determine that time has passed. There are two general approaches to solving this problem. One

approach, called **polling**, works by having one object (the *observer*) query another object (the *subject*) until some subject condition becomes true. To use polling in this case, the *WaterHeaterController* (the observer) would poll the *Clock* (the subject) in a loop to determine that some amount of time has passed. In the alternative approach, called **notification**, the observer is passive until the subject notifies it that some subject condition is true. To use notification in this case, the *WaterHeaterController* (the observer) would be passive until the *Clock* (the subject) notifies it that some amount of time has passed. Once notified, the *WaterHeaterController* can take action. These alternatives are illustrated in Figure 12-2-2.

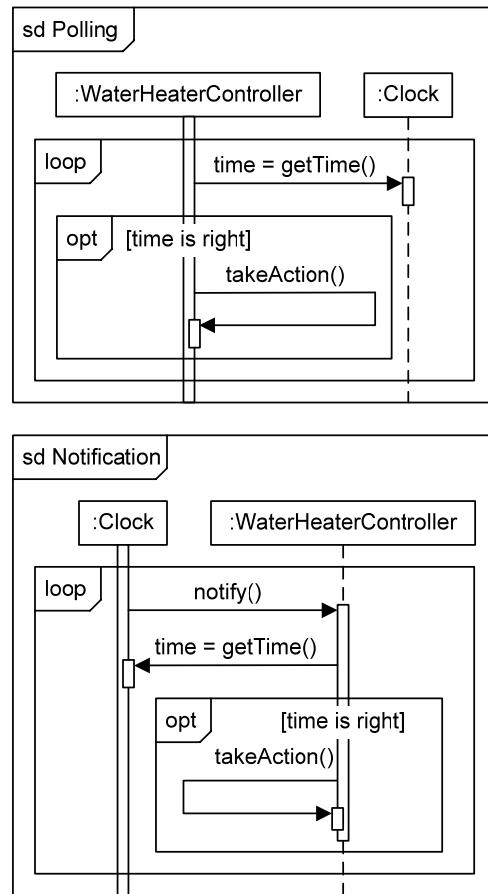


Figure 12-2-2 Polling Versus Notification

These alternatives at first appear more or less equivalent—they are about equally simple, hide the same information, couple the classes to roughly the same extent, and so forth—but they are not equivalent for two reasons. The first is the degree of cohesion. With polling, the *WaterHeaterController* is

responsible for finding out whether time has passed, but with notification, the `Clock` has this responsibility. When put in these terms, it is clear that this responsibility, because it has to do with time, belongs with the `Clock`, and putting it there makes both classes more cohesive.

The second consideration is one of efficiency. When polling, the `WaterHeaterController` is running continuously, but most of the time it is not doing any useful work. With notification, the `Clock` can sleep (suspend execution) until it is time for it to wake up and notify the `WaterHeaterController`. Computation occurs only when there is useful work to do.

These considerations point to notification as the better alternative for the interaction between the `WaterHeaterController` and the `Clock`, so we elaborate this alternative. The `WaterHeaterController` must have a `notify()` operation, so we add this to the class model. The `Clock` calls `notify()` every second.

Next we consider how the `WaterHeaterController` decides that the time is right to take some action and what action to take. The action to be taken is to adjust the water heater's thermostat to a new setting. The `ThermostatDevice` interface has an operation to do this job. This action needs to occur only when Caldera is in `NORMAL` mode and a time when the thermostat needs to be set to a peak time has just begun or ended. The `WaterHeaterController` knows its mode, so no interaction is needed to determine it. We assume that the interaction we are designing is the one that takes place when Caldera is running in `NORMAL` mode.

Who should determine whether the current time is one at which an action should take place? The `WaterHeaterController` knows the `weekendDays`, and it can get the `startTime` and `endTime` from the `peakTimes` `TimePeriod` objects. The `WaterHeaterController` can then compare the current time and day with the `weekendDays` and `peakTimes` to decide whether it needs to set the thermostat. Note that the `Clock` class will need to keep track of the day as well as the time for this to work. This interaction is specified in Figure 12-2-3, along with a class diagram showing the classes involved in the interaction.

This interaction will work, but let's consider an alternative. One thing that seems awkward is that the `WaterHeaterController` must query `TimePeriod` several times to obtain peak time start and end times. It would be better if this part of the interaction could be avoided.

Perhaps checking the times could be delegated to `TimePeriod`. But this computation involves the day as well as the time, so perhaps the day could be included with the period start and end times in `TimePeriod`. `TimePeriod` could then be entirely responsible for figuring out whether it was time to change the thermostat.

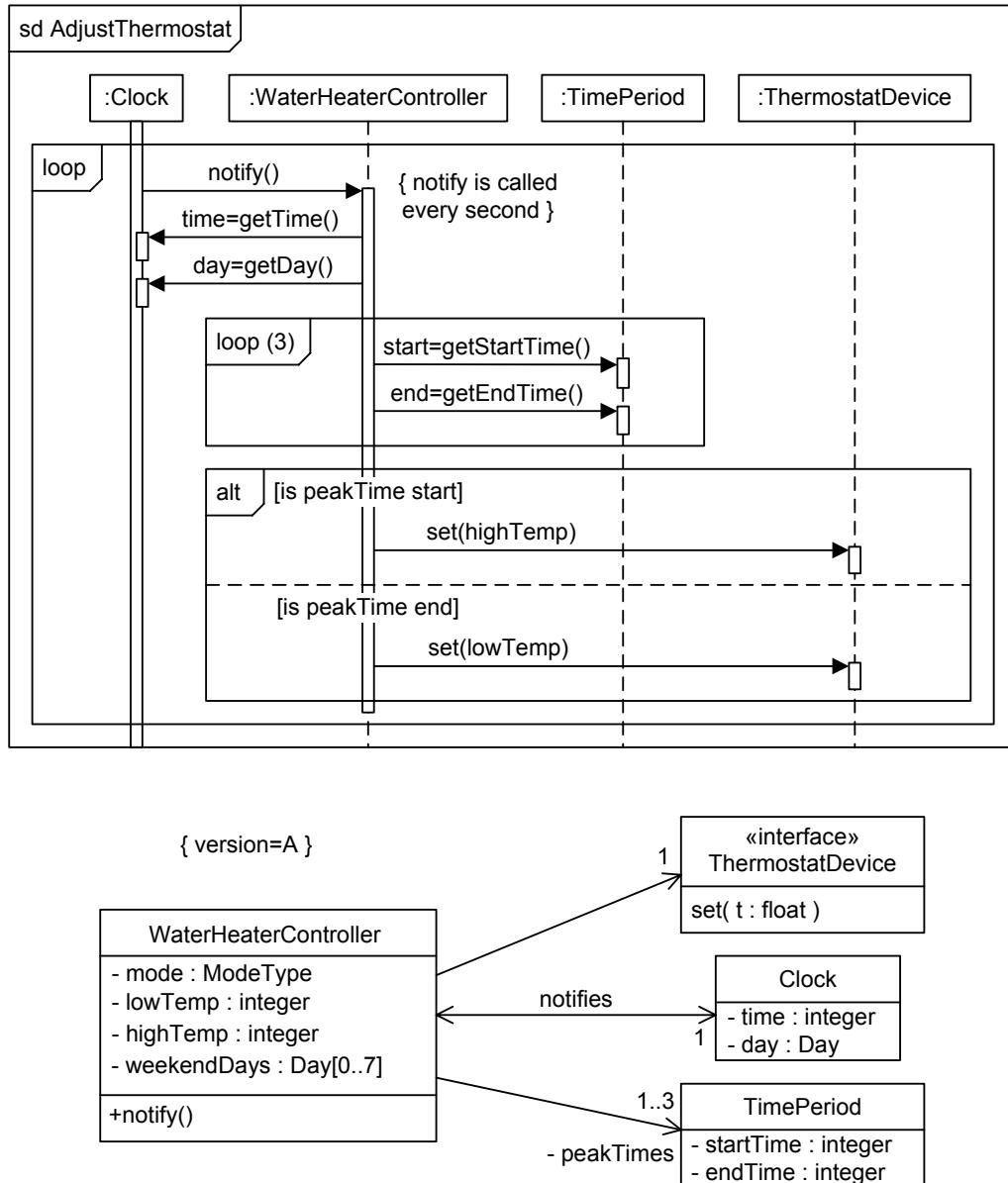
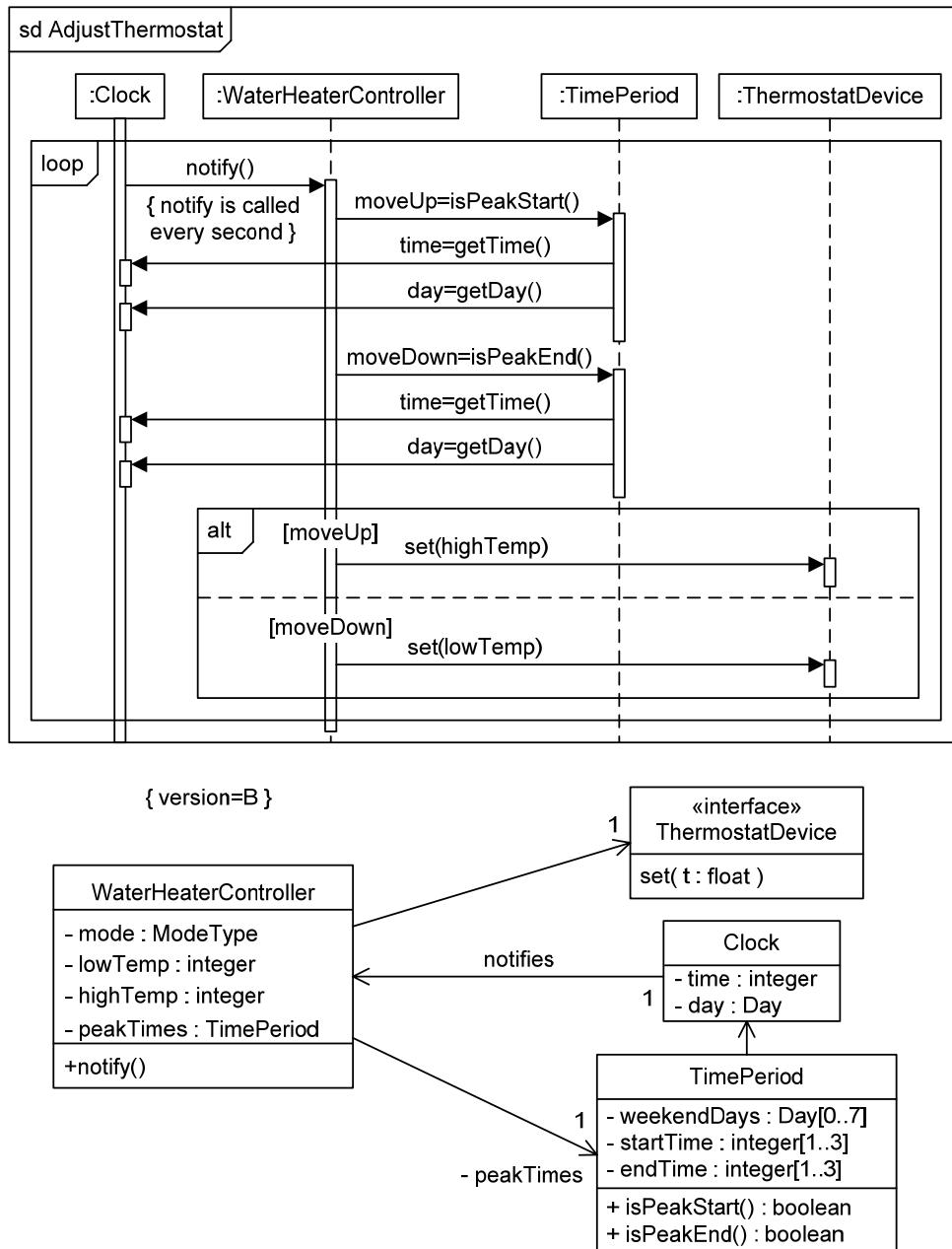


Figure 12-2-3 Adjusting the Thermostat in NORMAL mode, Version A

If you draw up this interaction, you will notice that the WaterHeaterController gets the time and day from the Clock and then passes it to the TimePeriod to be checked. Why not let TimePeriod interact with the Clock directly? This alternative interaction, along with the altered class diagram, appears in Figure 12-2-4.

**Figure 12-2-4 Adjusting the Thermostat in NORMAL mode, Version B**

In comparing these alternatives, we note that `WaterHeaterController` is simpler in version B, and the classes in version B are more cohesive. Fewer classes, however, are coupled in version A. Perhaps there is yet another alternative better than both? We leave consideration of this question to the reader.

In summary, we see from this example that components and interactions must be designed together—that is, there must be component and interaction co-design—and that considering alternatives is as important in mid-level design as it is in architectural and product design.

Section Summary

- Components and interactions are refined together during mid-level design—this is **component and interaction co-design**.
- Interaction design should be a top-down process, starting with program functions and operations and working inwards to the interactions that realize them—this is **outside-in design**.
- **Polling** is an interaction style in which an observer repeatedly queries a subject to find out when some subject condition becomes true.
- **Notification** is an interaction style in which a subject notifies an observer when some subject condition becomes true.

Review Quiz 12.2

1. Do all sequence diagrams begin with program operations or interactions with the environment?
2. Why is polling less efficient than notification?
3. Are the changes made to the Caldera class model in version B likely to affect any other interactions?

12.3 Interaction Modeling Heuristics

Control Style An important role for a program component to play is that of a controller.

A **controller** is a program component that makes decisions and directs other components.

Controllers are important in interaction design because they are the central figures in collaborations. They are usually the components that begin and end interactions, delegate jobs to other components, and return results.

A **control style** is a way that decision making is distributed among program components. We distinguish three broad control styles:

Centralized—In a program with a **centralized control style**, a few controllers make all significant decisions. Non-controller components merely hold data or carry out simple functions.

Delegated—A program with a **delegated control style** has decision making distributed through the program. Controllers make overall decisions and coordinate the activities of other components, but they delegate lower-level decisions to other components.

Dispersed—In a program with a **dispersed control style**, decision making is spread widely throughout the program; it is hard to identify controllers in such programs.

These control styles correspond to management styles in human organizations. The centralized style corresponds to *micro-management* in which supervisors make every decision. The delegated style corresponds to the *empowered teams* model wherein supervisors make major strategic decisions but leave it to teams of employees to decide how to implement the larger decisions. The dispersed style corresponds to a *leaderless* organization, such as occurs when a group of individuals get together for an impromptu activity.

These three styles identify regions of a continuum of responsibility distribution—many programs fall between these marker points. These styles have advantages and disadvantages, which we consider next.

Centralized Control

The main advantage of a program with a centralized control style is that it is easy to find where decisions are being made. If the controllers are not too big and complex, it is easy to understand how decisions are made and to alter the decision-making process. Unfortunately, there are dangers to a centralized control style:

- The controller can become too large and complex. If only a few components make all the decisions, and there are many decisions to be made, it stands to reason that the controllers will be large and complex units. Such controllers are called **bloated controllers**. Bloated controllers have low cohesion and are big modules, violating two modularity principles.
- Controllers may treat other components as data repositories, merely storing and retrieving data in them. This tends to increase coupling and destroy information hiding, violating two more modularity principles.

In general, a centralized control style should be used only when a program makes a few simple decisions.

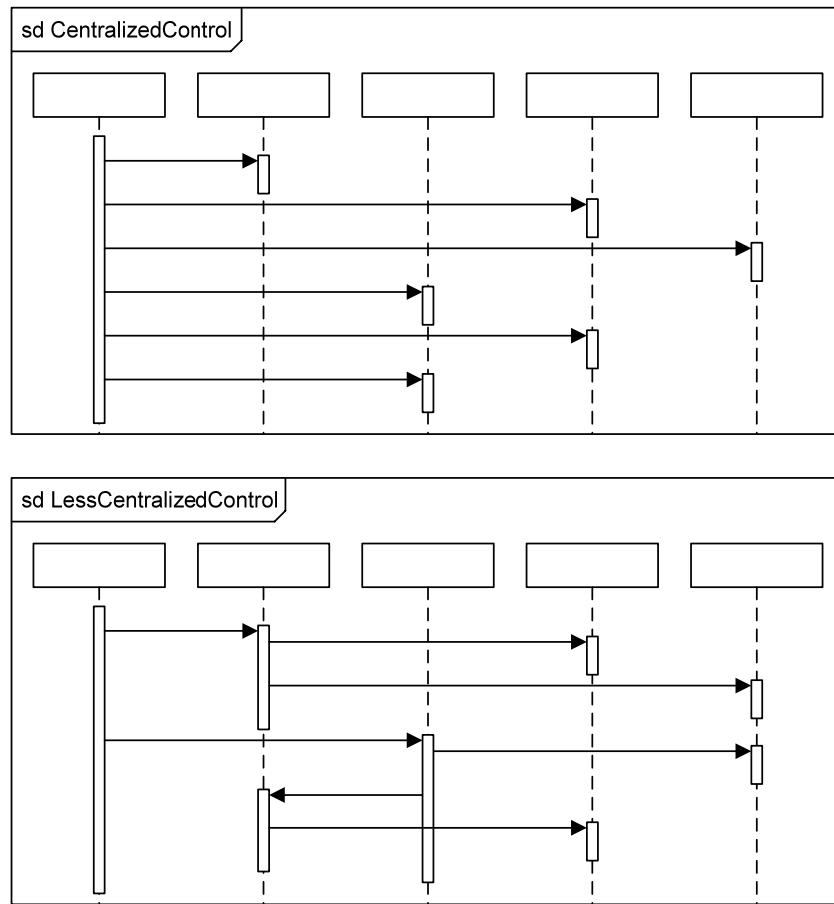
It is fairly easy to recognize an over-centralized control style during interaction design: Controllers tend to be the objects that receive the first message in sequence diagrams. If most messages in the diagram originate with the controllers as well, then the style is probably over-centralized. The schematic diagrams in Figure 12-3-1 on page 383 illustrate this situation.

This discussion suggests the following heuristic:

Avoid interaction designs where most messages originate from a single component.

Several more heuristics (some mentioned before) help avoid bloated controllers or an over-centralized control style:

Keep components small. This straightforward application of the Principle of Small Modules directly combats bloated controllers.



**Figure 12-3-1 Sequence Diagram Forms
Indicating Degree of Centralization**

Make sure operational responsibilities are not all assigned to just a few components. Concentration of operational responsibilities is characteristic of the centralized style.

Make sure operational responsibilities are consistent with data responsibilities. This heuristic is mainly aimed at increasing cohesion, but it also helps distribute operational responsibilities.

- Delegated Control** The delegated control style has several advantages:
- Controllers are coupled to fewer components, and overall program coupling is decreased. When responsibilities are delegated, controllers don't need to know about many components with which their delegates collaborate. This reduces coupling.
 - Information is hidden better. Rather than extracting data from components, modifying it, and returning it to the components (as in the

centralized control style), a delegated style encourages components to modify their data themselves.

- Programs are easier to divide into layers. As we will see, the Layered architectural style is a very important and powerful way to organize program modules. A delegated control style makes a layered organization easier to achieve.

The delegated control style really has no drawbacks—it is the preferred control style, especially for object-oriented systems. The main heuristic (in addition to those above) for realizing a delegated control style during interaction design is to

Have components delegate as many low-level tasks as possible. A component will be responsible for high-level tasks. These are typically accomplished by doing several low-level tasks. In other words, we can decompose functions into simpler functions. These lower-level tasks can often be done by collaborators.

Dispersed Control

Dispersed control is what happens when delegation is taken too far. In such designs, there are many small components holding little data and having a few small operations. Every task must be traced through dozens of interactions. There are several problems with this control style:

- It is too hard to understand the flow of control. A large number of messages must be traced to figure out how anything is done. This makes the design hard to understand and very hard to change.
- When components are too finely divided, they tend not to be able to do anything on their own. As a result, coupling increases.
- It is difficult to hide information. This is because the workings of a single component depend so much on how other components are implemented.

A dispersed control style is recognized during interaction design by the need for many messages emanating from each object. Designers should thus

Avoid interactions that require each component to send many messages.

Control Styles in AquaLush

To illustrate control styles and their application in interaction design, we will consider the interaction design for automatic irrigation in AquaLush. Recall that during automatic irrigation, AquaLush irrigates one zone at a time. It checks the current zone's critical moisture level every minute to see whether it has irrigated enough. It also checks to see whether the water allocation for that zone is exhausted. If either condition is true, the valves for the current zone are shut off, and AquaLush begins irrigating the next zone. When all zones are irrigated, the irrigation cycle is done.

Let us consider the interaction that occurs every minute when AquaLush is irrigating a zone. A draft design class model for the classes involved in this interaction appears in Figure 12-3-2.

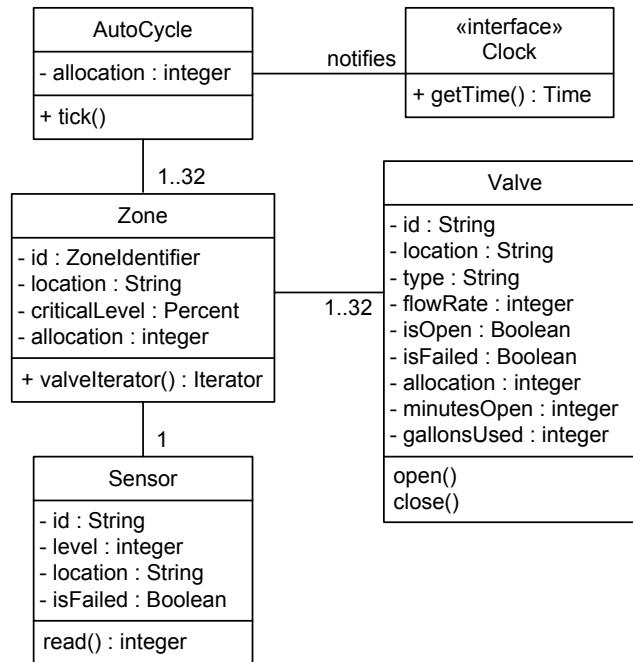


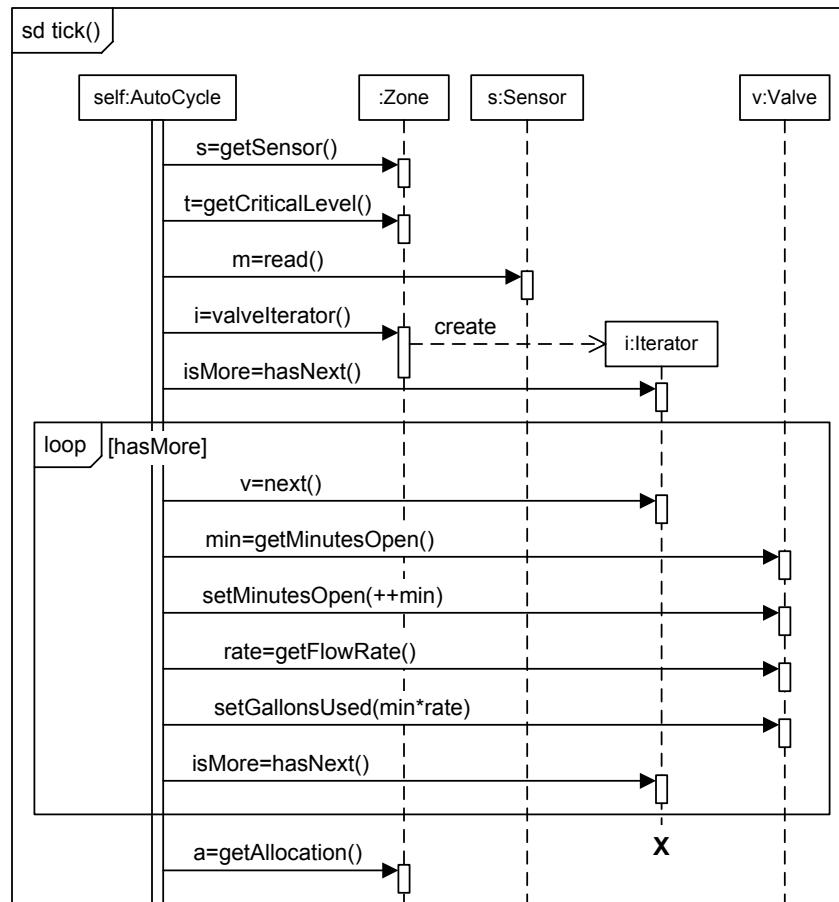
Figure 12-3-2 AquaLush Automatic Irrigation Classes

Every minute, a **Clock** object notifies the **AutoCycle** (the controller for this interaction) that a minute has passed by calling its **tick()** operation. Consider the diagram in Figure 12-3-3 as a design for the interaction that occurs when **tick()** is called. In this diagram it is assumed that it is not yet time to end the irrigation cycle.

In Figure 12-3-3, the **AutoCycle** object exercises complete control over checking the moisture level and water usage. It first obtains the irrigation Zone's **Sensor** object **s** and its critical moisture level **t**, and then queries the **Sensor** for its current moisture level **m**. Had **m** been greater than or equal to **t**, **AutoCycle** would have ended irrigation for the zone at this point.

Assuming this test fails, the **AutoCycle** object next gets an **Iterator** that it uses to obtain each **Valve** instance in a loop. For each **Valve**, the **AutoCycle** gets the minutes it has been open and then increments and sets this value.

AutoCycle next obtains the valve flow rate and sets the gallons used by the **Valve**. Presumably, **AutoCycle** sums the gallons used by each **Valve** so it can compare the total to the zone allocation **a** retrieved from the **Zone**. Had the water allocation been exhausted, **AutoCycle** would have stopped irrigation.

**Figure 12-3-3 Controlling Automatic Irrigation, Version A**

The control style in this interaction is clearly over-centralized. `AutoCycle` delegates nothing to its collaborators. This is a poor interaction design: The implementation of the collaborating objects is hardly hidden, the controller is tightly coupled to all the objects in the diagram, the controller lacks cohesion, and it is too big because it has too many low-level responsibilities. A much better design would have a delegated control style. Consider the sequence diagram in Figure 12-3-4 modeling the same interaction.

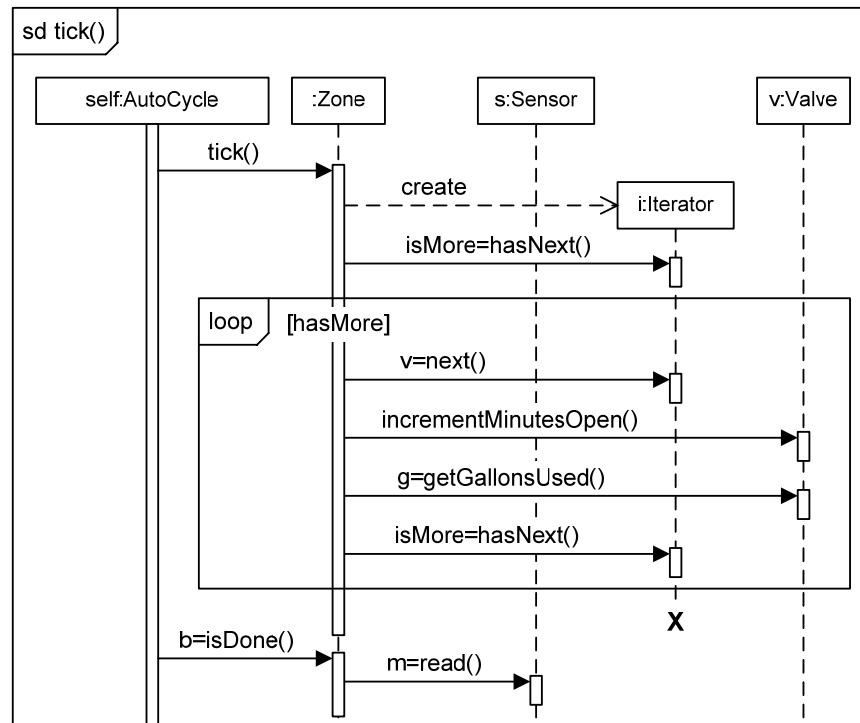


Figure 12-3-4 Controlling Automatic Irrigation, Version B

In this version, `AutoCycle` delegates responsibilities to its collaborators. When it receives a `tick()` from the `Clock`, `AutoCycle` passes it along to the `Zone`. The `Zone` instance runs through its `Valves` and tells each one to increment its minutes open counter. In the course of doing this, the `Valve` also recalculates the gallons of water it has used so far. During its iteration through the `Valves`, the `Zone` also sums the number of gallons used so far.

After the `Zone` has been told to update its status by calling its `tick()` operation, `AutoCycle` can simply ask the `Zone` if irrigation is complete. To determine this, the `Zone` need only do two things. First, it must compare the number of gallons used so far by its `Valves` with the `zoneAllocation` to check water usage. Second, it must query its `Sensor` to get the current moisture level and compare it with its critical moisture level.

Note that in this design information is better hidden, coupling is decreased, cohesion is increased, and the controller is greatly simplified. The delegated control style of version B is clearly superior to the centralized control style of version A.

The Law of Demeter

A well-known heuristic for good object-oriented interaction design is the **Law of Demeter**, stated in Figure 12-3-5.

*An operation of an object **obj** should send messages only to the following entities:*

- *The object **obj**;*
- *The attributes of **obj**;*
- *The arguments of the operation;*
- *The elements of a collection that is an argument of the operation or an attribute of **obj**;*
- *Objects created by the operation; and*
- *Global classes or objects.*

Figure 12-3-5 The Law of Demeter

The Law of Demeter is intended to ensure that objects send messages only to objects that are “directly known” to them. This helps ensure that information hiding constraints are not violated, that coupling is low, and that cohesion is high. The Law of Demeter rules out such things as querying an object for one of its attribute objects and then sending a message to that object. This helps discourage an over-centralized control style in favor of a delegated control style. For example, version A of the AquaLush automatic irrigation control interaction in Figure 12-3-3 violates the Law of Demeter, but version B does not.

Heuristics Summary

Figure 12-3-6 summarizes the interaction design heuristics in Section 12.3.

- Avoid interaction designs where most messages originate from a single component.
- Keep components small.
- Make sure operational responsibilities are not all assigned to just a few components.
- Make sure operational responsibilities are consistent with data responsibilities.
- Have components delegate as many low-level tasks as possible.
- Avoid interactions that require each component to send many messages.
- **Law of Demeter:** An operation of an object **obj** should send messages only to the following entities: the object **obj**; the attributes of **obj**; the arguments of the operation; the elements of a collection that is an argument of the operation or an attribute of **obj**; objects created by the operation; and global classes and objects.

Figure 12-3-6 Interaction Design Heuristics

Section Summary

- A **controller** is a program component that makes decisions and directs other components.

- A **control style** is a way that decision making is distributed among program components.
- **Centralized, delegated, and dispersed** are points along the continuum of control styles. A delegated style is usually preferred.
- The **Law of Demeter** is a well-known object-oriented interaction design heuristic.

Review
Quiz 12.3

1. What is a bloated controller?
2. What are the main differences between centralized, delegated, and dispersed control styles?
3. Suppose that `arg` is an argument of a Java method. What does the Law of Demeter have to say about the following line of code in this method:
`arg.getTitle().setFontSize(14)`?

Chapter 12 Further Reading

- Section 12.1** Sequence diagrams are among the most vaguely specified parts of UML. Our discussion of this notation is based on a compromise between the language standard, [Bennett et al. 2001], and [Booch et al. 2005].
- Section 12.2** Good discussions of interaction design are found in [Wirfs-Brock and McKean 2003] and [Larman 2005].
- Section 12.3** The discussion of control styles is based on [Wirfs-Brock and McKean 2003] which contains much more extension coverage of this topic. Larman [2005] discusses bloated controllers and the Law of Demeter, originally formulated by Lieberherr and others (see [Lieberherr and Holland 1989]).

Chapter 12 Exercises

The following product specification is used in the exercises.

Computer Assignment System (CAS)

The Computer Assignment System (CAS) must aid system administrators by keeping track of computers, computer users, and assignments of computers to users.

CAS has the following functional and data requirements:

- CAS must maintain the location, components, operational status, purchase date, purchase price, and assignment of every computer in the organization.
- CAS must maintain the name, location, and title of every computer user in the organization.
- Every computer tracked by CAS must have a unique identifier assigned by CAS.
- Every user tracked by CAS must have a unique identifier assigned by CAS.

- A computer can be assigned to at most one user, but a user may have arbitrarily many computers assigned to him or her.
- CAS must support queries about individual computers, users, and assignments based on computer and user identifiers.
- CAS must generate summary reports about all users, computers, and assignments.
- CAS must generate reports about computers, their costs, and their purchase dates so that accountants can compute capital expenditures, depreciation, and so forth.

Section 12.1

1. *Find the errors:* Which of the following expressions are valid sequence diagram message specifications?
 - (a) greetings
 - (b) greetings(int x)
 - (c) reply = greetings() + "friend"
 - (d) result := query
 - (e) query(x+fib(9))
 - (f) x<5]query()
 - (g) tryAgain(x/3+5);
 - (h) tryAgain(x=5)
 - (i) x=getCoordinate(first,units=pixels)
 - (j) sum = sum+nextElement()
 - (k) tryAgain(5, -, 9, -, y)
 - (l) tryAgain(x=x+5)
2. Explain the circumstances under which the operand of a loop fragment with the following parameters and guard would execute.
 - (a) loop (5,10) with guard [x>6]
 - (b) loop (5) with guard [x>6]
 - (c) loop (0,*) with guard [x>6]
 - (d) loop (5,10) with no guard
 - (e) loop (5) with no guard
 - (f) loop (0,*) with no guard
 - (g) loop with no parameters and guard [x>6]
 - (h) loop with no parameters and no guard
3. Draw a sequence diagram documenting the following interaction: An anonymous instance of a Game class is active and it sends an instance of the Dice class named die a roll() message. The die object holds a reference to an instance of the Random class called urn, and sends it the message nextInt(6). The result of this message is incremented by one and returned by die to the Game object. All messages are synchronous.
4. Draw a sequence diagram based on the Java code in Figure 12-E-1. Have your diagram model the main() operation. Include as much detail as you can.

```

import java.util.Date;

public class Daytime {
    public static void main( String[] argv ) {
        Daytime daytime = new Daytime();
        daytime.write();
    }
    public void write() {
        Date now = new Date();
        System.out.println( now );
    }
}

```

Figure 12-E-1 Java Code For Exercise 4

5. Draw a sequence diagram based on the Java program in Figure 12-E-2. Your diagram should illustrate the call `write(2)` directed to the `fibNumber` object in the `main()` method. Show all calls (including recursive calls) with execution occurrences.

```

public class Fibonacci {
    public static void main( String[] argv ) {
        Fibonacci fibNumber= new Fibonacci();
        fibNumber.write(2);
    }
    public void write( int n ) {
        int result = fib(n);
        System.out.println( result );
    }
    public int fib( int n ) {
        if ( n < 2 ) return 1;
        return fib(n-1) + fib(n-2);
    }
}

```

Figure 12-E-2 Java Code For Exercise 5

6. Can the return arrow be suppressed even when a diagram has no execution occurrences? Does this make it harder to read? When should return arrows be used in sequence diagrams?
7. What loop fragment parameters and guards are needed to perform the loop operand under the same conditions as a Java `do-while` loop?
8. Find some code that you have written for another class and document it with one or more sequence diagrams.
9. Formulate a third alternative for the Caldera thermostat adjustment interaction that replaces the `Clock` class with the `java.util.Timer` class (for notification) and `java.util.Calendar` (for getting the current day and time).

Section 12.2

Document this alternative with sequence and class diagrams. How does this alternative compare to the others?

10. Suppose that a program simulates checkout lines at a supermarket. An instance of a `Queue` class that holds instances of `Customers` simulates each lane. A `Checkout` object has 10 `Queue` objects. Time is simulated by a `Clock` instance. Every simulated minute, the `Clock` notifies the `Checkout` that time has passed. The `Checkout` then calls its own `addCustomers()` operation. This operation adds new `Customer` objects to each `Queue` by generating a random number between zero and four (using a `Random` object) and adding that many new `Customer` objects to the `Queue`. It then calls its own `processCustomers()` operation. This method removes `Customers` from each `Queue` by generating a random number between one and three (using a `Random` object) and removing that number of `Customers` from the `Queue` and destroying them. It also records statistics about the simulation for later display. Design this interaction and document it using a class diagram and one or more sequence diagrams.
11. Can you think of a circumstance where polling would need to be used instead of notification?

- CAS**
12. Make a design class model for the Computer Assignment System (described previously) that has `Computer` and `User` classes (at least). Design the interaction that occurs when a computer is assigned to a user. Assume that an assignment is made using computer and user identifiers. If an assigned computer is already assigned to some other user, then the old assignment is destroyed.
 13. Make an alternative interaction design for the problem described in the last exercise. Compare the two designs.

- Section 12.3**
14. What effects might different control styles have on the reusability of collaborating components?
 15. Which lines of code in Figure 12-E-3 on page 393 violate the Law of Demeter?

- CAS**
16. Analyze your interaction designs from exercises 12 and 13. What is their control style? If neither alternative exhibits a centralized or delegated control style, make a new interaction diagram illustrating the missing style. Which of your interaction designs is better, and why?

- AquaLush**
17. Modify the AquaLush class diagram in Figure 12-3-2 to include the attributes and operations needed to support the interaction design in Figure 12-3-4.
 18. Make a dispersed control style design model for the AquaLush automatic irrigation interaction pictured in Figures 12-3-3 and 12-3-4.
 19. Make sequence diagrams illustrating centralized and delegated control style interactions for AquaLush automatic irrigation control when irrigation is ended in a zone because the critical moisture level has been reached.

```

class ComputerPlayer {
    private Dictionary dictionary;
    public ComputerPlayer( Dictionary d ) {
        dictionary = d;
        dictionary.load();
    }
    public void generateWords( Board board ) {
        Iterator enum = dictionary.iterator();
        wordSet = new StringSet();
        wordSet.resources().setCapacity(256);
        while ( enum.hasNext() ) {
            String word = (String)enum.next();
            word.toLowerCase();
            if ( board.isOnBoard(word) )
                wordSet.add( word );
        }
    }
    public void findWords( Board board ) {
        wordSet = new StringSet();
        Dimension dim = board.getDimension();

        board.unUseAll();
        for ( int r = 0; r < dim.height; r++ )
            for ( int c = 0; c < dim.width; c++ )
                board.testPrefix( "", r, c );
    } // findWords
} // ComputerPlayer

```

Figure 12-E-3 Java Code for Exercise 15**Research Project**

20. Research the Law of Demeter and write an essay in which you discuss the different formulations of this law by different writers.

Team Project

21. Form a team of three. Make a complete mid-level design for the Computer Assignment System (described previously), including a complete design class model and as many interaction design models as necessary to document all interesting interactions.

Chapter 12 Review Quiz Answers**Review Quiz 12.1**

1. Dashed lines are used in lifelines to show the extent of an individual's existence and in interaction fragments to separate the area inside the fragment into operand regions.
2. A selector in a lifeline identifier is an expression enclosed in square brackets after the name portion of the identifier. It is used to pick out a member of a collection.
3. When an individual sends a synchronous message, it blocks or suspends execution until the message returns. When an individual sends an asynchronous

message, it does not block but continues execution without waiting for the message to return.

4. An operation is executing when some process is running its code. It is suspended when it is waiting for a synchronous message to return. An operation is active when it is either executing or suspended. An object can be active because it may have one or more active operations, but an object cannot be suspended. Presumably one would say that an object is suspended if one or more of its operations are suspended. However, an object may have one or more suspended operations while one or more of its operations are executing because several processes may be running on the object code. As a result, it makes no sense to say that an object is suspended when one or more of its operations are suspended.
5. Optional, break, and loop fragments must have exactly one operand. An alternative fragment can have one or more operands.

**Review
Quiz 12.2**

1. Sequence diagrams may describe the interactions in the implementation of any operation, including private class operations. Hence, sequence diagrams need not begin with program operation interactions with the environment.
2. When an observer object polls a subject, it executes continuously, even though it spends only a tiny fraction of its time doing useful work—mostly it is repeating its subject query. When notification is used, the observer is not executing, and the subject is doing whatever computations it needs to do, notifying the observer only when its triggering condition becomes true. With notification, no work is done making fruitless queries.
3. The Caldera class model in version B moves attributes from the `WaterHeaterController` to the `TimePeriod` class. Interactions that modify or query these attributes (such as those in which the homeowner checks and sets them) must be adjusted to account for the change.

**Review
Quiz 12.3**

1. A controller is a component that makes decisions and directs other components. A bloated controller is one that is too large and complex.
2. A centralized control style is one in which most decisions are made by only a few controller components. A delegated control style is one in which controller components make high-level decisions but distribute control of lower-level decisions to collaborators. A dispersed control style is one in which control is spread throughout a program with no real controllers at any level of abstraction.
3. If `arg` is an argument of a Java method, then the line of method code `arg.getTitle().setFontSize(14)` violates the Law of Demeter. The call `arg.getTitle()` returns an object that is (presumably) not the object holding the method, one of its attributes, a method argument, an element of a collection that is a method argument or an attribute, an object created by the method, or a global class or object. In other words, this call returns an object that the Law of Demeter says should not be sent a message; however, it is sent the message `setFontSize(14)`, violating the Law of Demeter.

13 Dynamic Mid-Level State-Based Design: State Models

Chapter Objectives

This chapter discusses modeling behavior by representing states and state transitions using UML state diagrams.

By the end of this chapter you will be able to

- Read and write basic UML state diagrams using internal and external transitions, transition events, guards, actions, and sequential composite states;
- Read and write state diagrams using advanced features including concurrent composite states, history states, and compound transitions; and
- Use state diagrams to specify finite automata acting as acceptors or transducers to model products, user interfaces, software components, or parts of software components.

Chapter Contents

- 13.1 UML State Diagrams
 - 13.2 Advanced UML State Diagrams
 - 13.3 Designing with State Diagrams
-

13.1 UML State Diagrams

States and State Transitions

A **state** is a mode or condition of being. People are in one state when asleep and another when awake. When awake, people may also be in several other states, such as happy, angry, sad, or thoughtful. Thus, states admit levels of abstraction, and higher-level states may have sub-states. For example, waking is a high-level state, while being happy is one of its sub-states. Furthermore, people may be in several states at the same time: Someone may be simultaneously healthy, wealthy, and wise. Thus, states may be occupied simultaneously.

These observations about the human condition also apply to software products, components, objects, and other computational entities. Entity states may be discerned at different levels of abstraction, states may contain sub-states, and an entity may be in several states simultaneously.

States must be discernible from one another, so an entity with several states must have varying characteristics that identify its states. The only varying characteristics that computational entities have are their variables and their relationships to other entities. Thus, computational entity states are characterized by values of their variables and perhaps the variables of other entities recording their relationships. Computational entity states are essentially configurations of variable values.

Being in a particular state may be interesting, but often more interesting are the details of how that state was attained. A **transition** is a change from one state to another. Transitions may be spontaneous, but usually some event triggers them. An **event** is a noteworthy occurrence at a particular time; events have no duration. Events include the arrival of a message or signal, the raising of an exception, or the passing of a moment of time. Transitions may be constrained to occur only under certain circumstances. Transitions may also themselves give rise to events. Generally, a transition causes a change from one state to a different state, but a transition that changes from a state to itself is also a theoretical possibility that turns out to be very useful in practice.

If computational entity states are configurations of variable values, then transitions are realized by making new assignments to certain variables. Implementing transitions is thus quite simple.

Finite Automata

If we abstract all an entity's details except its states and state transitions, we are left with something called a **finite state machine** or **finite automaton**. Finite automata model some entity's state-based behavior. In this chapter we discuss finite automata as tools for modeling the states and transitions of software products and their components.

Every finite automaton specification must contain

- Descriptions of the automaton's states in a way that allows them to be distinguished, such as by naming each one;
- Descriptions of transitions indicating each transition's source state, its target state, and the events that trigger it; and
- Designation of an initial state, the starting place for state transitions.

Finite automata are either deterministic or non-deterministic. A **deterministic finite automaton** is a finite automaton that has no spontaneous transitions and has a single transition that it must make in response to every event in each of its states. A **non-deterministic finite automaton** is one that is not deterministic. Theoretical computer scientists have shown that every non-deterministic finite automaton is equivalent to a deterministic finite automaton, so we can do all our modeling with either deterministic or non-deterministic automata. From now on we discuss only deterministic finite automata.

Despite their simplicity, finite automata are able to represent an enormous range of computational activity. Every real computer is a mechanism with only a finite (though enormous) number of states, well-defined state transitions, and a well-defined initial state. Thus, it is possible in principle to represent every possible computation of any real computer by a finite automaton. The fact that this is both practically impossible (because of the number of states) and pointless (because such a model would be too complex to be useful) does not blur the point that finite automata are a powerful tool.

In particular, finite automata are useful for analyzing, designing, specifying, and documenting the behavior of computational entities. They are important in both theoretical and applied computer science and in software engineering. Within software design, finite automata are important for problem specification and analysis as well as problem solution at all levels of abstraction.

The remainder of this section will introduce UML state diagrams, a sophisticated notation for describing finite automata. The next section will present advanced UML state diagram features. The final section presents examples illustrating how finite automata can be used in software design.

State Diagram Basics

The UML notation for specifying finite automata is the **state diagram**. In state diagrams, states are represented by rounded rectangles. The finite automaton initial state is designated by a special *initial pseudo-state* depicted as a large black dot at the tail of an arrow pointing at the initial state. A finite automaton may execute forever or it may halt in a *final state*. Final states are represented by a black dot inside a circle.

Transitions are represented by solid arrows labeled with one or more *transition strings* that describe the circumstances under which the transition is triggered and the actions that may ensue. Figure 13-1-1 illustrates these conventions in a state diagram describing the states and state transitions of a simple tape recorder.

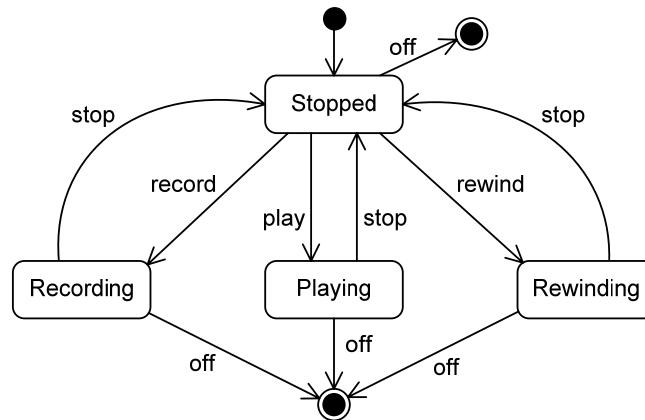


Figure 13-1-1 Tape Recorder State Diagram

This automaton starts in the **Stopped** state. From there, one of four events may occur; events that do not label a transition are ignored. The **off** event causes transition to a final state where the machine halts. The others trigger transitions to the **Record**, **Play**, and **Rewind** states. When in these three states, the only events that have an effect are **stop** and **off**.

Every transition string must begin with an *event-signature*. An event-signature may be empty, in which case it designates a state's *completion event*, which

occurs when the activity in the source state ends. Otherwise, it consists of an *event-name* possibly followed by a comma-separated *event parameter* list enclosed in parentheses.

The *event-signature* may be followed by a *guard*. The *guard's* Boolean expression must be true when the event designated by the *event-signature* occurs for the event to trigger the transition. Finally, the *guard* may be followed by a slash and an *action-expression* describing some action that occurs when the transition is made.

The transition string format is described in more detail in Figure 13-1-2.

State Diagram Transition String Format

event-signature guard / action-expression

Where:

event-signature—The empty string (which designates the source state completion event) or a non-empty *event-name* optionally followed by a comma-separated *event-parameter* list enclosed in parentheses. The *event-name* is a pathname. Each *event-parameter* has the form *parameter-name* : *type*, where *parameter-name* is a simple name and *type* is a type specification in a format not specified by UML. The *type* and the colon that precede it may be omitted. The parentheses do not need to appear if there are no *event-parameters*. The *event-parameters* are used to distinguish events with the same names.

guard—A *guard-condition* enclosed in square brackets. The *guard-condition* must be a Boolean expression in a format not specified by UML. The *guard-condition* must be true when the event occurs for the transition to be triggered.

action-expression—An optional description of a computation performed when the transition is triggered, written in a format not specified by UML. The slash must be omitted if there is no action-expression.

Figure 13-1-2 State Diagram Transition String Format

Several transition arrows with the same origination and termination points may be consolidated into a single arrow labeled with a list of transition strings, one per line.

State Symbol Compartments

State symbols have as many as three compartments, which are state symbol regions optionally separated by solid horizontal lines. The first compartment is the *name compartment*: It contains the state name. State names are optional and may be pathnames. The name compartment may be omitted. Also, the state name may be placed on a tab attached to the state symbol to save space within the symbol.

The second compartment lists internal transitions, one per line. An *internal transition* is a transition that is processed when a state is active, but it does not cause any state changes. Internal transitions are used to specify actions taken in response to events that occur when a state is active.

Internal transitions may be described either by transition strings in the previous format, or by specifications of the form

action-label / action-expression

where an *action-label* is characterized in Table 13-1-3 and an *action-expression* describes a computation in a format not specified by UML.

Action Label	Meaning
entry	Execute the associated <i>action-expression</i> (an <i>entry action</i>) upon state entry.
exit	Execute the associated <i>action-expression</i> (an <i>exit action</i>) upon state exit.
do	Execute the associated <i>action-expression</i> (a <i>do activity</i>) upon state entry and continue until state exit or action completion.
include	In this case the <i>action-expression</i> must name a finite automaton. The named automaton is a placeholder for a nested state diagram (discussed below).

Table 13-1-3 Internal Transition Action Labels

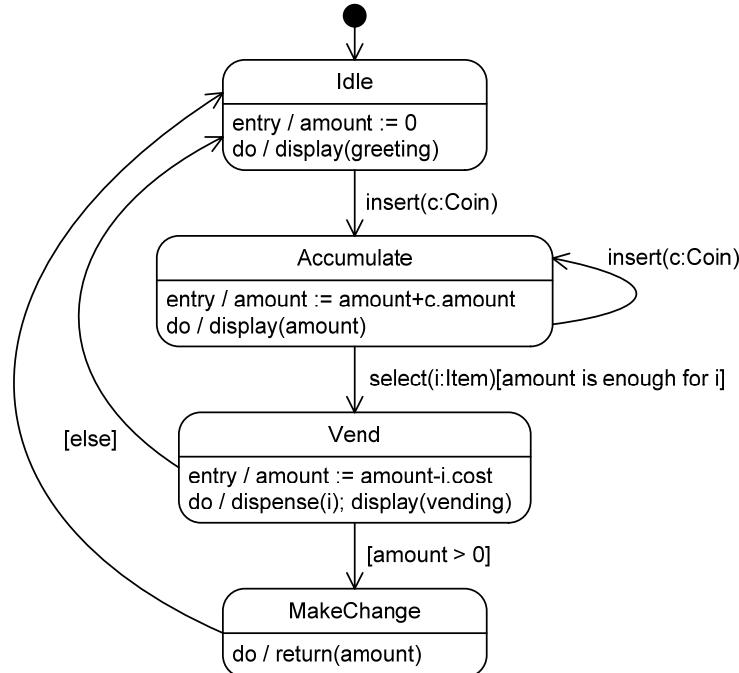
When an internal transition occurs, there is no change of state. Thus, for example, suppose a state has the following internal transitions:

entry / ringBell
scrollBtnPress(d:Direction) / scroll(d)

When this state is current and the scrollBtnPress event occurs, the scroll() action is invoked. However, the ringBell action is not invoked because the state is not reentered.

Figure 13-1-4 on page 400 illustrates the name and internal transition compartments. This finite automaton is a simple vending machine. When its initial Idle state is entered, the variable amount is set to 0. When customers insert coins, the machine changes to the Accumulate state and increments the amount variable by the value of the inserted coins. A customer then selects an item. If enough money has been inserted to pay for it, the machine changes state to Vend. In this state, the amount is decremented by the cost of the selected item and the item is dispensed. When this activity is complete, one of two transitions is made: If amount is greater than zero, the machine goes to the MakeChange state; otherwise, it returns to the Idle state. In the former case, when MakeChange has returned the correct change, its completion transition switches the automaton state back to Idle.

The third compartment in the state symbol is the *nested diagram compartment*. This compartment includes a nested state diagram, discussed next.

**Figure 13-1-4 Vending Machine State Diagram****Nested State Diagrams**

A state symbol without a nested state compartment represents a *simple state*, and one with a nested state compartment represents a *composite state*. There are two kinds of composite states:

- A *sequential composite state* contains a single state diagram composed of *sub-states* or *inner states* and the transitions between them. When the *outer state* is entered, an inner state is entered as well; when the outer state is exited, the current inner state is also exited.
- A *concurrent composite state* contains two or more sequential state diagrams in regions separated by dashed lines called *concurrent region boundary lines*. The automata described by these state diagrams execute concurrently. In other words, when the outer state is entered, a state in each of the concurrent automata is entered, and events are processed by each concurrent automaton simultaneously.

In this section we consider sequential composite states; concurrent composite states are discussed in the next section.

Sequential Composite States

A sequential composite state includes a nested state diagram that decomposes the composite state. When a sequential composite state is entered, both the composite state and one sub-state jointly become the current states. To make sure that the automaton is deterministic, there can be no doubt about which sub-state is entered when the composite state is entered. Therefore, either the nested state diagram must have its own initial

state (preferred), or every transition entering the composite state must terminate at a sub-state. If a transition that terminates at the composite state boundary occurs, the nested state diagram's initial state is entered. Similarly, when a sequential composite state is exited, its current sub-state must be exited as well. Transitions leaving the composite state may originate at the composite state boundary or at internal states. If either sort of transition occurs, then both the composite state and the current sub-state are exited. Transitions originating at a composite state boundary apply to every composite state sub-state.

To illustrate, consider the state diagram for a car's cruise control in Figure 13-1-5.

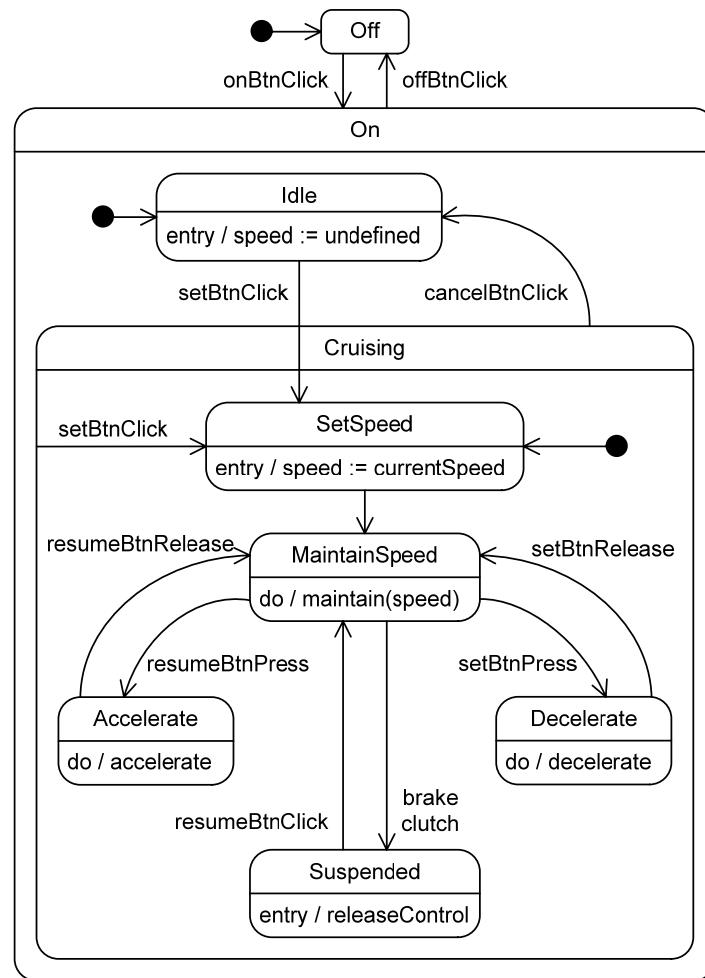


Figure 13-1-5 State Diagram with Nested Sequential States

This automaton starts in the Off state. When an `onBtnClick` event occurs it changes to the On state, which is a composite sequential state. The initial state inside On is Idle, so this sub-state is entered as well. If the `setBtnClick` event occurs, there is a transition to a sub-state of Cruising, another composite sequential state. At this point the three states On, Cruising, and SetSpeed are current states. Once the speed setting activity in SetSpeed is finished, a completion transition to MaintainSpeed occurs. Various events may cause other transition within or from the Cruising state. For example, if there is a `cancelBtnClick` event, then Cruising and its sub-states are exited and Idle becomes the current state (jointly with On). Similarly, no matter which states within On are current, all are exited and Off is entered if the `offBtnClick` event occurs.

Incidentally, notice that this example contains a transition labeled with two transition strings: `brake` and `clutch`.

Action Execution Order

Actions may be attached to transitions and appear as entry and exit actions of various states. Action execution order is well defined, even for composite states. Figure 13-1-6 states the rule governing action execution order.

When an initial state is entered, the state's entry actions are executed, followed by the entry actions of any initial sub-states.

When a transition causes exit from simple state A and entry to simple state B

1. Simple state A's exit actions are executed;
2. The exit actions of any exited composite states enclosing A are executed, in order from innermost to outermost exited states;
3. The transition action is executed;
4. The entry actions of any entered composite states enclosing B are executed, in order from outermost to innermost entered states; and
5. Simple state B's entry actions are executed.

Figure 13-1-6 Action Execution Order Rule

Figure 13-1-7 illustrates this rule. The finite automaton specified in this state diagram begins execution jointly in states AA and A. The entry actions of these states are executed from outermost to innermost, so `enterAA` is executed, followed by `enterA`. If event `x` occurs, the automaton exits states AA and A and enters states BBB, BB, and B. The action execution order is `exitA`, `exitAA`, `xAction`, `enterBBB`, `enterBB`, and `enterB`.

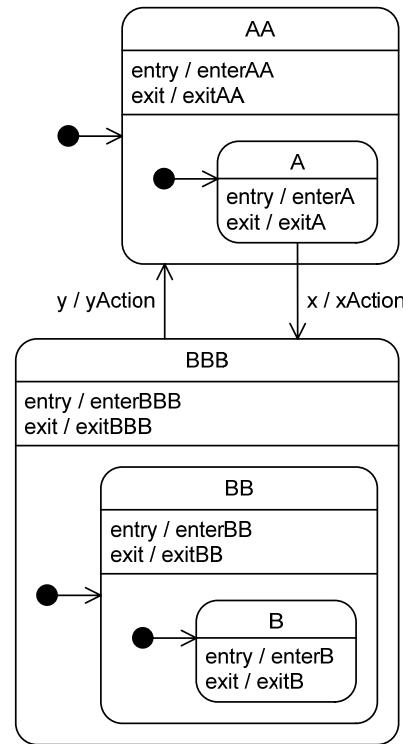
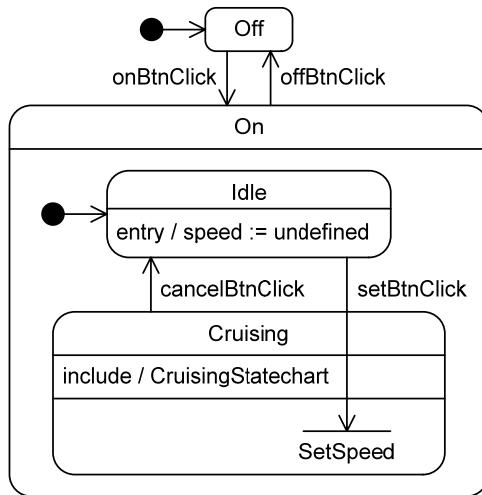


Figure 13-1-7 Action Execution Order Example

Stubbed States A nested state diagram may be suppressed and referred to by the include internal transition. But the problem with suppressing state diagrams like this is that there may be transitions to and from the states of the suppressed state diagram. What is to be done with them? State diagrams use a special stub symbol to represent sub-states of the suppressed state diagram that originates or terminates transitions to and from the enclosing state diagram. A *stub* or *stub symbol* is a short line labeled with a suppressed state name. To illustrate, Figure 13-1-8 on page 404 suppresses the state diagram in the *Cruising* composite state from Figure 13-1-5.

In this diagram, the state diagram nested in the *Cruising* state is suppressed and the *SetSpeed* sub-state appears as a stub so that the transition from the *Idle* state has a termination point.

Transitions terminating at stubs are labeled with transition strings, but those originating from stubs are not. Transition strings on transitions that originate from a stub are left off the diagram because they may include references to the suppressed state diagram that would not make sense.

**Figure 13-1-8 A Stubbed State**

Using Sequential Composite States

Sequential composite states simplify state models in two ways:

- They organize states into hierarchies. States are grouped by their containing states, making state models more understandable. Also, nested states can be suppressed, introducing an abstraction mechanism that makes state diagrams easier to read.
- They consolidate many transitions. If composite states were replaced by their nested state diagrams, all the transitions originating on the composite states would have to be copied onto each state in the nested state diagrams. Usually this would add many transitions, making the diagram much harder to read.

Although any finite automaton can be described by a state diagram with only simple states, such state diagrams often have many states and transition arrows and are hard to read. Sequential composite states are a powerful mechanism for making state models easier to understand.

Most state diagrams found in software engineering designs use only the portions of the state diagram notation presented up to this point. The next section presents some more exotic state diagram features useful in special situations.

State Diagram Heuristics

The following checks are easily made to help ensure that state diagrams are well formed and correct:

Check that every non-completion transition arrow is labeled. Leaving off transition strings is a common error in making state diagrams. Unlabeled transitions should only indicate completion events.

Check that no arrow leaves a final state. Execution halts in final states, so there can be no transitions from final states.

Check for black holes and white holes. A *black hole* is a non-final state with no outgoing transitions. Sometimes this is intentional, but more often this indicates a mistake. A *white hole* is a non-initial state with no incoming transitions. There is no way to get into white-hole states, so these states can always be removed without altering finite automaton behavior. White-hole states are there by mistake.

As with most design notations, good names for model elements help readers decipher the model. The following heuristics help make state diagrams more readable:

Label states with adjectives, gerund phrases, or verb phrases. States are ways that things are at some time, so they should be named with phrases that describe the ways a thing can be. Adjectives such as “on,” “off,” “suspended,” and “asleep” do this. A states can also be described in terms of the activity that goes on when an entity is in that state. Both gerund and verb phrases can describe activities. A *gerund* is a verb made into a noun by adding “-ing,” such as “sleeping,” “working,” or “running.” A *gerund phrase* is a gerund plus other words needed to fully describe an activity, such as “sleeping alone,” “working on the railroad,” or “running after the bus.” Verb phrases, such as “set up,” “monitor input,” or “compute salaries,” also describe actions or activities.

Name events with verb phrases or with noun phrases describing actions. Event signatures are the central items of transition strings. Events are action occurrences, so they can be described by verb phrases such as “press a button,” “select an item,” or “send a message.” Actions can also be described by noun phrases such as “button press,” “item selection,” and “message dispatch,” so these may label events as well.

Name actions with verb phrases. The actions that may be performed when an internal or external event occurs should be named with verb phrases.

Combine arrows with the same source and target states. State diagrams with many arrows are harder to read. One way to reduce the number of arrows is to attach several transition strings to one arrow.

Use stubs and the include internal transition to decompose large and complicated state diagrams. Very large state diagrams are daunting and difficult to read, much like very long procedures or functions. As with procedures and functions, the way to make them more readable is to decompose them.

A state diagram must describe a deterministic finite automaton, but it is very easy to make an automaton non-deterministic. The following heuristics help avoid this problem:

Make one initial state in every state diagram (including nested state diagrams). It must always be clear where a finite automaton begins execution, so every state diagram needs to identify where execution begins. If there are no transitions to a sequential composite state’s boundary, then it does not need an initial state because all transitions into the state go to specific sub-states. However, it is safer to identify an initial state anyway in case a transition to the composite state boundary is added later.

Check that no event labels two or more transitions from a state. The best way to check this is to go through states one by one and check the transitions directly from the state and the transitions from all enclosing sequential composite state boundaries. The latter are especially likely to cause problems.

Check that all guards on the same event are exclusive. This is really part of the previous check—if two or more transitions from a state are labeled with the same event, the automaton is still deterministic if their guards are exclusive. Guards are *exclusive* when they cannot be true at the same time.

Use [else] guards to help ensure that guards are exclusive and exhaustive. The safest way to make sure that several guards are exclusive is for one of them to be `[else]`. The `[else]` guard can also be used to make sure guards are *exhaustive*; that is, that at least one of them is true no matter what.

Heuristics Summary

Figure 13-1-9 summarizes the state diagram heuristics discussed in this section.

- Check that every non-completion transition arrow is labeled.
- Check that no arrow leaves a final state.
- Check for black holes and white holes.
- Label states with adjectives, gerund phrases, or verb phrases.
- Name events with verb phrases or with noun phrases describing actions.
- Name actions with verb phrases.
- Combine arrows with the same source and target states.
- Use stubs and the include internal transition to decompose large and complicated state diagrams.
- Make one initial state in every state diagram (including nested state diagrams).
- Check that no event labels two or more transitions from a state.
- Check that all guards on the same event are exclusive.
- Use `[else]` guards to help ensure that guards are exclusive and exhaustive.

Figure 13-1-9 State Diagram Heuristics

Section Summary

- A **state** is a mode or condition of being. States may have sub-states, and multiple states may be occupied simultaneously. Computational entity states are represented by configurations of variable values.
- A **transition** is a change from one state to another. Transitions may be spontaneous but are usually triggered by **events**, which are noteworthy occurrences at particular times.
- A **deterministic finite automaton** is a finite automaton that has no spontaneous transitions and has a single transition that it must make in response to every event in each of its states.

- **State diagrams** are a UML notation for describing finite automata.
- State diagrams have symbols representing states, initial states, final states, and transitions.
- State diagrams may have *sequential composite states* that include nested state diagrams. Nested state diagrams may be suppressed and appear in other diagrams.

Review
Quiz 13.1

1. What three things must every finite automaton specification include?
2. What is the difference between a deterministic and a non-deterministic finite automaton?
3. What is a completion event, and how is it designated in UML state diagrams?
4. What is an internal transition?
5. What happens when a transition from a composite state boundary occurs?

13.2 Advanced UML State Diagrams

Concurrent Composite States

Concurrent composite states contain two or more concurrent state diagrams separated by dashed lines called *concurrent region boundary lines*. The concurrent state diagrams specify finite automata that execute in parallel. For example, consider the model of a traffic light in Figure 13-2-1.

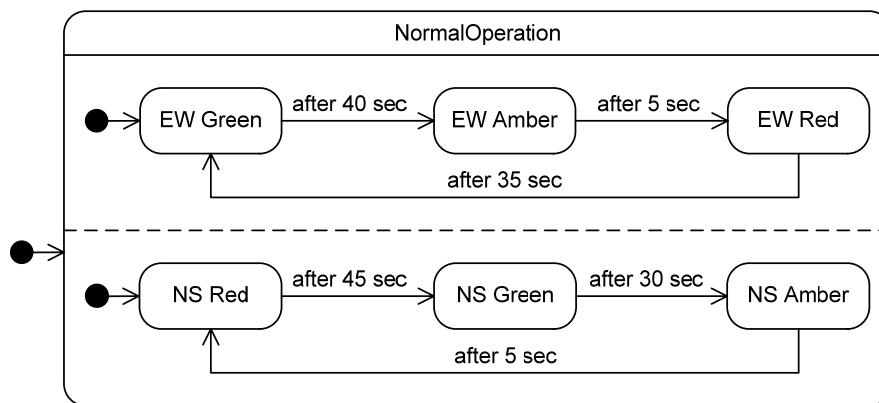


Figure 13-2-1 Traffic Light Concurrent Composite State

The key to understanding concurrent composite states is that when the composite state is entered, one state in each concurrent automaton is entered as well. Furthermore, exactly one state in each concurrent automaton is always current until the composite state is exited. The current state is thus a set that includes the composite state and one sub-state of each concurrent automaton.

In Figure 13-2-1, the indicated light is activated when the respective state is entered. For instance, the green lights facing north and south are activated

when the NS Green state is entered and the other north- and south-facing lights are deactivated. When the TrafficLight composite state is entered at the start of automaton execution, so are the EW Green and NS Red states. Hence, the current state at the start of execution is {TrafficLight, EW Green, NS Red}. After 40 seconds, a transition occurs in the top concurrent automaton, and the current state becomes {TrafficLight, EW Amber, NS Red}. Five seconds thereafter, two transitions occur, one in the top automaton and one in the bottom, making the current state {TrafficLight, EW Red, NS Green}. Execution continues along these lines.

There are two ways to make transitions into and two ways to make transitions from concurrent composite states. One way to make a transition into a concurrent composite state is illustrated in the previous example: A transition is made to the composite state boundary. In this case, each concurrent automaton's initial state is entered simultaneously.

The second way to make a transition into a concurrent composite state is to make a transition directly to one of its sub-states. However, this presents a problem: Which states should be entered in the other concurrent automata (the ones whose sub-states are not the target of the transition)? By default, the other concurrent automats' initial states are entered. Alternatively, specific states of one or more of the other concurrent automata can be chosen for entry. To pick out specific states, the incoming transition, with its transition string, terminates at a synchronization bar from which emerges unlabeled transitions to two or more sub-states in different concurrent regions. This technique is illustrated in the schematic example in Figure 13-2-2.

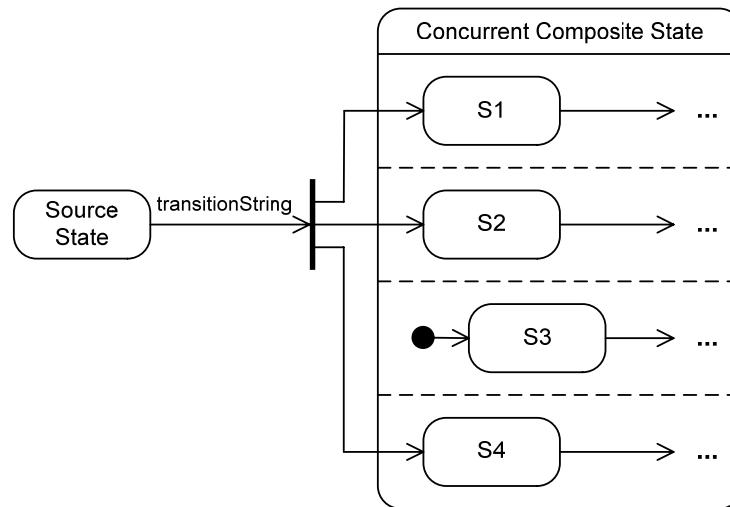


Figure 13-2-2 Entering Selected Concurrent Sub-States

In this example, when the transition from the Source State occurs, states S1, S2, and S4 are entered because they are explicitly designated, and state S3 is

entered by default because it is the initial state of an automaton that has no state as the target of the transition.

There are also two ways to leave concurrent composite states: transitioning from the composite state and transitioning from individual sub-states. A transition from the composite state triggered by any non-completion event causes all concurrent sub-states to be exited immediately. Transitions triggered by completion events are a little trickier. Each concurrent automaton must run to completion before the composite state is exited using a completion transition. Nested automata completion occurs when execution enters a final state. As a result, a completion transition from a concurrent composite state occurs only when execution in each nested automaton reaches a final state.

A transition from a sub-state to an external state causes immediate exit from each concurrent automaton's current state. Sometimes modelers may want a transition to occur only if certain states in two or more concurrent automata are current when the triggering event occurs. In this case, transition arrows are drawn from the chosen states to a synchronization bar outside the composite state, and a labeled transition is drawn from the synchronization bar to the transition's target state. This mechanism is illustrated in Figure 13-2-3.

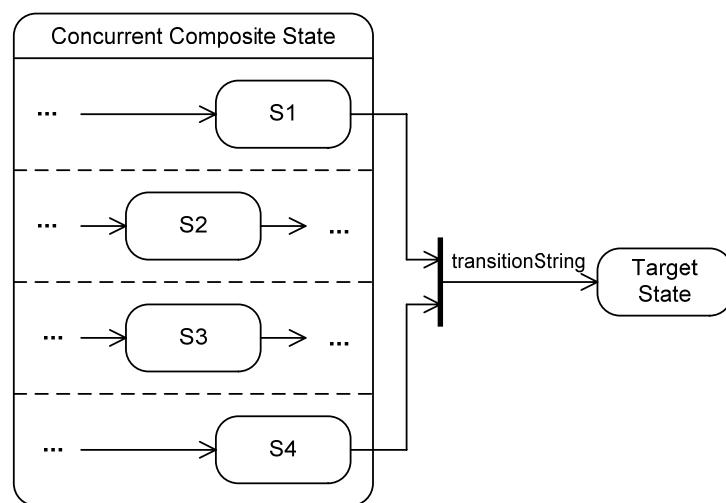


Figure 13-2-3 Leaving Selected Concurrent Sub-States

In this example, the transition to the Target State is made only when the indicated event occurs while states S1 and S4 are the current states of their respective automata. The current states of the other two concurrent automata have no effect on the transition. When the transition occurs, all concurrent states are exited.

Synch States Concurrent automata can synchronize their behavior by sending each other events and placing guards on transitions. State diagrams provide an additional synchronization mechanism called synch states. A *synch state* is a counter that keeps track of transitions. An automaton can register its transitions with a synch state and a concurrent automaton can delay its transitions if a synch state counter goes to zero. Synch states are represented in state diagrams by synch state symbols, which are small circles containing either a positive integer or an asterisk, indicating the counter's upper bound. The asterisk means no upper limit.

To illustrate, consider again the traffic light example in Figure 13-2-1. If the timing between the concurrent automata somehow becomes unsynchronized, it is possible the light could show green or amber in all directions. Inserting synch states can prevent this catastrophe, as shown in Figure 13-2-4.

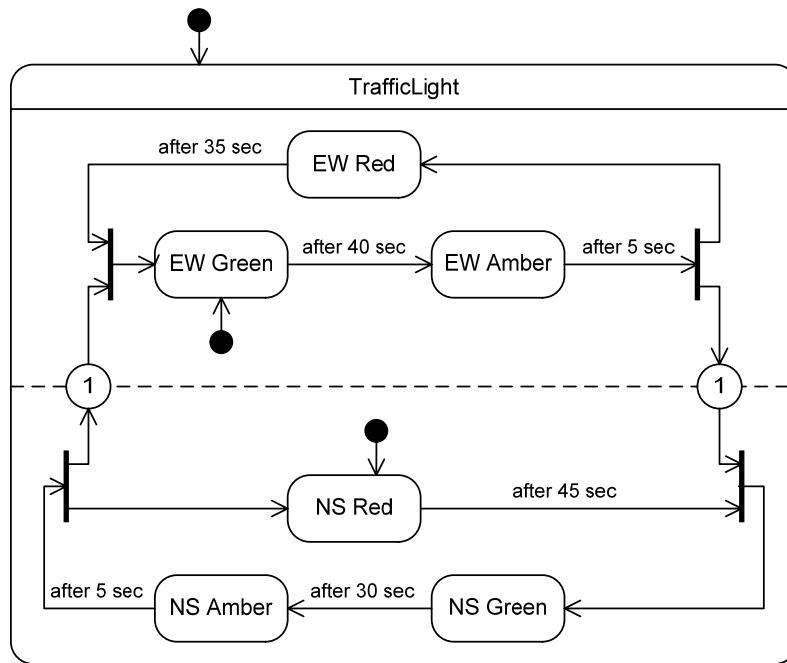


Figure 13-2-4 Traffic Light with Synch States

The synch state symbols are the circled 1s on the concurrent regions' boundary lines. As before, these concurrent automata start execution in the EW Green and NS Red states. After 40 seconds a transition is made from EW Green to EW Amber, and after another 5 seconds a transition is made from EW Amber to EW Red. This transition goes through a synchronization bar forking the transition so that it goes to a synch state. Also at this time, a transition is attempted from NS Red to NS Green. However, this transition goes through a synchronization bar joining the arrow from NS Red and a synch state. If this synch state has not received a transition from EW Amber,

then the transition from NS Red is blocked until the synch state does receive the incoming transition it needs. Thus, if the transition from NS Red to NS Green attempts to fire early, perhaps because the timing has somehow been disturbed, it cannot do so. The other synch state plays a similar role by preventing a premature transition from EW Red to EW Green. The synch states thus prevent the lights from being green or amber in all directions at once.

Synch states must always be arranged with their incoming transitions coming from forks and their outgoing transitions going to joins, as in Figure 13-2-5.

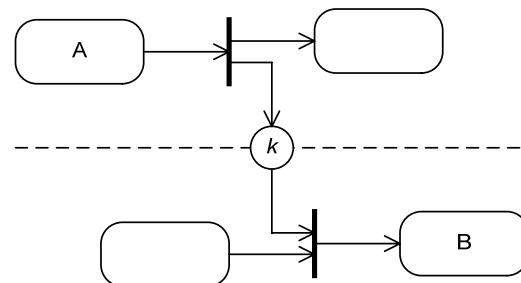


Figure 13-2-5 Using A Synch State

This arrangement ensures that state B is not entered until the transition leading from A has been traversed between 0 and k times since B was last entered; in other words, B is not entered until A has exited between 0 and k times as often as B has been entered.

The k in the synch state circle indicates the maximum number of incoming transitions that the synch state can count. If this number is exceeded, the automaton's behavior is undefined. An asterisk indicates no limit on the number of counted transitions.

Using Concurrent Composite States

Concurrent composite states simplify state diagrams by greatly reducing the number of states and transitions. Any set of concurrent state diagrams can be converted into an equivalent sequential state diagram, but the number of states in the sequential state diagram may be as much as the product of the number of states in each of the concurrent state diagrams.

Unfortunately, the simplicity of the diagram may not translate into understandability. Concurrent composite states are often hard to understand, especially when they have synch states.

Compound Transitions

We have already noted that arrows with the same source and target states may be combined into a single arrow with several transition strings. Transition junction points allow combinations of arrows with similar transition strings but *different* source and target states. In this case, the transitions from different sources can converge on a *transition junction point*

symbol (a diamond or filled circle), and transitions leaving the transition junction point symbol may disperse to different targets. The transition arrows must be labeled so that it is clear exactly which transitions are made in every possible circumstance. Several transition arrows connected using transition junction point symbols are called *compound transitions*. For example, consider the state diagram fragment in Figure 13-2-6.

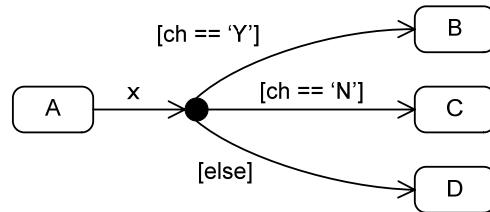


Figure 13-2-6 A Transition Junction Point

In this state diagram fragment, event *x* causes a transition from state A to either B, C, or D, depending on the value of the variable ch. The target state is determined based on the guards labeling the transitions emanating from the junction point symbol. Without using a junction point, there would be three arrows with a common event but different guards. The junction point consolidates these arrows, emphasizing that there is a common event and mutually exclusive guards.

Junction point symbols must not be used to fork or join transitions to concurrent regions inside a concurrent composite state: Synchronization bars must be used.

History States

A *history state* is a pseudo-state that serves as a marker indicating that the sub-state last active when the composite state was exited should be reentered. The *history state indicator* is a circled capital H. Consider the example in Figure 13-2-7.

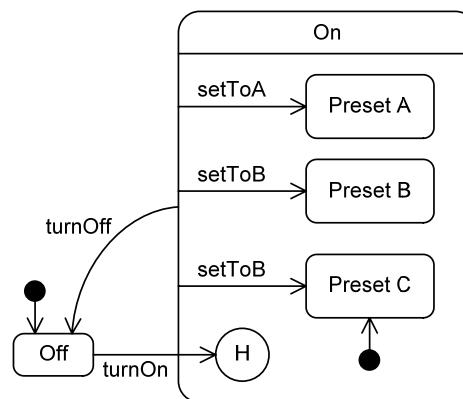


Figure 13-2-7 A History State Indicator

Many devices remember their state when they are turned off; radios remember their preset stations, for example. This state diagram models a device with three presets remembered when the device is turned off. When the device is first turned on, the history state is entered for the first time. By default, the composite state's initial sub-state (**Preset C**) is entered. Thereafter, when the **On** composite state is exited, it “remembers” the sub-state that was last active. When a transition is made to the history state, the last active sub-state is reentered.

Transitions to history state indicators are only allowed from outside the composite state that includes the history state indicator. A history state may have a single unlabeled transition arrow emanating from it to one of its peer states. This optional arrow indicates the state to be entered if a transition is made to the history state when the composite state has never been entered before. Thus, the state entered on transition to a history state when the composite state is entered for the first time can be different from the nested state diagram's initial sub-state.

Because a history state is only an indicator about which state to enter, it cannot have the paraphernalia of a true state. In particular, it cannot have regular labeled outgoing transition arrows, internal transitions, or its own sub-states.

Normally, transition to a history state indicator means that the last active state at the *same nesting level* as the history indicator is entered. States at deeper nesting levels are entered according to the usual rule, which is that each nested state diagram's initial sub-state is entered. A circled capital H* is a *deep history state indicator*. It means that the last exited sub-states of all *nested* state diagrams should be entered as well. Figure 13-2-8 illustrates a deep history state indicator.

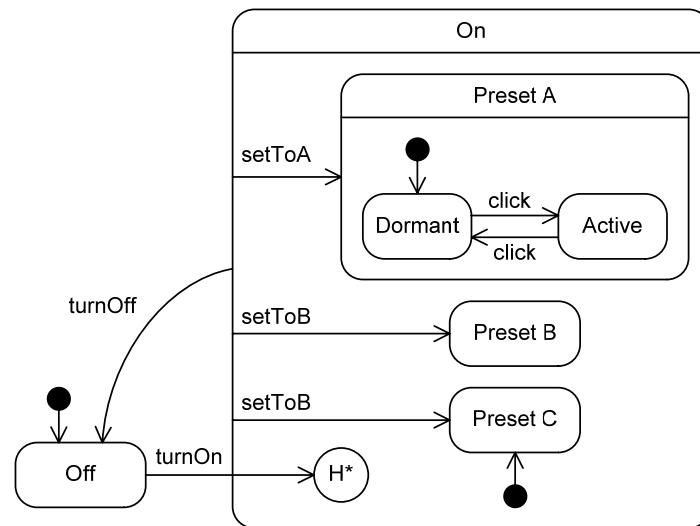


Figure 13-2-8 A Deep History State Indicator

Suppose that during execution the finite automaton is in states **Preset A** and **Active** when the *turnOff* event occurs. The **Off** state is entered. Now suppose that the *turnOn* event occurs. The transition to the deep history state indicator means that the states **Preset A** and **Active** are reentered.

Finally, if a sequential composite state with a history state ends up in a final state, then its history is “forgotten” and a subsequent transition to the history state behaves as if the composite state were being entered for the first time.

More State Diagram Heuristics

As mentioned earlier, every collection of concurrent automata is equivalent to a single sequential automaton, though it may have many more states. This means that every concurrent composite state can be replaced by a sequential composite state, though the nested state diagram may have considerably more states than the concurrent state diagrams do. However, concurrent state diagrams, especially those with synch states, are often very hard to construct and to understand. Concurrent state diagrams with synch states are also subject to deadlock when made incorrectly. Hence, we recommend the following state diagram construction heuristic:

Avoid concurrent composite states, especially those with synch states.

The following heuristics help designers make well-formed state diagrams with advanced features:

Designate an initial state in every concurrent region of a concurrent composite state. If there are no transitions to a concurrent composite state’s boundary, then concurrent regions do not need to have initial states. A transition to the composite state’s boundary could be added later, though, so initial states for each region should always be specified.

Check that transitions to several concurrent sub-states go through a fork. It is illegal to transition directly to more than one sub-state of a concurrent composite state without going through a fork.

Check that arrows connected to transition junction points are properly labeled. State diagrams must specify deterministic finite automata, so there can be no doubt which transition to follow under every possible circumstance. Transitions involving transition junction points must be constructed so that it is clear which transition to make in all cases.

Check that at most one unlabeled arrow emanates from each history state. History states are only markers, not real states, so they cannot be the sources of transitions.

Check that every sequential state diagram containing a history state has an initial state. If a transition is made to a history state, it is the first time the composite state has been entered, and no default entry state is marked, then the nested state diagram initial state is entered. If no initial state is marked, then the state diagram is ill formed. It is safer to always indicate an initial state.

Heuristics Summary Figure 13-2-9 summarizes the state diagram heuristics discussed in this section.

- Avoid concurrent composite states, especially those with synch states.
- Designate an initial state in every concurrent region of a concurrent composite state.
- Check that transitions to several concurrent sub-states go through a fork.
- Check that arrows connected to transition junction points are properly labeled.
- Check that at most one unlabeled arrow emanates from each history state.
- Check that every sequential state diagram containing a history state has an initial state.

Figure 13-2-9 State Diagram Heuristics

- Section Summary**
- *Concurrent composite states* contain two or more state diagrams specifying finite automata that execute in parallel.
 - Special rules ensure that when a concurrent composite state is entered, one state in every contained concurrent state diagram is entered simultaneously, and that when a concurrent composite state is exited, all contained concurrent state diagrams are exited simultaneously.
 - Concurrent state diagrams can be synchronized using guards, synchronization events, or special *synch states*.
 - Transition strings with common parts can share arrows in a *compound transition* consisting of arrows connected by *transition junction points*.
 - A *history state* is a pseudo-state indicating that the sub-state of the sequential composite state last active when the composite state was exited should be reentered.

- Review Quiz 13.2**
1. What are the two ways to enter a concurrent composite state?
 2. What happens if a synch state contains the number 1 but two transitions enter the synch state before any leave?
 3. What are transition junction points used for?
 4. Can a history state have an entry action?

13.3 Designing with State Diagrams

Acceptors and Transducers

Finite automata can be divided into two groups:

Acceptors or recognizers—Finite automata that respond to events but generate no actions are called **acceptors**, or **recognizers**. If events are

treated as inputs, then acceptors model the consumption of input that is either accepted or rejected (or recognized or not). Input acceptance is determined by whether the automaton is in an accepting state when it halts.

Transducers—Finite automata that both respond to events and generate actions are called **transducers**. If events are treated as inputs and actions as outputs, then transducers can be thought of as devices that transform inputs to outputs.

Acceptors are widely used to help design programs that must consume input and decide whether to accept it or not. Some programs, such as compilers, interpreters, and markup language processors, have this as a major task. Most programs must judge whether their input is acceptable, even if it is only a few commands or a simple input file. We will briefly consider how to use acceptors for this design task below.

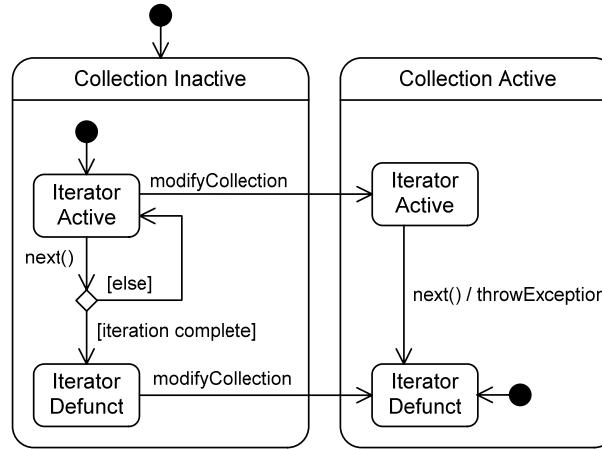
Transducers are used to help design programs or program components with complex state-based behavior. We saw examples of such applications in the first several sections of this chapter, and we consider a few more in this section.

Transducer Models

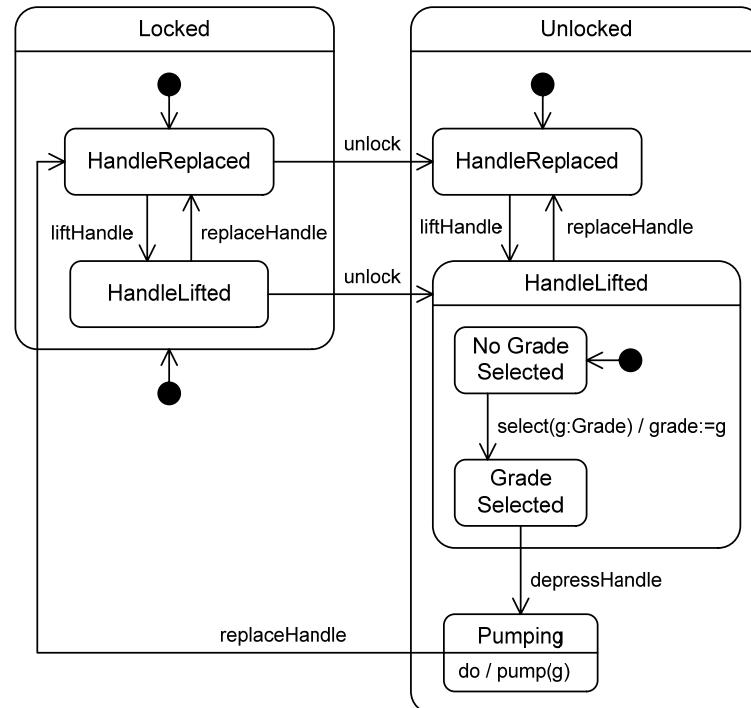
Transducers can be used to model state-based behavior throughout software design. In product design, transducers model the state-based behavior of entire products. Examples in this chapter that fall into this category include the tape recorder, the car's cruise control mechanism, and the traffic light. State diagrams are thus a useful modeling notation during software product design as well as during software engineering design.

Transducers are also useful for modeling software components. Although a single class can be modeled with a transducer, often several classes are modeled together because their states are highly interdependent. For example, consider the Java `Iterator` class and its interactions with any of several Java collection classes. An `Iterator` object provides sequential access to collection elements. If the collection an `Iterator` is traversing changes during traversal, the `Iterator` may be corrupted. A Java `Iterator` throws an exception if it detects that its associated collection has changed. For this to work, collections must keep track of whether they have changed state while an `Iterator` is traversing them, and the `Iterator` must use this information to determine whether to throw an exception. The state diagram in Figure 13-1 models this situation.

This state diagram begins when an `Iterator` instance is created. The `Iterator`'s `next()` operation can be called repeatedly to traverse the collection as long as the collection is not modified. If the collection is modified, everything will still work as long as either the `Iterator` is defunct or its `next()` operation is not called. If the latter occurs, the `Iterator` becomes defunct and an exception is thrown.

**Figure 13-3-1 Iterator and Collection States**

As mentioned previously, a transducer can also model a single class's states. For example, consider a program that controls the pumps at a gas station. It will certainly have a Pump class. A Pump is either locked or unlocked. When unlocked, the Pump can supply three grades of gasoline, but only when its handle is lifted. After gas is pumped, the Pump locks itself. This behavior is described in the state diagram in Figure 13-3-2.

**Figure 13-3-2 Pump Class State Diagram**

In summary, transducers can model state-based behavior at any level of abstraction, from an entire product to small software components.

Acceptor Models

Acceptors often model the behavior of small software modules. A common use of acceptors is to design a **lexical analyzer**, which is a program component that transforms a character stream into a stream of *tokens*, which are symbols recognized by the program.

For example, AquaLush needs to read a configuration file when it starts up. This task involves lexical analysis of the characters in the file to pick out the tokens that AquaLush can interpret to configure itself. The AquaLush SRS specifies a configuration file syntax. After studying this syntax, it is clear that AquaLush must recognize the tokens specified in Table 13-3-3.

Token Name	Token Description
endOfFile	End of the input file marker
leftBrace	{
rightBrace	}
zonelD	"Z" or "z" followed by one or more digits
sensorlD	"S" or "s" followed by one or more digits
valvelD	"V" or "v" followed by one or more digits
semicolon	";"
zoneKwd	The keyword "zone"
sensorKwd	The keyword "sensor"
valveKwd	The keyword "valve"
number	One or more digits
description	Characters between "<" and ">"

Table 13-3-3 AquaLush Tokens

All tokens except the end of file marker, the curly braces, the semicolon, and the descriptor must be separated by white space, such as spaces, tabs, newlines, and so forth. To recognize this set of tokens, an acceptor needs to consume any leading white space, then change state to reflect the input until it again arrives at white space, indicating the end of the token. Part of such an acceptor is pictured in the state diagram in Figure 13-3-4.

The events are named after the characters (such as "l") or classes of characters (such as digit) that are encountered in the input. The special character class other includes any character not named on an outgoing transition. The sub-states of the accepting composite state are token types;

arriving in one of these states means that a token is recognized. Each of these states has an exit action recording the recognized token (not shown). A few accepting states need to consume input past the end of the token as they transition to a final state. In these cases, the last consumed character is pushed back on the input stream by the `pushback` action. One more token type has been added to those in Table 13-3-3: The `badToken` type is needed to indicate that an invalid character sequence has been encountered. The only tokens not recognized by this acceptor are `sensorID`, `sensorKWord`, `valveld`, and `valveKWord`. These require quite a few more states, so they were left out to make the diagram smaller. The complete diagram appears in Appendix B.

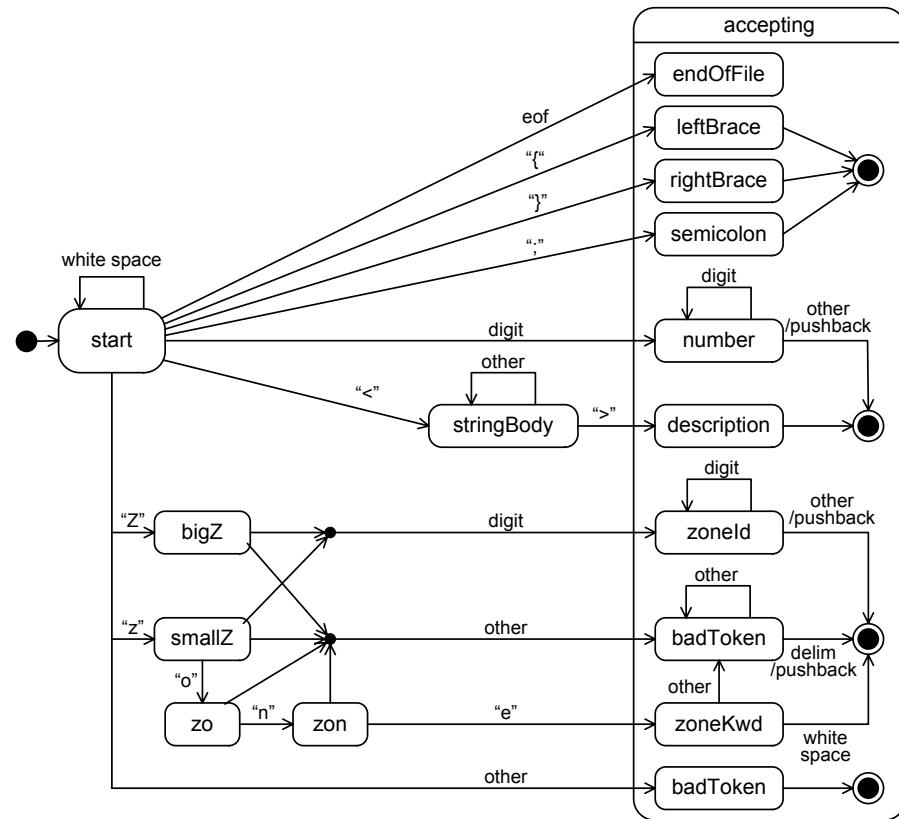


Figure 13-3-4 AquaLush Configuration File Lexical Analyzer

This acceptor is run on the characters read from the AquaLush configuration file and halts as soon as it recognizes a token. It is then run again on the remainder of the input stream to recognize the next token. In this way a character stream is turned into a stream of tokens.

Dialog Maps

Acceptors are also commonly used to model user interfaces. A **dialog map** is a state diagram whose nodes represent user interface states. Dialog maps are thus used to model user interface behavior. For example, the dialog map in Figure 13-3-5 models the behavior of a simple stopwatch program.

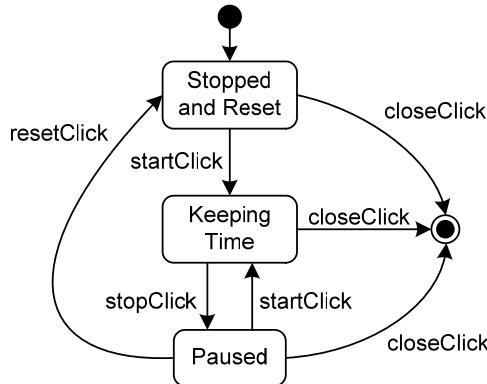


Figure 13-3-5 Stopwatch Program Dialog Map

This dialog map shows that the program first displays itself in the state **Stopped and Reset**. If the **startClick** event occurs, the user interface changes to the **Keeping Time** state. The **stopClick** event transitions to the **Paused** state. The **startClick** event again returns to the **Keeping Time** state, and the **resetClick** event returns to the **Stopped and Reset** state. If the **closeClick** event occurs in any state, the program halts.

User Interface Diagrams

A useful adjunct to a dialog map is a user interface diagram. A **user interface diagram** is a drawing of (part of) a product's visual display when it is in a particular state. User interface diagrams are static models of the visual display.

There are no particular rules governing user interface diagrams: They are simply drawings of the user interface in some state. However, one graphic device—the callout—is especially helpful with user interface diagrams because it is often not completely clear from a drawing alone what a particular element represents. Also, constraints on data values, enabled state, visibility, and so forth are often not clear from a diagram. Consequently, text may be needed to add missing information to user interface diagrams. A **callout** is a note attached to a line or arrow picking out part of a diagram. Figure 13-3-6 shows a simulated gas station user interface diagram with callouts.

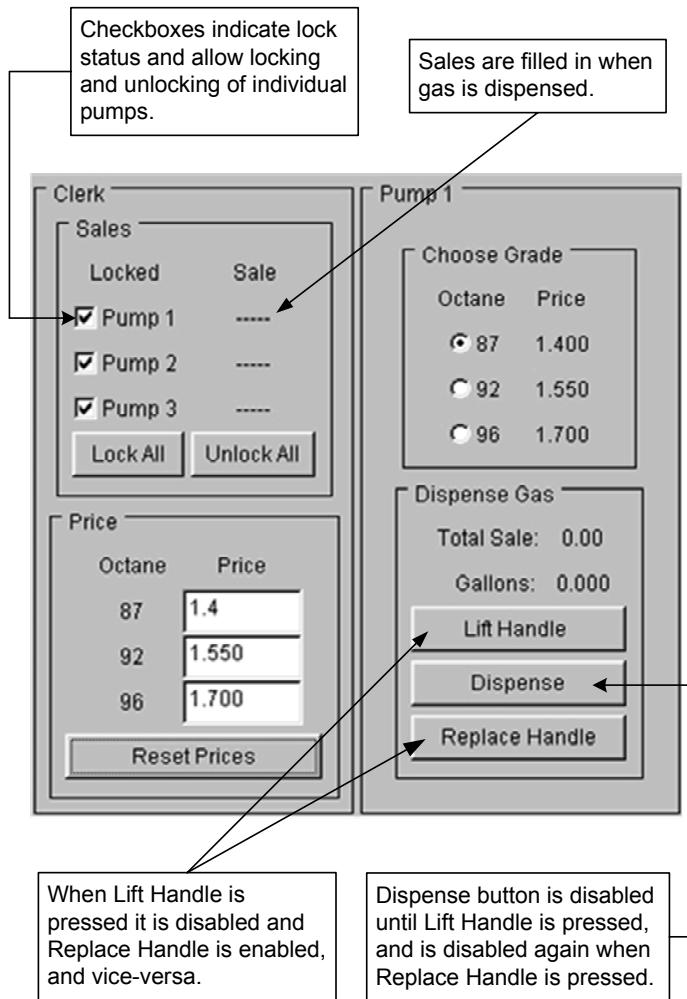


Figure 13-3-6 User Interface Diagram with Callouts

Connecting Dialog Maps to User Interface Diagrams

Each node in a dialog map represents a state of the user interface, and a user interface diagram shows the user interface in one of its states. A complete specification of a user interface should therefore satisfy the following condition.

Every user interface diagram should specify the visual form of a state in a dialog map, and every state in a dialog map should have its visual form specified by a user interface diagram.

Figure 13-3-7 illustrates the connection between user interface diagrams and dialog maps. This figure contains user interface diagrams for each of the three interface states of the stopwatch program whose dialog map appears in Figure 13-3-5.

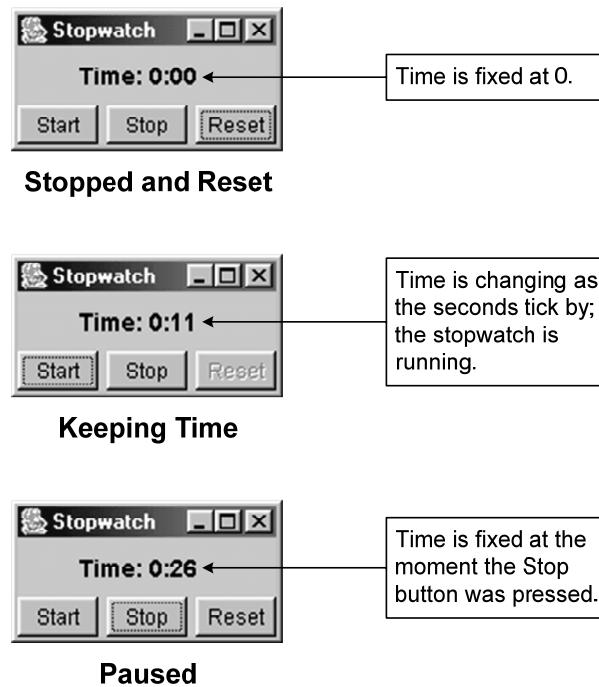


Figure 13-3-7 User Interface Diagrams of Stopwatch States

Sometimes the visual appearance of different states may be nearly the same. For example, one state may differ visually from another only in whether a control is enabled or in the contents of a label. In such cases it is acceptable to use the same user interface diagram to represent several states with the differences explained in callouts or some other text. This trick could easily have been taken with the stopwatch program because the **Stopped and Reset** state and the **Paused** state differ only in the contents of the time field.

Dialog Map Uses

Dialog maps are useful whenever the behavior of a program's user interface is under study. They can be used to help elicit stakeholder needs and desires and to model them to make sure they are well understood and documented. During user interface design resolution, dialog maps can be used to generate and document design alternatives. Dialog maps, especially in conjunction with detailed user interface diagrams, are important tools for documenting the final design of a user interface.

Both dialog maps and user interface diagrams are used to document the AquaLush user interface design recorded in Appendix B.

Section Summary

- An **acceptor** or **recognizer** is a finite automaton that responds to events but generates no actions.
- A **transducer** is a finite automaton that responds to events and generates actions.

- Transducers model the state-based behavior of entities ranging from entire products to small software components.
- Acceptors model the state-based behavior of small software components; they are usually used to recognize input.
- A common use of acceptors is as **dialog maps**, which are state diagrams whose nodes represent user interface states.
- Dialog maps are often used in conjunction with **user interface diagrams**, which are drawings of the visual form of user interface states, to develop, study, and specify user interfaces.

Review
Quiz 13.3

1. Must transducers always model individual classes?
 2. What is a lexical analyzer?
 3. How do user interface diagrams and dialog maps complement one another in user interface design?
-

Chapter 13 Further Reading

- Section 13.1-2** Finite state automata have been studied in computer science and used by software engineers for decades. Automata theory, the study of finite automata, is covered in many texts, including [Hopcroft and Ullman 1979]. Harel [1987] invented state diagrams, which he called *statecharts*. They are included almost unchanged in UML. UML state diagrams are discussed extensively in [Bennett et al. 2001], [Booch et al. 2005], [OMG 2003], [OMG 2004], and [Rumbaugh 2004].
- Section 13.3** Acceptors and transducers are discussed in [Hopcroft and Ullman 1979]. Wiegers [2003] discusses dialog maps.

Chapter 13 Exercises

The following product descriptions are used in the exercises.

Stud Finder	A hand tool vendor is making an electronic stud finder for the home-improvement market. The stud finder is a hand-held device that uses ultrasound to detect differences in densities as it is passed over a surface. The unit has a single button, a green LED called the <i>ready light</i> , and a red LED called the <i>finder light</i> . It can also emit a tone. The unit is off when the button is not pressed. In this state, neither light is on and it emits no sound. To use the stud finder, the user holds the unit against a surface and then presses and holds the button. The unit first calibrates itself by measuring the density of the material beneath it. The unit indicates that it is calibrating itself by turning on both lights and emitting a tone. Once calibrated, which takes about 1/10 of a second, the finder light goes out and the tone stops, leaving only the ready light illuminated. The user may then slide the unit along the surface. When the unit detects an increase in density, the finder light goes on and the tone is emitted. This continues
--------------------	--

until the unit reads a decrease in density, at which point the finder light goes out and the tone stops. The unit thus allows users to find studs. The unit turns off immediately when the user releases the button: All lights go off and the tone stops.

Stud finder calibration may fail in two ways:

- The surface may be too dense to allow accurate readings. In this case, during the calibration process the ready light goes off and the finder light and tone (already on to indicate that calibration is in process) remain on until the user releases the button.
- Initial calibration may appear to succeed, but when the user moves the unit, it reads a decrease in density. This indicates that the unit was calibrated over a stud. In this case, the ready light goes off and the unit beeps until the user releases the button.

Car Wash Control Program

A coin-operated car wash has a software system to monitor and control its activities. The car wash has three stalls, each with its own control pedestal.

A control pedestal accepts money and provides instructions for the kind of wash to do. When no car is in the washer, the pedestal is placed in a resting state where it displays a message inviting customers to deposit money. The pedestal accepts quarters and dollars. As money is deposited, the pedestal displays the total deposited so far. If the customer presses a coin return lever, all deposited money is returned and the pedestal returns to its resting state.

The system receives a sensor event when the car wash soap tank becomes empty. If this occurs, whatever washes are in progress are completed, then all pedestals are set to display an out-of-service message. They will not accept money or respond to button presses. When soap is added, another sensor event indicates that there is soap and the system becomes active again.

If enough money is deposited, the pedestal will respond to button presses for a regular wash (\$4), a super wash (\$5), or a wash and wax (\$6). The pedestal will then deliver change to its coin return, if necessary, and display a message telling the customer to enter the washer.

When a car enters the washer, a sensor sends an event that alerts the system to have the pedestal display a message for other customers to wait and to run the appropriate wash cycle. During this time the pedestal will not accept any money or attend to any button presses.

There is also a sensor that detects when a car leaves. If a car leaves before the cycle is done, the cycle is stopped and the pedestal is returned to its resting state. Otherwise, when the cycle completes, the system waits for the car to exit the washer. When the car exits the washer, or 30 seconds have passed, the pedestal is returned to its resting state.

The system records the time, date, and wash type selected by every customer and records this information in a file. When requested, it

produces either a complete activity log or a summary of the number of each kind of wash, for each day of a specified reporting period.

Section 13.1

1. *Find the errors:* Which of the following expressions are valid UML state diagram transition strings?
 - (a) The empty string
 - (b) buttonPress
 - (c) action(actionEvent)
 - (d) action(e : actionEvent)
 - (e) mouseClick [shiftKey down]
 - (f) [x < 0]
 - (g) [x < 0] / notify()
 - (h) / notify()
 - (i) mouseClick / notify()
 - (j) mouseClick() [shiftKey down] / notify
 - (k) buttonPress [x < 0] / notify(); button.disable()
2. Rewrite the cruise control state diagram in Figure 13-1-5 without composite states; that is, move all nested state diagrams into the outermost state diagram, adding transitions as necessary. Is the result easier or harder to understand?
3. In the state diagram in Figure 13-1-7, what actions are executed if the automaton is in state B and event y occurs?
4. In a room with two doors, it is common to have two switches controlling a single light: one by each door. Flipping either switch changes the light's state from on to off, or vice versa. Make a state diagram illustrating this situation. Your state diagram should track the states of switches A and B (Up or Down) and the state of the Light (On or Off). The events are all switch flips.
5. Make a state diagram showing the states and transitions for a landline telephone. Include the states hung-up, off-hook, dialing, connected, and ringing. Include the events pick up, hang up, and dial.
6. A single-page pager has two buttons: an activation button and a selector button. When it is off, pressing either button turns the pager on. When it is on, the pager goes into its ready state. From the ready state, a selector button press displays a page (typically a phone number) or the message "no page" if no page is recorded. Pressing the activation button at this point erases any recorded page and returns the pager to its ready state. Another selector button press during page display causes the query "off" to be displayed. If the activation button is pressed, the pager turns off; if the selector button is pressed again, the pager returns to its ready state. When the pager is not in its ready state, failure to press any button for five seconds returns the pager to its ready state. If a page is received when the pager is on, the pager records the page and immediately beeps until a button is pressed or it has beeped five times, after which it returns to its ready state. The pager holds only one

page, so each page replaces any undeleted previous page. Turning off the pager erases any recorded page. Make a state diagram describing this product.

7. An expensive novelty gift maker is designing a special grow light for houseplants. This device will monitor the hours of sunlight each day, along with the sunlight's intensity, using a light meter. When light levels are low, the grow light will come on to augment the natural light. When the sun goes down, the grow light will come on for as long as necessary to provide an equivalent minimum number of hours of daylight. The user interface to this device is an on-off switch and a dial specifying the minimum number of hours of daylight for the plant. Make a state diagram to model this product.
8. *Find the errors:* Identify at least five errors in the state diagram in Figure 13-E-1.

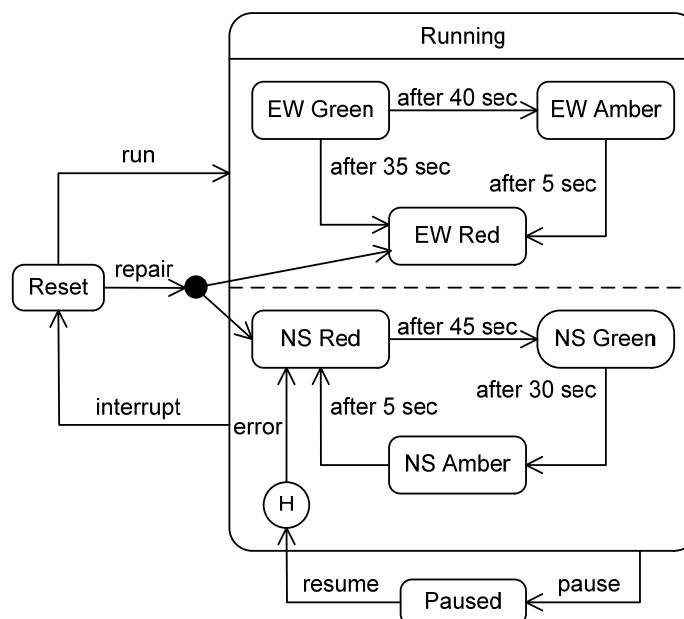


Figure 13-E-1 An Erroneous State Diagram for Exercise 8

9. In a room with two doors, it is common to have two switches controlling a single light: one by each door. Flipping either switch changes the light's state from on to off, or vice versa. Make a state diagram with a concurrent composite state that includes regions for each switch and the light illustrating this situation. Your state diagram should track the states of switches A and B (Up or Down) and the state of the Light (On or Off). The events are all switch flips.
10. Modify the state diagram you made in the last exercise to model the possibility of a power failure. Use one or more history states to ensure that the lights and switches are in the same state that they were in when the power failed.

11. Construct a state diagram with no concurrency specifying a finite automaton equivalent to the concurrent automaton modeling the traffic light shown in Figure 13-2-1.
12. Make a state diagram with a concurrent composite state with synch states illustrating the concurrent workings of a producer and a consumer. The producer repeatedly creates items and the consumer repeatedly consumes them. Your model should guarantee that the consumer does not enter its consume state more often than the producer leaves its produce state.
13. The producer and consumer model in the last exercise implicitly assumes an infinite store for produced items. Make another model with a concurrent composite state in which you have a producer, a consumer, and a queue running concurrently. The producer must wait if the queue is full, and the consumer must wait if it is empty.
14. Convert the model you made in the last exercise into a state diagram with no concurrent composite states.

Section 13.3

15. The *parity* of a bit string is even if the bit string has an even number of ones and odd otherwise. Design an acceptor to recognize all even parity bit strings.
16. Design acceptors to recognize the following bit strings:
 - (a) All bit strings with exactly one 1
 - (b) All bit strings beginning with at least one 0, followed by at least one 1, followed by at least one 0
 - (c) All bit strings ending and beginning with the same bit
 - (d) All bit strings with exactly two 1s
 - (e) All bit strings containing the pattern 101
17. Design an acceptor to recognize all sequences of digits and dashes with the format of a social security number; that is, *ddd-dd-dddd*, where *d* is any digit.
18. Design an acceptor to recognize all sequences of plus or minus signs, digits, and dots, with the format of a real number. The number may begin with a sign (or not), must have at least one digit before and one digit after the decimal point, and may have no decimal point.
19. Design a transducer to model the Stud Finder.
20. Design a transducer to model the Car Wash Control Program.
21. Design a user interface for the cruise control mechanism whose behavior is described in Figure 13-1-5. Document your design with a dialog map and user interface diagrams.
22. Design a user interface for a gas station simulation based on the diagrams in Figures 13-3-2 and 13-3-6. Document your design with a dialog map and user interface diagrams.
23. Design an alternative user interface for the AquaLush Web simulation. Document your design with a dialog map and user interface diagrams.

Research Project

24. Consult David Harel's original paper about state diagrams ([Harel 1987]) and write an essay discussing and illustrating how UML state diagrams differ from Harel's original proposal.

Chapter 13 Review Quiz Answers**Review Quiz 13.1**

1. Every finite automaton specification must include a list of its states; specification of its transitions (including the source state, the target state, and the event that triggers it); and designation of the initial state.
2. The behavior of a deterministic finite automaton is completely specified in response to every event and condition that may occur. In contrast, the behavior of a non-deterministic finite automaton is not so tightly specified—it may have spontaneous transitions or make one of several alternative transitions in response to events.
3. A completion event occurs when the processing in a state ends. It is designated in UML state diagrams by an empty event signature in the transition string.
4. An internal transition is an event that a finite automaton responds to without changing state.
5. When a transition from a composite state boundary occurs, the state and all its sub-states are exited. The exit actions of the states are executed from the innermost to the outermost nested states.

Review Quiz 13.2

1. A concurrent composite state may be entered by a transition to its boundary or by a forked transition to individual states in each component concurrent state diagram. In the former case, each concurrent automaton's initial sub-state is entered simultaneously. In the latter case, each of the forked transition target states is entered simultaneously, along with the initial sub-states of any regions not containing states that are targets of the transition.
2. If a synch state contains the number 1 but two transitions enter the synch state before any leave, the automaton's behavior is undefined. The state diagram is ill formed.
3. Transition junction points are used to combine transition arrows that share portions of their transition strings, simplifying the state diagram and making it easier to read.
4. A history state cannot have any internal transitions because it is not really a state: It is merely an indicator that a previously exited state is to be reentered.

Review Quiz 13.3

1. Transducers can model individual classes, groups of classes (or other kinds of components), entire products, or parts of classes.
2. A lexical analyzer is a program component that transforms a character stream into a stream of *tokens*, which are symbols recognized by the program.
3. User interface diagrams model the static visual form of the user interface in some state, while dialog maps represent the dynamic behavior of the user interface. User interface diagrams and dialog maps together model the two aspects of user interface design: visual form and behavior.

14 Low-Level Design

Chapter Objectives

This chapter concludes our survey of engineering design with a look at low-level detailed design. The last section discusses engineering design finalization. The diagram in Figure 14-O-1 shows where this material fits into the detailed design process.

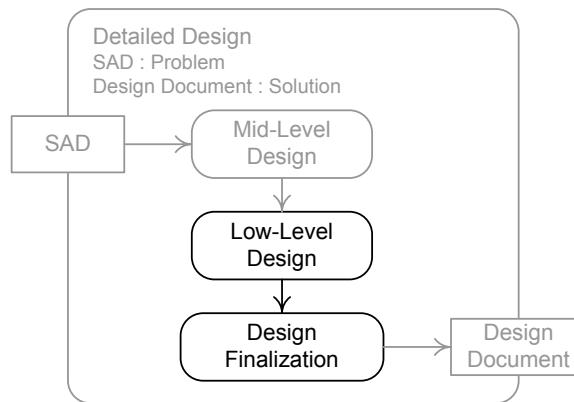


Figure 14-O-1 Detailed Design Process

By the end of this chapter you will be able to

- Explain visibility and accessibility, define visibility categories, and explain references and aliasing;
- Explain how to hide information by limiting visibility and not extending access beyond visibility;
- Read and write operation specifications with declarative behavior specifications using operation contracts;
- Read and write pseudocode algorithm specifications;
- Read and write data structure diagrams; and
- Finalize an engineering design.

Chapter Contents

- 14.1 Visibility, Accessibility, and Information Hiding
- 14.2 Operation Specification
- 14.3 Algorithm and Data Structure Specification
- 14.4 Design Finalization

14.1 Visibility, Accessibility, and Information Hiding

Low-Level Design Specifications

Recall from Chapter 8 the various elements of a complete detailed design specification summarized by the DeSCRIPTR-PAID acronym. The DeSCRIPTR specifications are mainly created during mid-level design:

- Program component decomposition (De) and component relationships (R) are shown in static models, such as class diagrams.
- State and transition modeling (S, T) is done with state diagrams.
- Collaborations (C) are documented in interaction diagrams (such as sequence diagrams).
- Component properties (P) are documented in text.
- Responsibilities (R) are documented in part in text.

However, some DeSCRIPTR specifications are not fully specified by mid-level design models. Specifically, individual operation responsibilities are often not spelled out in sufficient detail in class or module descriptions. Also, operation interface semantics and pragmatics are not fully specified in mid-level static design models (such as class diagrams).

This chapter considers how to specify operation responsibilities and interfaces, and also how to state the PAID specifications of low-level detailed design. In particular, this chapter considers the following topics:

- Packaging (P) and implementation (I) specifications, particularly as they relate to information hiding, considered in this section.
- Operation specifications, which elaborate operation responsibilities (R) and state operation interfaces (I), presented in the second section.
- Algorithms (A) and data structure and type (D) specifications, discussed in the third section.

The final section discusses detailed design finalization.

Program Entity Visibility

A **program entity** is anything in a program that is treated as a unit. Variables, constants, sub-programs, packages, classes, attributes, and operations are all examples of program entities. Most programming languages allow various kinds of program entities. Many can be named by binding an identifier (a string) to them.

A **name** is an identifier bound to a program entity.

Virtually every programming language supports variable and sub-program naming. Many support named constants, and object-oriented languages support class, attribute, and operation naming.

It may or may not be possible to refer to a particular entity by name at various points of a program's text. For example, in Java a variable declared

in a method may be referred to by name only within the method. This fact leads to the concept of *visibility*.

A program entity is **visible** at a point in a program text if it can be referred to by name at that point.

The portion of a text over which a program entity is visible is its **visibility**.

Consider the Java code skeleton in Figure 14-1-1.

```

File: package1/PublicClass.java
package package1;
public class PublicClass{
    private String privateAttribute;
    String packageAttribute;
    public void method() {
        String localVariable;
        ...
        // point A
        ...
    }
    ...
    // point B
    ...
} // end package1.PublicClass

File: package1/PackageClass.java
package package1;
class PackageClass{
    ...
    // point C
    ...
} // end package1.PackageClass

File: package2/PackageClass.java
package package2;
import package1.*;
class PackageClass{
    ...
    // point D
    ...
} // end package2.PackageClass

```

Figure 14-1-1 Visibility Regions in a Java Program

In this code the variable `localVariable` is visible at point *A* but not at points *B*, *C*, or *D*—in fact, `localVariable` is visible only within `method()`.

In contrast, `privateAttribute` and `packageAttribute` are visible at points *A* and *B* because both are visible throughout `package1.PublicClass`. However, `privateAttribute` is visible only in `package1.PublicClass`, while `packageAttribute` is visible at point *C*, though not at point *D*. Class `package1.PackageClass` is visible at points *A*, *B*, and *C* but not at point *D*. Finally, class `package1.PublicClass` is visible at points *A*, *B*, *C*, and *D*.

Types of Visibility

There are several types of visibility commonly available in programming languages and systems:

Local—An entity with **local visibility** is visible only within the module where it is defined. In the example from Figure 14-1-1, `localVariable` and `privateAttribute` have local visibility. In most programming languages, sub-program local variables and parameters have local visibility.

Non-local—An entity with **non-local visibility** is visible outside the module where it is defined, but not visible everywhere in a program. In the example shown in Figure 14-1-1, `packageAttribute` has non-local visibility because it is visible in a module outside the module where it is defined (`package1.PackageClass`), but not in the entire program (that is, not in `package2.PackageClass`). Many programming languages offer such intermediate levels of visibility. For example, in C++ a function or class may be declared a *friend* of a class. All attributes and functions declared in a class are visible to its friend functions and classes, though not necessarily to the entire program.

Global—An entity with **global visibility** is visible everywhere in a program. In the example from Figure 14-1-1, `package1.PublicClass` has global visibility because it can be imported and used in any class file. Most languages allow global visibility.

Every named entity falls into one of these three visibility categories. We also say that an entity is **exported** from the module where it is defined when it is visible outside that module. Exported entities thus have either non-local or global visibility.

Object-oriented languages have additional visibility categories especially for attributes and operations, though they vary from language to language. We have already discussed these in connection with UML class diagrams, but we review them in this section and discuss their relation to local, non-local, and global visibility.

Private—An attribute or operation with **private visibility** is visible only within the class where it is defined. Private features have local visibility.

Package—An attribute or operation with **package visibility** is visible in the class where it is defined as well as all classes in the same package or namespace. Package visibility is a form of non-local visibility.

Protected—An attribute or operation with **protected visibility** is visible in the class where it is defined as well as all its sub-classes. This is another form of non-local visibility.

Public—An attribute or operation with **public visibility** is visible in the class where it is defined and anywhere this class is visible. Public attributes and operations have either non-local or global visibility, depending on the visibility of their enclosing classes. For example, in Java a public attribute or operation of a public class has global visibility, but a public attribute or operation of a non-public class has non-local visibility.

Programming languages adapt these object-oriented visibility categories in various ways. For example, Java modifies protected visibility to include classes in the same package and extends the visibility categories to apply to classes as well as attributes and operations. C++ does not have a package construct, but it does have friend classes and functions, which are similar.

Accessibility

If a program entity is visible, it can be used. For example, if a variable is visible at some point in a program text, we can assign it a value or obtain its value; if an object is visible, we can send it messages. But sometimes we can use a program entity even when it is not visible. For example, in C++ an object passed through a reference parameter into a method can be used even if it is not visible inside the method—the parameter gives us a way to get to the object even if we can't refer to it by name. We need another concept to describe this situation.

A program entity is **accessible** at a point in a program text if it can be used at that point.

A program entity is accessible everywhere it is visible, but it may also be accessible even where it is not visible. Let's consider how this works by looking in more detail at variables.

A **variable** is a programming language device for storing values. A variable has several attributes, including its name, value, and address. The **address** is the location of the storage cell in computer memory where the variable's value is stored. Usually it is an unsigned integer that the computer hardware interprets as designating a word of memory. The diagram in Figure 14-1-2 shows the relation among the attributes of a variable.

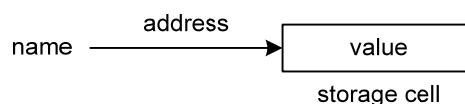


Figure 14-1-2 Attributes of a Variable

The variable's name is associated with its address, and the address designates a storage cell in computer memory where the variable's value is stored. We may also use a variable's name as shorthand to refer to its associated storage cell or its value.

The principal mechanisms for accessing variables (and hence their storage cells or values) without using their names are references and aliases, which we consider next.

A **reference** is an expression that evaluates to an address where a value is stored.

A reference to a variable or its value is an expression that evaluates to the variable's address; in other words, the place where the variable's value is stored. A variable may hold a reference as its value. A variable holding a reference provides a way to access a storage cell, and hence a program entity, without using its name. For example, the diagram in Figure 14-1-3 shows the configuration of two variables after the indicated C++ code fragment is executed.

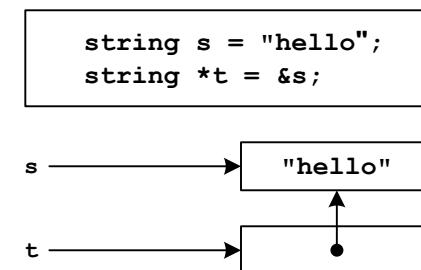


Figure 14-1-3 A Reference to a Named Variable

The first line of code creates a new `string` object and assigns it to the variable `s`. The identifier “`s`” is the name of the variable that holds the new string value and, by extension, the name of the storage cell and the string value itself. The second line of code creates a variable `t` whose value is a reference to the storage cell associated with `s`. We can use the reference stored in `t` to access `s` without using its name.

In Java, unlike C++, all object and array variables actually hold *references* to objects and arrays, not the objects or arrays themselves. Hence, objects and arrays in Java *never* have names and can be accessed only through references. This policy makes some things in Java easier (the syntax of references) and other things harder (methods cannot be kept from changing objects sent to them as parameters). Although technically Java objects and arrays don't have names, from now on we regard the name of the *first* variable holding a reference to an object or array (the one holding the reference returned by a constructor) as the name of that object or array.

The second way to access a program entity without using its name is through an alias.

An **alias** is a variable with the same address as another variable.

Note that if two variables have the same address, then they must share the same storage cell and, hence, have the same value.

Some programming languages allow aliases, though most do not. To illustrate aliases, consider the picture of the state of a C++ program after execution of the indicated code shown in Figure 14-1-4.

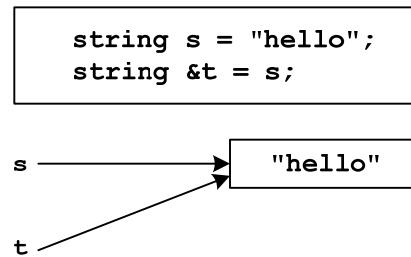


Figure 14-1-4 A Variable with an Alias

The first line of code creates a new `string` object and assigns it to the variable `s`. As before, the identifier “`s`” is regarded as the name of the variable, its storage cell, and the new `string` object. The second line of code creates a variable `t` whose address is the same as the address of `s`. The first identifier bound to a value is its name, and any identifiers bound subsequently are considered aliases, so “`t`” is an alias. We can use the alias “`t`” to access `s` without using the name “`s`”.

A program entity may be accessible through an alias even where it is not visible. In particular, the pass-by-reference mechanism available in several languages (such as Pascal or C++) for passing parameters to sub-programs makes a formal parameter an alias for the actual parameter used as a sub-program argument. The alias can be used to access the argument variable even though it may not be visible inside the sub-program.

In summary, a program entity may be accessible even where it is not visible. The most common ways this happens are

- Passing a reference as an argument;
- Passing an argument by reference (which uses aliasing); and
- Returning a reference from a sub-program.

The entity referenced or aliased is thereby made accessible in a portion of the program text where it is not visible.

Making a variable accessible in a portion of a program where it is not visible is called *extending access beyond visibility*. In general, extending access

beyond visibility is not a good idea, but there are occasions when it is appropriate. We discuss these next.

Accessibility and Information Hiding

Information hiding is a fundamental design principle. Its intent is quite clear, but how to achieve it may not be. How should a module shield its internal processing details from other modules? In fact, hiding information is not hard, as the following observation makes clear.

The key technique for hiding information is to restrict access to program entities as much as possible.

This technique is applicable at all levels of abstraction in every kind of system, regardless of programming language. Restricting access to program entities relies on two strategies:

Limiting Visibility—Program entities should be visible in the smallest possible portion of a program. Absent further action (such as passing a reference outside the region), limiting visibility cuts off access to program entities and hides them in the regions where they are visible. Such regions include, from smallest to largest, blocks, sub-programs, classes, compilation units, packages, libraries, and programs. Designers should limit program entity visibility to regions near the front of this list.

Not Extending Access—Limiting visibility does no good if references are passed around willy-nilly or variables are aliased. Designers must be especially careful not to extend access beyond visibility by accident. For example, suppose a Java class contains an attribute hidden by giving it private visibility. If the attribute has a fundamental data type (such as `int`), then its value may be returned as the result of a query operation without exposing the attribute. But if the attribute is an object or an array, then returning its value as the result of a query operation returns a reference to that object or array. The caller can then access the attribute through this reference! To keep the attribute hidden, the query operation must make a copy of the attribute (called a **defensive copy**) and return the copy instead.

Information Hiding Heuristics

Many well-known low-level design heuristics follow from the strategies of limiting visibility and not extending access. The following heuristics help limit visibility:

Restrict the scope of declarations to the smallest possible program segment.

Make class attributes at least protected and preferably private.

*Make class helper operations at least protected and preferably private. A **helper operation** is an operation needed to implement an exported operation.*

Avoid global visibility.

Avoid package visibility.

The following heuristics help in not extending access beyond visibility:

Don't initialize class attributes with references passed to the class—make defensive copies instead.

Don't pass or return references to class attributes—pass or return defensive copies instead.

Don't pass parameters by reference (which uses aliasing).

Don't make aliases.

Extending Access Beyond Visibility

There are two cases when it is appropriate to extend access beyond visibility, despite the Principle of Information Hiding. The first case is when modules must share an entity to collaborate.

Often, modules collaborate by sharing access to some entity not visible to them both. For example, suppose a *Producer* module generates values processed by a *Consumer* module and that the values are sent from *Producer* to *Consumer* through a Queue *q*. Both the *Producer* and the *Consumer* *must* have access to *q*, which is (let us suppose) not visible to one or both of them. In this case, it is appropriate (in fact, necessary) to pass *q* to the *Producer* and the *Consumer* using a reference or an alias.

In such cases the shared entity is not meant to be hidden from the modules that use it, so the Principle of Information Hiding is not violated—in fact, the system is *intended* to work by sharing some program entity. This principle would be violated, however, if other modules not involved in the collaboration had access to the shared entity. An entity shared among only a few modules can be made widely accessible (by giving it global visibility, for example), but then this detail of the workings of the program is not hidden from other modules, which *is* a violation of the Principle of Information Hiding. Limiting visibility and passing a reference or alias to the modules involved extends access in a controlled way and hides information appropriately.

The second case when it is appropriate to extend access beyond visibility is when the Principle of Information Hiding is purposely violated to achieve some other design goal. One example is the occasional need to violate this principle to achieve performance goals. It is much faster to pass a large value to a sub-program using a reference parameter or by passing it by reference than to pass it by value (which requires making a copy). If performance goals can be met only by passing a reference or using pass-by-reference, then this is what must be done, although doing so exposes the argument to modification by the sub-program.

We will encounter other examples where a module exposes its internal processing to another module when we consider design patterns. Some fundamental design patterns can work only by violating the Principle of Information Hiding. In such cases the advantages of using the pattern outweigh adhering to the design principle.

In summary, extending access to an entity beyond its visibility should be done only to provide access to entities explicitly shared in module

collaborations or in conscious violation of the Principle of Information Hiding for the sake of other important design goals.

Heuristics Summary

Figure 14-1-5 summarizes the information-hiding heuristics discussed in this section.

- | |
|---|
| <p>Limit Visibility</p> <ul style="list-style-type: none"> □ Make program entities visible in the smallest possible program region. □ Restrict the scope of declarations to the smallest possible program region. □ Make class attributes at least protected and preferably private. □ Make class helper operations at least protected and preferably private. □ Avoid global visibility. □ Avoid package visibility. <p>Don't Extend Access</p> <ul style="list-style-type: none"> □ Don't initialize class attributes with references passed to the class—make defensive copies instead. □ Don't pass or return references to class attributes—pass or return defensive copies instead. □ Don't pass parameters by reference. □ Don't make aliases. |
|---|

Figure 14-1-5 Information Hiding Heuristics

Section Summary

- A **program entity** is something in a program treated as a unit.
- A **name** is an identifier bound to a program entity.
- A program entity that can be referred to by name at some point in a program is **visible** at that point, and the portion of the program over which the entity is visible is its **visibility**.
- Program entities have **local, non-local, or global visibility**.
- Attributes and operations have special types of visibility: Private features have local visibility, protected and package features have non-local visibility, and public features have at least non-local and sometimes global visibility.
- A program entity is **accessible** at any point in a program where it may be used. Entities are accessible by name or through references or aliases.
- The principal means of hiding information is to limit accessibility by restricting visibility and not extending access to program entities.
- Access may have to be extended in special circumstances.

Review Quiz 14.1

1. What is the difference between visibility and accessibility?
2. What is the difference between passing a reference as an argument and passing an argument by reference?

3. What two strategies are integral to restricting access to program entities?
 4. What are two occasions when it is appropriate to extend access beyond visibility?
-

14.2 Operation Specification

Operation Specification Elements

Operations realize program behavior, so designs must specify them down to some level of detail. Certainly operation names will appear somewhere in a design—for example, in sequence and class diagrams. Sequence diagrams may also show operation arguments and return values, and class diagrams may show operation signatures and visibilities.

Although important, these details are hardly enough to document operation responsibilities and interfaces well enough for other designers, programmers, or maintainers to understand them. For one thing, a signature specifies only an operation’s interface syntax and not its semantics or pragmatics (interface specifications are discussed in Chapter 9). Sequence diagrams give only a rough indication of an operation’s responsibilities and some clues about its interface semantics and pragmatics. Depending on the skill of the programmers, a designer may also need to specify the processing details (the algorithms and data structures) of difficult or crucial operations.

Specification of an operation’s responsibilities and interface can be collected in one place, in structured text called an **operation specification**, or *op-spec*. Op-specs should include the following fields:

Class or Module—The operation’s class (in object-oriented designs) or module (in non-object-oriented designs). This is basic identification information.

Signature—The operation name, the names and types of its parameters, and its return type. The signature may include additional information about the operation (such as the exceptions it throws and its visibility) depending on the specification language used. The signature specifies the operation’s interface syntax.

Description—A sentence or two informally stating the operation’s responsibilities.

Behavior—A detailed account of what the operation does, including the constraints on its arguments, the conditions under which it is called, what values it returns, other effects it has on its environment, and what actions it takes when it encounters undesirable conditions (such as exceptions it throws). This portion of the op-spec states the operation’s semantics and pragmatics.

In addition, an op-spec may include a coding specification:

Implementation—A detailed description of the algorithm and data structures used to implement this operation.

We discuss the contents of operation specifications in more detail in this section and notations for implementation specification in the next section.

Class and Signature	<p>In object-oriented design and programming, an operation's class and signature are the means needed to identify it. Because polymorphism allows different classes to have operations with the same name, an operation's class is necessary to establish its identity. Polymorphism also allows a single class to have several operations with the same name, so the number and type of an operation's arguments are also part of its identity. These must all appear in the operation specification to identify the operation.</p> <p>Additional information may be specified about an operation in its signature, depending on the specification language used. For example, Java signatures include operation visibility (public, private, protected, or package); whether the operation is a class or instance operation (static or not); whether the operation may be executed concurrently with other operations in the class (synchronized or not); and the checked exceptions that may be thrown by the operation.</p>
Module and Signature	<p>Although an operation's module may not be necessary to identify it in a non-object-oriented system, the module is still important because it determines, in part, what is visible to the operation. Furthermore, an operation should be put into a module based on whether it really belongs there (cohesion), whether it links the module too closely or in undesirable ways with other modules (coupling), and whether it shields or exposes module implementation details (information hiding). Thus, the choice of module is an important design decision that should be recorded explicitly in the op-spec, even if it appears in whole or in part in other diagrams.</p> <p>Similarly, even in systems that do not support operation polymorphism, operation signatures are very important. The parameters and return type determine what data flows into and out of an operation, and the rest of the op-spec is probably not understandable without this information.</p>
Description	<p>Sometimes a short textual description of an operation's responsibilities is more helpful in understanding what it does than all the rest of the information in the operation specification put together. Operation descriptions should be short and informal: More detailed and precise specifications appear in the op-spec's behavior specification.</p>
Behavior	<p>A behavior specification states an operation's interface semantics and pragmatics. There are two ways to do this:</p> <ul style="list-style-type: none">▪ Describe operation inputs, calling constraints, and results, but <i>not</i> the processing that the operation uses to transform its inputs into its outputs.

- Describe an **algorithm** (a sequence of steps that can be performed by a computer) for transforming the operation's inputs into its outputs. Note that this algorithm is meant not to dictate an implementation, but only to describe what the operation is supposed to do. Programmers are free to implement the operation however they like, provided its inputs and outputs are identical to the designer's algorithm.

A behavior description that does not use an algorithm is called a **declarative specification**; an algorithmic behavior description is called a **procedural specification**. Operation behavior can be described either declaratively or procedurally.

For example, suppose we want to specify the behavior of an operation with the following Java signature:

```
public static int findMax( int[] a )
    throws IllegalArgumentException
```

We could say that this operation takes a non-null `int` array with at least one element as input and returns the largest value in the array as its result. This is a declarative specification.

Alternatively, we could say that the operation throws an `IllegalArgumentException` if its argument is `null` or has no elements. Otherwise, it saves the first value of the array in a variable, then loops through the remaining elements and checks each one to see if it is larger than the value in the variable. If so, it assigns the larger element into the variable. It returns the value of the variable as its result when the loop terminates. This is a procedural specification.

Declarative specifications are preferable for several reasons:

- Declarative specifications are more abstract because they ignore implementation details, and consequently they are more concise than procedural specifications.
- Declarative specifications focus on an operation's interface, making it clearer how an operation interacts with its callers.
- Procedural specifications may bias programmers toward the implementation used in the specification. A procedural specification may be good as an explanation for designers, but not for realization in code.

Designers should use declarative rather than procedural specifications to describe operation behavior. *Operation contracts* are a good way to make declarative specifications: We discuss them next. Various notations can be used for procedural specifications. We discuss some in the next section.

Operation Contracts

A **contract** is a binding agreement between two or more parties. Contracts usually state each party's rights and obligations. Viewing software interactions in this light, an operation caller is obliged to provide valid arguments and to call the operation under the correct circumstances. In

return, the caller has the right to expect the called operation to provide a certain computational service. The mirror image of caller rights and obligations are the rights and obligations of the called operation: It is obliged to provide advertised services, and it has the right to expect that it is being called in the right circumstances with valid arguments.

The rights and obligations in an operation contract are stated using assertions.

An **assertion** is a statement that must be true at a designated point in a program.

An assertion that must be true when an operation is called states the calling operation's obligations and the called operation's rights. An assertion that must be true when an operation returns states the called operation's obligations and the calling operation's rights. These two kinds of assertions have names.

A **precondition** is an assertion that must be true at the initiation of an operation.

A **postcondition** is an assertion that must be true upon completion of an operation.

By stating the rights and obligations of the parties in the interaction that occurs when an operation is called, an operation's preconditions and postconditions specify a contract between an operation and its caller:

- The preconditions express the caller's obligations and the called operation's rights.
- The postconditions express the caller's rights and the called operation's obligations.

Let's illustrate these ideas with an example. Consider again the maximum finding operation with the Java signature:

```
public static int findMax( int[] a )
    throws IllegalArgumentException
```

Recall that the argument must not be null and must contain at least one element. We can state assertions in many ways; here we use a hybrid notation combining Java and English:

`pre: (a != null) && (0 < a.length)`

The `pre:` symbol marks this assertion as a precondition.

Upon completion, the `result` (return value) is the largest element in the array. Furthermore, if preconditions are violated, then this operation throws an exception. We can state these postconditions as follows.

post: for every element x of a , $x \leq result$
 post: throws `IllegalArgumentException` if preconditions are violated

The `post:` symbol is the indicator for a postcondition. The keyword `result` is used in a postcondition to indicate the value returned by the operation.

Table 14-2-1 shows the entire op-spec for the `findMax()` operation.

Signature	<code>public static int findMax(int[] a) throws IllegalArgumentException</code>
Class	Utility
Description	Return one of the largest elements in an int array.
Behavior	<pre>pre: (a != null) && (0 < a.length) post: for every element x of a, x <= result post: throws <code>IllegalArgumentException</code> if preconditions are violated</pre>

Table 14-2-1 An Operation Specification

Using pre- and postconditions to specify an operation by stating the rights and obligations of the operation and its callers is called **design by contract**.

Class Invariants A **class invariant** is an assertion that must be true of any class instance between calls of its exported operations. In other words, a class invariant constrains the class's states, though it may be violated briefly during a state change. For example, suppose that a class has two attributes x and y , and a class invariant stating that y must be twice x . This invariant may not be true when an operation that changes the values of x and y is executing, but it will be true before and after the operation executes.

A class invariant augments every exported operation's contract. In effect it is added to every precondition and postcondition. Usually class invariants specify relationships among class attributes that must be maintained by the class's operations.

Class invariants must

- Be established by class constructors, and
- Be preserved by every exported operation in the class.

To illustrate class invariants, consider a simple program used in a health and fitness club to assign lockers to patrons. When asked for a locker, the program issues a key card for a locker as far from already issued lockers as possible (the idea being to spread people out in the locker room). When a patron is done with a locker, the program scans the key card and returns the locker to the pool of free lockers.

One of the classes in this program is `LockerRoom`, which is in charge of knowing which lockers have been issued, issuing lockers, and handling freed lockers. The UML class diagram in Figure 14-2-2 shows `LockerRoom`.

LockerRoom
- roomSize : int { final } - isFree : boolean[*] { ordered } - numFree : int
+ LockerRoom(size : int) + issueLocker() : int + freeLocker(locker : int) + isFull() : boolean

Figure 14-2-2 The LockerRoom Class

The idea behind this design is that a `LockerRoom` object is created with `roomSize` lockers numbered from 0 to `roomSize`-1. A list of Boolean values records which lockers are free. The constructor sets up the attributes to represent an empty locker room. The other operations issue a locker, return a locker to the free locker pool, and indicate whether all lockers have been issued.

We can state several assertions about the attributes in this class. First, `roomSize` must be at least 1. Second, the `numFree` attribute must vary between 0 and `roomSize`. Third, the number of true values in the `isFree` array must be `numFree`. These assertions can all be stated succinctly as the following invariants.

```
inv: 0 < roomSize
inv: 0 <= numFree && numFree <= roomSize
inv: numFree is the number of true elements in isFree
```

The `inv:` symbol marks these assertions as class invariants.

Invariants need to be part of an operation's behavior specification because they influence pre- and postconditions and because they constrain operation implementation. For example, the first invariant above dictates a constructor precondition. The second and third influence preconditions and also constrain programmers in coding the operations.

Table 14-2-3 provides operation specifications for the `LockerRoom` class. The format has been altered from the one used in Figure 14-2-2 to take advantage of the fact that we are specifying several operations from the same class.

Some of the operations specified in Table 14-2-3 modify variables. This is stated in postconditions in expressions using the notation `variable@pre`, which refers to the value of the variable at operation initiation.

Note how the pre- and postconditions of `issueLocker()`, together with the class invariants, entail that only a single value of `isFree` is changed from true to false, and that the corresponding locker number is the one returned by the operation. Thus, the invariants work with the pre- and postconditions to specify operation behavior.

Class	LockerRoom
Invariant	inv: $0 < \text{roomSize}$ inv: $0 \leq \text{numFree} \&& \text{numFree} \leq \text{roomSize}$ inv: numFree is the number of true elements in isFree
Signature	public LockerRoom(int size) throws IllegalArgumentException
Description	Create and initialize an instance.
Behavior	pre: $0 < \text{size}$ post: throw IllegalArgumentException if the precondition is violated post: roomSize == size and numFree == size
Signature	public boolean isFull()
Description	Indicate whether lockers are available.
Behavior	post: result == (numFree == 0)
Signature	public int issueLocker() throws IllegalStateException
Description	Return the locker in the middle of the largest segment of available lockers.
Behavior	pre: isFull() is false post: throw IllegalStateException if precondition is violated post: $0 \leq \text{result} \&& \text{result} < \text{roomSize}$ post: result is in the middle of the largest segment of available lockers post: isFree[@pre[result]] == true and isFree[result] == false post: numFree == numFree[@pre - 1]
Signature	public void returnLocker(int locker) throws IllegalArgumentException
Description	Put a locker back into the available lockers pool.
Behavior	pre: $0 \leq \text{locker} \&& \text{locker} < \text{roomSize}$ pre: isFree[locker] == false post: throw IllegalArgumentException if precondition is violated post: isFree[locker] == true post: numFree == numFree[@pre + 1]

Table 14-2-3 LockerRoom Operation Specifications

What to Put in Assertions Contracts specify the rights and obligations of an operation and its callers. Operation users must understand how and when to call the operation and what to make of its results. The following heuristics help designers write preconditions:

Specify restrictions on parameters. For example, parameters may not be allowed to be null or must have values in a certain range, or there may have to be relationships among parameter values.

Specify conditions that must have been established. An operation may require that other operations be called before it, or it may require that certain files or other resources be available and in a certain state.

Specify empty preconditions as true or none. If an operation has no preconditions, they may be stated simply as `true` (the weakest statement, since it really says nothing), or as `none`.

The following are heuristics for writing postconditions:

Specify relationships between the parameters and the results. This assertion explains how the operation outputs are related to its input.

Specify restrictions on the results. The range of the result may be limited, or the result may have certain values under certain conditions.

Specify changes to parameters. Parameters passed by reference, or reference values passed by value, allow the parameters to be changed. These changes need to be documented.

Specify responses to violated preconditions. Operations can respond to precondition violations in many ways, such as adjusting bad inputs or conditions, ignoring the violations, or raising exceptions.

Class invariants are supposed to be about the state of the class between executions of exported operations. State is defined by attributes, so invariants should be about attributes. In particular, class invariants should constrain the values of attributes in the following ways:

Specify restrictions on attributes. Attributes may need to be in certain ranges or have only certain kinds of values.

Specify relationships among attributes. Attributes jointly define the state of a class, so often certain relationships among them must be maintained.

For example, the size of a list may have to be between limits specified in other attributes.

Developing Operation Specifications

Operation specifications should be developed gradually, in conjunction with the rest of a design. Classes and operations tend to be quite fluid early in mid-level design, as many alternatives are considered and class and sequence diagrams evolve. Making op-specs at this early stage is a waste of time because they will likely change. As the design stabilizes, work can begin on op-specs, starting with those least likely to change and progressing to others over time.

It is not a good idea to leave writing op-specs until the very end of design, for two reasons:

- Specification details are likely to be forgotten if they are not written down, so an operation specification should be written for an operation as soon as it appears that the details are set.
- If writing op-specs is left until the end of the design effort, it's likely it won't be done well.

Not every operation needs an op-spec. Especially in object-oriented systems, a large fraction of the operations in most classes are simple attribute access (set or get) operations. These operations tend to be trivial—set operations check parameters and constraints and then alter an attribute, while get operations merely return the value of an attribute. It is a waste of time to develop operation specifications for these or other equally simple operations. Operation specifications should be developed only for interesting operations that really require a designer’s attention.

Heuristics Summary Figure 14-2-4 summarizes the heuristics for writing assertions discussed in this section.

Precondition Heuristics

- Specify restrictions on parameters.
- Specify conditions that must have been established.
- Specify empty preconditions as true or none.

Postcondition Heuristics

- Specify relationships between the parameters and the results.
- Specify restrictions on the results.
- Specify changes to parameters.
- Specify responses to violated preconditions.

Class Invariant Heuristics

- Specify restrictions on attributes.
- Specify relationships among attributes.

Figure 14-2-4 Assertion Heuristics

Section Summary

- An **operation specification** is a structured text specifying an operation’s interface, and responsibilities, and possibly its implementation.
- Behavior specifications can be **declarative** (non-algorithmic specifications) or **procedural** (algorithmic specifications).
- An **assertion** is a statement that must be true at a designated point in a program.
- A **precondition** is an assertion that must be true at the initiation of an operation; a **postcondition** is an assertion that must be true upon completion of an operation.
- Operation pre- and postconditions specify a **contract** detailing the rights and obligations of the operation and its callers.
- Using contracts to state declarative operation behavior specifications is called **design by contract**.

**Review
Quiz 14.2**

1. What operation specification fields are most concerned with detailing an operation's interface?
 2. What is the difference between a declarative and a procedural specification?
 3. What is a class invariant?
-

14.3 Algorithm and Data Structure Specification

**Algorithm
Specifications**

Algorithms may be stated in an operation specification for two reasons:

- An algorithm may be stated to provide a procedural specification of the operation's behavior. In this case programmers are not constrained to use the algorithm as stated, just one with the same inputs and outputs.
- An algorithm may be stated to specify that the algorithm is to be used in the implementation. In this case programmers are obliged to use the specified algorithm in their code.

Declarative behavioral specifications are generally better than procedural specifications, as discussed in the last section. Usually, designers have enough to do without bothering to tell programmers what algorithms to use in their code, and neither reason for specifying algorithms in an operation specification should arise often. Nevertheless, there may be occasions when algorithm specification is necessary.

Well-known algorithms may be specified by name. For example, a designer may dictate that a search operation use a binary search or an interpolation search. However, relatively few algorithms are known by name, so usually algorithm specification requires stating each step in the computation.

A **minispec** is a step-by-step description of how an operation transforms its inputs to outputs. Calls to other operations may be included in minispecs. Minispecs may be written in a programming language, but more often they are written in pseudocode.

Pseudocode is English augmented with programming language constructs. It offers much of the ease and power of expression of English together with the precision of a programming language, if it is written well. There are many versions of pseudocode and no widely accepted standards; most designers simply make up a pseudocode as they go along.

Figure 14-3-1 illustrates pseudocode. This pseudocode specifies the binary search algorithm. Note that some lines are almost program code, while others are closer to English.

```

Inputs: array a, lower bound lb, upper bound ub,
        search key
Outputs: location of key, or -1 if key is not found

lo = lb
hi = ub

while lo <= hi and key not found
    mid = (lo + hi) / 2
    if      (key = a[mid]) then key is found
    else if (key < a[mid]) then hi = mid-1
    else                      lo = mid+1

    if key is found then return mid
else                           return -1

```

Figure 14-3-1 Pseudocode for Binary Search**Data Structure Specification**

An **abstract data type (ADT)** is a set of values (the **carrier set** of the type) and operations for manipulating those values. Arrays, stacks, queues, lists, and trees are all examples of abstract data types. ADTs are mathematical entities that can only be used in a computer program if given an implementation. ADT implementations must realize both aspects of the type: There must be an implementation for the values of the carrier set in computer memory, and an implementation of the operations for manipulating the values. Algorithms realize the operations, and data structures realize the values. A **data structure** is thus a scheme for storing values in computer memory.

Data structures generally employ two implementation strategies:

Contiguous Implementation—Values are stored in adjacent memory cells.

Linked Implementation—Values are not necessarily stored in adjacent memory cells and are accessed using pointers or references.

Data Structure Diagrams

A simple graphical notation is traditionally used to depict data structures. This notation has no generally accepted name; here we call it the **data structure diagram**.

Data structure diagrams use rectangles to represent memory cells. If the cell has a name, it is placed near the rectangle's boundary. The value stored in a cell can be designated by a label written inside the cell's rectangle.

Contiguous data structure implementations are represented by placing a series of memory cell rectangles in a row or column, forming an array or record. Some array or record elements may be suppressed and indicated by an ellipsis (...). The array or record may be named by a label placed near it.

Indices, like names, may be placed next to the cells they designate and field names may be placed beside memory cell rectangles to name them. Ellipses may also be used to indicate ranges of array values. Figure 14-3-2 illustrates an array pictured using this notation.

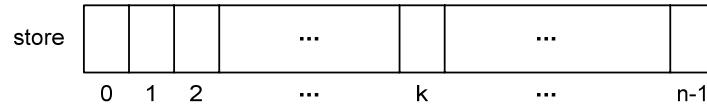


Figure 14-3-2 Array Data Structure Diagram

References or pointers to memory cells are represented by pointer arrows originating in the memory cell rectangles where the references or pointers are stored and terminating at the edges of the memory cells to which the references or pointers refer. A dot represents the null pointer or reference. Linked data structure implementations are represented by connecting disjoint collections of arrays or records with pointer arrows. As with arrays, a sequence of similar data elements can be suppressed and indicated by an ellipsis. For example, the diagram in Figure 14-3-3 depicts a binary tree.

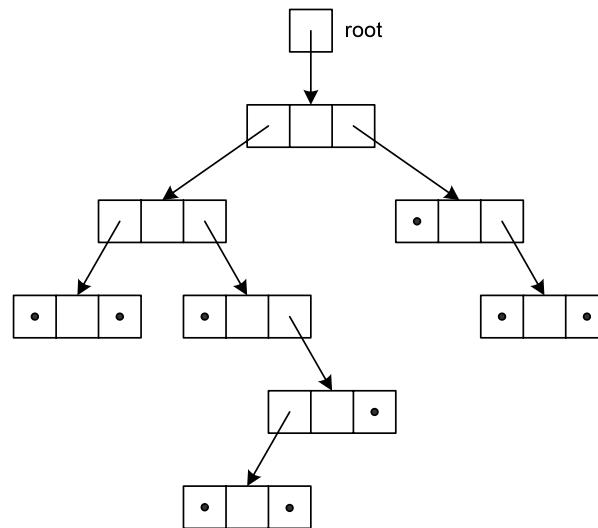


Figure 14-3-3 Binary Tree Data Structure Diagram

The contiguous structures are records with three fields storing pointers to the left and right sub-trees and the data stored at the node. Values in the data fields are not shown. The **root** memory cell stores a pointer to the root node.

Arbitrarily complex structures with linked and contiguous components can be pictured using this notation. For example, the diagram in Figure 14-3-4 represents a hash table with chaining used to resolve collisions.

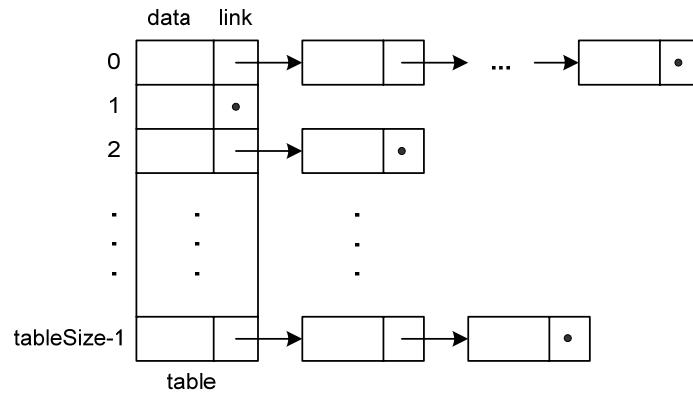


Figure 14-3-4 Hash Table Data Structure Diagram

Figure 14-3-4 shows the hash table as an array of records. Each record has a **data** and a **link** field. The middle portion of the array is suppressed, as indicated by the vertical ellipses in and beside it. The **link** field holds a pointer to a singly linked list of records whose keys collide at that hash table location. The linked lists show the various possibilities: The first list is long, and part of it is suppressed, as indicated by the ellipsis; the second list is empty, the third has a single element, and so on.

Comments can be added to data structure diagrams as free-floating text or in callouts. Other symbols can be added provided they are explained in a legend (as in box-and-line diagrams) or in accompanying text. Sometimes other shapes (such as circles) are used to represent parts of data structures, or arrowheads are removed to simplify the diagram when the direction of the pointers is clear from context (as in tree diagrams).

Data structure diagrams are a convenient notation for generating, evaluating, improving, and documenting data structure designs. They can be used in conjunction with operation specifications to specify abstract data type implementations, or used alone to specify data structures.

Data Structure Diagram Heuristics

Data structure diagrams are easy to read when they are not cluttered. The following heuristics help keep them simple:

Label record fields only once. The labeled record serves as a key to the other records with the same shape and keeps the diagram from being cluttered with lots of text.

Use ellipses to simplify large, repetitive structures. There is no need to draw huge pictures of large arrays or long linked lists.

A few final rules of thumb make it easier to read data structure diagrams:

Draw linked structures so that the pointers point down the page or from left to right. English readers are used to things flowing down and to the right, so it is usually easier for them to follow pointers that flow in these directions.

Identify unusual or additional symbols with a legend. New or different symbols may be introduced into data structure diagrams, but they will be hard to understand without explanation.

Heuristics Summary

Figure 14-3-5 summarizes the data structure diagram heuristics discussed in this section.

- Label record fields only once.
- Use ellipses to simplify large, repetitive structures.
- Draw linked structures so that the pointers point down the page or from left to right.
- Identify unusual or additional symbols with a legend.

Figure 14-3-5 Data Structure Diagram Heuristics

Section Summary

- A **minispec** is a step-by-step description of how an operation transforms its inputs to outputs.
- Minispecs may be written in a programming language or **pseudocode** (English augmented with programming language constructs).
- Data structures may be specified in **data structure diagrams**.

Review Quiz 14.3

1. Why do algorithmic specifications appear in operation specifications?
2. What is a data structure?
3. What are contiguous and linked implementations?

14.4 Design Finalization

Design Phase Completion

Low-level detailed design puts the finishing touches on a design document that guides programmers and testers in coding and checking a software product. Once the design document is complete, the engineering design activity is finished and the design phase of the software life cycle is done.

The last engineering design resolution activity is *design finalization*: checking the design to make sure that it is of sufficient quality and is well documented. In this section we discuss the design finalization task.

Design Document Contents

Let's first review the contents of a design document. Recall that a product's engineering design is documented in a design document with two parts: a software architecture document (SAD) and a detailed design document (DDD). Recall from Chapter 9 that the SAD includes the following parts:

Product Overview—This section summarizes the product vision, stakeholders, target market, assumptions, constraints, and business

requirements. It may simply refer readers to the project mission statement.

Architectural Models—This section specifies the program's major components and their responsibilities, relationships, properties, collaborations, and so forth, using a variety of models that typically includes box-and-line diagrams, class models, interaction models, and state models.

Mapping Between Models—This section includes text and tables explaining the connections between architectural models.

Architectural Design Rationale—This section explains the most important architectural design decisions.

Recall from Chapter 11 that the DDD refines the SAD and should include the following sections:

Mid-Level Design Models—This section specifies the modules comprising the major system components and their responsibilities, properties, relationships, collaborations, and so forth. It typically includes class models, interaction models, and state models.

Low-Level Design Models—This section contains operation specifications and may contain specifications about algorithms and data structures.

Mapping Between Models—This section uses text and tables to connect various detailed design models.

Detailed Design Rationale—This section explains the most important detailed design decisions.

Glossary—This section lists terms and their definitions.

All the design resolution models have a place in the design document. As an illustration, Appendix B contains the SAD and DDD constituting the AquaLush design document.

Design Document Quality Characteristics

During design finalization, designers ensure that the design is adequate and that the design document does a good job documenting the design. Both these goals are accomplished by examining the design document to see whether it has the following quality characteristics:

Feasibility—It must be possible to realize the design. The designers must check the design thoroughly to convince themselves that it can be implemented.

Adequacy—The design must specify a program that will meet its requirements subject to constraints.

Economy—The design must specify a program that can be built on time and within its budget. By the end of the engineering design activity, enough detail is available that developers should be able to make accurate project time and cost estimates.

Changeability—The design should specify a program that can be changed easily, especially in response to anticipated future needs.

Well-Formedness—The design document must use design notations properly.

Completeness—The design document must specify everything that programmers need to implement the product. In practice, this means that the design document should include all required sections, all needed DeSCRIPTR-PAID specifications, and enough detail that programmers do not need to resolve architectural or mid-level design questions on their own.

Clarity—The design document must be as easy to understand as possible given the design's complexity.

Consistency—Every design specification can be realized in a single product. Consistency is especially hard to achieve across levels of abstraction in a design document because it contains highly abstract architectural design specifications, mid-level design specifications of medium abstraction, and low-level design specifications of low abstraction.

Design Reviews

Recall from Chapters 5 and 10 that a **review** is an examination and evaluation of a work product or process by qualified individuals or teams. A **critical review** is an evaluation of a finished product to determine whether it is of acceptable quality. A design finalization review is a critical review.

Designers can check a design document using many review techniques:

Desk Check—An assessment, perhaps using a checklist, by the design's author, who proofreads the design specification.

Walkthrough—An informal examination by a team of reviewers.

Inspection—A formal, checklist-driven design examination by a trained team of inspectors, each playing fixed roles and following a strict process. Inspections are discussed in detail in Chapter 5.

Audit—An examination by experts outside the design team.

Active Review—An examination of parts of a design by experts who answer questions about the design in their areas of expertise. Active design reviews are discussed in detail in Chapter 10.

A Critical Review Process

All these kinds of reviews can and should be used throughout the design process. Desk checks, audits, and active reviews are probably the most effective forms of review during design finalization. One way to combine these forms of review in a comprehensive critical design review is shown in the activity diagram in Figure 14-4-1.

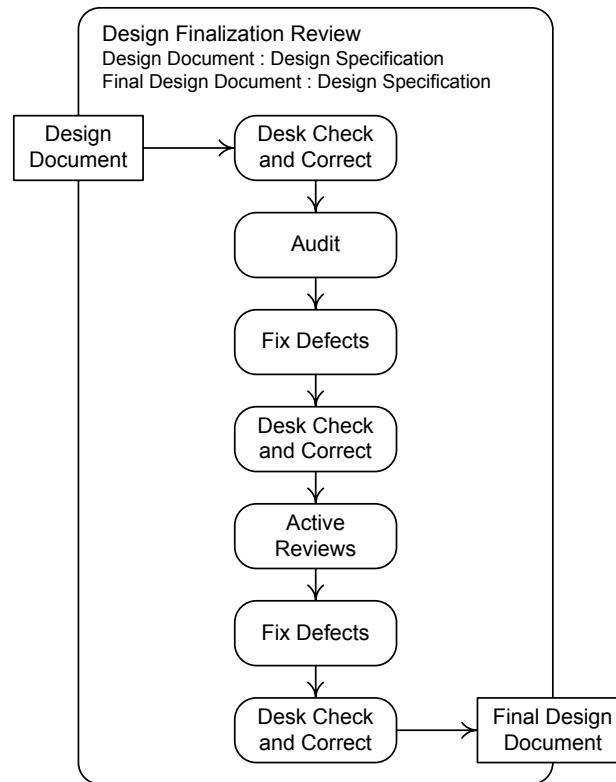


Figure 14-4-1 Design Finalization Review Process

In this process, designers first desk check the design to assess well-formedness, completeness, clarity, and consistency, and fix any problems they find. Next, they perform a design audit to assess feasibility, economy, and adequacy. All defects are corrected. The designers should then once again desk check and correct the design. If the auditors find serious problems in the design, the design document may have to be redone from a much earlier stage of the design process. The project may have to revert to its beginning stages; this flow is left off of the diagram.

Next, active reviews are done for any perceived weak points in the design. Reviewer questions and reactions also provide information to assess clarity, completeness, and consistency. All defects are corrected. Finally, the designers should once more desk check and correct the design. Once again, the design may not survive the review and may have to be largely or completely redone, but this is not shown on the diagram.

Continuous Review	The expense and frustration of carrying a design through to finalization only to have it break down under review argues strongly in favor of conducting reviews throughout the engineering design process.
-------------------	--

Designers should adopt a policy of *continuous review* to mitigate this risk. The most effective kinds of reviews are inspections and active reviews. Both are expensive, but it is cheaper to find errors early in the design process than at its end or, even worse, during coding and testing.

Section Summary

- The engineering design specified in the design document must be *finalized* to ensure that the design is of sufficient quality and is well documented.
- The design document consists of the software architecture document (SAD) and the detailed design document (DDD).
- The design document should specify a design that is feasible, adequate, economical, and changeable; the document should be well formed, complete, clear, and consistent.
- Design finalization reviews are **critical reviews** that evaluate a finished product to see if it is of sufficient quality.
- Continuous reviews during the design process mitigate the risk that major errors found during finalization will force the design to be partly or completely redone.

Review Quiz 14.4

1. What is a design rationale?
 2. What is the quality characteristic of well-formedness?
 3. What is a walkthrough and when should it be used in the design process?
-

Chapter 14 Further Reading

Section 14.1

Fuller discussions of visibility, variables, aliases, and other programming language design topics may be found in [Clarke and Wilson 2001] and [Sebesta 2004]. [Riel 1996] discusses many information-hiding heuristics at length.

Section 14.2

Bertrand Meyer introduced design by contract. He covers these ideas in brief in an article ([Meyer 1992]), and in greater depth in a textbook ([Meyer 1988]).

Section 14.3

Pseudocode and data structure pictures have been used for decades in data structures and algorithms books, though nowadays most books use a particular programming language to specify algorithms. A book that uses both pseudocode and data structure diagrams is [Cormen et al. 2001].

Section 14.4

Inspections are thoroughly discussed in [Gilb and Graham 1993]. See [Parnas and Weiss 1985] for discussion of active design reviews.

Chapter 14 Exercises

Section 14.1

1. *Fill in the blanks:* A _____ is a programming language device for storing values. A variable has several attributes, including its _____, _____, and _____. Values are stored in a _____. A variable's value can also be a _____, which allows access to an entity without using its name.

2. Consider the Java code in Figure 14-E-1. Rewrite this code to decrease variable visibility without changing the algorithms or data structures.

```

public class LockerRoom {
    int numLockers; // in locker room
    int numInUse; // by patrons
    Locker inUse; // linked list of lockers
    Locker crnt; // current locker in list
    Locker last; // one step behind crnt
    public LockerRoom( int roomSize ) {
        numLockers = roomSize;
        numInUse = 0;
        inUse = null;
    }
    // remove a locker from use
    public void returnKey( int key ) {
        if ( key < 0 || numLockers <= key )
            return;
        // delete locker from list
        last = null;
        crnt = inUse;
        while ( crnt != null ) {
            if ( key == crnt.getNumber() ) {
                if ( last == null )
                    inUse = crnt.getNext();
                else
                    last.setNext( crnt.getNext() );
                numInUse--;
                return;
            }
            last = crnt;
            crnt = crnt.getNext();
        }
    } // returnKey
} // end LockerRoom

```

Figure 14-E-1 Java Code for Exercise 2

3. Draw pictures like those in Figures 14-1-2 and 14-1-3 illustrating the variables declared in the following Java code fragments:
 - (a) `int a = 3; int b = 4;`
 - (b) `int a = 3; Integer b = new Integer(4);`
 - (c) `int a = 3; Integer b = new Integer(a);`
 - (d) `Integer a = new Integer(3); Integer b = a;`
4. Does Java support passing operation arguments by reference? Is there a way to make an alias in Java?
5. How can Java package visibility be abused to make a class with package visibility effectively globally visible?

6. Java objects and arrays technically have no names because all object and array variables hold references. Is there a way to create a class without a name in Java?
 7. Consider the following argument: In Java an attribute can be declared `final`, making it a constant; if an attribute is `final`, there is no need to make a defensive copy before returning it as the result of a get operation. Does this argument work?
- Section 14.2**
8. Write pre- and postconditions for the following operations of a standard `Stack` class:
 - (a) `Stack()`
 - (b) `boolean isEmpty()`
 - (c) `push(Object o)`
 - (d) `pop() throws IllegalStateException`
 - (e) `Object top() throws IllegalStateException`
 9. Write pre- and postconditions for the following operations of a `BoundedStack` class whose constructor accepts a maximum size when it is created. Write a class invariant as well.
 - (a) Class invariant
 - (b) `BoundedStack(int maxSize)`
 - (c) `boolean isEmpty()`
 - (d) `boolean isFull()`
 - (e) `push(Object o) throws IllegalStateException`
 - (f) `pop() throws IllegalStateException`
 - (g) `Object top() throws IllegalStateException`
 10. A parking garage control program has a `Garage` class responsible for keeping track of the number of spaces, the number of free spaces, the number of occupied spaces, and whether it is full or not. The `Garage` class has `enter()` and `leave()` operations called when a vehicle enters or leaves. It also has a `setSpaces(int n)` operation used to modify the total number of spaces (in case of construction or some other problem with some spaces) and a `setOccupiedSpaces(int n)` operation (to correct any miscounts). Finally, it has operations to indicate whether the garage is full or empty. Write op-specs for the operations in the `Garage` class. Include class invariants.

Section 14.3

11. Are pseudocode and data structure diagrams static or dynamic modeling notations?
12. Write pseudocode for the following algorithms:
 - (a) Change all the characters of a string to uppercase.
 - (b) Find the maximum and minimum elements of an array.
 - (c) Create a set that is the union of two input sets.
 - (d) Create a set that is the intersection of two input sets.
 - (e) Find the median elements in an unsorted array.
13. Draw data structure diagrams representing the following data structures:
 - (a) A singly-linked circular list

- (b) A doubly-linked non-circular list
 (c) An array implementation of a stack
 (d) A linked implementation of a queue
 (e) An adjacency list representation of a graph
14. A *sparse matrix* is a large two-dimensional array of numbers with very few non-zero elements. It usually saves space to store only the non-zero elements in a linked data structure. Design such a structure and represent it using a data structure diagram.
15. A *trie* is a data structure in which words are stored in a tree whose internal nodes hold letters and whose leaves hold the words. The root holds a null character assumed to begin each word. A path from the root to a leaf traces out the first several letters of the word stored at the leaf. Words with common prefixes share initial paths through the trie. For example “astound” and “aster” would be stored at the leaves of a trie with the null character at the root, a child node of the root storing “a,” a child node of this node storing “s,” a child node of this node storing “t,” and then two child nodes of this node storing either the words or further letters tracing out more characters in each word. (Tries are discussed in detail in data structures books such as [Kruse and Ryba 1999].) Design an implementation of this data structure and an algorithm to search it using pseudocode and a data structure diagram.
16. Design an alternative implementation of the trie data structure described in the last exercise. Document your design with pseudocode and data structure pictures.
- Section 14.4** 17. *Fill in the blanks:* A _____ is an examination and evaluation of a work product or process by qualified individuals or a team. They take many forms, ranging from _____, in which an individual checks his or her own work, to _____, in which a team of outside experts is called in to review something. The two most effective kinds are _____ and _____.
18. Propose an alternative finalization critical review process and explain why you think it would be effective.
- Team Project** 19. Form a small team. Write Java code to illustrate local, package, private, public, and protected visibility in Java.

Chapter 14 Review Quiz Answers

Review Quiz 14.1

- An entity is visible at a point in a program’s text if it may be referred to by name there. An entity is accessible at a point in a program’s text if it may be used there. If a program entity is visible at a point in a program’s text, then it is accessible there, but an entity may be accessible at a point in a program’s text even if it is not visible there (using references or aliases).
- Passing a reference as an argument means that a reference value has been passed into a sub-program. The sub-program can then use this reference to access an entity. Passing an argument by reference means that a sub-program

variable has been given the same address as the argument variable. The sub-program can then access the argument variable using this local variable.

3. The two strategies needed to restrict access to program entities are to limit visibility and not extend access.
4. It is appropriate to extend access beyond visibility when doing so is essential to realize a collaboration or when efficiency or some other overriding concern is judged to be more important than information hiding.

**Review
Quiz 14.2**

1. The class or module and signature fields specify an operation's interface syntax, while the behavior field specifies its interface semantics and pragmatics.
2. A declarative specification does not use an algorithm to describe an operation's behavior, while a procedural specification does.
3. A class invariant is an assertion that must be true of any class instance between calls of its exported operations. Class invariants are in effect additional pre- and postconditions of all public and package operations.

**Review
Quiz 14.3**

1. Algorithmic specifications appear in operation specifications either to provide declarative behavior specifications (which may not be adopted during implementation) or to specify the operation's implementation.
2. A data structure is a scheme for storing values in computer memory.
3. A contiguous implementation is a data structure in which values are stored in adjacent memory cells (as in arrays and records); a linked implementation is a data structure in which values are stored in non-adjacent memory locations and accessed via pointers or references (as in linked lists).

**Review
Quiz 14.4**

1. A design rationale is an explanation of why a design is the way that it is. It usually consists of a discussion of important, surprising, or hard-to-change design decisions. These discussions list design alternatives, evaluations of each one, and an explanation of why a particular alternative was chosen.
2. A design document is well formed if it uses design notations correctly.
3. A walkthrough is an informal examination of work product by a team of reviewers.

Part IV Patterns in Software Design

The fourth part of this book presents a “starter set” of patterns that software engineering designers can use to generate and improve designs. More designs can be added to this basic repertoire through further reading or design experience.

Chapter 15 introduces the study of patterns in software design and presents several important architectural styles.

Chapter 16 introduces the problem of collection iteration and presents the Iterator pattern as a paradigm example of a mid-level design pattern. It also presents the mid-level design pattern classification scheme used in the remaining three chapters.

Chapter 17 presents patterns featuring broker classes that mediate the interaction between client and supplier classes.

Chapter 18 presents patterns that have generator classes that collaborate with producer classes to create producer class instances on behalf of clients.

Chapter 19 presents patterns that have reactor classes that register with invoker classes to provide ongoing reactive services on behalf of clients.



15 Architectural Styles

Chapter Objectives This chapter introduces software design patterns and considers architectural styles, which are patterns for software architecture.

By the end of this chapter you will be able to

- Explain what a software design pattern is and why design patterns are important;
- Explain the basis for recent interest in software design patterns;
- Describe and explain the Layered architectural style and use it in architectural design; and
- Explain the application, form, and consequences of several additional architectural styles, including the Pipe-and-Filter, Shared-Data, Event-Driven, and Model-View-Controller styles.

Chapter Contents 15.1 Patterns in Software Design
15.2 Layered Architectures
15.3 Other Architectural Styles

15.1 Patterns in Software Design

Why Are Patterns Important?

Expert designers usually produce much better designs than novices. What do experts know that novices do not? Among other things, experts have a store of solutions to common problems they have used successfully in the past. Experts draw on this store when they encounter problems similar to those they have solved before. Often, experts don't apply their experience consciously: They simply recognize that a problem is one that can be solved in a way they have used before.

Patterns are important because they capture expert design knowledge and make it accessible to both novices and other experts. Novice designers can benefit from learning solution patterns that experts use, without needing design experience. Expert designers can benefit from studying patterns too: They can broaden their repertoire of patterns and deepen their understanding of the patterns they already know.

Software design patterns provide other benefits:

Promoting Communication—Pattern names are a shorthand for discussing design alternatives among designers. Shared knowledge of pattern advantages, disadvantages, and uses makes discussion and evaluation easier and faster.

Streamlining Documentation—Design documentation that names well-known patterns with a few words about how they are realized can replace pages of explanation about a program’s form and behavior. For example, noting that one module responds to changes in another using the Observer pattern can replace several paragraphs of explanation and justification about how two modules interact.

Increasing Development Efficiency—Languages and components that support or incorporate standard patterns make design and implementation easier. For example, Java libraries provide classes that make implementing certain patterns easier, and many Java components incorporate well-known patterns. This makes it easier and faster to create and implement good designs in Java than in some other languages.

Supporting Software Reuse—Choosing and standardizing patterns for a problem domain promotes software reuse and, hence, quality and productivity. Certain patterns, especially at the architectural level, may prove best for solving problems in particular domains. Standardizing patterns prevents designers from solving the same problems again and again. It also sets the stage for collections of components or frameworks that can be reused in developing new applications.

Providing Design Ideas—Patterns can serve as the starting point for a design or as a model for improving a design. When blocking out a program’s architecture, for example, a designer might decompose the program to fit an architectural pattern with characteristics appropriate for the problem at hand. Or, if a design has a weakness, rearranging it to fit a pattern that addresses that weakness can improve the design.

Software engineering designers have only recently begun exploring design patterns at high levels of abstraction, though low-level patterns have been studied for decades. Older and more established design disciplines have long been aware of the value of design patterns and most have “handbooks” or “pattern books” documenting solutions to standard problems. The fact that software engineering design now has its own pattern books is a sign that the field is maturing. Somewhat surprisingly, software designers were inspired to study design patterns by a building architect.

Christopher Alexander

In the late 1970s, building architect Christopher Alexander introduced a new approach to building design based on patterns. Although building architects have used pattern books for millennia, Alexander’s patterns were different because they were at a higher level of abstraction and were arranged and related to one another to form what Alexander calls a “pattern language.”

Alexander argued that

- Good building design patterns are consequences of human anatomy, psychology, physiology, sociology, and politics and so are things in the world that can be discovered and verified.

- Great architecture has always relied on patterns, but they have never been studied systematically.
- Anyone can make great buildings once the patterns are known and understood.

Alexander's research program, which he has been pursuing for over 30 years, is to discover and document the patterns that make buildings great, teach them to lay people, and encourage them to design their own buildings. In pursuing this goal, Alexander has written a pattern catalog called *A Pattern Language*. This book contains 253 patterns, among which are the following examples:

Four-Story Limit—In an urban area, keep most buildings four stories high or fewer. This will keep occupants in contact with the street, enhance safety, and encourage community.

South-Facing Outdoors—Always build structures north of the outdoor spaces that go with them. Spaces to the north of buildings get little light and tend to be barren, gloomy, and unused. South-facing outdoor spaces are used more and are linked to adjacent indoor spaces.

Warm Colors—Choose light and surface colors to achieve a yellow-red light tone in a room. Studies show that people are more comfortable in yellow-red light than in green-blue light.

Alexander's work has not been widely accepted among architects, and his efforts to get lay people to design great buildings for themselves have usually failed, as he himself admits. Alexander's work has, however, inspired a community of software developers to generate work in design patterns sometimes called *the patterns movement*; this may ultimately turn out to be Alexander's greatest contribution to design.

Patterns in Software

Software engineers began discussing software patterns at conferences in the late 1980s. Journal articles began appearing in 1992, and conferences devoted to design patterns started in 1994. In the early 1990s, software engineers also began to study software architecture as a distinct and important area of research. It soon became clear that there was considerable overlap between the work being done in software patterns, software architecture, some past work in programming language idioms, and even data structures and algorithms.

We begin our discussion of software design patterns with a definition of patterns. A **pattern** is a model proposed for imitation. Applying this characterization to the realm of software design leads to the following definition.

A **software design pattern** is a model proposed for imitation in solving a software design problem.

Models are used at all levels of abstraction in software design, so software design patterns can be at different levels of abstraction as well. We distinguish the following four levels of patterns:

Architectural Styles—These are high-level models for entire programs and sub-systems. **Architectural styles** describe kinds of architectural components and the interactions between them. These patterns come into play during architectural design.

Mid-Level Design Patterns—Usually called just *design patterns*, **mid-level design patterns** are at an intermediate level of abstraction and model collaborations between several modules (typically classes). Patterns at this level of abstraction have been the focus of the design patterns movement. As their name suggests, mid-level design patterns provide guidance during mid-level design.

Data Structures and Algorithms—**Data structures** and **algorithms** are patterns for storing, retrieving, and manipulating data. Well-known examples of data structures include contiguous lists, linked lists, hash tables, and binary search trees; well-known examples of algorithms (besides those involved in implementing data structures) include searching algorithms, such as binary search, and sorting algorithms, such as quicksort, merge sort, and heapsort. Although usually treated as a distinct topic in computer science, the study of data structures and algorithms is really a branch of software engineering design. These patterns are most useful in designing single modules or classes, so they provide guidance during low-level design. Data structures and algorithms have been studied since the earliest days of computing, and theory and practice in this area is firmly established. This topic is thoroughly dealt with in specialized texts and not covered in this book.

Programming Idioms—A **programming idiom** models a standard way to do something in a particular programming language. Idioms are useful mainly during programming, but a few idioms are sufficiently general to guide low-level design. Many books discuss idioms for particular programming languages. Further discussion of programming idioms is beyond the scope of this book.

The other sections of this chapter present several architectural styles, and the remaining four chapters of the book discuss many mid-level design patterns. The architectural styles and mid-level patterns in this part of the book form a “starter set” of design patterns that every software designer should know.

Design Pattern Catalogs

Realization of the importance of design patterns has spurred creation of pattern catalogs. These are much like the pattern books used in building architecture or interior design and the handbooks used in engineering.

The documentation for a design pattern in a software patterns book consists of at least the following items:

Name—The name of the pattern.

Application—When to apply the pattern. Often the application of a pattern is explained in terms of the problem solved by the pattern and contexts where the problem arises.

Form—The (static) structure and (dynamic) behavior of the pattern. Structure may be described in text supplemented with one or more static models and behavior may be described in text with one or more dynamic models.

Consequences—The advantages and disadvantages of using the pattern.

Some pattern catalogs supply additional information, such as other names for the pattern, implementation details, examples of its use, known uses, and related patterns.

Section Summary

- Knowing design patterns confers many advantages, including providing familiarity with effective design alternatives, promoting communication, streamlining documentation, increasing developer efficiency, and supporting software reuse.
- Christopher Alexander, a building architect, inspired software designers to discover and document patterns.
- A **software design pattern** is a model proposed for imitation in solving a software design problem.
- Design patterns occur at different levels of abstraction with special names: **architectural styles**, **mid-level design patterns**, **algorithms** and **data structures**, and **programming idioms**.
- Software pattern catalogs collect and describe patterns.

Review Quiz 15.1

1. How can design patterns streamline design documentation?
2. When did the patterns movement in software engineering begin?
3. What information beyond a pattern's name, application, form, and consequences might appear in a pattern description?

15.2 Layered Architectures

Architectural Layers

Perhaps the most widely used of all architectural styles is the **Layered architectural style**. A Layered-style program is divided into an array of modules or layers. Each layer provides services to the layer “above” and makes use of services provided by the layer “below.”

Every layer is a module that provides a cohesive set of services and has a well-defined interface. Every program unit must be in exactly one layer; in other words, the layers partition the set of program units. The static structure of the Layered style is to partition software units into modular layers.

The dynamic structure of the Layered style is nothing more than a constraint on interactions between layers. Each layer can be constrained to use *only* the layer directly below it—this is a *Strict Layered style*. Sometimes this constraint is relaxed slightly to allow each layer to use all the layers below it—this is a *Relaxed Layered style*.

It is crucial to understand the communications that the Layered style allows and prohibits. The key concept is the difference between the using and calling relationships.

Module *A* **uses** Module *B* if a correct version of *B* must be present for *A* to execute correctly.

Module *A* **calls** (or **invokes**) module *B* if *A* triggers execution of *B*.

A module may use another without ever calling it. For example, Module *A* may read a file that Module *B* is supposed to have written, or *A* may use a device that *B* is supposed to have initialized. On the other hand, Module *A* may call Module *B* without using it. For example, Module *A* may call Module *B* (an observer) to notify it that *A* has changed. It does not matter to *A* whether *B* is correct, so *A* does not use *B*.

In a Layered style, each layer is allowed to use only the layers below it—either the single layer directly below it (in the Strict Layered style), or all the layers below it (in the Relaxed Layered style). In other words, each layer is allowed to depend on the layer below it being present and correct. A layer may *call* other layers above and below it, as long as it does not *use* them.

To summarize, the Layered architectural style has the following form:

Static Structure—The software is partitioned into layers that provide a cohesive set of services with a well-defined interface.

Dynamic Structure—Each layer is allowed to use only the layer directly below it (Strict Layered style) or all the layers below it (Relaxed Layered style).

Representing Layers

The box-and-line diagrams in Figure 15-2-1 show two common ways of depicting Layered-style architectures.

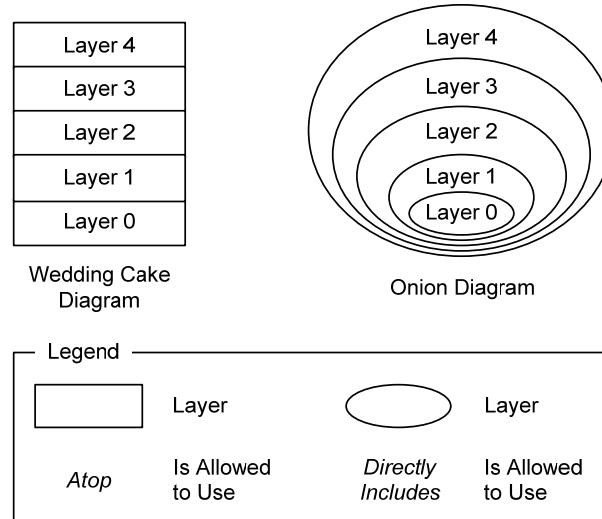


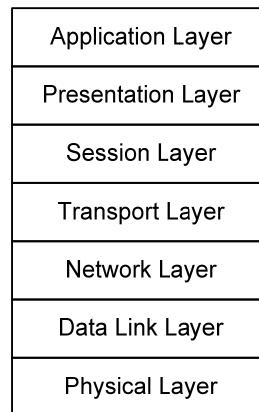
Figure 15-2-1 Box-and-Line Diagrams of the Layered Style

The *wedding cake diagram* on the left is typically used to show the connection between the layers in communications protocols, such as the International Standards Organization's Open Systems Interconnection (ISO OSI) model, or the layers in user interface and windowing systems, such as the X Window System. The *onion diagram* on the right often illustrates operating system layers, with the kernel at the core.

A Relaxed Layered-style diagram extends boxes downwards, or circles inwards, to show uses relationships between non-adjacent layers.

These diagrams show the layers in an architectural decomposition and the layers each layer is allowed to use. However, they do not display layer responsibilities, interfaces, properties, relationships, states and transitions, or collaboration details, all of which must be documented in other ways.

Forming Layers The components in a Layered-style program often implement services at different levels of abstraction. For example, network protocols are usually designed and implemented in layers that provide communications services at various levels of abstraction. The ISO OSI Reference Model has seven layers, as shown in the diagram in Figure 15-2-2. The bottom or **Physical Layer**, at the lowest level of abstraction, is responsible for transmitting bits over a channel. The layer above it, the **Data Link Layer**, uses the **Physical Layer** and is responsible for detecting and correcting transmission errors. The **Data Link Layer** provides an error-free channel to the layer above it, the **Network Layer**. In like fashion, each layer has responsibilities at higher levels of abstraction. At the top level is the **Application Layer**, where the services provided by the entire layered sub-system interface with the clients that use them.

**Figure 15-2-2 ISO OSI Reference Model**

Other programs are layered for reasons besides providing levels of abstraction. For example, many architects place user interface and application domain code in different layers. This is done not to separate levels of abstraction but to make the user interface easy to change without disturbing the core functionality of the program. Layers are often formed to increase changeability and reusability.

When to Use the Layered Style

There are many excellent reasons for adopting a Layered architectural style, principally due to the fact that layers make good modules. In particular, layers have the following characteristics:

- Layers are by definition highly cohesive, thus satisfying the Principle of Cohesion.
- Layers support information hiding.
- Layers are constrained to use only lower layers. This means that layers are not strongly coupled to the layers above them. If a program uses the Strict Layered style, then each layer is strongly coupled only to the layer immediately below it. As a result, overall Layered-style architectures are loosely coupled.
- Layering helps sub-divide complex portions of a program into separate modules, which simplifies the program.

One consequence of these characteristics is that Layered-style programs are easy to modify because changes can be made to a layer independently of the rest of the program. In fact, entire layers can be replaced without altering other parts of the program. Layered architectures are highly changeable, so this pattern can be used when changeability is an important quality attribute.

One kind of change handled especially well by Layered systems is porting to new platforms. Typically, only the bottom layer must be replaced or rewritten to port a Layered program to a new platform. This is

demonstrated by the AquaLush architecture, which uses a device interface layer to accommodate hardware changes.

The form of Layered architectures, both static and dynamic, is very simple. Components form a linear static structure, and their collaborations are highly constrained. This simplicity of form and behavior increases reliability and maintainability.

Layered programs encourage reuse because layers, being independent modules with well-defined interfaces, should be easily reusable.

These many advantages help explain why Layered architectures are so widely used. But there are a few drawbacks:

- It is often necessary to pass data through many layers, which can slow performance significantly. Some programs alleviate this problem by moving from a Strict to a Relaxed Layered style, but doing so may lose some of the previously listed advantages; for example, coupling increases, simplicity decreases, and information is less well hidden.
- Multi-layered programs can be hard to debug because operations tend to be implemented through a series of calls across layers.
- It may be hard to get the layers right. If there are too few layers, they may not isolate changes, may be too big and complicated, and may not be replaceable and reusable. If there are too many layers, they may lack cohesion, may increase collaboration across layers, and may damage performance as data is processed and passed through many layers.

Despite these disadvantages, the Layered architectural style is still extremely useful and valuable. In particular, designers should use a Layered style to increase changeability, maintainability, reliability, and reusability, but perhaps not if performance is a major concern. As a rule, if no other architectural style seems right for a given problem, try a Layered style.

A Layerd-Style Example

A *citation management system* records bibliographic citations, searches them, and produces lists of references in a variety of bibliographic styles. Such programs must have user interfaces that are easily changed. They must also be able to incorporate new citation styles easily.

The need to make the user interface easily changeable suggests that it should be a separate architectural component. Also, the part of the program that formats citations should be separate from the part that stores, searches, and retrieves data because these latter functions are fairly stable. This suggests three major program components: a user interface module, a citation formatter module, and a citation storage and retrieval module. Notice that these components partition the program and each has high cohesion.

Clearly the user interface module must use the others, and the citation formatter needs to use the storage and retrieval module. If we broaden the responsibilities of the citation formatter to include citation management, then we have a Strict Layered architecture, pictured in Figure 15-2-3.

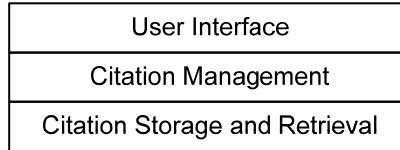


Figure 15-2-3 Citation Management System Layers

The User Interface layer is responsible for user interaction. It uses the Citation Management module to store, retrieve, and format citations. The Citation Management module is responsible for keeping track of all citations and formatting them as required. When it needs to store, search, or retrieve citations it does so using the Citation Storage and Retrieval module. The Citation Storage and Retrieval module uses some sort of persistent storage mechanism—it may be a database or information retrieval system—to hold citation records.

The citation records have fields for every kind of citation and do not change often. Also, the storage and retrieval functionality of the Citation Storage and Retrieval module is quite stable, so this module does not often change. The Citation Management module changes when new formatting styles are added to the program, but this does not affect the Citation Storage and Retrieval module and affects the User Interface module only to the extent that support for the new styles must be added to the user interface. Finally, the User Interface module can change without any effect on the other two modules.

Style Summary

Figure 15-2-4 summarizes the Layered architectural style.

Name: Layered Application: Structure a program into an array of cohesive modules with well-defined interfaces to realize levels of abstraction, increase changeability, and increase reusability. Form: The program is partitioned into cohesive modules with well-defined interfaces arranged in a chain or sequence from highest to lowest: the layers. Each layer is allowed to use only the layer immediately below it (Strict Layered style), or all the layers below it (Relaxed Layered style). Consequences: Layers should be cohesive, hide information, be simple, and be coupled only to the layer or layers beneath them. All these characteristics make layers easy to alter or replace, improving changeability. The simplicity of a Layered architecture increases reliability and maintainability. Layers are likely to form reusable components. Layered programs may be hard to debug; program behavior must often be realized with communications across several layers, which may cause performance problems and be more work to program. It may be hard to form a good collection of layers.

Figure 15-2-4 The Layered Architectural Style

Section Summary

- The single most widely used architectural style is the **Layered architectural style**.
- The static structure of the Layered style is that the program is partitioned into modules that form a linear array of **layers**, each of which has high cohesion and a well-defined interface.
- The Layered style has a dynamic constraint: Each layer may use only the layer directly beneath it (the Strict Layered style) or all layers beneath it (the Relaxed Layered style).
- Layers are formed to realize levels of abstraction or for other reasons, such as increased reliability and greater changeability.
- Layers should be simple modules with high cohesion and low coupling that hide information well.
- Layers may decrease efficiency and make debugging more difficult; it may be hard to form good layers.
- The Layered style increases changeability (especially portability), maintainability, reliability, and reusability, but it may damage performance.

Review Quiz 15.2

1. What sorts of pictures are typically made of Layered architectures?
 2. What constraints are imposed on the collaboration between layers in a Layered architecture?
 3. Explain how the Layered style supports reuse.
 4. Explain why Layered-style programs can be hard to debug.
-

15.3 Other Architectural Styles

Styles Covered

This section surveys several well-known architectural styles, including the Pipe-and-Filter, Shared-Data, Event-Driven, and Model-View-Controller styles. There are several other styles; this collection is a “starter set” that you can add to over time.

Pipe-and-Filter Style

A **filter** is a program component that transforms a stream of input data to a stream of output data. A **pipe** is a conduit over which a stream of data flows. The **Pipe-and-Filter architectural style** is a dynamic model in which program components are filters connected by pipes. This style does not model the static form of a program.

A classic example of a Pipe-and-Filter program is a compiler. A compiler reads a stream of characters comprising the source code of a program and produces a stream of machine-language instructions comprising the object code of the compiled program. The processing that occurs in between is shown in the box-and-line diagram in Figure 15-3-1.

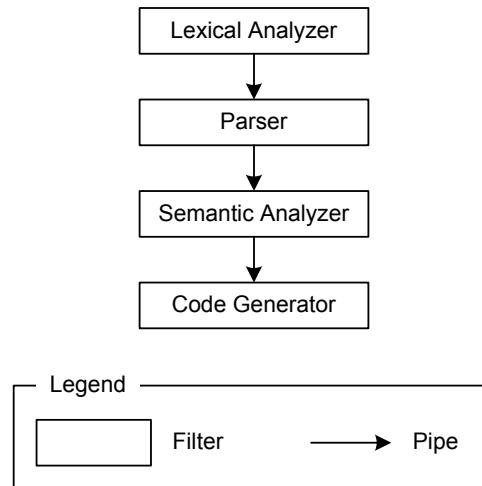


Figure 15-3-1 Pipe-and-Filter-Style Compiler

The Lexical Analyzer filter reads a stream of characters from a source code file and transforms it into a stream of *tokens*, which are significant units in the programming language. These tokens are sent down a pipe to the Parser filter, which transforms the tokens into a *syntax tree* representing the declarations and statements in the program. The syntax tree is sent down a pipe to the Semantic Analyzer filter, which processes the syntax tree by doing type checking and adding annotations to it. The annotated syntax tree is then sent through a pipe to the Code Generator filter, which transforms it into a stream of machine-language instructions written to an object code file.

The Pipe-and-Filter style is used for programs that transform input streams into output streams, as in this example. This style does not fit programs that interact with people or react to circumstances in their environment.

Characteristics of Filters and Pipes

Filters read input and produce output by enriching, refining, or converting it to another format. *Enriching* the input means adding information to it, *refining* the input means compressing it or extracting information from it, and *converting* the input means changing its format. In the compiler example from Figure 15-3-1, the Semantic Analyzer enriches a syntax tree by adding annotations to it, and the Lexical Analyzer converts a stream of characters into a stream of tokens.

Filters generally do not have externally visible states and do not communicate with other components in a program. Instead, they merely transform input data to output data. This makes it easy to write and debug filters because they can be treated as stand-alone programs.

A filter may begin to produce output before it has consumed all its input, though sometimes it may need to process all its input before it can produce any output. Because filters are independent of each other and because they

may produce output before consuming all their input, they may execute concurrently, which can increase program performance.

If filters run concurrently, pipes need to synchronize them. A pipe may be a buffer that holds its input filter's output until its output filter is ready to accept it. Sometimes, however, a pipe is implemented simply as a sub-program call. In either case, filters need to block if they are ready for input but their input pipes are empty, or if they are ready for output but their output pipes are full.

Pipe-and-Filter Topologies

Pipes and filters can be fit together in all sorts of ways, but pipes that double back to previous filters have complexities involving timing and deadlock. Consequently it is best to restrict the shape of a Pipe-and-Filter architecture to an *acyclic graph*, or a form that does not include any loops.

A simple linear arrangement of pipes and filters is called a *pipeline*. Pipelines suffice for many applications, but occasionally filters that split or merge data streams are needed. For example, suppose a program needs to convert a file listing employees, their employee numbers, and their salaries into a file listing the employees, their supervisors, and the difference between their salaries and the mean company salary. One design for this job is shown in Figure 15-3-2.

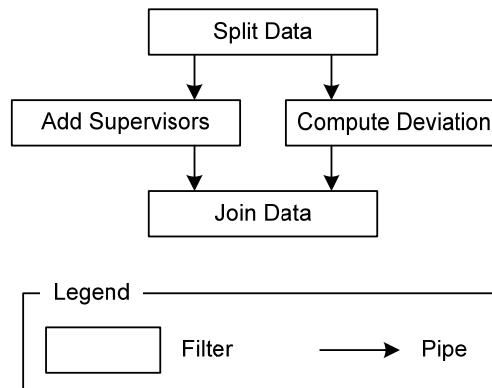


Figure 15-3-2 Splitting and Joining Data Streams

The **Split Data** filter reads the input data file, pulls out the name and identifier data, and sends it down a pipe to the **Add Supervisors** filter. It also pulls out the salary data and sends it to the **Compute Deviations** filter. The **Add Supervisors** filter replaces each employee number with that employee's supervisor and sends the result down a pipe to **Join Data**. The **Compute Deviations** filter calculates the difference between each salary and the mean company salary and sends the result down a pipe to the **Join Data** filter. Finally, the **Join Data** filter puts the two data streams together and prints the resulting report.

Advantage and Disadvantages

The Pipe-and-Filter style has the following advantages:

- Filters can be modified or replaced easily, making it simple to change the program to fix a problem or modify its behavior.
- Filters can be rearranged with little effort, making it easy to develop several programs that do similar tasks.
- Filters are highly reusable.
- Concurrency is supported and is relatively easy to implement, provided synchronizing pipes are available.

This style has the following disadvantages:

- Filters communicate only through pipes, making it difficult for them to coordinate their activities.
- Filters usually consume and produce very simple data streams, such as streams of characters, which means that they may have to spend a lot of effort converting input into a usable form and then converting results back to a simple format for output.
- Error handling is difficult; error information can only be output or passed along a pipe. The difficulty of error detection and recovery makes this style inappropriate when reliability and safety are important.
- Gains from concurrency may be illusory. Pipes may not synchronize filters effectively, and some filters may need to wait for all their input before doing any output.

Pipes and filters are used in the UNIX shell, which provides an excellent illustration of these advantages and disadvantages. The shell includes many filter programs (such as `grep`, `sort`, `uniq`, and `cut`), and it provides pipes as well as programming languages for writing Pipe-and-Filter programs. It is easy to write new filters for special jobs. The filters are extremely reusable and are easy to connect together with pipes to accomplish innumerable tasks. The filters also run concurrently, which is sometimes very efficient.

However, it is difficult to coordinate UNIX filters and to use them for tasks other than text processing because they all read and write streams of characters. The only way to handle errors is to issue error messages on the `stderr` output stream. Also, the fact that filters run concurrently is less important if a filter like `sort` is used because it must wait for all its input before it can do its job.

In summary, the Pipe-and-Filter style is an excellent model for component behavior when the program as a whole transforms a stream of input into a stream of output. It provides for rapid development, reusability, and good performance, but it does not promote reliability and safety.

Shared-Data Style

The **Shared-Data architectural style** is characterized by one or more *shared-data stores* used by one or more *shared-data accessors*. The shared-data accessors store, delete, and modify data in the shared-data stores. The accessors communicate solely through the shared-data stores. The way accessors are activated provides two variants of the style:

- The shared-data stores activate the accessors when the shared-data stores change. This variant is called the **Blackboard architectural style**, and the shared-data stores are called *blackboards*.
- The shared-data stores are passive and are queried by the accessors, which may run continuously or be controlled by some other component. This variant is called the **Repository architectural style**, and the shared-data stores are called *repositories*.

An example where this style might be used is a Software Development Environment (SDE). SDEs have many tools supporting developers, such as programs for planning and scheduling, design, editing, testing, and compilation. These tools typically work off the same data. For example, a class diagram editor and a code editor should both display the same classes, attributes, and operations, and changes to these entities in one editor should be reflected in the others. One way to design such a system is to have a shared-data store holding all project data that each tool, acting as a shared-data accessor, reads and writes in the course of its operation. This architecture is illustrated in Figure 15-3-3.

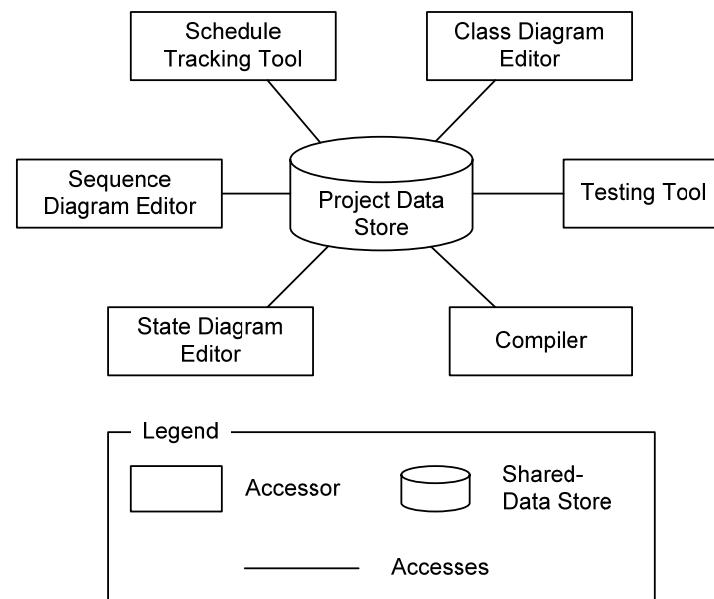


Figure 15-3-3 Shared-Data-Style SDE

Advantages and Disadvantages

The Shared-Data architectural style has the following advantages:

- Shared-data accessors communicate only through the shared-data store, so they can be changed independently, replaced, or deleted, and new ones can be added. This increases maintainability and changeability.
- The independence of shared-data accessors increases program robustness and fault tolerance.
- Placing all data in the shared-data store makes it easier to secure the data and ensure its quality.

The Shared-Data style has the following disadvantages:

- Forcing all communication through the shared-data store may degrade performance.
- If the shared-data store fails, the entire program is crippled; thus, it may be a source of unreliability.

In summary, the Shared-Data style is appropriate when several independent program components must access common persistent data.

Event-Driven Style

The **Event-Driven architectural style**, also called the **Implicit-Invocation architectural style**, has a special *event dispatcher* that mediates between components that announce and are notified of events. An **event** is an important occurrence that happens at a particular time. Components register interest in events by requesting that the event dispatcher notify them when the events occur. Components announce events to the event dispatcher, which then notifies all components registered for those events.

The event dispatcher is often built into a system's infrastructure. For example, in Visual Basic, programmers write code within skeletons associated with user interface widgets, such as buttons or scrollbars. When these user interface widgets are used (for instance, when a button is pressed), the associated code is executed. Visual Basic thus has a built-in event dispatcher that programmers can assume will dispatch events.

As an illustration of this style, consider a building monitoring system consisting of many sensors scattered around a building. The sensors might be smoke detectors, motion sensors, or door and window sensors. All detectors register events with an event dispatcher. Monitoring programs may register with the event dispatcher to check on fire safety, building security, and environmental safety. The diagram in Figure 15-3-4 illustrates this architecture.

The Event-Driven style dictates a dynamic model for program design, but not a static model. However, the event dispatcher is usually a distinct module, and as noted earlier, it is often built into the program's infrastructure. In either case, it is separate from the other modules in the program.

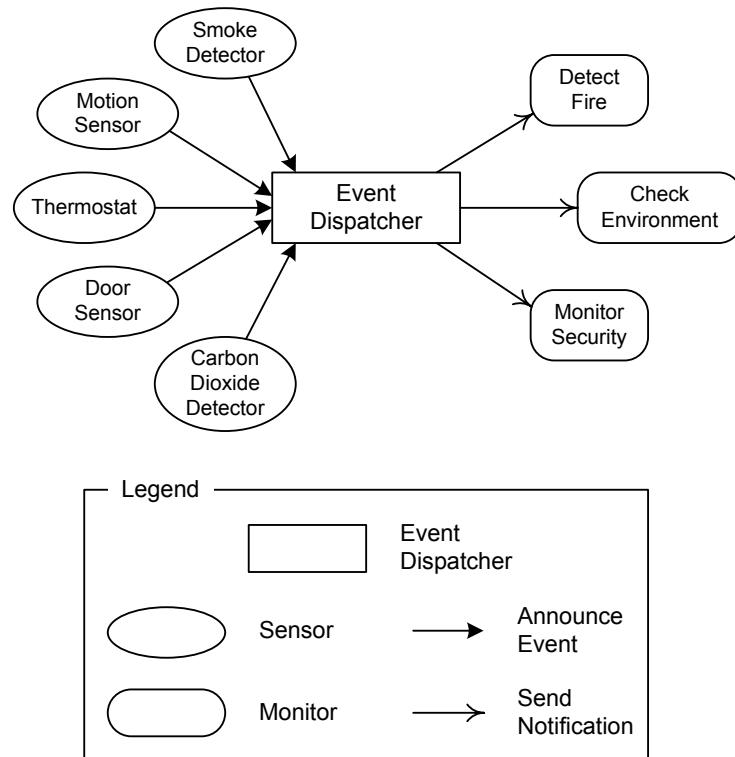


Figure 15-3-4 An Event-Driven Program

Stylistic Variations Event-Driven systems may vary in many ways:

- Events may be simple notifications, or they may carry packets of data from the event announcer to the event responders. In particular, an event may identify its originator, allowing event responders to interact with the event announcer directly.
- Events may have priorities or timing constraints honored by the event dispatcher, or the event dispatcher may manipulate events. For example, an event dispatcher may drop a long series of queued mouse events and dispatch only the last one. This allows responding components to catch up with a user who is doing things rapidly with a mouse.
- Events may be dispatched *synchronously* or *asynchronously*. In other words, the dispatcher may wait for the operations it invokes to return (synchronous dispatch) or, using threads or processes, execute them without waiting for them to return (asynchronous dispatch).
- Registering interest in events may be constrained in various ways. For example, components may be allowed to register and unregister with the event dispatcher arbitrarily during execution, or registrations may have to be set up when the program is coded.

Advantages and Disadvantages The components that announce events have no connection whatsoever to the components that respond to them. Hence, this style completely decouples components announcing events from components responding to them. Furthermore, event announcers are independent of one another, as are event responders. As a result, there are several advantages to using this style:

- It is easy to add, remove, or change components, so programs written in this style are changeable and maintainable.
- The independence of program components supports reusability, robustness, and fault tolerance.

The drawbacks of this style result from component independence and the unpredictability of events:

- Although events may carry data, component interaction is awkward when mediated by the event handler. Event-driven systems usually also support explicit operation invocation to solve this problem, though this couples components.
- Components that announce events cannot guarantee that any components respond to them or have any expectations about the order in which components respond to them. This sometimes makes it difficult to write correct programs.
- Event traffic tends to be highly variable: Often the event dispatcher is idle, and at other times it is swamped with events. This may make it difficult to achieve performance goals.

The Event-Driven style is a good choice for a program that must react to unpredictable events in its environment. Programs with complex graphical user interfaces are an excellent example in this regard. It is impossible to tell what a user may choose to do next, so the program must be ready to react to a myriad of possible inputs. This is the reason most graphical user interface code uses this style and many user interface development frameworks have implicit event dispatchers.

Model-View-Controller Style

The **Model-View-Controller (MVC) architectural style** is a model for setting up the relationship between a user interface and the domain-specific portion of a program. MVC divides a program's components into three categories:

Models—The portions of the program that realize problem-domain function. The models hold data and operations for achieving the computational goals of the program independent of its user interface.

Views—Components that display data to users. The data displayed in views comes entirely from one or more models. Labels and graphics are examples of views.

Controllers—Components that receive and carry out commands from users. Controllers may alter views or models. Buttons and scrollbars are examples of controllers.

The views and controllers together comprise a program's user interface; models are entirely separate from the user interface.

- MVC Static Structure** The static structure of the MVC style uses the Layered style: A program is partitioned into a user interface module, consisting of views and controllers, and an application domain module, which contains all the models. The Layered-style dynamic constraint that a higher layer is allowed to use only the layer directly beneath it is also satisfied in the MVC style. Views and controllers are allowed to use models, but models do not use controllers or views. The structure of the MVC style is shown in the UML class diagram in Figure 15-3-5.

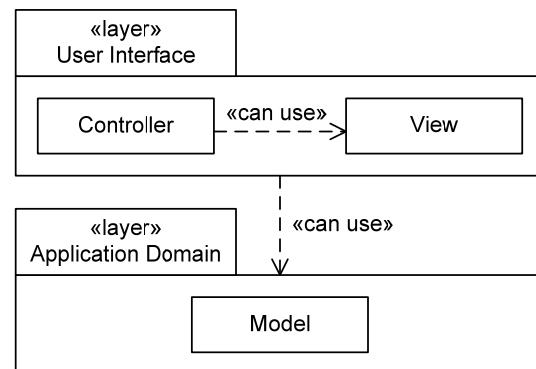


Figure 15-3-5 MVC-Style Static Structure

In this figure, the packages represent program layers. The **User Interface** layer includes one or more controllers and views, represented here by **Controller** and **View** classes, and the **Application Domain** layer contains one or more models, represented here by the **Model** class. Controllers can use Views, and both Controllers and Views can use Models.

- MVC Dynamic Structure** The dynamic structure of the MVC style is fairly simple. The user only manipulates controllers. A controller may alter a view or a model. If a view is altered by a controller, it redisplays itself, perhaps interrogating one or more models to obtain the data it needs for display.
- If a controller alters a model, the model responds by doing whatever computations are required. Once it is done, the model issues a notification that it has changed. This may be done by announcing an event or by directly notifying controllers and views. In either case, the model issues an event or notification only, and does not depend on any response. As a result, the model does not use any views or controllers and it is not strongly coupled to them. Views and controllers that receive the model's notification can reconfigure or redisplay themselves, interrogating models as necessary to obtain data. The sequence diagram in Figure 15-3-6 illustrates this interaction in a schematic example with a single view,

controller, and model. The model notifies the view and controller of changes by calling a notification operation rather than announcing events.

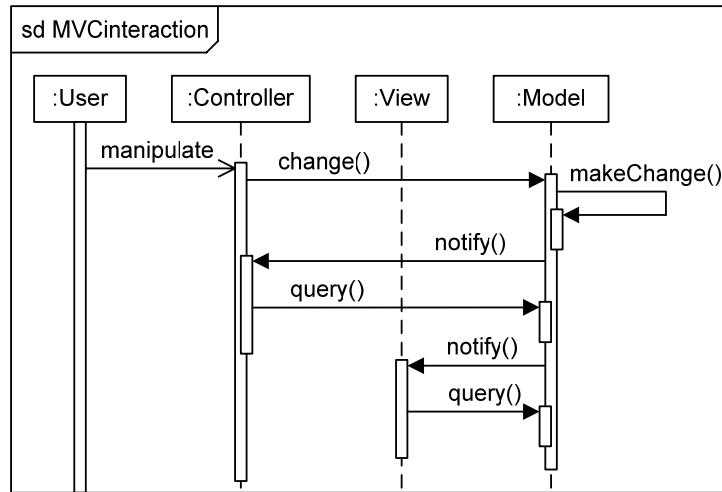


Figure 15-3-6 MVC-Style Dynamic Structure

Figure 15-3-6 shows a paradigm interaction in which a **User** manipulates a **Controller** object. The **Controller** changes a **Model** instance. The **Model** makes the indicated change and then notifies the **Controller** and a **View** object. Each queries the **Model** for data to update itself. A similar course of events ensues if the model changes in response to some application-domain event, such as the passing of time.

Advantages and Disadvantages

The MVC style is very widely used because it has the following advantages:

- Views and controllers can be added, removed, and changed without disturbing the model. The user interface components are almost completely decoupled from the application-domain components, making it easy to change the user interface. Because user interfaces are among the most volatile portions of programs, this greatly increases program changeability and portability.
- Views can be added or changed, even during execution. Separating views from controllers makes it is easy to add, remove, or change views without affecting the way the user interacts with the program. This provides view flexibility and configurability without confusing users.
- The components of the user interface can be changed, even at runtime. The look and feel of the program can be changed by replacing views and controllers.

However, there are also some disadvantages to using the MVC style:

- Views and controllers are often hard to separate. For example, a text box is typically both a view *and* a controller component. This makes it harder to change views and controllers independently, but does not

affect the separation of program layers, so it is a relatively minor problem. A single user interface component plays both view and controller roles.

- Frequent updates may slow data display and degrade user interface performance. When a model changes, it simply notifies the views and controllers that it has changed. It is up to them to figure out how the model has changed and update themselves. This may require many query operations from the views and controllers to the model, slowing user interface update operations.
- The MVC style makes user interface components highly dependent on model components. Although the model does not know anything about views and controllers, the views and controllers depend intimately on the models. Hence, if the models change, the views and controllers usually must be changed too.

Despite some drawbacks, the MVC style is an excellent and widely used style for designing the connection between the user interface and the rest of a program.

Heterogeneous Architectures

Several architectural styles may appear in the same program, either in different sub-systems or in combination. For example, a program using the MVC style to realize user interface and application domain layers may also use the Event-Driven style to model the dynamics of controller and view notification when models change. An architecture that employs two or more architectural styles is called a **heterogeneous architecture**.

Style Summaries

Figures 15-3-7 through 15-3-10 summarize the architectural styles discussed in this section.

<p>Name: Pipe-and-Filter</p> <p>Application: Model the collaboration of components in programs that transform input streams to output streams.</p> <p>Form: This style does not specify a static form. Its dynamic form is a collection of data stream transformers (filters) connected to one another by data stream conduits (pipes). Filters are independent components that are simple to construct and highly reusable. They may run in parallel, synchronized by pipes.</p> <p>Consequences: Pipe-and-Filter-style systems are easy to construct, have reusable components, are easy to change, and may have good performance thanks to concurrent filters. However, filters are hard to coordinate and have difficulty handling errors cooperatively (degrading reliability and safety). In addition, performance gains may not materialize if the filters are not well synchronized or do not lend themselves to concurrent processing.</p>

Figure 15-3-7 Pipe-and-Filter Architectural Style

Name: Shared-Data
Application: Model several independent program components using persistent data.
Form: This style does not specify a static form. Its dynamic form is one or more shared-data stores accessed by one or more shared-data accessors. The shared-data accessors do not directly interact. The accessors are activated by the shared-data stores when they change (the Blackboard style), or the shared-data stores are passive (the Repository style).
Consequences: It is easy to change, replace, remove, or add shared-data accessors in Shared-Data-style programs, making them changeable, maintainable, robust, and fault tolerant. The shared-data stores contribute to data security and quality. However, they are a single point of failure, which may decrease reliability.

Figure 15-3-8 Shared-Data Architectural Style

Name: Event-Driven or Implicit-Invocation
Application: Model a program that must react to unpredictable sequences of inputs.
Form: This style does not specify a static form. In its dynamic form, one or more event announcers send events to an event dispatcher. When an event occurs, the event dispatcher invokes procedures of components that have registered interest in the event.
Consequences: Components that announce events are independent of other components that announce events and of components that respond to them, which are themselves independent of one another. Hence, components are easy to change, replace, remove, or add, making Event-Driven-style programs changeable, maintainable, robust, and fault tolerant.
Interacting entirely through the event dispatcher is awkward. The independence of components may make it hard to establish program correctness. Program performance may degrade when many events occur in quick succession.

Figure 15-3-9 Event-Driven Architectural Style

Name: Model-View-Controller (MVC)

Application: Model a program with interactive user interfaces.

Form: This style is a specialization of the Strict Layered style. The top user interface layer contains view modules for data display and controller modules for user input. The bottom application domain layer contains model modules embodying the data and operations implementing core program function. Controllers may change views or models, and models may change on their own. When a model changes, it notifies views and controllers, which respond by querying the model and updating themselves.

Consequences: The user interface is decoupled from the models, so it may be changed independently, increasing changeability and maintainability. Views and controllers may be changed even at runtime, increasing flexibility and configurability. However, views and controllers are highly dependent on models, decreasing application domain layer changeability and maintainability. User interface update performance may be a problem.

Figure 15-3-10 Model-View-Controller Architectural Style

Section Summary

- The **Pipe-and-Filter** style has data-transforming components (**filters**) connected by data-stream conduits (**pipes**).
- Pipe-and-Filter-style programs are non-interactive programs that transform input streams to output streams, often through a number of intermediate data-transforming steps.
- The **Shared-Data** style has one or more shared-data stores used by independent shared-data accessors.
- Shared-Data-style programs tend to have persistent data used by several program components that do not need to communicate directly.
- The **Event-Driven** or **Implicit-Invocation** style has an event dispatcher that invokes operations of components interested in particular events when those events are announced.
- The Event-Driven style decouples event announcers and responders and is a good choice for programs that must be changeable and react to unpredictable events.
- The **Model-View-Controller (MVC)** style divides program components into controllers, views, and models, with the former in a user interface layer and the latter (models) in an application domain layer.
- The MVC style should be used to separate an interactive user interface from the remainder of a program, thus increasing user interface flexibility, changeability, and configurability.
- Architectures are often **heterogeneous**, meaning they are combinations of several architectural styles.

**Review
Quiz 15.3**

1. What is a pipeline?
2. What is the difference between the Blackboard and Repository styles?

3. Can an event in an Event-Driven architecture hold data?
 4. How does the MVC style insulate the user interface from changes in the application domain?
-

Chapter 15 Further Reading

- Section 15.1** Christopher Alexander's books ([Alexander et al. 1977] and [Alexander 1979]) are very interesting reading on their own, as well as being the inspiration for the software patterns movement. The "Gang-of-Four" book [Gamma et al. 1995] was the first mid-level design patterns book, and it is still the best place to find the most important mid-level design patterns. The *Pattern Languages of Program Design* conference began in 1994 and has occurred every two years since then. The proceedings of this conference are a good source for lesser-known design patterns. Architectural styles are discussed extensively in [Shaw and Garlan 1996] and [Buschmann et al. 1996]. There are numerous data structure and algorithm textbooks on the market, including many that use Java as the implementation language, such as [Standish 1998]. Excellent programming idioms books include [Bloch 2001] and [Coplien 1992].
- Section 15.2** The Layered architectural style is discussed in [Shaw and Garlan 1996], and extensively in [Buschmann et al. 1996]. Discussion of the uses relation and its importance for the Layered style is found in [Clements et al. 2003].
- Section 15.3** The Pipe-and-Filter style is discussed extensively in [Buschmann et al. 1996]. The Shared-Data style is covered in [Clements et al. 2003]; Buschmann et al. [1996] discuss the Blackboard style in depth. Shaw and Garlan [1996] discuss Implicit-Invocation systems. The MVC style was documented by Krasner and Pope [1988] and is discussed in [Buschmann et al. 1996].

Chapter 15 Exercises

The following program description is used in the exercises.

Process Monitor

A manufacturing process must maintain the pH of a solution in a vat within certain bounds. The pH of the vat is monitored by taking samples every 12 minutes using six pH sensors located at various places in the vat. The samples are stored, and every 12 minutes a control chart is produced showing a moving window of the most recent sample means.

The control chart displays between 10 and 40 data points, along with quality control statistics. Its parameters are set by the user.

The program displays and evaluates the control chart, as well as flagging the out-of-control points (data points indicating that the process may be out of control).

The program stores all data permanently in data files for later analysis. Each data file contains one day's pH readings, along with the times the readings were taken.

The program redraws the control chart every 12 minutes. It is also redrawn whenever the user changes any of the parameters governing the chart.

This program must be highly reliable. It must accommodate several kinds of pH sensors and allow new brands of sensors to be introduced later. Control charts are an established technique for monitoring industrial processes and are not likely to change.

- Section 15.1**
1. *Fill in the blanks:* Design patterns occur at several levels of abstraction, including _____, which are patterns at the architectural level; _____, which are patterns involving classes and their interactions; _____, which are patterns for implementing abstract data types and efficient operations; and _____, which are patterns for using a particular programming language well.
 2. Choose a data structure that you know well and write a pattern summary for the data structure that lists its name, application, form (including elements, structure, and behavior), and consequences. Feel free to use diagrams in your summary.
 3. Consider the relationships between modules described below, and for each one, indicate whether Module *A* uses Module *B*, and vice versa:
 - (a) Module *A* calls Module *B* to compute statistics on network traffic that *A* then displays to a user.
 - (b) Module *A* parses an expression in part by calling *B* to parse the terms in the expression. Some terms are themselves expressions, so *B* sometimes calls *A* as well.
 - (c) Module *A* throws an exception caught by *B*.
 - (d) Module *A* sets a non-local variable that Module *B* checks before deciding whether it needs to sort a list.
 - (e) Modules *A* and *B* exchange data by writing it to a file that they both read and write.
 - (f) Module *A* calls Module *B* whenever its second parameter is greater than zero.
 - (g) Module *A* sleeps until it is interrupted by Module *B*, which occurs whenever there is work for *A* to do.
 4. The Client-Server style is an architecture in which software is partitioned into a server and one or more clients. The server often mediates access to an expensive resource, such as a database or a printer, or mediates communication between clients, such as a mail server. The clients are constrained to communicate with the server and not with one another. Is the Client-Server style a special case of the Layered style? Explain why or why not.
 5. The “above” relation between layers in a Layered-style architecture is a partial order (a reflexive, anti-symmetric, and transitive relation). Explain how this relation is a partial order. Does this relation accord with a Strict or Relaxed Layered style?

6. Propose a Layered style architecture for the Process Monitor program. Draw a diagram to illustrate your design, explain each layer's responsibilities, and provide a design rationale.

AquaLush 7. Briefly discuss how well the AquaLush architecture conforms to the Layered style.

Section 15.3

8. Propose a Pipe-and-Filter-style architecture for the Process Monitor program. Draw a diagram to illustrate your design, explain what each filter does, and discuss filter synchronization.
9. Propose a Shared-Data-style architecture for the Process Monitor program. Draw a diagram to illustrate your design, explain what each data accessor does, and explain how data accessors are activated.
10. Is your design in the last exercise a Blackboard or a Repository style? Propose an alternative using the Blackboard style (if you used a repository) or the Repository style (if you used a blackboard).
11. Could an entire program be written so that all communication was in the form of events? Discuss the advantages and disadvantages of such a design.
12. Propose an Event-Driven-style architecture for the Process Monitor program. Draw a diagram to illustrate your design and explain the events announced and handled in your design.
13. Does the MVC architectural style rule out having more than two layers? Explain your answer.
14. Propose a Model-View-Controller architecture for the Process Monitor program. Draw a diagram to illustrate your design and list the responsibilities of the models, views, and controllers in your design.
15. Evaluate the designs you proposed for the Process Monitor program in the previous exercises and write an essay in which you explain which design is the best.
16. Use a scoring matrix (discussed in Chapters 5 and 10) to evaluate the designs you proposed for the Process Monitor program in the previous exercises. Use it to select an alternative.
17. Combine the best features of the designs you proposed for the Process Monitor program in the previous exercises. Write a design rationale in which you explain your design decisions.
18. Make a matrix whose rows are the architectural styles discussed in this chapter and whose columns are architecture quality attributes, such as performance, reliability, and maintainability. Rate each style on a one-to-five scale (five being the highest) with respect to each quality attribute.
19. Obtain a copy of Christopher Alexander's *A Pattern Language* ([Alexander et al. 1977]) and summarize 10 patterns that you think are especially interesting.

Research Projects

20. How many well-known architectural styles are there? Do some research in the library and on the Internet to come up with an estimate.
21. How does the UNIX shell handle splitting and joining data streams using pipes and filters? Write a UNIX command line or a shell program to illustrate splitting and joining data streams.

Chapter 15 Review Quiz Answers

Review Quiz 15.1

1. Design patterns streamline design documentation because many design details can be expressed by simply noting that a particular design pattern is used.
2. The patterns movement began in the late 1980s and gained impetus during the decade of the 1990s. Patterns are now a well-established part of the software engineering design literature.
3. A pattern description might list other names for a pattern, details about how it is implemented, examples of its use, important programs in which it was used, and related patterns.

Review Quiz 15.2

1. Layered architectures are usually depicted using a wedding cake diagram that shows modules stacked one above another or using an onion diagram that shows components in concentric ovals.
2. Each module in a Layered architecture is supposed to use only the module immediately below it (Strict Layered style). This constraint is often loosened to allow each module to use all the modules below it (Relaxed Layered style) to make programs simpler, faster, and easier to program.
3. In a Layered architecture, each layer should contain cohesive elements and be coupled only to the module below it. This should make it relatively easy to extract a layer and use it in another program. Furthermore, similar programs are likely to require similar layers, making layer reuse even easier.
4. Realization of many program features in a program with a Layered architecture will span several—perhaps most—layers. If there is a bug, it is hard to determine which layer is at fault, making it hard to debug the program.

Review Quiz 15.3

1. A pipeline is a kind of Pipe-and-Filter architecture in which no filter splits the data stream into parts or joins data streams together, and in which there are no cycles. In other words, it is a linear sequence of filters connected by pipes.
2. The Blackboard style is a Shared-Data style in which the shared-data store (the blackboard) triggers the shared-data accessors when it is changed. In contrast, the Repository style is a Shared-Data style in which the shared-data store (the repository) is entirely passive. The accessors must either run continuously or be triggered by some other component.
3. An event in an Event-Driven architecture can hold data, thus conveying data from the event announcer to the components that respond to the event.
4. The MVC style does not insulate the user interface from changes in the application domain; a disadvantage of this style is that changes in the application domain generally result in changes in the user interface as well.

16 Mid-Level Object-Oriented Design Patterns

Chapter Objectives

This chapter is the first of four presenting a small catalog of mid-level design patterns. It introduces the Iterator pattern as a paradigm mid-level design pattern and introduces a pattern classification system used to arrange the patterns discussed in the remaining chapters.

By the end of this chapter you will be able to

- Explain the problem of collection iteration, design alternatives to solve this problem, and state their advantages and disadvantages;
- Describe and explain the Iterator pattern and show how it solves the collection iteration problem;
- Use the Iterator pattern in mid-level detailed design; and
- Explain the broker, generator, and reactor pattern categories.

Chapter Contents

- 16.1 Collection Iteration
 - 16.2 The Iterator Pattern
 - 16.3 Mid-Level Design Pattern Categories
-

16.1 Collection Iteration

Mid-Level Design Patterns

This chapter is the first of four covering design patterns at an intermediate level of abstraction. Such patterns typically model collaboration between several medium-sized modules. These mid-level design patterns are an important resource for mid-level detailed design. We introduce this topic by discussing the Iterator design pattern as a paradigm. Before considering the Iterator pattern, we discuss the general problem of collection iteration to provide context and as a case study in mid-level design. This section covers collection iteration; the next presents the Iterator pattern. The final section explains the mid-level classification scheme used to structure discussion in the remaining chapters.

Collection Iteration

A **collection** is any object that holds or contains other objects, such as an array, a set, a list, a tree, a table, and so forth. Collections must provide some means of traversal so that components that use the collection, which we call *clients*, can access each element of the collection in turn. Traversal and access of each element in a collection is termed **iteration over the collection**, or **collection iteration**.

Some collections have operations that clients can use for iteration. For example, array and list collections usually provide an operation that takes an array or list position argument and returns the element at that position, along with an operation returning the size of the collection. It is an easy matter for clients to use these operations to iterate over the list in a `for` loop. Other collections do not have such operations. The elements of sets, hash tables, and trees, for example, do not have position numbers, so these collections do not provide position-number-based access operations. Because many collections do not have operations that can be adapted for iteration, some way to iterate over collections must be supplied. An **iteration mechanism** is a language feature or a set of operations that allows clients to access each element of a collection.

If an iteration mechanism must be supplied for some collections, it makes sense to supply a uniform mechanism for all collections. This has the advantages of being easier for developers to understand and remember as well as making designs and code easier to change. To illustrate the latter point, suppose a program stores some data in a list over which it must iterate. If programmers later decide to replace the list with a hash table, the iteration portion of the code will not have to be changed if it already makes use of an iteration mechanism common to all collections.

Iteration Mechanism Requirements

Every collection iteration mechanism must provide the following four iteration control facilities:

Initialize—Prepare the iteration mechanism for traversal.

Access Current Element—Provide client access to the current element in the collection.

Advance Current Element—Make the next element in the collection the current element.

Completion Test—Indicate when traversal is complete.

In addition, iterators may provide facilities to back up or reverse direction during traversal, skip over certain entities, and so forth; different facilities are needed in different circumstances.

Iteration mechanisms should also meet the following design requirements to satisfy a wide range of needs:

Information Hiding—The internal structure of the collection must not be exposed.

Multiple Simultaneous Iterations—Several simultaneous iterations must be able to occur without interfering with one another. To see why this is important, suppose that a program must compute the shortest path between every pair of nodes in a graph, which are stored in a collection. A brute-force algorithm iterates over all the nodes and, for each one, computes the shortest path to every other node, which involves iterating over the collection again. Two iterations are occurring simultaneously, and the algorithm will not work if they interfere with one another.

Collection Interface Simplicity—A variety of collection traversals and iteration controls must be available without cluttering the collection’s interface. Collections can be traversed in many ways, and there may be a need to support many sorts of iterations. Furthermore, there may be a need for fine iteration control. For example, binary tree iterations include in-order traversal, pre-order traversal, post-order traversal, and level-order traversal, along with the reverse of all these. In addition, clients might need to back up or reverse direction during iteration. Adding support for all of these facilities to a tree collection class interface would complicate it greatly.

Flexibility—Iteration mechanisms must allow clients flexibility in processing during collection traversal. For example, it should be easy for clients to abort a traversal early, alter processing of the current elements based on values of previous elements, and so forth.

These requirements can be used to evaluate several collection iteration mechanism design alternatives.

Iteration Mechanism Design Alternatives

Design alternatives for implementing iteration mechanisms can be classified in two dimensions: where the iteration mechanism resides and whether iteration control is internal or external.

There are three possibilities for where an iteration mechanism resides:

Programming Language—Some languages have constructs for iterating over collections. For example, a `for` loop used to iterate over collections has recently been added to Java. It has the following form:

```
for ( elementType e : collection ) {
    // process element e
}
```

where `elementType` is the class of elements in a `collection` and `e` is a variable. The iteration mechanism is initialized when the loop starts. The test for iteration completion is done at the top of the loop. The loop control variable `e` holds the current element of the `collection` during collection traversal, and it is assigned the next element (if any) at the end of the loop.

Collection—Collections can include iteration facilities as part of their service to clients. Such *built-in* iteration mechanisms usually take the form of iteration operations added to the collection’s interface.

Iterator—An **iterator** is an entity that provides serial access to each element of an associated collection. Iterators house iteration mechanisms.

Building collection iteration into a programming language satisfies all iteration mechanism requirements: It is a powerful language feature that is simple to use. This is an excellent design alternative, but it depends on language features that are not often present. As this alternative may not be available to designers, we do not consider it further.

Placing an iteration mechanism in the collection has the advantage of hiding the collection's implementation from clients, while placing it in an iterator may expose the collection's internal processing to clients.

Iteration mechanism control can work in one of two ways:

External—An iteration mechanism with **external iteration control** provides collection elements as directed by the client. In this case the client controls element access.

Internal—An iteration mechanism with **internal iteration control** accepts operations and applies them to each element of the collection on the client's behalf. In this case the iteration mechanism controls element access rather than the client.

These alternatives are illustrated in Figure 16-1-1.

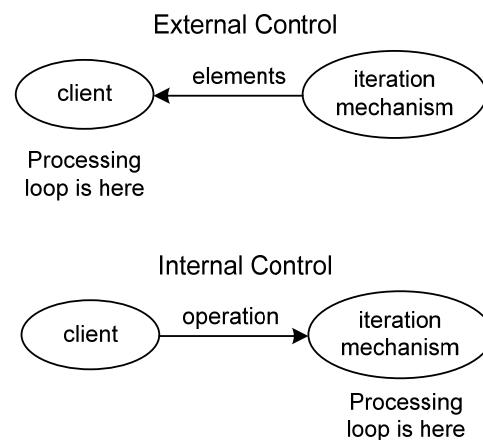


Figure 16-1-1 External Versus Internal Iteration Control

With external control, the client sets up a loop and asks the iteration mechanism to supply collection elements one at a time for processing. With internal control, the client passes an operation to the iteration mechanism, which then uses a loop to apply the operation to each collection element.

Combining these two dimensions of design variation gives four iteration mechanism design alternatives, summarized in Table 16-1-2.

		Residence	
		Collection	Iterator
Control	External	Collection with built-in external control	Iterator with external control
	Internal	Collection with built-in internal control	Iterator with internal control

Table 16-1-2 Iteration Mechanism Design Alternatives

Built-In Internal Control

When an iteration mechanism has internal control, the client must package processing for each collection element and give it to the iteration mechanism. The iteration mechanism then traverses the collection and applies the processing package. Processing is usually packaged in an operation.

To illustrate, suppose a **Collection** class provides a built-in internal iterator, as illustrated in the class diagram in Figure 16-1-3.

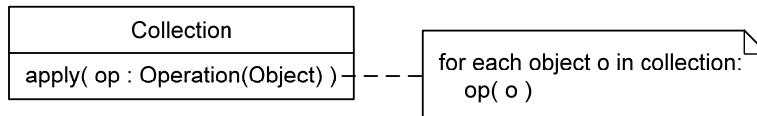


Figure 16-1-3 Internal Control of Collection Iteration

The **apply()** operation takes an operation **op()** as its parameter and applies it to each element of the **Collection**. For example, suppose a client wants to print out the elements of a **Collection**. The client could first define an operation like the one in Figure 16-1-4.

```

printObject( Object o ) {
    println( o )
}
  
```

Figure 16-1-4 An Operation Applied During Iteration

The client then simply calls the collection's **apply()** operation with **printObject()** as its parameter (here, **u** is a client object and **c** a collection object):

```
c.apply( u.printObject )
```

A collection with built-in internal iteration control can easily hide its implementation, and adding one or two **apply()** operations to the collection interface does not overly complicate it. Hence, collections with built-in internally controlled iteration mechanisms satisfy some iteration mechanism requirements well.

Supporting multiple simultaneous iterations is not so easy. We discuss why when considering built-in external controls; for now, we simply assert that built-in iteration mechanisms do not support multiple simultaneous iterations well.

The biggest drawback of built-in internally controlled iteration mechanisms is lack of flexibility. This problem stems from having internal control. External control always provides fine control over iteration, while internal control always limits client control, often severely. For example, suppose a client wants to do something as simple as searching a collection. There is no need to continue searching once the desired element is found, so an

obvious need is the ability to abort iteration. A client can easily do this when using an iteration mechanism with external control. But how might this work when control is internal? The operation a client passes to the collection does not have access to the collection itself, and even if it did, the collection does not provide any way to abort iteration. Adding facilities to support this need would complicate the collection interface and make using the iteration control mechanism more difficult. Furthermore, doing so would address only one problem; what about other client needs for more flexibility during collection traversal?

We conclude that built-in iterators with internal control fail to satisfy all requirements for a general-purpose iteration control mechanism.

Built-In External Control	A collection with a built-in externally controlled iteration mechanism must provide the four previously discussed iteration facilities to clients. Usually this is done by providing one or more operations. For example, a collection might provide the following operations to control iteration:
---------------------------	---

`reset()`—Initialize iteration.
`getNext()`—Obtain the current value and advance to the next element.
`hasMore()`—Return false when traversal is complete.

Other iteration control facilities require additional operations. This design alternative significantly adds to the collection's interface, in violation of the requirement that the collection interface be kept simple.

Another difficulty with this design alternative is that it does not easily support multiple simultaneous iterations. Suppose, for example, that a client must perform two simultaneous iterations using the previously defined operations. The client initializes the first iteration by calling `reset()` and then begins a `while` loop controlled by `hasMore()` with `getNext()` in its body, as shown in Figure 16-1-5 (c is a collection instance).

```
c.reset();
while ( c.hasMore() ) {
    element = c.getNext();
    // second iteration starts here
}
```

Figure 16-1-5 Multiple Simultaneous Iteration Framework

Having obtained an element of the collection, the client is ready to begin the second iteration over the same collection, as indicated by the comment. However, calling the collection's `reset()` operation again to initialize the second iteration will instead abort the first iteration! The problem is that there is no way for the client to indicate to the collection which iteration it is trying to control with various calls of the iteration operations.

The only solution to this problem is to have the collection provide some kind of identifier that the client can use to distinguish different iterations. For example, the `reset()` operation might return an iteration identifier that the client must then supply as a parameter to `hasMore()` and `getNext()` to distinguish iterations. The client might need to call a `release()` operation to tell the collection to retire an iteration identifier. This solution is inconvenient, complicated to implement in the collection, and prone to error, as clients may mix up iteration identifiers or forget to release iteration identifiers. It also further complicates the collection interface.

In summary, this design alternative hides the implementation of the collection and provides clients with flexible iteration control, but it does not support multiple simultaneous iterations and complicates the collection interface.

Iterator Advantages and Disadvantages

Placing the iteration mechanism in iterators separate from collections helps satisfy two iteration mechanism requirements:

Multiple Simultaneous Iterations—Each iterator can be responsible for managing one traversal of its associated collection independently of all other iterators. Clients can then use several iterators at once to traverse the iteration in various ways simultaneously.

Collection Interface Simplicity—The iteration control interface is in iterators, not the collection. The only addition to the collection is a facility (usually an operation) for creating iterators.

The only potential disadvantage of using iterators is that if they are not well implemented the collection implementation may be exposed. Iterators are separate from collections, but they must “look inside” a collection to access its elements. Thus, the implementation of the collection must be exposed to iterators and *only* to iterators. With care, this can be done; we discuss how in the next section.

We previously noted how built-in iteration mechanisms fail to support multiple simultaneous iterations and how externally controlled built-in iteration mechanisms complicate collection interfaces. The advantages of iterators over built-in collection iteration mechanisms clearly favor iterators as the better design alternative.

Note, however, that internally controlled iterators work much as internally controlled built-in iteration mechanisms and have the same drawback: They do not provide flexibility during collection traversal.

The Iterator Pattern

We have now considered all four iteration mechanism design alternatives. Iteration mechanisms built into collections do not provide good support for multiple simultaneous iterations. Internally controlled iteration mechanisms lack processing flexibility. Externally controlled built-in iteration mechanisms complicate the collection interface. The only design

alternative that satisfies all requirements for a general-purpose iteration mechanism is an externally controlled iterator.

The **Iterator pattern** is an object-oriented design pattern for externally controlled iterators. This pattern is covered in the next section.

Robust Iteration Mechanisms Before considering the Iterator pattern, we explore one more aspect of the design of iteration mechanisms: What happens if the collection being traversed changes during iteration?

It is not altogether clear what should happen in different cases. For example, what should occur if the last element in a collection is removed after an external iteration mechanism has said that it is there but before it has been retrieved? Arbitrary decisions can be made about the behavior of an iteration mechanism in such circumstances. At a minimum, however, an iteration mechanism able to tolerate changes to its associated collection should meet the following requirements:

Fault Tolerance—The program should not crash.

Iteration Termination—Iteration should halt; that is, the iteration mechanism should not cause infinite loops.

Complete Traversal—Elements present throughout any changes to the collection during iteration should not be missed if the iteration continues after the collection is changed.

Single Access—Elements should not be accessed more than once during iteration.

An iteration mechanism specification conforming to these four requirements is *coherent*. Note that a coherent specification may allow or prohibit continued iteration in the face of change, and may include or exclude from the iteration elements added to or removed from the collection during traversal. We can now characterize robustness.

A **robust iteration mechanism** is one that conforms to some coherent specification of behavior when its associated collection is changed during iteration.

Making an iteration mechanism robust can be difficult or computationally expensive. A copy of the collection may have to be made before iteration begins, or code may have to be inserted in the collection, the iteration mechanism, or both to coordinate processing when the collection is changed.

Whether an iteration mechanism is robust or not, it is imperative that its programmers *know* whether it is robust. Many obvious algorithms for common processing tasks depend on an iteration mechanism being robust. For example, the obvious way to remove unwanted elements from a collection is to traverse it, inspect each element, and delete those that should be removed. This will not work with a non-robust iteration

mechanism. Non-robust iteration mechanisms can also cause failures in concurrent programs. Clients must be aware of robustness to guard against such difficulties by ensuring that collections do not change during traversal by non-robust iteration mechanisms.

Section Summary

- A **collection** is an object that holds or contains other objects.
- **Collection iteration** is the process of serial access of each element of a collection.
- Collection iteration is realized by **iteration mechanisms** that can have internal or external control and can either be built into collections or be implemented in separate entities.
- An **iterator** is an entity that provides serial access to each element of an associated collection.
- Externally controlled iterators best satisfy the requirements for a general-purpose iteration mechanism.
- An important virtue of iteration mechanisms is **robustness**, meaning that the iteration mechanism performs correctly when its associated collection changes during iteration.

Review Quiz 16.1

1. What four iteration control functions must every iteration mechanism have?
 2. How can an iteration mechanism reside in a programming language?
 3. What is the major drawback of an internally controlled iteration mechanism?
 4. What four requirements should a robust iterator adhere to?
-

16.2 The Iterator Pattern

Iterators

As previously discussed, the best design for a general-purpose iteration mechanism is one in which it is placed in an entity separate from its associated collection. Objects implementing iteration mechanisms are called **iterators**.

Each iterator is responsible for managing access to the elements of a collection. Iterators provide access to collection elements without exposing the collection's implementation details. Because iterators are separate from each other, they do not interfere with one another during collection traversal. Finally, iterators have their own interface, thus keeping the collection interface simple while providing the potential for a rich set of traversals and operations to control them. Iterators thus elegantly satisfy iteration mechanism design requirements.

An iterator with external control is an **external iterator**, and one with internal control is an **internal iterator**. As previously noted, external iterators are preferred as a general solution because they provide maximum flexibility. Thus, the ideal iteration mechanism is an external iterator.

The Iterator design pattern is an object-oriented model for an external iterator.

An Analogy

A collection object is like a warehouse filled with items carefully arranged for easy inventory and rapid access. Allowing clients to walk into the warehouse and poke around would not be a good idea: Items might be misplaced or lost, and clients might get in each other's way. Furthermore, every client would have to learn how the warehouse was arranged to use it effectively. Instead, the warehouse can provide each client with one or more dedicated clerks, which correspond to collection iterators. Clerks know how to find things in the warehouse, they are careful to preserve the order of the warehouse, and they coordinate their activities so that they do not interfere with one another.

A clerk can work in one of two ways. An *external clerk* (corresponding to an external iterator) retrieves each item in turn and presents it to the client. An *internal clerk* (corresponding to an internal iterator) receives instructions from clients about what to do with each item in the warehouse and then carries out these instructions.

Iterator Operations

As noted in the last section, every iterator must implement four basic control functions:

Initialize—Prepare for traversal.

Access Current Element—Provide client access to the current element in the collection.

Advance Current Element—Make the next element in the collection the current element.

Completion Test—Indicate when traversal is complete.

Iterators may have additional functions for added control, such as reversing traversal direction, backing up, and skipping elements.

These functions can be packaged into operations in different ways. For example, an iterator might provide the four basic functions as four separate operations and additional functions in other operations.

Alternatively, the initialize function might be done at iterator creation, and only then. In this case, an iterator could be used to traverse a collection once, and then it would be “used up”: A new iterator would have to be created to traverse the collection again. An iterator might place the last three fundamental behaviors in a single operation that returns a Boolean to indicate whether traversal is complete, sets a parameter to the current element if it is not, and advances to the next element of the collection. Thus, an iterator could have just a single (rather complicated) operation.

**Iterator
Pattern
Structure**

The Iterator pattern is an object-oriented generalization of an external iterator. Each collection object includes one or more iterator creation methods that return various kinds of iterators for accessing its elements. Each iterator has operations implementing the four fundamental behaviors previously discussed. The iterator must maintain a reference to its collection so that it can access elements. This is reflected in the class diagram in Figure 16-2-1.

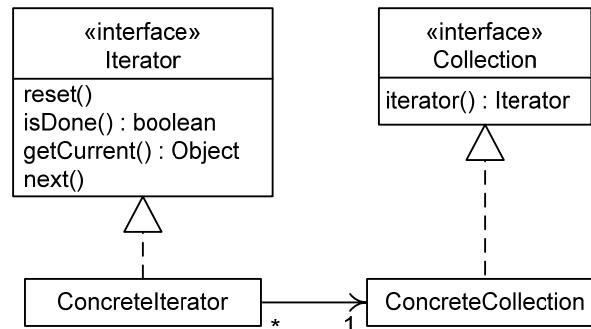


Figure 16-2-1 Iterator Pattern Structure

The use of interfaces increases program changeability and maintainability. Clients should declare all their variables to be of type `Iterator` or `Collection` and use only the `Iterator` and `Collection` operations wherever possible. If a need arises to change the type of collection used, only the code that actually creates the collection needs to be changed.

**Iterator
Pattern
Behavior**

A client creates an iterator object whenever it needs to traverse a collection. The client then requests access to objects in the collection, and controls collection traversal, by sending messages to the iterator.

The dynamics of iterator use are illustrated in the sequence diagram in Figure 16-2-2. The `isDone()` operation regulates a loop that requests each item in the collection in turn with the `getCurrent()` operation and then moves the iterator to the next element with the `next()` operation.

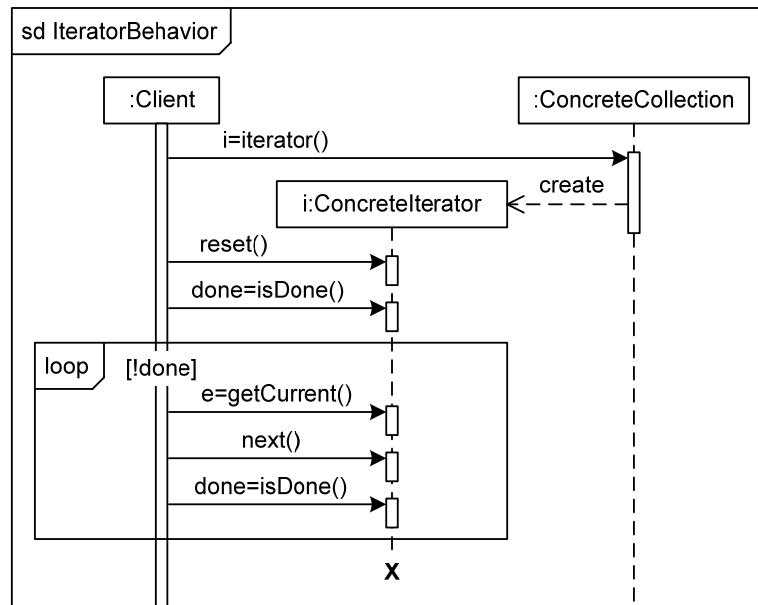


Figure 16-2-2 Iterator Pattern Behavior

When to Use Iterators

Iterators should be used as a matter of course with collection objects; in fact, iterators may often be the only way to access the elements of a collection (short of removing them all and then putting them back in the collection).

Iterators support information hiding, help simplify collection interfaces, decouple modules by supporting multiple concurrent traversals, and make programs more changeable. The Iterator pattern is among the most widely known and used of all design patterns.

External iterators require several operation calls to access each element of a collection, so they sometimes may cause efficiency problems. The greatest problem that may arise using the Iterator pattern is that an iterator may fail when its collection is changed during iteration. In other words, the greatest problem with using iterators is lack of robustness. Iterators should be robust, or at least their non-robustness should be well documented.

Implementation Considerations

Iterators usually need access to the internals of a collection to do their job. If this is not done carefully, collection internals are exposed, violating the Principle of Information Hiding. There are several ways to implement

iterators so that the internal details of collections are shielded from the rest of the program:

Make Collection Internals Visible to Iterators—An iterator can be implemented in a portion of the program where collection internals are visible. In Java, this can be done in two ways: (a) iterators can be made inner classes of the collection class, or (b) a collection and its iterators can be in the same package and the parts of the collection the iterator needs can be given package visibility. The problem with alternative (b) is that all the other classes in the package with the collection also have access to the collection's internals. Alternative (a) is clearly better. Another advantage to making iterators inner classes is that they can also be made private, completely hiding the implementation of iterator classes.

Extend Access to Collection Internals to Iterators—The crucial private members of a collection class can be passed to iterators using references or aliases, allowing the iterators, but no other entities, to access collection internals. For example, suppose that a collection stores elements in a linked list. The collection can pass the list head when it creates a new iterator instance. This strategy is illustrated in Figure 16-2-3.

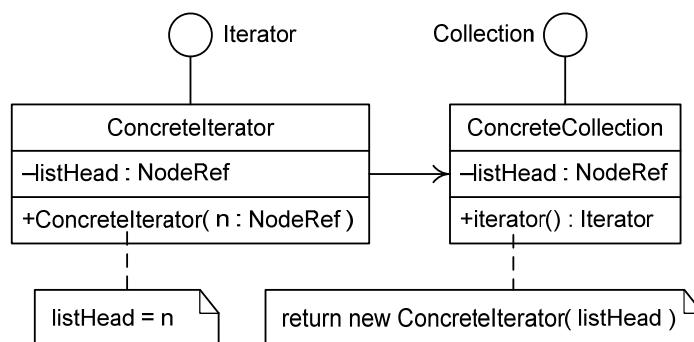


Figure 16-2-3 Extending Access to an Iterator

The `ConcreteCollection`'s `iterator()` operation creates a new `Concretelterator` and passes it `listHead`, a reference to the first node in the list. The `Concretelterator` records this reference in its own private `listHead` attribute, giving it access to the list stored in the `ConcreteCollection`. Because the `listHead` attributes are private in both classes, no other class can access the list.

Of these design alternatives, using an inner class is preferred because it hides information best and because it has better cohesion. A concrete

iterator belongs with its associated concrete collection; making it an inner class puts it where it belongs.

Iterators in AquaLush

AquaLush has many collections that must be traversed often. For example, it has collections of zones, valves in a zone, and broken devices. Zone collections must be traversed during automatic irrigation. Valve collections must be traversed to open and close valves when irrigation in a zone is begun or ended and to compute water allocations. Broken device collections must be traversed to generate reports.

One particularly interesting kind of iteration occurs when valve collections are traversed. AquaLush only uses valves that are not broken. A client can iterate over all the valves in a collection and test each one to see if it is working, or a special *working valve iterator* can be written to deliver only working valves to clients. An iterator that provides access to elements of a collection meeting some criterion is called a **filtering iterator**.

Java Enumerations and Iterators

Most class libraries provide iterators to go along with various collection classes, or mechanisms for helping make iterators for custom-built collection classes. The Java `Iterator` interface is for robust external iterators. Classes that implement this interface traverse classes that implement the `Collection` interface, which include most of the collection classes in the Java class libraries.

The `Iterator` interface specifies three operations: `hasNext()` returns true if and only if the iterator's collection has not been completely traversed, `next()` returns the current element in the collection and advances the current element, and `remove()` deletes the last element retrieved by `next()` from the collection. The `remove()` operation is provided to handle the common need to delete an element from a collection during traversal in a robust way. This operation is optional, so some `Iterators` might not support it. If an element is deleted from a collection during iteration without using the `remove()` operation or is changed in some other way, many `Iterator` objects will throw a `ConcurrentModificationException`, making them robust. `Iterators` are not forced to behave this way, so great care must still be exercised when using them.

Java also has older non-robust iterators that implement the `Enumeration` interface. Programs that change a collection during traversal with an `Enumeration` iterator may crash or go into an infinite loop. These iterators have been superseded by the `Iterator` interface and should no longer be used.

Pattern Summary

Figure 16-2-4 summarizes the Iterator pattern.

Name: Iterator

Application: Create iterators to go along with collections and use them to traverse collections.

Form: A Concreteliterator class realizes an Iterator interface with standard iteration operations. A ConcreteCollection class has an operation for creating Concreteliterator instances. In use, a Client obtains a Concreteliterator instance from a ConcreteCollection and then uses the Iterator operations to access each element of the ConcreteCollection.

Consequences: An iterators hides the implementation of its associated collection, simplifies its interface, and allows multiple simultaneous traversals of the collection. Iterators may be less efficient than other means of access. An important concern is iterator robustness; that is, whether an iterator's behavior when its associated collection is changed during traversal is well defined.

Figure 16-2-4 The Iterator Pattern

Section Summary

- Iterators provide various kinds of traversal, processing flexibility, and program changeability without complicating the collection interface.
- The **Iterator pattern** is a mid-level design pattern for implementing external iterators.
- The Iterator pattern should be used whenever collections need to be traversed.
- Object-oriented languages typically provide iterators in standard class libraries. Java, for example, provides two in the `java.util` package.

Review Quiz 16.2

1. What is the difference between internal and external iterators?
2. Why are interface types part of the Iterator pattern?
3. How can iterators gain access to collection internals without exposing them to the rest of the program?
4. What is a filtering iterator?

16.3 Mid-Level Design Pattern Categories

Too Many Patterns?

Since they became an important topic in the mid-1990s, several hundred mid-level software design patterns have been published. Although many of these patterns are obscure, several dozen of them, at least, are quite useful. Design patterns do not help designers if they cannot be remembered and called to mind when appropriate. How can designers keep so many patterns in mind?

Few people can keep dozens of design patterns at their fingertips without something to help remember them. Probably the best mnemonic aid is a good classification system based on problem types. When a particular

problem crops up, the designer can recall a group of patterns that solve that kind of problem.

The mid-level design patterns presented in this and the next few chapters are divided into three categories based partly on their form, which helps group them, and partly on the sorts of problems they solve, which helps designers remember them.

There are many other pattern classification systems in the literature, but most of them do not provide much help in remembering patterns. The classification used in this book makes it easier to both remember and use design patterns. However, only about a dozen patterns are presented here; there are probably many others that do not fit into the three categories presented. These categories are just a starting point for a larger design pattern classification system.

Pattern Categories

Most of the mid-level patterns discussed here are models for collaborations among three classes. One class is always a client that needs a service, and the other two cooperate, sometimes interacting with the client, to provide the service. For example, in the case of the Iterator pattern, a **Client** needs access to each element of a **Collection**, and an **Iterator** provides this access. The **Collection** and **Iterator** work together to provide a service to the **Client**, which in this case must also do some work controlling the iteration.

We divide mid-level design patterns into the following three categories:

Broker—**Broker patterns** feature a client that needs a service from a supplier, but there is some problem with the supplier providing the service directly to the client. The problem is solved with a *broker* that mediates the interaction between the client and the supplier. An analogy from human interactions is a stock broker who buys and sells stocks on behalf of a client. Clients do not have access to the stock exchanges, so they must go through the broker to obtain these services from a stock exchange.

Generator—The client in a **generator pattern** needs a new instance of a product, but is not able to instantiate the product class directly. A generator class supplies the product instance to the client, solving the problem. Generator classes are analogous to interior designers who obtain furnishings from manufacturers and then customize and install them on behalf of clients.

Reactor patterns—**Reactor patterns** are the most complex mid-level design patterns considered, especially in their behavior. The client generally creates a reactor instance that registers with a target to carry out tasks in response to target events on behalf of the client. Reactors are like enterprises that supply continuing services to clients, such as security services or lawn services. When some event occurs, such as an attempted break-in or the lawn growing, these enterprises swing into action and take care of the situation on behalf of their clients.

Chapter 17 discusses broker patterns, Chapter 18 generator patterns, and Chapter 19 reactor patterns. The characteristics of each category are discussed along with specific patterns.

Section Summary

- Patterns can be classified according to their form and use, which makes them easier to remember and use during design.
- **Broker patterns** have a broker that mediates interactions between a client and a supplier.
- **Generator patterns** have a generator that creates instances of a product on behalf of a client.
- **Reactor patterns** have a reactor that registers with a target to respond to target events on behalf of a client.

Review Quiz 16.3

1. How many classes usually participate in the patterns presented in this book?
 2. What is the difference between broker and generator patterns?
-

Chapter 16 Further Reading

Section 16.1 Collection iteration is discussed in [Gamma et al. 1995].

Section 16.2 The Iterator pattern is discussed at length in [Gamma et al. 1995].

Section 16.3 The pattern classification used in this book is based mainly on Michael Norton's work (see [Norton 2003]). Gamma et al. [1995] and Buschmann et al. [1996] both present alternative pattern classifications.

Chapter 16 Exercises

Section 16.1

1. Design and code a `Vector` class in Java with a built-in externally controlled iteration mechanism. Realize the four iteration control facilities in four operations.
2. Design and document a single operation that incorporates all four iteration mechanism control functions.
3. Design an internally controlled iterator for a `Vector` class. Document your design with a class diagram. Be sure to explain how the iterator gains access to the internals of the `Vector` class.
4. Is the iteration mechanism you designed for the last exercise robust? If not, how could you make it robust?
5. Design a built-in iteration mechanism with internal control to provide traversal algorithms for a `BinaryTree` class. Include the `BinaryTreeNode` class in your design. Use a class diagram and code snippets to present your design.

6. Design a `Vector` class with a built-in externally controlled iteration mechanism that supports multiple simultaneous iterations along the lines suggested in Section 16.1. Write pseudocode to document how each operation works.
7. State two things that a robust iterator might do when its associated collection is changed during iteration. Make sure these behaviors satisfy the four requirements for specifying a robust iterator.
8. If an iterator is robust, it needs to know whether its associated collection has changed. Propose a strategy that iterators and collections can use to keep the iterator informed of the state of the collection. What are the advantages and disadvantages of your strategy?

Section 16.2

9. Design your own `Vector` class along with an external iterator class called `VectorIterator` using the Iterator design pattern. Provide implementations of the iterator operations in your design in notes attached to the operations in your class diagram. Also explain how an iterator obtains a reference to its associated `Vector` instance.
10. Consider your solution to the last exercise. What happens if two simultaneous iterations are attempted? If these iterations interfere with one another, how could you change your design to support multiple simultaneous iterations?
11. Consider your solution to the last exercise. What happens if the `Vector` is changed (elements are added or deleted) in the middle of iteration? If your iterator is not robust, how can you make it robust?
12. Design and code an iterator for an arbitrary array (an array of `Objects`) in Java.
13. The Java `ArrayList` class permits null elements to be added. Design a filtering iterator that traverses an `ArrayList` but returns only non-null elements. Write pseudocode to illustrate how to implement the iterator operations.
14. Write Java code to make two simultaneous traversals of a collection using an iterator for that collection.

Section 16.3

15. Explain the Iterator pattern in terms of the roles of a broker pattern.
16. The Iterator pattern is a model for using an external iterator. Let us call a pattern for using an internal iterator the *Iteration Agent* pattern. Complete the following activities:
 - (a) Make a class diagram to describe the static structure of the Iteration Agent pattern.
 - (b) Make a sequence diagram to describe the behavior of the Iteration Agent pattern.
 - (c) Write a pattern summary like the one in Figure 16-2-4 to describe the Iteration Agent pattern.
17. An iteration control structure has recently been added to Java. Find out about this control structure and write an account of how to use it.

Research Projects

18. Operations cannot be passed as arguments in Java. Find out about the Command pattern and explain how to use it to implement a built-in internally controlled iteration mechanism in Java.
19. The C++ Standard Template Library provides iterators. Are they robust?
- Team Projects** 20. Form a team of two. Design, implement, and test List and ListIterator classes using the Iterator pattern. Include add() and remove() operations in the List class. Use a singly linked list for the collection. Document your design with a class diagram and a data structure diagram. Write your code in Java and make the ListIterator implement the `java.util.Iterator` interface and be robust. Also, make the ListIterator class separate from but in the same package as the List class.
21. Repeat the last exercise, but make the ListIterator a private inner class in the List class.

Chapter 16 Review Quiz Answers

Review Quiz 16.1

1. Every iteration mechanism must provide an initialization function that prepares the iteration mechanism and the collection for traversal, a function to determine whether iteration is complete, a function to fetch the current element from the collection, and a function to advance the current element.
2. Some programming languages provide special control structures for iterating over collections. For example, in Smalltalk there is a built-in do control structure that can be used with any collection to iterate over its elements. In such cases the iteration mechanism is not built into collections or separate entities, but is part of the language itself.
3. The major drawback of an internally controlled iteration mechanism is that it limits processing flexibility. Clients are unable to exercise fine control over collection traversal or the processing done at each step based on previous steps.
4. A robust iterator must conform to several requirements if iteration is continued after its associated collection is changed. It must not crash, not fail to terminate, provide access to every persistent collection element, and provide access to elements no more than once.

Review Quiz 16.2

1. An internal iterator accepts an operation from a client and applies it to all elements of a collection on the client's behalf. An external iterator mediates the client's interaction with a collection, providing each element of the collection to the client in succession. The client can then process each element.
2. The Iterator pattern uses interface types to increase changeability. Designers and programmers can express designs and write code in terms of the `Iterator` and `Collection` interfaces. The resulting code and designs will not need to be changed even if particular collection objects change.
3. Iterators can be given access to collection internals without exposing them to the rest of the program by either placing the iterators in a portion of the program where the collection internals are visible or extending access to collection internals to iterators. In C++, for example, collection class internals can be made visible to iterator classes by making the iterator classes friend

classes of the collection class. Extending access to collection internals can be done by the collection passing references to crucial data structures to iterators when they are created.

4. A filtering iterator is an iterator that provides access to elements of a collection meeting some criterion.

**Review
Quiz 16.3**

1. The patterns presented in this book mainly involve three classes, one of which is always a client. The client needs some service, and the other two classes work together to provide the service, sometimes in collaboration with the client.
2. The broker patterns feature an intermediate class that works with a supplier to provide a *service* to a client, while generator patterns have an intermediate class that works with a producer to provide a class *instance* to a client.

17 Broker Design Patterns

Chapter Objectives

This chapter focuses on broker design patterns and presents several important broker patterns.

By the end of this chapter you will be able to

- Illustrate and explain the structure and behavior of broker patterns, and list their uses;
- Describe, illustrate, and explain several important broker patterns, including the Façade, Mediator, Class and Object Adapter, and Proxy patterns; and
- Use these patterns in mid-level design.

Chapter Contents

- 17.1 The Broker Category
 - 17.2 The Façade and Mediator Patterns
 - 17.3 The Adapter Patterns
 - 17.4. The Proxy Pattern
-

17.1 The Broker Category

Broker Patterns

Broker patterns are the simplest mid-level design patterns. All broker patterns have instances of a **Broker** class that mediate the interaction between **Client** and **Supplier** class instances. The static model of the broker patterns is shown in Figure 17-1-1.



Figure 17-1-1 Broker Pattern Structure

In a broker pattern, the **Client** can access the **Broker**, and the **Broker** can access the **Supplier**. These access associations are required to realize the behavior characteristic of broker patterns, illustrated in Figure 17-1-2.

The **Client** requests a **Supplier** service from the **Broker**. The **Broker** then interacts with the **Supplier** to obtain the service from the **Supplier** on behalf of the **Client**.

These static and dynamic models are very abstract and show only the essentials of the broker patterns. Particular patterns in this category may elaborate these schematic models. More of the classes in the pattern may be directly accessible to one another than is shown in the model in Figure

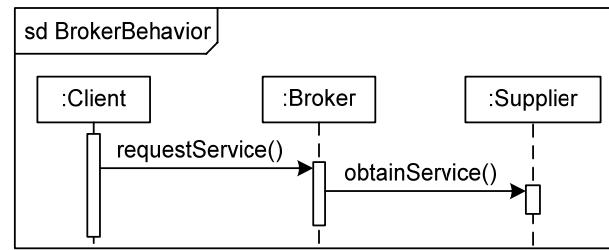


Figure 17-1-2 Broker Pattern Behavior

17-1-2, and the messages sent between them may be considerably more complex than those shown. Also, the models do not show super-classes and interfaces, which are frequently part of patterns in this category.

Consequences of Using Broker Patterns

Brokers patterns can be used for the following reasons:

Simplify the Supplier—A broker may augment or enhance a supplier’s services without complicating its interface or its design. This keeps the supplier from becoming bloated and hard to understand, making it more maintainable and reusable.

Decompose the Supplier—A complex supplier may be decomposed into parts with a broker presenting a uniform interface to the client and deciding how to route client requests. Decomposing suppliers often makes programs easier to understand, and hence more changeable and maintainable, as well as making their components more reusable.

Facilitate Client/Supplier Interaction—A broker can make it easier for the client and supplier to interact. For example, a broker might present a different interface to the client, the supplier, or both, or the broker may handle interaction details to make it easier for the client to obtain the services it needs. Brokers may thus be introduced to make programs easier to understand or as a way to make program changes easier.

The basic broker form combined with these uses lead to the particular patterns in the broker category.

Broker classes mediate communications between clients and suppliers, and in doing so they may slow this interaction, causing a performance penalty. On the other hand, a broker may sometimes be able to provide some services to clients more cheaply than if the clients interacted directly with the suppliers. As a result, brokers do not always damage performance.

The Iterator Pattern as a Broker Pattern

The Iterator pattern is a broker pattern, though it also incorporates a generator pattern, which we discuss in the next chapter. To see this, note that the task of an iterator fits the role of a broker: An iterator provides access to the elements of a collection on behalf of a client. The static structure of the Iterator pattern also fits the broker model: The Iterator pattern has a Client (the class that needs to traverse a collection), the ConcretelIterator is a Broker, and the ConcreteCollection is a Supplier. The class

diagram in Figure 17-1-3 reproduces the Iterator class diagram but adds an explicit Client and stereotypes identifying the broker pattern roles.

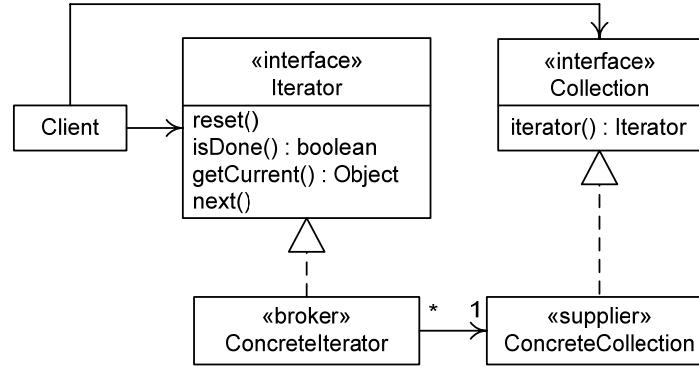


Figure 17-1-3 Iterator as a Broker Pattern

The Client holds a reference to the ConcreterIterator by way of the Iterator interface. The diagram includes several elaborations of the basic broker pattern structure, including a reference from the Client to the ConcreteCollection and two interfaces, but it still matches the general form of the broker pattern.

The Iterator pattern dynamic model may seem quite different from the broker pattern model, but a closer look reveals that the Iterator pattern does exhibit broker behavior. Consider the diagram in Figure 17-1-4.

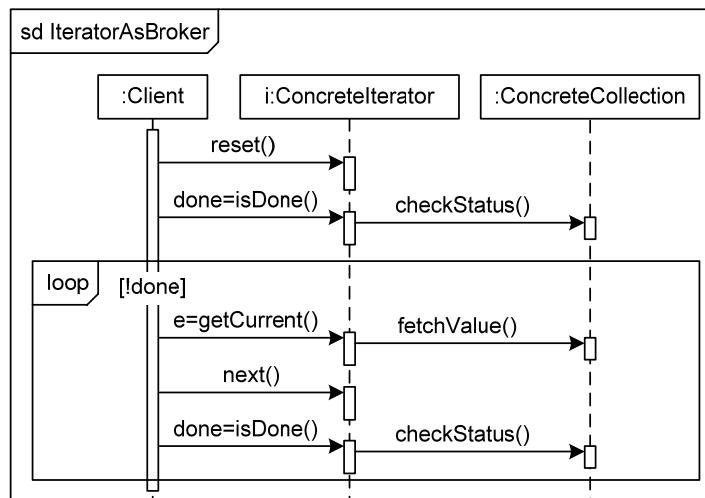


Figure 17-1-4 Iterator Broker Pattern Behavior

This diagram is a modification of Figure 16-2-2 that suppresses Iterator instance creation and destruction and elaborates the interaction between the Iterator and the ConcreteCollection that is suppressed in Figure 16-2-2.

Figure 17-1-4 emphasizes how the `Iterator` meets `Client` requests by obtaining data from the `ConcreteCollection`, thus serving a broker role. The `checkStatus()` and `fetchValue()` operations represent collection operations that might be called during interaction between an iterator and its associated collection.

An iterator brokers the interaction between a client and a collection to augment the service provided by the collection without complicating the collection's interface and design. The `Iterator` pattern is thus an example of a broker pattern used to simplify the supplier.

Pattern Summary

Figure 17-1-5 summarizes the broker pattern category.

<p>Name: Broker Category</p> <p>Application: Provide a class to mediate between a client and a supplier as a means to simplify or decompose the supplier or to facilitate communication between the client and the supplier.</p> <p>Form: Broker patterns have three classes: a client, a broker, and a supplier. The client interacts with the broker, which in turn interacts with the supplier on the client's behalf.</p> <p>Consequences: Broker patterns simplify or decompose suppliers, or ease communication between them. Broker patterns thus increase changeability, modifiability, and reusability. By interposing additional processing, they may degrade performance.</p>
--

Figure 17-1-5 The Broker Category

Section Summary

- **Broker patterns** have a broker that mediates interactions between a client and a supplier.
- Broker patterns are used to facilitate client-supplier interaction, decompose suppliers, and simplify suppliers.
- The `Iterator` pattern is a broker pattern: The iterator is a broker and the traversed collection is the supplier.
- The `Iterator` pattern is used to simplify the supplier.

Review Quiz 17.1

1. What access or navigation relationships are vital between objects participating in a broker pattern?
2. In the `Iterator` pattern, an iterator is a broker used to simplify a supplier (the associated collection). How does the iterator simplify its associated collection class?

17.2 The Façade and Mediator Patterns

Facilitating Interaction with Façades

The **Façade pattern** is a broker pattern that eases interaction between a client and a sub-system of suppliers by providing a simpler interface to the sub-system. Sub-systems may contain many classes with complex interfaces

and relationships, but often clients only need basic services that can be supplied through a simple interface. The broker class, called a *façade*, provides basic, simplified services to clients by taking upon itself the job of dealing with a complex sub-system. Clients needing more flexible and powerful access to the sub-system can still access it without using the façade.

An Analogy

A travel agent provides a façade to the travel industry, which is a large, complex, and intricately interrelated world. Many travelers are content to let their travel agents arrange trips for them, but more intrepid souls are still free to make their own arrangements with airlines, cruise lines, train systems, rental car agencies, and hotels.

Façade Pattern Structure

The Façade pattern adds one or more classes to a complex sub-system. These classes contain operations for simplified use of the sub-system but do not restrict access to the sub-system for clients who need it.

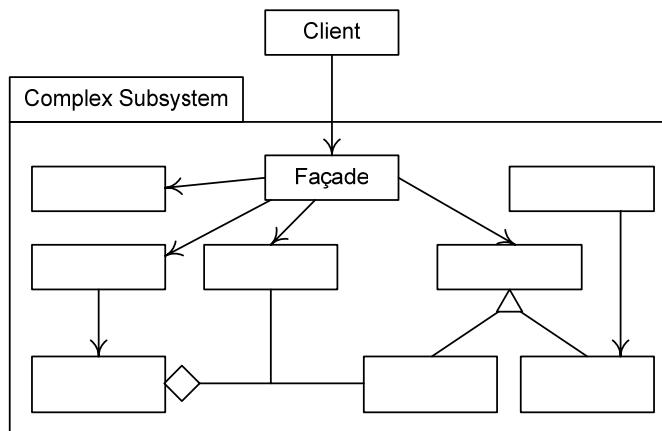


Figure 17-2-1 Façade Pattern Structure

In Figure 17-2-1 the Façade class is a broker that mediates the interaction between the Client class and the many supplier classes in the sub-system. The Façade pattern has a broker pattern form.

Façade Pattern Behavior

Façade class operations do nothing more than delegate activities to other portions of the sub-system to which they provide an interface. The details of such delegations depend on the sub-system in question. In all cases, however, the behavior follows the general outline of a broker pattern interaction.

Some Examples

A graphical user interface sub-system is a complicated thing. The portion of a system mainly concerned with the application domain (and the

programmers of that portion of a system) can interact with the GUI sub-system in all its intricacies. However, a much better approach is to form a GUI sub-system façade for application domain interaction.

A memory management sub-system in an operating system, or in an application program that needs to manage its own memory, will typically be quite complex. Memory management systems must keep track of allocated portions of memory. Allocation and deallocation schemes may be quite complicated, particularly if garbage collection or relocation of blocks is used. Most clients of memory management systems only need to obtain and release blocks of memory, so a simple façade supplying these services is advisable.

When to Use the Façade Pattern

Besides providing a simplified interface to a complex sub-system, the Façade pattern confers another advantage: Clients accessing a sub-system through a façade are decoupled from the internal classes of the sub-system. The structure of the sub-system may thus be altered radically without affecting clients. The Façade pattern is therefore a good candidate for situations in which either simplified access to a complex sub-system or decoupling from sub-system internals is desirable, or both.

There are many situations where one or both of these goals are desirable. In Layered-style architectures, for example, each layer is supposed to be decoupled from all other layers so that it can be modified or replaced without affecting the rest of the system. Also, layers tend to be rather complex. Using a façade to implement each layer's interface helps decouple layers and provides services at the right level of abstraction and complexity for each layer, no matter how the layers are implemented internally.

Facades can also be very useful when sub-systems are reused. The façade can provide an interface providing the correct abstractions and complexity for the system into which the reused sub-system is placed. Note that the Façade pattern in this respect is like the Adapter pattern, discussed in the next section.

Mediation and Multiple Façades

Sometimes objects collaborate in complex ways that resemble a negotiation, with each object having to exchange messages with several others. This increases coupling and often makes for complex and fragile code in the participating classes. The **Mediator pattern** provides a means of reducing coupling and simplifying code by controlling and coordinating interaction. In the Mediator pattern, rather than interacting with one another, collaborating objects interact *only* with a special mediator class. The objects are coupled only to the mediator, which contains all the code for coordinating the collaboration. The difference between unmediated and mediated collaboration is illustrated in the schematic diagrams in Figure 17-2-2.

The upper portion of Figure 17-2-2 shows four collaborating objects (as boxes), their links (as lines between them), and some of the messages flowing between them (as arrows). The bottom diagram shows the same

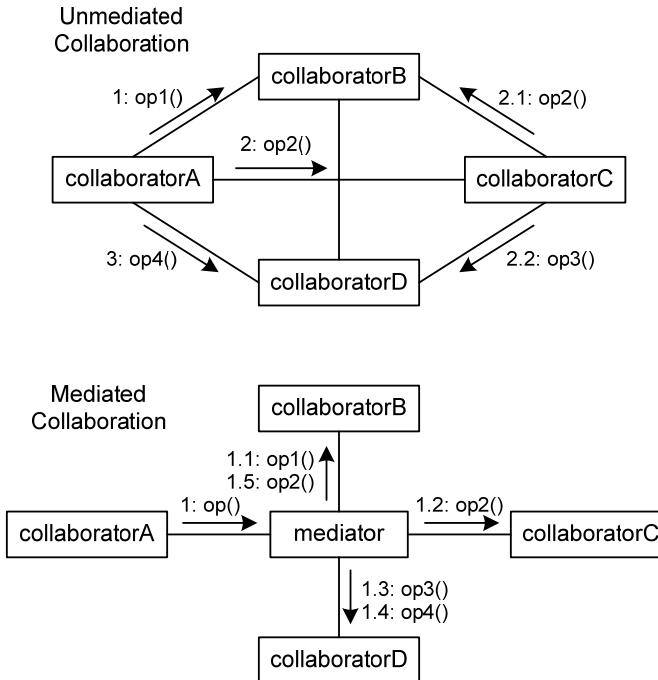


Figure 17-2-2 Unmediated and Mediated Collaborations

collaborating objects and a **mediator** object that coordinates the interaction. Note how each collaborator is linked only to the **mediator** and how the message flow is directed by the **mediator**.

Another way to think about the **Mediator** pattern is as a multi-way Façade pattern. From the point of view of a single collaborator, which we can regard as a client, the other objects with which it interacts form a complex sub-system. The **mediator** class provides a simplified interface to this sub-system and decouples the client from it. But each collaborator is in the same boat: Each can regard the **mediator** as a façade decoupling it from, and simplifying its interactions with, a complex sub-system.

An Analogy

The **Mediator** pattern is akin to many situations in everyday life where many individuals must coordinate their activities or negotiate some outcome, and it is much easier for one individual to mediate the interaction than for each individual to interact with all the others. Consider, for example, the problem of scheduling a meeting for a group of people. A free time must be found for each person and a meeting place arranged. Every person could negotiate with every other and with the agency responsible for room scheduling, and eventually a meeting time could be hashed out. However, it is much easier for a single person, who may not even be going to the meeting, to take charge of this task. Each meeting participant and the room allocator need only interact with the **mediator**.

Mediator Pattern Structure

The Mediator pattern has a **Mediator** class that is associated with every collaborator. Each collaborator must know about the **Mediator**. The connection from all collaborators to the **Mediator** can be captured in a **Collaborator** super-class; its sub-classes are called “colleagues” in Figure 17-2-3.

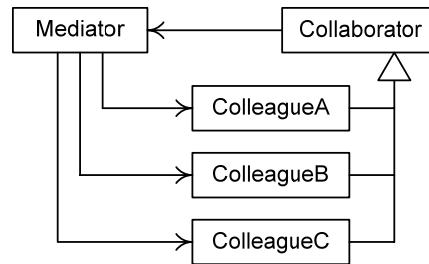


Figure 17-2-3 Mediator Pattern Structure

Three colleague classes are shown in this diagram; the number varies depending on the application in which the pattern is used.

The Mediator pattern is a broker pattern, though it does not look like one in Figure 17-2-3. The client and supplier roles are played by different collaborators at different times. The diagram in Figure 17-2-4 rearranges the classes and adds role stereotypes to make the pattern’s classification as a broker pattern clear.

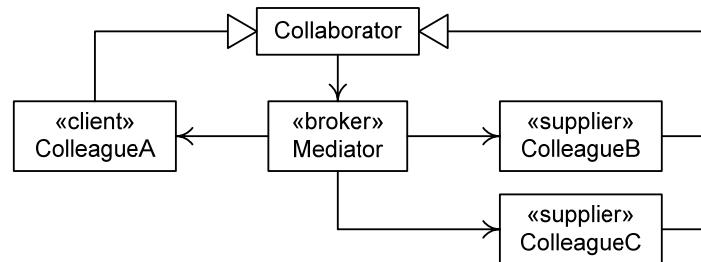
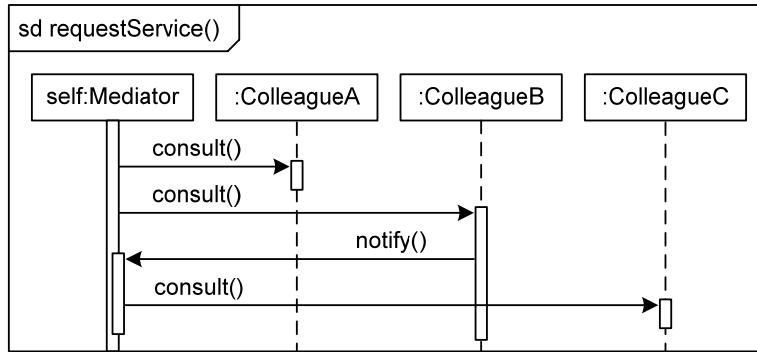


Figure 17-2-4 Mediator as a Broker Pattern

In Figure 17-2-4, ColleagueA is acting as a client and ColleagueB and ColleagueC as suppliers. ColleagueA has a navigation arrow to the **Mediator** inherited from its **Collaborator** parent. All the classes and relationships characteristic of a broker pattern are present.

Mediator Pattern Behavior

Interaction is begun by one of the collaborators or a client object outside the collaboration. In either case, the **Mediator** object directs the collaboration, with all communication going between it and the colleagues in the interaction. A schematic view of Mediator pattern behavior is shown in the sequence diagram in Figure 17-2-5.

**Figure 17-2-5 Mediator Pattern Behavior**

As in any broker pattern, the **Mediator** first receives a request for service. It then controls an interaction with the collaborating colleagues to supply the service. If one collaborator needs help from another, it notifies the **Mediator** object, which obtains the needed service from another collaborator. In Figure 17-2-5, three collaborating objects are shown, though in any particular application of the pattern there may be fewer or more.

A Mediator Pattern Example

Let's turn to a more concrete example to illustrate the Mediator pattern. Boggle is a word game played on a 4x4 grid of letter tiles. The letter tiles are placed in the grid at random, and players have three minutes to spell out words from the grid. The letters in words must appear on adjacent tiles (including diagonally adjacent), and no tile can be used twice in the same word. Players are awarded points for each word they find that is at least three letters long and is not found by any other player.

Suppose we are designing a program to play Boggle over the Internet with human and computer players, and we are considering the collaboration that must occur to score the set of words generated by a player. Each player is represented by an instance of a **Player** class. The game grid is an instance of a **Board** class. Also, there is a **Dictionary** class used to ensure that words proposed by players are legitimate.

Implementing the scoring collaboration without a mediator could be done as pictured in Figure 17-2-6.

This sequence diagram shows one way to realize the scoring collaboration in a game with three players. When a **Player** object is asked for its score, it can loop through all the words in its found word set and complete the following actions for each one:

1. Make sure it is really on the grid by asking the **Board** object to check it, realized by the call of `isOnBoard()`.
2. Make sure it is really a word by asking the **Dictionary** object to check it, realized by the call of `contains()`.

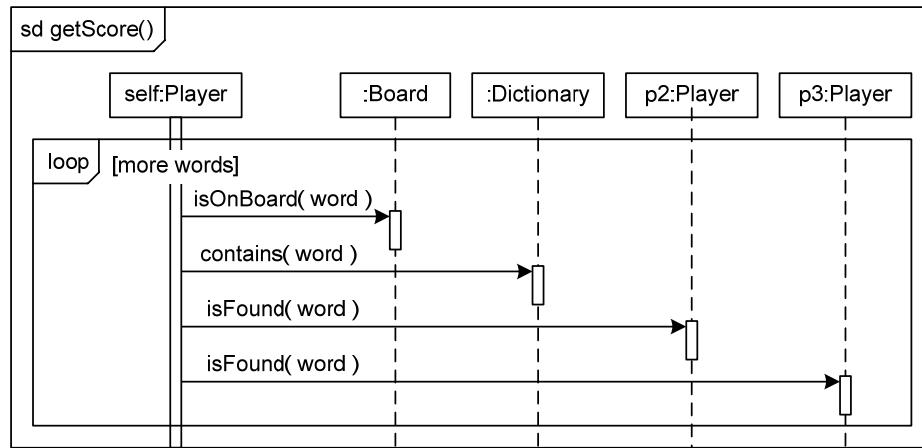


Figure 17-2-6 Boggle Scoring Without a Mediator

3. Ask the other **Player** objects if they found the word. (Remember that only words found by a single player score points.) This part of the job is done by multiple calls of the **isFound()** operation.

Note that the message traffic in this diagram demands that each **Player** hold references to the **Board**, the **Dictionary**, and every other **Player**. Furthermore, the scoring algorithm is built into the **Player** class, complicating the class and saddling it with a responsibility that might be better placed elsewhere.

Incidentally, there are many other ways this collaboration might be done. For example, the **Board** object might be responsible for vetting words by both checking that they are on the grid and that they are in the **Dictionary**. In every alternative without a mediator, however, the collaborators must hold references that couple them to one another, and the scoring algorithm is spread among the collaborators.

Applying the Mediator pattern to this problem introduces a mediator object in charge of scoring. The sequence diagram in Figure 17-2-7 on page 520 illustrates one way to use this pattern.

In this model, the mediator (**Scorer**) is told to compute the scores. It gets each **Player**'s word set, validates all words with the **Board** and **Dictionary**, and checks the sets against one another to compute each **Player**'s score. The **Scorer** then tells each **Player** its score. In this design, the **Scorer** holds references to all the collaborators, but none of them hold references to one another—they do not even need references to the **Scorer**. The collaborators are completely decoupled, and the entire scoring algorithm is centralized in the **Scorer**.

As with the design alternative without a mediator, there are many other ways the Mediator pattern could be applied. For example, the **Scorer** could compute each **Player**'s scores separately in response to a request by each

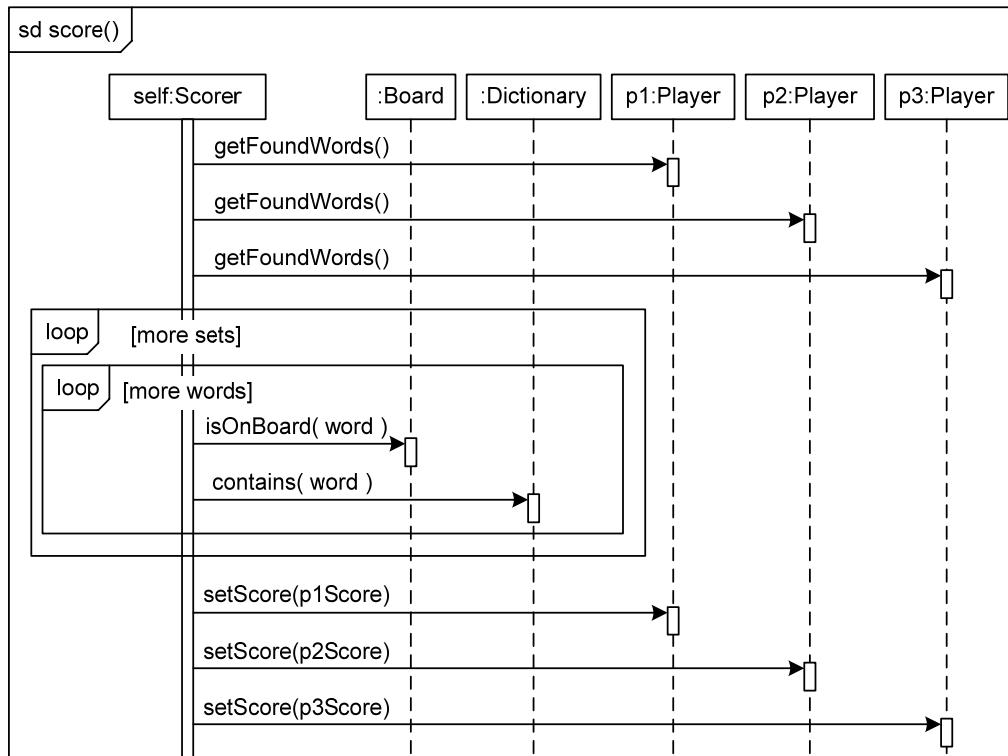


Figure 17-2-7 Boggle Scoring with a Mediator

Player to do so. In all cases, however, using the Mediator pattern decouples collaborators and moves the scoring algorithm into the mediator class.

We can summarize the differences between these designs:

- The design without a mediator requires each Player to keep references to its peers and to the Board and Dictionary objects. In the design with a mediator, no Player needs references to any other objects at all. The Scorer needs references to each Player and to the Board and Dictionary objects.
- The design without a mediator places the scoring code in the Player class. This code complicates the Player class and does not really belong there. The design with a mediator moves all this code into the Scorer class, simplifying the Player class and increasing its cohesion.

The design with the mediator has lower coupling and better cohesion. The Player class is more reusable with the Mediator pattern. Finally, if the scoring algorithm changes, the Player, Board, and Dictionary classes can be left alone and only the Scorer class modified.

When to Use the Mediator Pattern

Mediators should be used to encapsulate any complex interaction between several collaborators. The Mediator pattern has the following advantages:

- It decouples collaborators, making them more changeable and reusable.
- It centralizes control of an interaction in the mediator class, making it easier to change, thus increasing modifiability.
- It simplifies the collaborators, making them easier to understand, and hence to change. It may also increase collaborator cohesion.

The only potential drawback of the Mediator pattern is that forcing collaborator interaction through the mediator may compromise performance.

Mediators, Façades, and Control Styles

Control styles are described in Chapter 12 as ways that decision making is distributed among program components. Three control styles are distinguished there: centralized, delegated, and dispersed. The Façade and Mediator patterns provide models for making control less distributed and more centralized—they provide means for moving from a dispersed to a delegated style, or from a delegated to a centralized style.

In Chapter 12 we stressed that an over-centralized style can lead to bloated controllers, while a dispersed style can lead to programs with control so spread out that they are hard to understand and change. These considerations should be kept in mind when applying the Façade and Mediator patterns.

Apply the Façade and Mediator patterns to move from a dispersed to a delegated control style, but not to move from a delegated to a centralized style.

Pattern Summary

Figures 17-2-8 and 17-2-9 summarize the Façade and Mediator patterns.

Name: Façade

Application: Add an interface class to a sub-system.

Form: The façade class is a broker that mediates the interaction between a client class and a collection of supplier classes in a sub-system.

Consequences: The façade class makes a sub-system easier for clients to use, lowers the coupling between the client and the sub-system, and may increase reusability by adapting a sub-system's interface to client needs.

Figure 17-2-8 The Façade Pattern

Name: Mediator

Application: Add an intermediate class to control the interaction of several collaborating classes.

Form: The mediator class is a broker that decouples several collaborating classes and facilitates their interaction. Each collaborator holds a reference to the mediator, which holds a reference to each collaborator. Collaborator interaction goes through the mediator.

Consequences: The mediator class decouples collaborating classes, making them more reusable and changeable. It encapsulates an interaction, making the interaction easier to change.

Figure 17-2-9 The Mediator Pattern

Section Summary

- The **Façade pattern** is a broker pattern used to provide a simple interface to a complex sub-system without restricting access to the sub-system for clients who need it.
- The Façade pattern also helps decouple sub-systems from one another.
- Façades are useful when many clients need a simple interface to a complex sub-system, in building layer interfaces in a Layered-style architecture, and in adapting a sub-system interface to client needs.
- The **Mediator pattern** is a broker pattern used to encapsulate control of an interaction and decouple collaborators.
- Mediators are useful when several objects carry out a complex interaction; they decrease coupling and increase reusability, changeability, and cohesion.

Review Quiz 17.2

1. State an analogy, different from the one in the text, for the Façade pattern.
2. Name three advantages of using the Façade pattern.
3. State an analogy, different from the one in the text, for the Mediator pattern.
4. Explain how a mediator can be thought of as a multi-way façade.
5. What kind of control styles does the Mediator pattern encourage?

17.3 The Adapter Patterns

Adaptation and Reuse

Reuse has many advantages but sometimes can be difficult. One standard reuse problem occurs when a component has the wrong interface for a system in which it might be reused. Interface incompatibility problems range from slight inconveniences, such as not having the right operation signatures, to serious deficiencies, such as lacking crucial operations or not being thread-safe.

One way to solve such interface incompatibility problems is to introduce a class to broker the interface between a client and a reusable class with the wrong interface. This sort of broker pattern facilitates interaction between a client and a supplier by providing a new interface for the supplier.

An **adapter** or **wrapper** is a component that provides a new interface for an existing component. An **Adapter** or **Wrapper pattern** is a broker pattern that provides a new interface for existing software so that it can be reused. Adaptation for reuse is an old technique that has been used since the beginning of software development. The Adapter patterns provide object-oriented adapters in two varieties: One uses inheritance and one uses delegation.

An Analogy The Adapter patterns are well named because the term “adapter” brings to mind examples of physical devices analogous to software adapters.

Electrical adapters fit into the electrical outlets standard in one part of the world and convert the current and provide an outlet fitting plugs used in a different part of the world. Plumbing adapters make the switch from one pipe’s diameter and threading to another’s.

Adapters provide a new interface to something whose inherent interface makes it unusable in some context. Sometimes adapters add functionality; for example, an electrical adapter may have an on-off switch. These are the essential characteristics of the software adapter patterns.

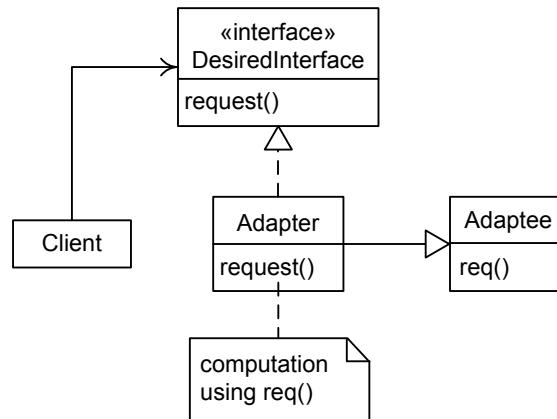
Class and Object Adapter Patterns A class (the *adaptee* class) may be given a new interface by an *adapter* class in two ways:

- The adapter may sub-class the adaptee. The adapter can inherit adaptee operations with appropriate semantics and pragmatics, override those with inappropriate semantics or pragmatics, and add operations needed for the new interface. This is the **Class Adapter pattern**.
- The adapter may hold a reference to the adaptee and delegate most work to the adaptee object. This approach is the **Object Adapter pattern**.

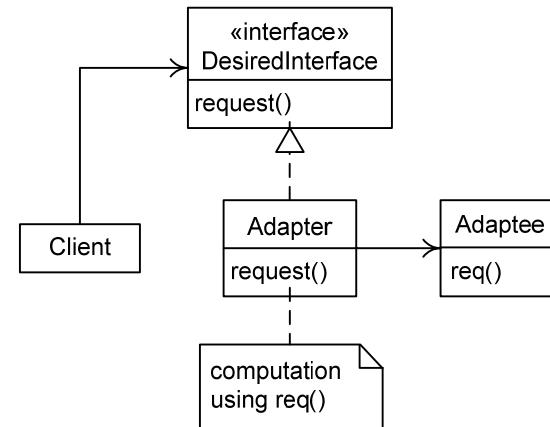
These are the two adapter patterns. Between them, they provide the means necessary to give virtually any class a new interface.

Adapter Pattern Structure The Class and Object Adapter patterns are structurally very similar. The class diagram in Figure 17-3-1 illustrates the Class Adapter pattern. The **Adapter** class realizes the **DesiredInterface** by implementing the **request()** operation. This implementation reuses the **req()** operation that has been inherited from the **Adaptee** class.

The **Adapter** class is the broker and the **Adaptee** is the supplier when this pattern is viewed as a broker pattern. Note that the broker and supplier collapse into a single class because of the inheritance relationship between the **Adapter** and **Adaptee** classes. The **Client** accesses the **Adapter** through **DesiredInterface**, and the **Adapter** can access the **Adaptee** because the **Adapter** is a sub-class of the **Adaptee**. Thus, the Class Adapter pattern has the static structure of a broker pattern.

**Figure 17-3-1 Class Adapter Pattern Structure**

The only real difference between the static structures of the Class Adapter and Object Adapter patterns is that in the latter the **Adapter** class is not a sub-class of the **Adaptee** class. Instead, the **Adapter** class holds a reference to an **Adaptee** object and delegates work to it in the **Adapter**'s realization of the **DesiredInterface**. Compare the class diagram in Figure 17-3-2 illustrating the Object Adapter pattern to the one in Figure 17-3-1 for the Class Adapter pattern.

**Figure 17-3-2 Object Adapter Pattern Structure**

Both of these Adapter patterns clearly fit the static structure of a broker pattern.

Adapter Pattern Behavior

The Adapter patterns behave as one would expect of broker patterns. The diagram in Figure 17-3-3 illustrates how the Object Adapter pattern handles a request.

Because an **Adapter**'s job is to change an interface, it rarely handles requests from a **Client** on its own—it merely passes them along to the **Adaptee**.

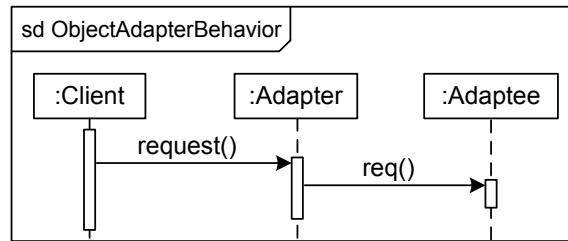


Figure 17-3-3 Object Adapter Pattern Behavior

An Adapter Pattern Example

Many classes are designed with no attempt to make them safe in a multi-threaded environment, either because their designers did not consider the possibility or because they purposely did not do so for reasons of efficiency and ease of implementation. Such classes often cannot be reused in a concurrent program. However, they may be easily adapted for such use.

For example, suppose a Java program helps run a small engine repair business by keeping track of repair jobs in a priority queue. The class diagram and skeleton Java program code in Figure 17-3-4 illustrate the **PriorityQueue** class in this program.

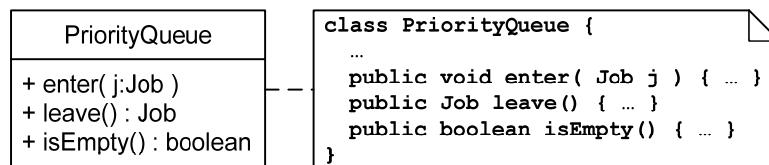


Figure 17-3-4 A PriorityQueue Class

Such a class does not need to be thread-safe in a sequential program used only by the shop manager. But if the business implements a concurrent program serving a Web site that allows various stakeholders (such as customers, repairmen, and managers) to submit jobs, see job statuses, reset job priorities, and so forth, the class as implemented in Figure 17-3-4 will lead to bugs. For example, suppose that two users controlling two threads in the program enter jobs into the priority queue at almost the same time. One thread could be halfway through executing the **enter()** operation when the other thread also starts executing the **enter()** operation. Both threads manipulate the data structure implementing the priority queue. They are likely to interfere with one another, corrupting the data structure and causing the program to fail. The program must be changed to prohibit more than one thread from manipulating the **PriorityQueue** data structure at the same time, making the program thread-safe.

A Java object may be made thread-safe using either the Class or Object Adapter patterns, depending on the characteristics of the adaptee class. If the adaptee class has no changeable public attributes, it can be sub-classed, and its methods can be overridden, then a thread-safe class adapter can be made by extending the sequential adaptee class and overriding all its methods that alter the class with thread-safe methods. This will guarantee that only one thread may change the priority queue at one time, ensuring that it will not be corrupted. For example, the `PriorityQueue` class in Figure 17-3-4 can be made thread-safe using a class adapter, as shown in Figure 17-3-5.

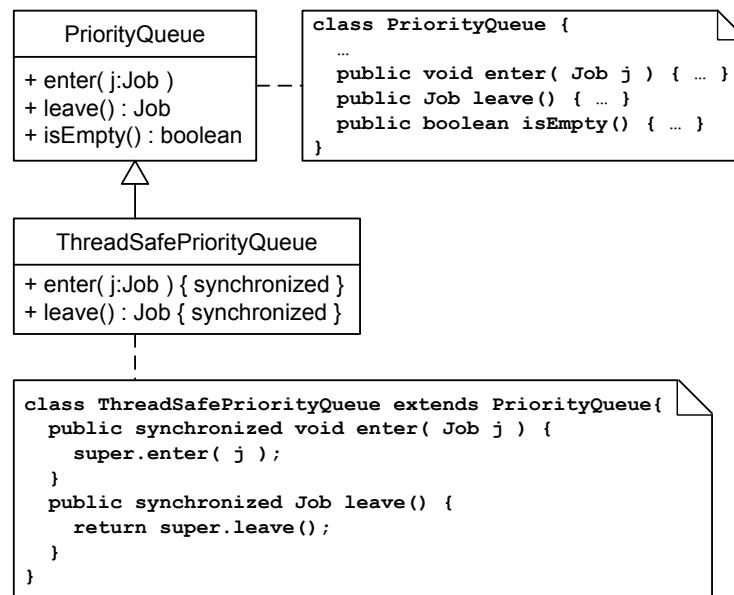


Figure 17-3-5 Thread-Safe Class Adapter

When an operation is *synchronized* in Java, only one thread may execute that operation, or any synchronized operation of the object, at one time. The operations that change the priority queue data structures in the adapter sub-class (`enter()` and `leave()`) override the super-class operations and are synchronized, but all they do is delegate the work to the super-class operations. The `isEmpty()` operation does not change the data structure, so it does not need to be synchronized.

If the adaptee has changeable public attributes, it cannot be sub-classed, or its methods cannot be overridden, then the Object Adapter pattern must be used. In this case, a class with the appropriate interface can be created, and it can create and hold an instance of the adaptee class. The adapter methods must be made thread-safe, and the adapter must hide any public attributes of the adaptee, providing access through thread-safe access methods. For example, the `PriorityQueue` class can be wrapped with a thread-safe object adapter, as shown in Figure 17-3-6.

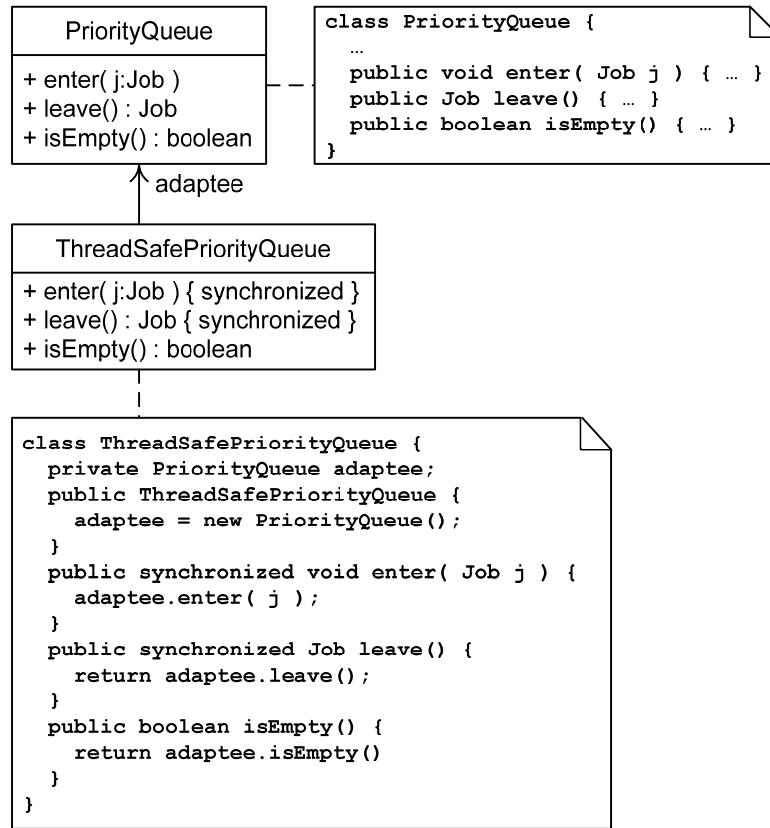


Figure 17-3-6 Thread-Safe Object Adapter

As when using the Class Adapter pattern, the `enter()` and `leave()` operations in the adapter are synchronized, while the `isEmpty()` operation is not. The Object Adapter pattern requires that all operations be implemented in the adapter because none are provided through inheritance. The adapter must also create an instance of the adaptee so it has something to delegate to.

Java uses adapters in its standard class libraries to make collection classes thread-safe. Most of the Java collection classes are not thread-safe, but special factory methods in `java.util.Collections` produce thread-safe collection classes by wrapping them as described in this section.

When to Use the Adapter Patterns

Adapters should be used whenever a client needs a different supplier class interface. This need can arise for the following reasons:

- The current context of use expects a certain interface. This is the main circumstance in which an adapter is used. For example, the `java.util.Collections` class has operations to produce an unmodifiable collection from a modifiable collection. The unmodifiable collection objects are adapters. In the adapters, all operations that change a collection throw an `UnsupportedOperationException`, while

all query operations are delegated to the adaptee—that is, the modifiable collection. In this case the adapter changes the adaptee's interface by removing several operations from it.

- A simplified interface is needed. The Façade pattern is used to provide a simplified interface to a complex sub-system, and the Adapter patterns can be used to provide a simplified interface to a complex class.
- Operations with slightly different functionality are needed. Very often, an interface will not work because functionality has been packaged slightly differently than is needed, or some functionality is missing. It is usually easy to repackage or add functionality in an adapter. The Java thread-safe collection classes are an example of this reason for using an adapter. The thread-safe collection classes simply add synchronization functionality to the collection operations that need it.

If part of an adaptee's interface is reusable and part is not, then the Class Adapter pattern will be easiest to use. The adapter sub-class can override the portions of the adaptee super-class's interface that do not fit, and it can add missing operations as necessary. The Class Adapter pattern cannot block access to adaptee attributes and operations, however, which is sometimes desirable.

Sometimes the Object Adapter pattern must be used rather than the Class Adapter pattern, even though it will typically require more work to implement. For example, if the entire adaptee interface needs to be replaced, several adaptee classes must be used, the adaptee cannot be sub-classed, or some aspect of the adaptee's behavior must be completely shielded, then the Object Adapter pattern must be used. The Object Adapter pattern may be somewhat less efficient than the Class Adapter pattern, however, because every operation must make additional operation calls to delegate work to the adaptee object.

Pattern Summary

Figures 17-3-7 and 17-3-8 summarize the Adapter patterns.

Name: Class Adapter

Application: Change another class's interface to meet the needs of a client class.

Form: The adapter is a sub-class of an adaptee and overrides, removes, or adds operations as necessary to meet a client class's needs. The adapter acts as a broker that mediates the interaction between a client class and the adaptee super-class.

Consequences: The Class Adapter pattern promotes reuse by resolving interface incompatibilities between clients and adaptees. It can also add functionality to an adaptee or simplify an interface like the Façade pattern does. Sometimes this pattern cannot be used; for example, when the adaptee cannot be sub-classed or some of its operations cannot be overridden.

Figure 17-3-7 The Class Adapter Pattern

Name: Object Adapter
Application: Change another class's interface to meet the needs of a client class.
Form: The adapter can access the adaptee and provides the interface necessary to meet a client class's needs. The adapter delegates most of the work to the adaptee, doing only what it must to provide the interface that the client needs. The adapter acts as a broker that mediates the interaction between a client class and an adaptee class.
Consequences: The Object Adapter pattern promotes reuse by resolving interface incompatibilities between clients and adaptees. It can also add functionality to an adaptee or simplify an interface like the Façade pattern does. This pattern can always be used, but it may be a lot of work to create the adapter if the interface is large or complex.

Figure 17-3-8 The Object Adapter Pattern

- Section Summary**
- The **Adapter patterns** wrap a class to provide it with a new interface so it can be reused in a different context.
 - The adapter class may sub-class the adaptee, replacing part of its interface. This is the **Class Adapter pattern**.
 - The adapter class may access the adaptee and completely replace its interface, though delegating most work to the adaptee. This is the **Object Adapter pattern**.
 - The Adapter patterns should be used when a client expects a particular interface, when a simplified interface is needed, or when only slightly changed functionality is needed.

- Review Quiz 17.3**
1. What roles do the classes in the Class and Object Adapter patterns play when they are viewed as broker patterns?
 2. Can a single adaptee be given two new interfaces simultaneously?
 3. Suppose a class has 15 operations, of which three must be changed to satisfy the needs of a client. Assuming the class can be sub-classed, would the Class or Object Adapter pattern be preferable?

17.4 The Proxy Pattern

- Stand-Ins** Some objects are expensive to create because they are large or require a great deal of computation, are difficult or expensive to use because they are in a remote location, or need protected access for security or safety reasons. In such cases, a stand-in for the object may be used that
- Has exactly the same interface as the real object,
 - Handles routine or illegitimate messages without accessing the real object, and
 - Delegates messages that it cannot handle to the real object (possibly creating it if necessary).

Such a stand-in is called a *proxy* or a *surrogate*. The **Proxy pattern** is the mid-level design pattern that models these situations.

Stand-ins for objects not yet created are called *virtual proxies*. Local stand-ins for non-local objects are called *remote proxies*. Stand-ins that control access to other objects are called *protection* or *access proxies*. Other sorts of proxies are possible but have no special names.

An Analogy

A manager called to a meeting he or she is unable to attend may send a subordinate to the meeting in his or her place. Such a substitute is a good analogy for a proxy object. Someone standing in for a supervisor will play the role of the supervisor in the meeting in most ways and will be able to answer questions, make or refuse commitments, and represent the group's point of view on his or her own. However, in some cases the stand-in must contact the supervisor to get information or authorization before proceeding. A proxy acts in much the same way. It interacts with the rest of the system just like the object it represents and can often respond to messages on its own, but in certain cases, it must delegate the message to the object it represents.

Proxy Pattern Structure

A proxy must have the same interface as the real object that it represents. In the class diagram in Figure 17-4-1, both the *ProxySupplier* and the *RealSupplier* that it represents implement the same *Supplier* interface. A proxy must also maintain a reference to the real object that it represents so that it can forward messages that it cannot handle to the real object.

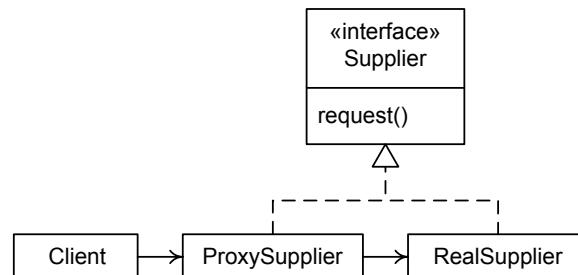
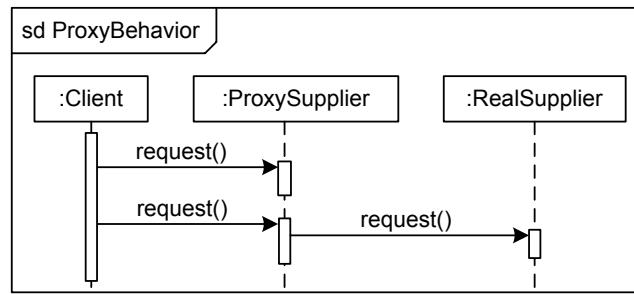


Figure 17-4-1 Proxy Pattern Structure

This pattern clearly has the broker pattern structure, with the *ProxySupplier* acting as a broker that either facilitates the interaction between the *Client* and the *RealSupplier* or augments the function of the *RealSupplier*.

Proxy Pattern Behavior

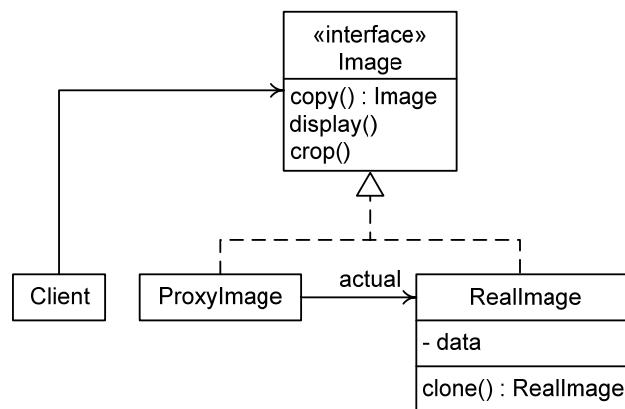
A proxy receives all messages intended for the real object that it represents, thus hiding from clients the fact that a proxy is involved. If the proxy can handle a request, it does so; if not, it forwards the request to the real object that it represents. This behavior is illustrated in Figure 17-4-2.

**Figure 17-4-2 Proxy Pattern Behavior**

The first time the **ProxySupplier** receives a request from the **Client**, it handles the request itself. The second time it receives a request, the **ProxySupplier** decides it cannot handle the request and forwards it to the **RealSupplier**. This behavior also conforms to the broker pattern behavioral paradigm.

A Proxy Pattern Example

Suppose you are writing an image processing application that must store many images; for example, it might be an image editor or an animation program. Copying images takes a lot of time and images use a lot of memory, so the program should avoid unnecessary copying. One way to do this is to have an image proxy created when an image is copied. The image proxy does not hold the image, but it keeps a reference to the original image and uses it until either the proxy or the original are told to do something that changes the image. At this point the image proxy creates a new copy of the original. The advantage of this strategy is that images are copied only when necessary—in this case only when the copy becomes different from the original. This strategy is called *copy-on-write*, and it is often employed in applications that manipulate large objects. Figure 17-4-3 shows the classes involved in this example.

**Figure 17-4-3 Image Proxy Structure**

To illustrate how this scheme works, suppose that a **Client** object has a **ReallImage** object and makes a copy of it using the `copy()` operation. The `copy()` operation returns a new **ProxylImage** object with a reference to the original **ReallImage** object. The situation is then as shown in the object diagram in Figure 17-4-4.

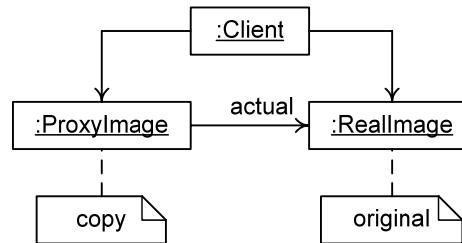


Figure 17-4-4 An Original Image and Its Proxy Copy

If the **Client** instance calls the `ProxylImage.display()` operation, the `ProxylImage` instance calls the original `ReallImage.display()` operation. However, if the `ProxylImage.crop()` operation is called, which would change the image, the `ProxylImage` instance calls `ReallImage.clone()` to obtain another `ReallImage`. It then calls the `ReallImage.crop()` operation to alter the image data in the clone object. The configuration of the objects after a call of `ProxylImage.crop()` is shown in Figure 17-4-5.

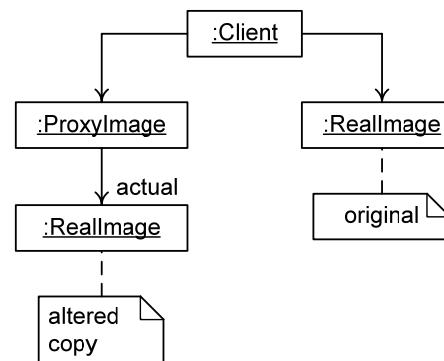


Figure 17-4-5 An Original and an Altered Copy

Note that the **Client** object is not involved in the manipulations that occur when a new `ReallImage` object is created. The **Client** object merely interacts with `Image` objects, never knowing or caring whether the `Image` objects it sends messages to are `ProxylImage` or `ReallImage` objects.

This example illustrates the use of a virtual proxy that saves time and memory by deferring creation of real suppliers until they are needed: The `ProxylImage` delays creation of a new real supplier (a `ReallImage`) until it is required.

**When to Use
the Proxy
Pattern**

The Proxy pattern should be used whenever the services directly provided by some object need to be managed or mediated in some way without changing the object's interface. If the interface must be changed, then one of the other broker patterns should be used.

Virtual proxies can be used to delay the creation or loading of large and time-consuming objects to preserve space and ensure rapid responses to requests. A virtual proxy can be used to start a new thread to create an object in the background. Virtual proxies can also save space and time by arranging to delay copying of objects until a copied object is actually changed, as in the previous example. This can increase efficiency by not making copies of large objects that are never changed.

Remote proxies can hide the fact that an object is not locally present, handling the communication necessary to access the remote real object and hiding this communication from clients.

Protection proxies can ensure that only authorized clients access real objects, that they are accessed in legitimate ways, and that only one client at a time attempts to change an object.

**Pattern
Summary**

Figure 17-4-6 summarizes the Proxy pattern.

Name: Proxy

Application: Provide a stand-in for another object that is not yet created, is not locally available, has access restrictions, or is unavailable for other reasons.

Form: The proxy class is a broker with the same interface as the supplier for which it stands in. The proxy eases the interaction between the client and the supplier it replaces or augments the function of the supplier. The proxy may handle some requests; it passes on those it cannot handle to the supplier.

Consequences: The Proxy pattern makes it possible to defer expensive operations until they are necessary (virtual proxies), provides an elegant way to treat remote objects as if they were local (remote proxies), and provides a mechanism for implementing supplier access restrictions (protection or access proxies).

Figure 17-4-6 The Proxy Pattern

**Section
Summary**

- Sometimes it is useful to have one object stand in for some other object—this is the idea behind the **Proxy pattern**.
- Use the Proxy pattern whenever some object's services need to be managed or mediated in some way without changing its interface.
- *Virtual proxies* stand in for objects that have not yet been created.
- *Remote proxies* stand in for objects that are not locally present.
- *Protection or access proxies* are used to provide secure or unique access to objects.

**Review
Quiz 17.4**

1. Give an example, different from the one in the text, of an analogy for the Proxy pattern.
 2. What is the relationship between the interfaces of a proxy class and the supplier class that it stands in for?
-

Chapter 17 Further Reading**All Sections**

All the patterns discussed in this chapter were first documented in [Gamma et al. 1995].

Chapter 17 Exercises

The following problem descriptions will be used in the exercises.

Indexing Sub-System

Web search engines and other sorts of information-retrieval systems use indexes to search for documents. These indexes are much like the indexes at the backs of books, with words keyed to the documents that contain them. Not every word is a good index term. For example, common words such as “the,” “of,” “in,” and so forth are useless as index terms.

Frequently or infrequently occurring words also tend not to be good index terms. They can be eliminated, or a term-weighting scheme can be used to assign a weight or importance to each word in a document. It often helps to “conflate” different word forms into a special index term. Conflation is usually done by finding the “stem” of a word, so it is called “stemming.” For example, documents containing the words “compute,” “computing,” “computer,” or “computation” often have the same topic, though one or another of these words may not appear in every document. They may be conflated into the special index term “comput.” The index terms of the document that a document references, or the index terms of documents that reference a given document, can be used to alter a document’s index terms or change their weightings.

As one might imagine, there is great variety in the way that an indexing sub-system might process a text to generate a list of index terms for it. Indexing sub-systems generally start with a list of all the words in a document. They may then remove common words using various lists, stem words using various algorithms, weight words using a variety of term-weighting schemes, and alter the index terms or their weights based on references to or from other documents. Indexing sub-systems are thus complex collections of algorithms for analyzing a document and creating a list of index terms for it.

CardShark

CardShark is a program for playing poker over the Internet with other people or with computer players that mimic humans. The essence of poker is gambling, so each player bets. Bets are placed in a pot taken by the

winner of the hand. Each player must begin a round by placing an ante (a small fixed amount) in the pot. During or after the cards are dealt or exchanged for other cards, depending on the game, betting takes place. Betting begins with the player to the left of the dealer and proceeds clockwise. Each player may call (meet the current bet), raise (increase the current bet), or fold (drop out of the hand). Betting continues until every player has called or folded since the last raise.

Suppose that the designers of CardShark have `Player` and `Pot` classes. Betting could be managed through the collaboration of all `Player` and `Pot` objects as follows:

1. The dealing `Player` sends an ante message to the `Pot` object and then tells the `Player` to its left to ante. The notified `Player` sends an ante message to the `Pot` object and then tells the `Player` to its left to ante. This continues until the dealing `Player` is told by the `Player` to its right to ante, which ends the ante process.
2. When it is time to bet, the dealing `Player` tells the `Player` to its left to bet. The notified `Player` calls, raises, or folds.
3. If a `Player` calls or raises, it sends a message adding the correct amount to the `Pot` and then notifies the `Player` on its left to bet, sending along the amount of the bet and who last raised.
4. If a `Player` folds, it tells the `Player` to its right to send future bets to the `Player` on its left.
5. When a `Player` is told to bet, it checks to see whether it was the last one to raise. If so, betting is done.

Section 17.1

1. *Fill in the blanks:* A broker pattern has a broker class that mediates the interactions between a _____ and a _____. Brokers can be used to simplify the _____, to decompose the _____, or to facilitate interactions between the _____ and the _____.
2. The text argues that the Iterator pattern is a broker pattern based on its use, static structure, and behavior. Take these considerations into account in deciding whether the internal iterator pattern discussed in Chapter 16's Exercises (the *Iteration Agent* pattern) is also a broker pattern.
3. Discuss the similarities and differences between broker patterns and the architectural design strategy of using a virtual device layer.

Section 17.2

4. The Façade pattern provides an interface to a sub-system analogous to the way that public attributes and operations provide an interface to a class. Classes have inaccessible attributes and operations as well, and it might be useful for sub-systems to have private components, with only the façade class made public. Discuss the mechanisms that Java provides to accomplish this for a sub-system (that is, to have a public façade class but inaccessible component classes).

5. Discuss the connection between the Façade pattern and the Layered architectural style.
6. Using the Façade pattern, propose a façade class for an indexing subsystem of the sort described at the beginning of the Exercises section.
7. Draw object diagrams showing the links between the objects needed to realize interactions described in the two design alternatives for Boggle scoring illustrated in Figures 17-2-6 and 17-2-7.
8. Apply the Mediator pattern to the interaction realizing betting in the CardShark program (described at the beginning of the Exercises section). Explain your solution in words, as done in the problem description.
9. Apply the Mediator pattern to the interaction realizing betting in the CardShark program (described at the beginning of the Exercises section). Make UML class diagrams illustrating the design alternative without a mediator outlined in the problem description and your design using the Mediator pattern.
10. Apply the Mediator pattern to the interaction realizing betting in the CardShark program (described at the beginning of the Exercises section). Make UML sequence diagrams illustrating the design alternative without a mediator outlined in the problem description and your design using the Mediator pattern for implementing the following collaborations:
 - (a) Initializing the Pot object with antes
 - (b) Making a round of bets
11. Discuss the advantages and disadvantages of using the Mediator pattern in the design of the interaction realizing betting in the CardShark program (described at the beginning of the Exercises section).

Section 17.3

12. Make a diagram like Figure 17-3-3 illustrating the dynamic behavior of the Class Adapter pattern.
13. What is the difference between the Class Adapter pattern and simple inheritance?
14. Discuss the Java primitive data type wrapper classes (such as `Integer`, `Float`, and `Character`) with respect to the Adapter patterns. How well do they fit these patterns? Does the Class or the Object Adapter pattern model them most closely?
15. Suppose that an adapter class must provide a new interface to an adaptee class, but the adapter object must be generated from the adaptee object at runtime. Can the Adapter patterns be used to solve this problem?
16. Make a list of circumstances under which it is impossible to use the Class Adapter pattern and the Object Adapter pattern must be used instead.

17. Java provides several collection classes that implement unordered collections of objects with no duplicates; that is, they conform to the `java.util.Set` interface. Mathematical sets have several standard operations that do not appear in these classes: namely, the union, intersection, and relative difference operations, though there are other operations that do the same thing as these. Suppose an application needs a `HashSet` with these missing operations.
- (a) Design a `MathHashSet` class with `union()`, `intersection()`, and `difference()` operations using the Class Adapter pattern. Document your design with a class diagram containing Java code in comment symbols.
 - (b) Design a `MathHashSet` class with `union()`, `intersection()`, and `difference()` operations using the Object Adapter pattern. Document your design with a class diagram containing Java code in comment symbols.
 - (c) Which design alternative is better, and why?
18. Java provides special collection wrappers to make collections unmodifiable in the `java.util.Collections` class. Suppose an application needs an unmodifiable `HashSet`. The `Set` operations of the unmodifiable class that would change it should throw an `UnsupportedOperationException`.
- (a) Design a `ConstantHashSet` class that cannot be modified using the Class Adapter pattern. Document your design with a class diagram containing Java code in comment symbols.
 - (b) Design a `ConstantHashSet` class that cannot be modified using the Object Adapter pattern. Document your design with a class diagram containing Java code in comment symbols.
 - (c) Which design alternative is better, and why?

- Section 17.4**
19. Both the Object Adapter pattern and the Proxy pattern have broker classes that delegate virtually all work to a supplier class. Explain the differences between these two patterns.
20. Based on the example of a virtual proxy illustrated by Figures 17-4-3 through 17-4-5, write Java-like pseudocode for the constructor and the `copy()`, `display()`, and `crop()` operations of the `ProxyImage` class, and write pseudocode for the `copy()` operation of the `RealImage` class.
21. The example illustrated in Figures 17-4-3 through 17-4-5 shows what happens when a client makes changes to an `ImageProxy` object. What happens if an `ImageProxy` object is created by copying a `RealImage` object, and then the original `RealImage` object is changed? In particular, how can the original `RealImage` object contact any proxies holding copies to notify them to make `RealImage` copies? Suggest at least two alternative ways to solve this problem and describe them using class diagrams and pseudocode.
22. List some of the ways that the various broker patterns are similar to one another or differ from one another. For example, proxies, façades, mediators, and adapters can all augment the functionality of the

suppliers, but that is not their main purpose. Iterators, on the other hand, exist only to provide a capability that the supplier does not have.

Research Projects

23. Do the `java.util.Collections` unmodifiable collection wrappers use the Class or Object Adapter patterns? Obtain the Java source code from <http://java.sun.com> to find the answer to this question. Could the other Adapter pattern have been used?
24. The Composite, Decorator, State, Strategy, and Bridge patterns are other patterns discussed by Gamma et al. [1995] that fit the broker model. Find out about one or more of these patterns and write an essay in which you explain how they fit the broker paradigm.
25. Form a team of two or three. Choose a broker pattern, think of a simple example to illustrate it, and write a Java program implementing your example. Create a Web page in which you explain the pattern and illustrate it with your example program.

Chapter 17 Review Quiz Answers

Review Quiz 17.1

1. In a broker pattern, the client must have access to the broker, which in turn must have access to the supplier. Other access relationships may be present (for example, the client may be able to access the supplier without going through the broker), but these are not essential to the broker paradigm.
2. In the Iterator pattern, an iterator simplifies its associated collection class by providing an iteration mechanism separate from the collection class. This simplifies the collection in two ways. First, the collection interface does not include operations for controlling iteration. Second, the collection implementation need not include iteration mechanisms.

Review Quiz 17.2

1. The Façade pattern is analogous to many situations where people hire brokers or specialists to handle some complicated aspect of their affairs that they prefer not to handle themselves. For example, people often hire mortgage brokers to set up their mortgages; lawyers to make wills, set up trusts, and so forth; financial advisors, stock brokers, or investment companies to handle their investments; and insurance agents to find and recommend insurance policies.
2. The Façade pattern can simplify the interface to a complex sub-system. It can lower the coupling between a client and a sub-system by reducing the number of interactions between the client and the sub-system and the number of classes in the sub-system with which the client must interact. It can also add functionality not otherwise present in the sub-system.
3. The Mediator pattern is analogous to a situation at an auction. Each bidder could negotiate with all the others, but an auctioneer can mediate the interaction by keeping track of the bids, announcing them to all the bidders, recognizing bidders who want to make new bids, and deciding when bidding is done.
4. A mediator frees each object in an interaction from having to deal with the other objects in the interaction. If the interaction is complex, the mediator presents a simplified interface to each collaborator. It also decouples the collaborators and encapsulates control of the interaction. Presenting a

simplified interface, decoupling an object from other objects it interacts with, and (possibly) adding functions are jobs done by a façade object. A mediator plays this role for each object in an interaction, so it presents a façade to each object, acting as a multi-way façade object.

5. The Mediator pattern encourages less-distributed control styles, providing a means to move from a dispersed to a delegated or from a delegated to a centralized control style.

**Review
Quiz 17.3**

1. The adapters in the Class and Object Adapter patterns both play the role of brokers, and the adaptees both play the role of suppliers.
2. *Two-way adapters* are classes providing several interfaces to a single adaptee. Multiple inheritance or multiple interface implementation can realize two-way class adapters. Often one of the interfaces needed is the original adaptee interface. In this case implementation is easy because the sub-classing adapter only needs to add the operations from the remaining interfaces. Sometimes language or individual class restrictions make it impossible to use the class adapter pattern to implement a two-way adapter. It is always possible to use the object adapter pattern instead. An object adapter can be written from scratch to implement as many interfaces to an adaptee as necessary.
3. If a class has 15 operations but only 3 must be changed to satisfy the needs of a client, then it will be a lot of work to use the Object Adapter pattern because every operation will have to be implemented in the adapter. If, on the other hand, the Class Adapter pattern is used, the adapter can simply inherit the 12 operations that do not need to be changed and override the 3 that need to be changed. It would be better to use the Class Adapter pattern in this case.

**Review
Quiz 17.4**

1. There are many times in life when one person or thing stands in for another; these are analogous to uses of the Proxy pattern. For example, an ambassador is a stand-in for a country's head of state in another country. Ambassadors are like remote proxies, providing a local presence for a person in a remote location.
2. A proxy class and the supplier class that it stands in for must have exactly the same interface. Otherwise, the proxy could not stand in for the supplier.

18 Generator Design Patterns

Chapter Objectives

This chapter continues our catalog of mid-level design patterns with the addition of several generator patterns.

By the end of this chapter you will be able to

- Illustrate and explain the structure and behavior of generator patterns and list their uses;
- Describe, illustrate, and explain several important generator patterns, including the Factory Method, Abstract Factory, Singleton, and Prototype patterns; and
- Use these patterns in mid-level design.

Chapter Contents

- 18.1 The Generator Category
 - 18.2 The Factory Patterns
 - 18.3 The Singleton Pattern
 - 18.4 The Prototype Pattern
-

18.1 The Generator Category

Instance Creation

Most object-oriented languages and systems provide two ways to create new objects: instantiating a class using one of its constructors or cloning an existing object. However, there are many reasons why a client may not be able to avail itself of one of these possibilities. In such cases a client may need to ask another class to generate an instance on its behalf. This is the essence of the **generator pattern** category.

An Analogy

To illustrate the need for generator classes, suppose a vehicle traffic simulator is used to study traffic patterns, wear and tear on roads, and other characteristics of a road system. Vehicles are generated randomly according to the distribution of the various sorts of vehicles expected to use the road system.

Suppose that the design for this program specifies a **Vehicle** interface implemented by classes **CompactCar**, **MidSizeCar**, **SUV**, **MiniVan**, **HeavyVan**, and so forth, all of which flow over instances of class **Road**. When a new **Vehicle** is created and placed on a **Road**, the vehicle type must be chosen at random. Where should the calculations for creating new **Vehicles** occur? They are clearly not intrinsic to **Roads**—a vehicle manufacturing class is needed. Such a **VehicleFactory** class contains an operation that creates and returns **Vehicles** of various classes at random according to their expected distributions. A **Road** does not obtain new **Vehicles** by using constructors

for particular classes directly, but instead by invoking an operation that creates instances for it.

Generator Patterns

A generator pattern has a Client that needs an instance of a Product class and a Generator that creates or obtains access to such an instance on behalf of the Client. The static structure of generator patterns is pictured in Figure 18-1-1.

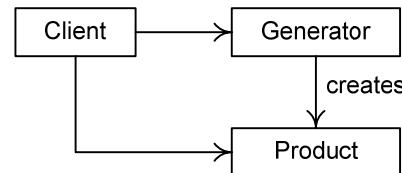


Figure 18-1-1 Generator Pattern Structure

The Client must have access to the Generator. The Generator creates an instance of the Product and then delivers it to the Client, which can then access it. This sequence of events is depicted in Figure 18-1-2.

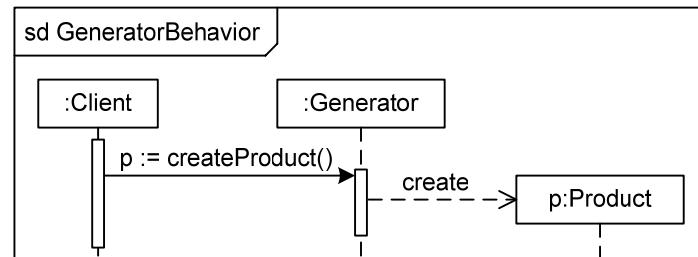


Figure 18-1-2 Generator Pattern Behavior

A generator may not always have to create a new instance of a product. In some cases, it may just provide access to existing instances of the product. We will see this variation in the dynamic behavior of generator patterns in later sections.

In the traffic simulation example mentioned earlier, the Road class is the client. It needs various sorts of Vehicle sub-class instances, which are the products. The VehicleFactory class is a generator that produces these products for Road.

Factory Methods

A generator must have at least one special operation that creates and returns new product instances (called `createProduct()` in Figure 18-1-2). These operations are crucial to generator pattern behavior, and for that reason they have a special name.

A **factory method** is a non-constructor operation that creates and returns class instances.

The operation in `VehicleFactory` that creates and returns `Vehicle` objects is an example of a factory method.

Factory methods are an example of a programming idiom. Factory methods are widely used in mid-level design patterns and in object-oriented programming in general. For example, the Iterator pattern uses a factory method in the collection to provide iterators, and object adapters are often created using a factory method.

Factory methods create new instances using constructors or cloning, so they do not rely on any special technique for class instantiation. They are the means by which responsibility for product object creation is removed from a client and placed in a generator, which is the purpose of generator patterns. When a generator assumes responsibility for product object creation in a factory method, the following capabilities become available:

- Access to product constructors can be restricted. There may be reasons to prohibit general access to product constructors; if there are, a factory method is needed to provide instances to clients.
- Private data can be provided to new product objects. Classes that must collaborate closely sometimes need to share data not accessible to other classes. Some languages have special mechanisms allowing a class to provide privileged access to its private data to certain other classes, such as the friend class mechanism in C++. Another way to do this is to have the class holding the private data be a generator with a factory method for its collaborating product class or classes. It can then pass the private data to new product objects in the factory method. For example, in implementing the Iterator pattern, the iterator may need access to the internal details of its associated collection class. The collection class has an `iterator()` factory method, and this method can create an iterator object and pass the object the data it needs to access the elements of the collection.
- Product objects can be configured after creation. In some cases, it may be necessary to set up new product objects after their creation in ways not possible using their constructors alone or using mechanisms hidden from clients. For example, if several product objects must be created with references to each other, then some objects must be created first and must receive references to objects created later. Factory methods provide a way to do this without burdening clients with the work.
- Product class bindings can be deferred until runtime. If a client creates an instance of a product by calling its constructor or cloning it, the particular class instantiated is built into the code at compile time. If a factory method is used to create a product instance, the type of its return value can be an interface type or a super-class, and the class of the

product object actually returned can be determined at runtime by the factory method. This adds enormous flexibility to product object creation.

These capabilities make the use of factory methods essential in generator patterns.

Reasons for Using Generator Patterns

There are several reasons for using generator patterns:

Product Creation Control—If a client does not have access to a product's constructor, then a generator is needed to access the constructor on behalf of the client. Usually constructors, or their classes, are not available to clients so that instance creation can be kept under strict control. For example, clients may not be able to instantiate classes directly so that only a certain number of instances are created.

Product Configuration Control—A client may need a generator to create a product instance to free itself of this responsibility or to configure the instance properly. For example, the product instance may need to be configured using private data in the generator.

Client and Product Decoupling—Generators are often used to create and configure products appropriate to the situation as a way of decoupling the client and the product. For example, a program may be able to interact with all kinds of input and output devices but may need to create the right device drivers when it starts up. The main parts of the program can be decoupled from the intricacies of the device layer by having a generator figure out which device drivers to instantiate.

Pattern Summary

Figure 18-1-3 summarizes the generator pattern category.

Name: Generator Category

Application: Make a class to create new product instances on behalf of a client, thus providing greater control over product instance creation and removing this responsibility from the client.

Form: Generator patterns have three classes: a client, a generator, and a product. The client requests product instances from the generator, which creates and configures them for the client.

Consequences: Generator patterns provide a way to exercise control over product creation and configuration and to decouple clients from products. This helps hide information, reduces coupling, and makes programs more changeable, configurable, and maintainable.

Figure 18-1-3 The Generator Category

Section Summary

- **Generator patterns** have generator classes that create product instances for clients.
- Generators must have special **factory methods** that create and return new class instances.

- Factory methods allow transfer of responsibility for object creation from a client to a generator.
- Factory methods enable product constructor access to be restricted; product objects to be configured after creation, possibly with private data from the generator; and specific product classes to be chosen at runtime.
- Generators may be used to control product instance creation, control product instance configuration, or decouple clients and products.

**Review
Quiz 18.1**

1. How do generator and broker patterns differ in structure and behavior?
 2. What sort of operation must always be present in any class playing the generator role in a generator pattern?
 3. In the traffic simulation example, which of the three reasons mentioned for using a generator pattern accounts for introducing the `VehicleFactory` generator class?
-

18.2 The Factory Patterns

Factory Methods Versus Factory Patterns

As noted in the previous section, a factory method is an operation that creates and returns a new class instance. Factory methods are present in all generator patterns; thus, the Factory patterns do not simply use factory methods to provide clients with new product instances—all generator patterns do that. Rather, they configure the generator and product classes in certain ways to decouple clients and products. More specifically, the **Factory patterns** use interfaces to generalize the generator and product classes, decoupling the client from the product and allowing greater flexibility in object creation.

The Factory patterns decouple clients from products by taking advantage of interfaces in two ways:

- The generator class with the factory methods can be changed, allowing variability in factory method implementations.
- Instances of a variety of classes that implement the product interface can be returned by a factory method, allowing great flexibility in results.

There are two Factory patterns:

Factory Method—The **Factory Method pattern** has a generator with a factory method for some product; the generator usually contains other operations besides the factory operation. This pattern decouples the product and the client using inheritance and interfaces to generalize the factory method and the product class.

Abstract Factory—The **Abstract Factory pattern** has a generator that is a **factory class**, which is a class that exists only as a container for factory methods producing instances of different, though usually related, products. This pattern decouples the client and the products produced by the factory class using factory class and product interfaces.

An Analogy Consider automobile factories: Automobiles are manufactured on assembly lines, and each line produces a certain kind of vehicle. Assembly lines can manufacture the same kinds of vehicles in different factories; for example, Chrysler, Ford, and Subaru factories can all have SUV assembly lines. The cars that come off the assembly lines are different, though they are all SUVs. For example, the Chrysler assembly line produces Durangos, the Subaru SUV line produces Outbacks, and the Ford line produces Explorers.

The Factory Method pattern is analogous to different automobile plants producing the same kind of vehicle. The analogy begins by observing that factory methods are like assembly lines. The Factory Method pattern has a generator class that is like a vehicle factory because it contains a factory method. Different generator classes that implement the same abstract factory method are like different plants that all make the same kind of vehicle. The products are the same kind because they all realize the same interface. This analogy is captured in the UML diagram in Figure 18-2-1, which names elements of the Factory Method pattern after analogous parts of automobile factories.

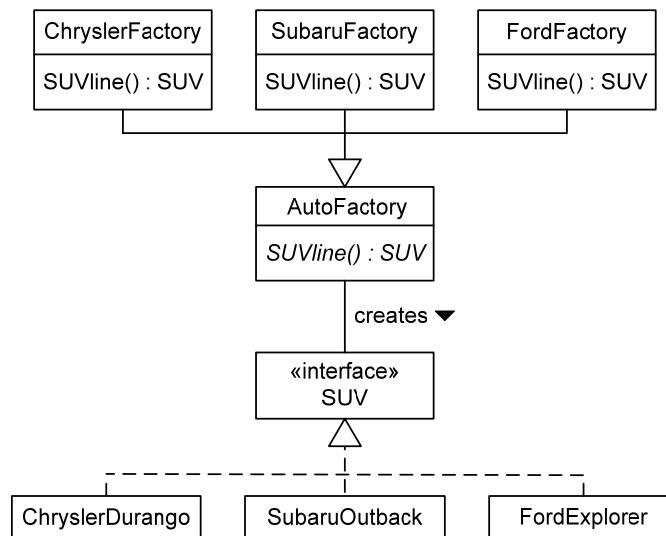


Figure 18-2-1 Automobile Factories and the Factory Method Pattern

Considering automobile factories again, a single factory might have several assembly lines for a variety of vehicles. For example, a single plant might have lines producing SUVs, sedans, and light trucks. Different manufacturers might have factories with lines that produce exactly the same mix of vehicles. This situation is analogous to the Abstract Factory pattern, in which a factory interface specifies several factory methods and the generator classes that realize this interface each produce several products.

Factory Method Pattern Structure

The static structure of the Factory Method pattern conforms closely to the generator pattern paradigm, with the addition of an abstract class and an interface. A generator class, called `ConcreteFactory` in the diagram in Figure 18-2-2, is a sub-class of an abstract Factory class that specifies an abstract factory method. The `ConcreteFactory` class implements the factory method. The factory method creates and returns new instances of a `ConcreteProduct` class, which realizes a Product interface.

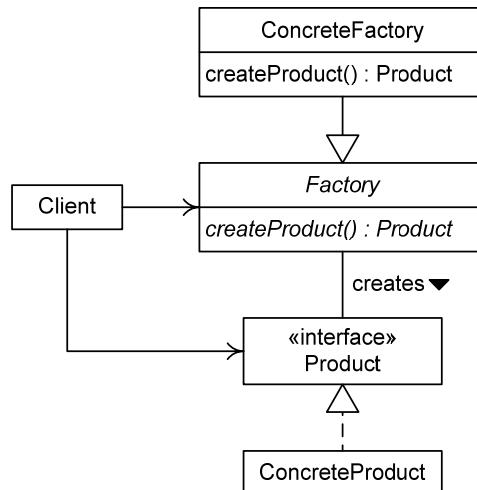


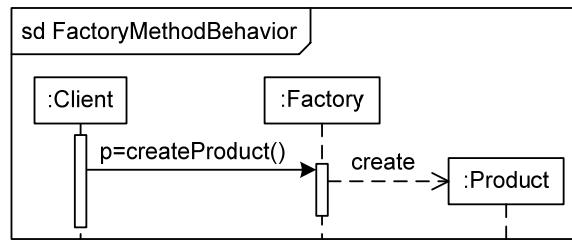
Figure 18-2-2 Factory Method Pattern Structure

The `Client` may use different `ConcreteFactories` because they all conform to the `Factory` interface, and the factory methods may create and return to the `Client` different `ConcreteProducts` because all products conform to the `Product` interface. This structure allows the Factory Method pattern to decouple the client class from the product classes.

The Factory Method pattern can be simplified by removing the abstract generator super-class or the product interface. For example, a generator might not implement an abstract factory method, but the factory method might still produce a variety of objects conforming to a product interface based on its arguments or other factors. For example, the Java `unmodifiableCollection()` method in the `Collections` class returns a collection of the same type as its parameter.

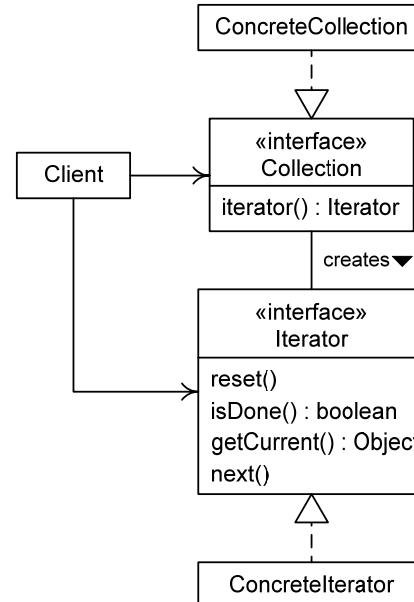
Factory Method Pattern Behavior

The behavior of the Factory Method pattern matches the generator pattern paradigm exactly. The `Client` gets a new `Product` instance from the `Factory` using its factory method. The behavior of the objects in this pattern is illustrated in the sequence diagram in Figure 18-2-3.

**Figure 18-2-3 Factory Method Pattern Behavior**

The Iterator and Factory Method Patterns

We have discussed the Iterator pattern as an example of a broker pattern, but it also incorporates the Factory Method pattern for iterator object creation. This means that it is also an instance of a generator pattern. The class diagram in Figure 18-2-4 presents the Iterator pattern in a way that emphasizes its use of the Factory Method pattern.

**Figure 18-2-4 Iterator as a Generator Pattern**

This diagram merely rearranges the components of the Iterator pattern structure to match the form of the Factory Method pattern structure presented in Figure 18-2-2. This diagram makes clear that the **Collection** interface matches the **Factory** abstract class in the Factory Method pattern, while the **Iterator** interface matches the **Product** interface. The dynamic structure of the iterator-creation portion of the Iterator pattern exactly matches the Factory Method pattern's dynamic structure as well.

Placing the `iterator()` factory method in an interface ensures that a Client can get an `Iterator` for any `Collection` using the same factory method, and the Client need not worry about which `ConcretelIterator` the `Collection` provides. This arrangement decouples the client from the product and makes it easier to use collection classes. To demonstrate, suppose that a Java program needs to use an `ArrayList` collection in some circumstances and a `HashSet` in other circumstances. In all cases, the collection must be traversed. This can be accomplished easily because of the flexibility provided by collection iterators adhering to the Factory Method pattern. Consider the code fragment in Figure 18-2-5.

```

Collection stuff;
if ( circumstance == x )
    stuff = new ArrayList();
else
    stuff = new HashSet();
...
Iterator it = stuff.iterator();
while ( it.hasNext() ) {
    Object o = it.next();
    // process a collection element
}

```

Figure 18-2-5 Generating an Iterator with Factory Method

This client code can iterate over any collection because the Factory Method pattern decouples iterator creation from the collection that is being used.

We now see that the Iterator pattern incorporates the Factory Method pattern in its first phase, when an iterator object is created. It is thus a generator pattern in its first phase and a broker pattern in its second phase, when the iterator object is used to traverse its associated collection.

When to Use the Factory Method Pattern

The main reason to use the Factory Method pattern is to decouple a client from the particular version of some product it uses. All products created by a factory method conform to a product interface, but beyond that constraint the factory method is free to create whatever particular products are needed. Decisions about product creation can be changed easily and deferred until runtime.

The Factory Method pattern also benefits from the advantages of using a factory method: The factory method can relieve the client of responsibilities for object instantiation and configuration.

Abstract Factory Pattern Structure

The Abstract Factory pattern is a *restriction* of the Factory Method pattern because the generator class is solely a factory class: It contains only factory methods, but the generator in the Factory Method pattern can, and usually

does, contain other operations. On the other hand, the Abstract Factory pattern *generalizes* the Factory Method pattern because the generator class has several factory methods producing several different but usually related products. This pattern's structure is pictured in Figure 18-2-6.

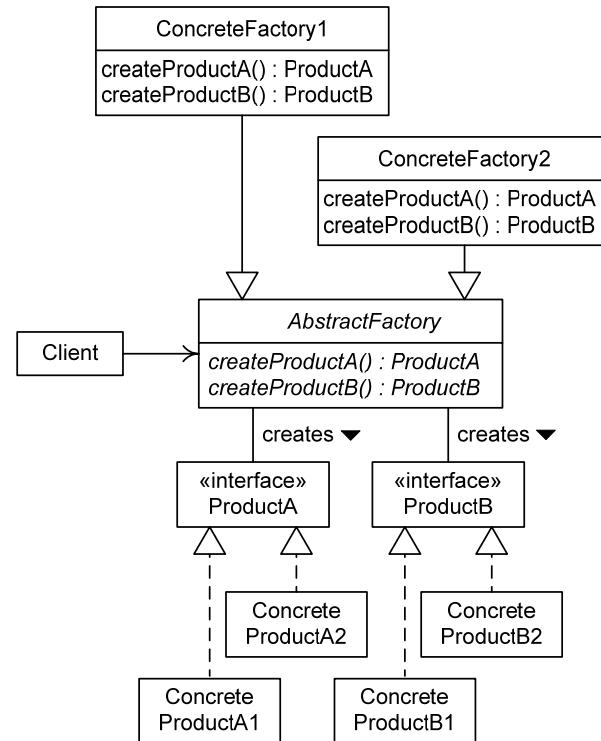


Figure 18-2-6 Abstract Factory Pattern Structure

There can be arbitrarily many concrete factories and products; this diagram shows two of each. The **AbstractFactory** interface must be instantiated to form a container for individual factory methods, such as **ConcreteFactory1** and **ConcreteFactory2**. Each concrete factory has methods for creating product instances, in this case **ProductA** and **ProductB**, but the objects each factory creates may differ. Hence the methods of **Factory1** create **ProductA1** and **ProductB1**, and the methods of **Factory2** create **ProductA2** and **ProductB2**.

A client may select or instantiate a concrete factory and then use it to generate the concrete products that it needs. The client can be written in terms of the abstract factory and product interfaces, thus decoupling it from concrete factories and products. Furthermore, the concrete factory can be chosen at runtime if desired, giving the client great flexibility.

Abstract Factory Pattern Behavior

The Abstract Factory pattern requires that the client have access to a concrete factory. The client might create one itself, be passed one as a parameter, use a factory method to obtain one, or have access to one in

another class. Once the client has access to a concrete factory, however, the client uses it in conformance with the standard generator pattern, as indicated in the sequence diagram in Figure 18-2-7. This diagram shows a client using `ConcreteFactory2` to obtain two concrete products.

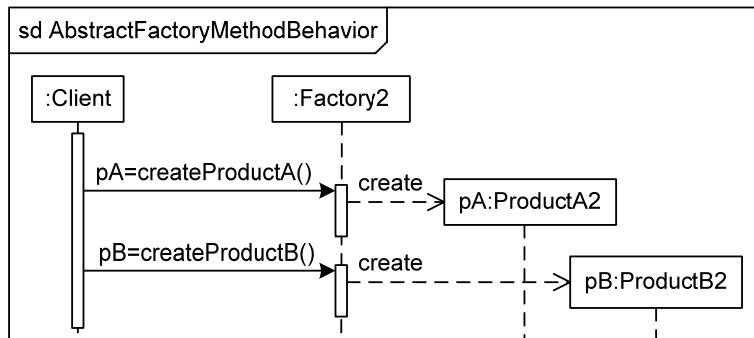


Figure 18-2-7 Abstract Factory Pattern Behavior

An Abstract Factory Pattern Example: AquaLush

Recall that AquaLush runs either in a real product or as a simulation on a Web page. When running in a real product, AquaLush communicates with hardware devices through virtual devices; it communicates with simulated devices when running as a simulation. One way to make it easy for AquaLush to meet these requirements is to use the Abstract Factory pattern to have two device factories: a simulated device factory and a real device factory. AquaLush can use one factory when running as a simulation and the other when running in the actual product.

The class diagram shown in Figure 18-2-8 illustrates how the Abstract Factory pattern can be applied in this case.

The Client (the part of AquaLush that needs devices) asks a `DeviceFactory` object for `SensorDevices`, `ValveDevices`, and so forth. The `DeviceFactory` object is set in one place to be either a `SimDeviceFactory` or a `RealDeviceFactory`, depending on whether AquaLush is running as a simulation or as the real product:

```

DeviceFactory factory;
if ( isSimulation ) then factory = new SimDeviceFactory();
else factory = new RealDeviceFactory();

```

Now AquaLush can create devices of the appropriate kind by using the factory methods of the new `DeviceFactory` object:

```

SensorDevice s1 = factory.createSensorDevice( sPort1 );
ValveDevice v1 = factory.createValveDevice( vPort1 );

```

The Abstract Factory pattern makes this code much simpler than it would be without use of this pattern.

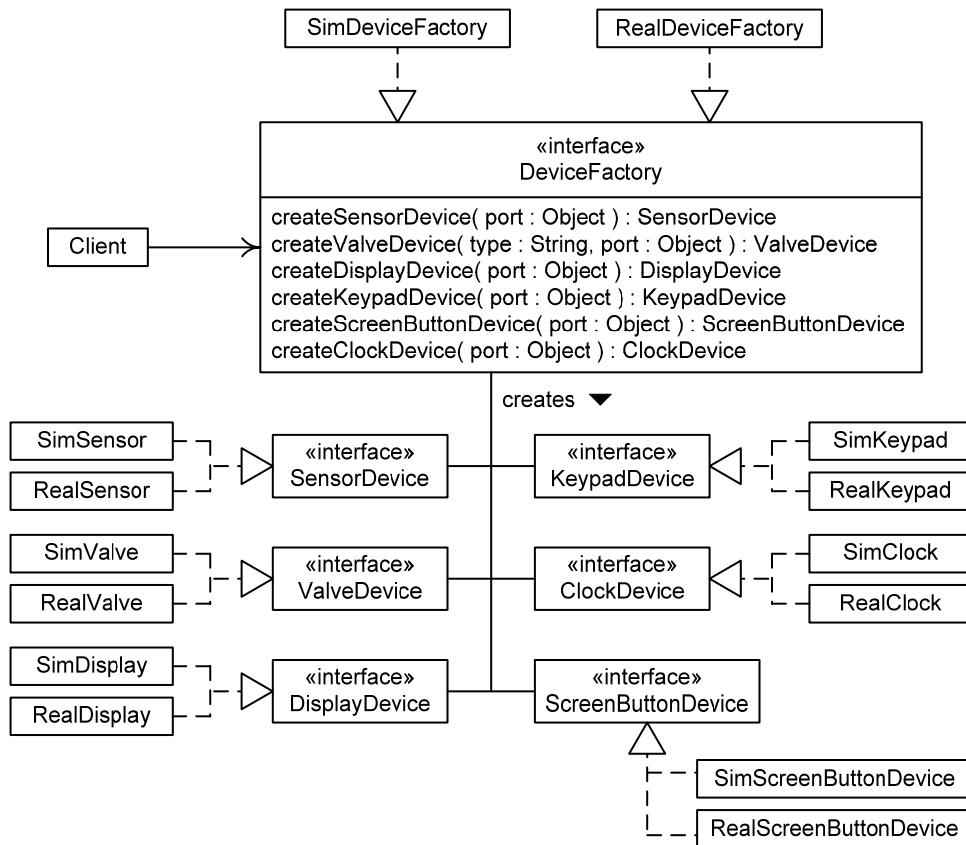


Figure 18-2-8 Abstract Factory Pattern for Creating AquaLush Devices

When to Use the Abstract Factory Pattern

The Abstract Factory pattern, like the Factory Method pattern, helps decouple clients from concrete product classes: Clients can be written entirely in terms of product interfaces. This is useful for program configuration and modification. For example, a program's user interface might be written in terms of user interface widget interfaces or abstract classes along with an abstract factory for creating widget instances.

Concrete factories for particular windowing systems would implement the abstract factory. For example, a Java Swing factory could produce Swing widgets and a Java AWT factory could produce AWT widgets. The program can choose which factory to use to set up its user interface when it starts up. An abstract factory and its concrete factories are also good candidates for reuse.

The Abstract Factory pattern can also enforce constraints about which classes need to be used with other classes. Classes that are supposed to be used together can all be products of a particular concrete factory.

One drawback of the Abstract Factory pattern is that it may be a lot of work to make a new concrete factory. Every concrete factory must have factory methods for every product in the abstract factory, even if some are

not needed. This may require implementing concrete product classes as well as the concrete factory class itself.

Pattern Summary

Figures 18-2-9 and 18-2-10 summarize the Factory Method and Abstract Factory patterns.

Name: Factory Method

Application: Allow clients to obtain a variety of product classes without hard-coding this choice. Clients can choose product classes at runtime.

Form: A generator implements a factory method specified in an abstract super-class. The factory method produces one of a variety of instances of product classes that implement a product interface.

Consequences: The client is decoupled from concrete product classes, which need not be chosen until runtime. The generator relieves the client of responsibility for creating and configuring products. This makes programs easier to configure and change.

Figure 18-2-9 The Factory Method Pattern

Name: Abstract Factory

Application: Allow clients to obtain a variety of sets of related product classes without hard-coding this choice. Clients can choose sets of product classes at runtime.

Form: A generator implements an abstract factory interface that specifies several factory methods for a set of products. Each factory method produces one of a variety of instances of product classes that implements product interfaces.

Consequences: The client is decoupled from concrete product classes, which need not be chosen until runtime. The generator relieves the client of responsibility for creating and configuring a set of related products. This makes programs easier to configure and modify. The abstract factory and concrete factories are good candidates for reuse.

Figure 18-2-10 The Abstract Factory Pattern

Section Summary

- **Factory patterns** use abstract classes and interfaces to decrease the coupling between clients and products.
- The **Factory Method pattern** is a generator pattern in which the factory method can be varied to create instances of different product classes that all conform to a product interface.
- The **Abstract Factory pattern** is a generator pattern whose generator is a factory class holding several factory methods for creating a set of related products.
- The Factory patterns allow clients to choose concrete classes at runtime.

Review Quiz 18.2

1. What is the relationship between factory methods and factory patterns?
2. What is a factory class?

3. In what way is the Abstract Factory pattern a restriction of the Factory Method pattern?
 4. In what way is the Abstract Factory pattern a generalization of the Factory Method pattern?
-

18.3 The Singleton Pattern

Unique Instances There are many circumstances in which software entities must be unique. For example, components such as print spoolers, window managers, disk controllers, memory managers, and database managers may need to be unique because they must have exclusive control of a resource. Other needs for unique entities arise in particular problem domains. In object-oriented systems, the need for unique entities amounts to a need for singleton classes.

A **singleton class** is a class that can have only one instance.

Few object-oriented programming languages or systems provide means for declaring singleton classes, but most have facilities that can be used to make sure that a class is instantiated only once. The **Singleton pattern** uses such facilities to make singleton classes. Variations of this pattern allow some fixed number of instances to be created.

Wide Access The Singleton pattern also provides global access to the instance of the singleton class. Often classes that are good candidates for singleton status also need to be accessible throughout a program. Rather than passing references to the singleton instance through many objects to provide wide access, this pattern allows the singleton instance to be globally accessible. Variations of the Singleton pattern can restrict access.

An Analogy It is rather difficult to come up with everyday examples of unique individuals that are accessible everywhere. The government, considered as a single entity, comes to mind. The best analogies, however, are probably the gods of the Greek, Roman, or Norse pantheons. The gods were unique entities who could be invoked by name anywhere (as can globally visible program entities), so they were, in a sense, like singleton classes.

Singleton Operations There must be some way to instantiate a singleton class to get an instance, but instantiation must be controlled so that only a single instance is ever made. One way to do this is to modify the class constructor to ensure that only a single instance is ever created. But this tends to be awkward—the constructor would have to take some other radical action, such as throwing an exception, if a client tried to create a second instance. Clients thus have to catch exceptions, making the class inconvenient to use.

A better solution is to make the constructor private or protected, prohibiting its use by clients. Instances are instead created by a factory method that ensures only a single instance of the class is ever created. This factory method needs to be associated with the class, but it must be available for use before any class instances exist because it creates the only one. This, in effect, means that the singleton factory method must be a class operation. This also provides the opportunity to make the factory method globally visible. Public class operations are visible wherever the class is visible, and classes can usually be given global visibility. Thus, a singleton class must have a class factory method.

Restricting the visibility of the class or the factory method limits the accessibility of the singleton instance as well. As a result, it is easy to vary this pattern to achieve more restrictive singleton accessibility.

Singleton Pattern Structure

The class diagram in Figure 18-3-1 illustrates the structure of the Singleton pattern. There is a private class variable holding a reference to the single instance, and the Singleton class constructor is also private. The public `instance()` operation is a factory method that instantiates the class only if it has not been instantiated before, thus guaranteeing instance uniqueness.

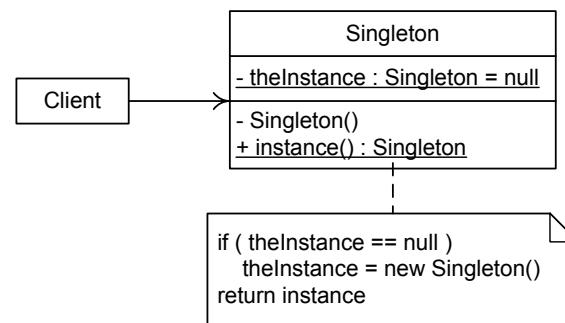


Figure 18-3-1 Singleton Pattern Structure

The Singleton pattern is a generator pattern, but it does not look like one in this diagram. The confusion occurs because there are three classes in generator patterns, but only two classes appear in Figure 18-3-1. However, the Singleton class plays two roles: It is both the generator and the product. Hence, this pattern is still a generator pattern even though its structure does not at first appear to match the paradigm. The dynamic structure of the pattern is a closer fit to the generator pattern standard.

Singleton Pattern Behavior

Using a singleton class could not be easier: A client simply calls the class `instance()` factory method to obtain the unique instance of the class. The diagram in Figure 18-3-2 shows what happens when a client requests the unique instance.

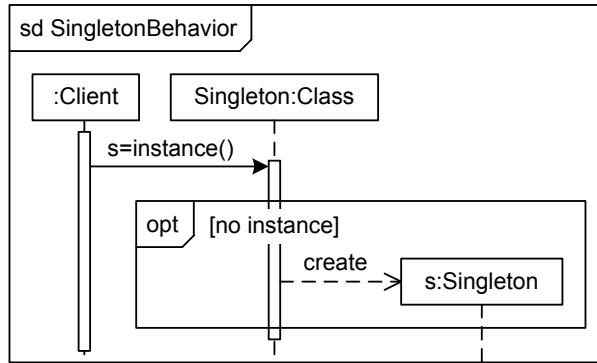


Figure 18-3-2 Singleton Pattern Behavior

Note that `Singleton` appears both as an individual of type `Class` and as the type of the object `s`. This is because both the `Singleton` class and its single instance `s` participate as individuals in this interaction.

This sequence diagram makes clear that the `Singleton` is a generator and the single instance of the class is its product. Thus, the `Singleton` pattern does indeed conform to the generator paradigm.

A Singleton Pattern Example: AquaLush

Many parts of `AquaLush` depend on time. The time is displayed in the control panel user interface, where it can be changed. The time determines when `AquaLush` begins irrigation in automatic mode, and it is used to compute how much water has been consumed during irrigation. `AquaLush` will not work correctly if its parts get out of temporal synchronization. The easiest way to prevent this is to have a single clock used by the entire program. The clock must also be easily accessible. Therefore, a `Clock` class is an ideal candidate for a singleton class. Other programs that rely on time may likewise benefit from making `Clock` a singleton.

Figure 18-3-3 is a class diagram illustrating the `AquaLush Clock` class.

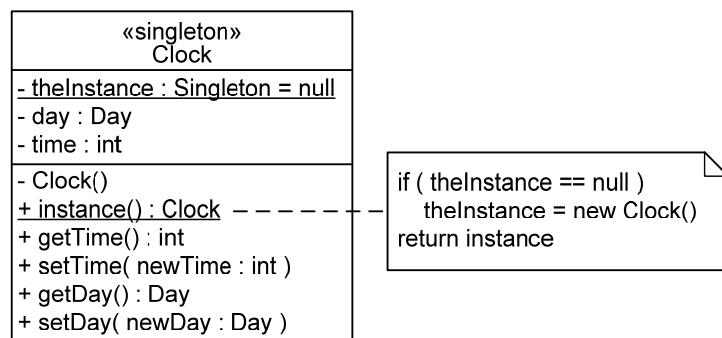


Figure 18-3-3 AquaLush Singleton Clock Class

A UML «singleton» stereotype has been used to emphasize that the `Clock` class is a singleton. Notice how the Singleton pattern mechanisms (the class variable and operation) are added to the class without disturbing its other attributes and operations—any class can easily be made into a singleton class.

When to Use the Singleton Pattern

Singleton classes should be used whenever it is important that only a single instance of a class exist and that that single instance be widely accessible. Singletons are easy to implement and incur no performance penalties, so it is good to use them whenever these constraints are present.

The Singleton pattern can also be used, with slight modifications, when a limited number of instances greater than one are desired. For example, if a system has four hardware ports, a `HardwarePort` class might allow up to four instances, one for each physical port. The Singleton pattern can easily be modified to provide this functionality.

Access restrictions are usually easy to add by restricting the visibility of either the class or the factory method.

Singletons Versus Classes without Instances

Programs written in languages that allow class operations and variables can have classes with operations and attributes that do not need to be instantiated at all. Even when classes are instantiated, their class variables are unique and shared across all instances. Why not just use classes with class operations and variables instead of singleton classes when a unique entity is needed?

Indeed, a class can play the same role as a singleton class instance, and doing this has some advantages in concurrent programming. But this approach also has the following drawbacks:

- Many languages do not allow classes to be values assignable to variables. Therefore, classes cannot be assigned to attributes, passed as parameters, and so forth the way that singleton class instances can. This may complicate code.
- Singleton classes can be sub-classed, and the singleton factory method can return a sub-class instance. This increases design flexibility, but it works only with instances, not classes.
- Classes can easily replace only single instances. If a limited number of instances greater than one are desired, the Singleton pattern can easily be modified to satisfy this need, but the classes without instances approach cannot.

Implementing the Singleton Pattern in Java

Java has no library support for the Singleton pattern, but the features of the language make singleton implementation straightforward. The singleton class is public, but its constructor is private or protected. (Private visibility is better because any classes in the same package can access protected members.) The class contains a static, private instance field holding a reference to the single instance and a static, public factory method for

accessing the instance. The factory method creates the instance if it does not exist and always returns the single instance. Otherwise, the singleton class is designed and implemented like any other class.

Pattern Summary Figure 18-3-4 summarizes the Singleton pattern.

Name: Singleton

Application: Ensure that a class has only a single globally accessible instance. The pattern can be modified to guarantee an upper bound on the number of instances or more restricted accessibility.

Form: The static structure is a modified generator structure with the singleton class acting as both the generator and the product. Clients call the singleton class factory method that creates (if necessary) and returns the singleton class instance.

Consequences: The singleton class is easy to implement and use. This pattern makes programs less prone to error, and hence more reliable and robust, by constraining the number of instances of a class.

Figure 18-3-4 The Singleton Pattern

Section Summary

- A **singleton class** is a class with only one instance.
- There is often need for unique entities in programs, and singleton classes can fill this need.
- The **Singleton pattern** is a generator pattern that guarantees that only a single instance of a class is created.
- The Singleton pattern can be modified to place an upper bound on the number of instances of a class or to provide limited access to instances.

Review Quiz 18.3

1. Why must the singleton class factory method be a class operation?
2. Why must the attribute holding the single instance of the singleton class be a class variable?
3. Why must the singleton class constructor be private or protected?

18.4 The Prototype Pattern

Cloning The usual way to create a new instance of a class is to call a class constructor. An alternative is to make a clone.

A **clone** is a copy of an object.

Clones are usually made by a factory method that produces a copy of an object at the moment when the factory method is called. This operation is conventionally called `clone()`. It has no parameters. Because `clone()` makes an exact copy of its host object at a moment in time, it produces an object

whose state is typically different from the state of a pristine object created by a constructor.

A question immediately arises when composite entities such as objects are copied: What should be done with attributes holding references? To illustrate this difficulty, consider a simple `Stack` class implemented in Java as shown in Figure 18-4-1.

```
class Stack {
    private final int maxSize;
    private int top;
    private Object[] store;

    public Stack( int size ) {
        store = new Object[maxSize = size];
        top = -1
    }

    public Stack clone() {
        ...
    }
    // Remainder omitted
}
```

Figure 18-4-1 Outline of a Stack Class Implementation

Suppose that a `Stack` instance exists, and it is copied using the `clone()` operation. If the `clone()` operation copies each attribute from the original into the copy, then the copy and the original will share the `store` array. The situation will then be as depicted in the data structure diagram in Figure 18-4-2.

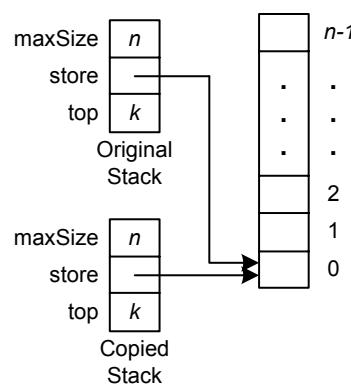


Figure 18-4-2 Shallow Copy of a Stack

If either the original or the copy of the `Stack` object is changed, then the shared data structure will be corrupted and the program will fail.

When values stored in an entity (including references) are reproduced in the copy, the copy operation is said to be **shallow**. In contrast, a copy operation is **deep** when copies are made of all referenced entities in the original composite, and references to the new entities are placed in the copy. Here “all referenced entities” means entities referenced in the original composite entity, entities referenced in any referenced composite entities, and so forth, to some semantically meaningful limit. For example, a deep copy of a `Stack` creates a new `store` array rather than merely copying a reference to it. However, the *references* in the `store` array are copied rather than the *entities* referenced because a `Stack`, as a collection, is understood to hold references rather than objects.

A deep copy of a `Stack` would produce data structures like those depicted in Figure 18-4-3.

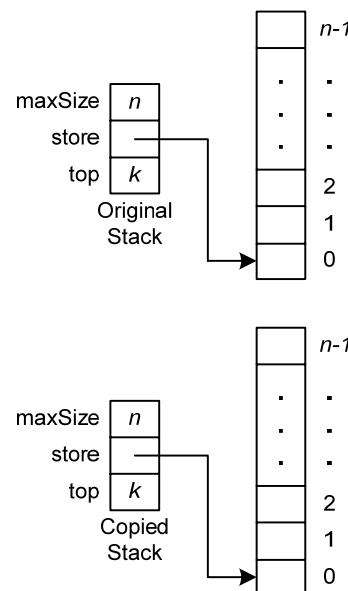


Figure 18-4-3 Deep Copy of a Stack

A `clone()` operation that makes a deep copy of the original `Stack` object solves the problem provided that clients understand that the `Stack` copy holds references to all the objects in the original `Stack`, not copies of these objects.

There is no standard copy behavior for `clone()` operations. If all the fields of a composite entity are primitive values or constants, then a shallow copy cannot lead to any difficulties. Sometimes, as in the example just considered, a deep copy is required. In many cases, copies must share references to some mutable objects and not others. Each case is different and must be considered individually.

Cloning in Java Java provides support for cloning with a `Clonable` interface and a protected `clone()` method in the `Object` class. Any class using the built-in cloning mechanism is supposed to

- Implement the `Clonable` interface;
- Define a concrete public or protected `clone()` operation; and
- In the `clone()` operation, obtain a new object by calling `super.clone()`.

If every class in a hierarchy does these three things, then an object of the correct class is created without using a constructor. But if even a single a class in the hierarchy does not implement the `Clonable` interface, then a `CloneNotSupportedException` is thrown when the `clone()` operation is called.

The default `clone()` operation makes a shallow copy. Consequently, an overriding `clone()` operation that needs to make a deep copy must first obtain the shallow-copy clone from its super-class and then modify it to make a deep copy before returning it. For example, an implementation of the `Stack` class using the built-in mechanism might look like the example in Figure 18-4-4.

```
class Stack implements Clonable {
    private final int maxSize;
    private int top;
    private Object[] store;

    public Stack( int size ) {
        store = new Object[maxSize = size];
        top = -1
    }

    public Object clone()
        throws CloneNotSupportedException{
        Stack result = (Stack)super.clone();
        result.store = (Object[])store.clone();
        return result;
    }
    // Remainder omitted
}
```

Figure 18-4-4 Stack Class Implementing Clonable

Note that after obtaining the `Stack` clone, the `store` array must also be cloned to make a deep copy.

The Java cloning mechanism is confusing, inconvenient, and error prone. Furthermore, it does not allow for making deep copies of final fields. It is probably better to simply implement a `clone()` factory method that

invokes a constructor and sets all fields as required. For example, the alternative implementation shown in Figure 18-4-5 of a `clone()` method for the `Stack` class works perfectly well.

```
class Stack {
    private final int maxSize;
    private int top;
    private Object[] store;

    public Stack( int size ) {
        store = new Object[maxSize = size];
        top = -1
    }

    public Stack clone() {
        Stack result = new Stack( maxSize );
        result.size = size;
        for ( int k = 0; k < size; k++ )
            result.store[k] = store[k];
        return result;
    }
    // Remainder omitted
}
```

Figure 18-4-5 Stack Class with a Clone Operation

In this case, the `clone()` operation uses the regular constructor and adjusts the fields of the clone to match the original. Alternatively, the `clone()` operation could use a copy constructor to create the clone. A **copy constructor** is a constructor that takes an instance of its class as an argument and creates a clone of its argument. A copy constructor must be used when a final field must be set in creating the clone because only a constructor can set a final field.

Cloning and the Prototype Pattern

We have gone into so much detail about cloning because it is the basis for the **Prototype pattern**. In this pattern, a client obtains a new class instance by asking an object to clone itself. Using a `clone()` operation is not using the Prototype pattern. The Prototype pattern decouples clients from particular product classes by using a Prototype interface to hide them, thereby making it easier to configure and modify programs.

An Analogy

Except for objects reproducing themselves, there are many common examples of using prototypes to obtain new objects. For example, suppose that your watch battery runs out. You can take the battery as a prototype to an electronics or battery store and ask for a replacement. You do not care what type of battery it is, only that you get another one of the same type so

that your watch runs again. Hence you, the client, are decoupled from the type of entity you need a copy of, just as in the Prototype pattern.

Prototype Pattern Structure

The Prototype pattern has a Client with a reference to a Prototype with a `clone()` factory method, as pictured in Figure 18-4-6.

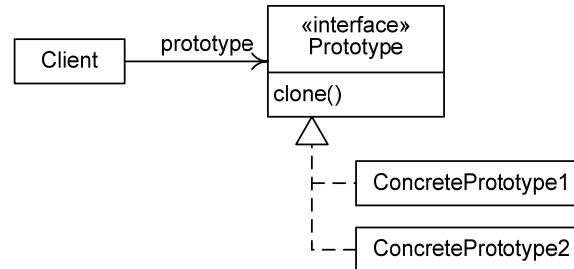


Figure 18-4-6 Prototype Pattern Structure

Although two `ConcretePrototype` classes are shown in the figure, there can be any number of them. The `Client` only needs to know that the prototype has the characteristics needed to meet its needs and a `clone()` operation. Otherwise, the actual type of the prototype is irrelevant to the `Client`.

This structure does not appear to fit a generator pattern, but as with the Singleton pattern the class diagram is deceiving because the `Prototype` is both the generator and the product. If the `Prototype` is put into the diagram twice to emphasize its different roles, the resulting picture has the structure of a generator pattern.

Prototype Pattern Behavior

The behavior of the Prototype pattern fits the generator model, as shown in Figure 18-4-7.

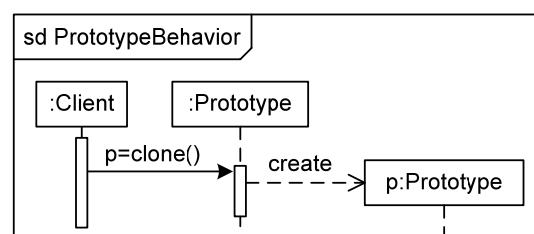


Figure 18-4-7 Prototype Pattern Behavior

A Prototype Pattern Example

A software architecture editor should be able to handle arbitrary box-and-line diagrams as well as several UML notations. A nice feature of such an editor would be to allow users to invent new icons or connectors and add them to a palette for later use. These new graphic entities would be created by users and have no defining class. They could be graphic primitives, such as rectangles, lines, and images, or composites made up of other graphic

elements. However, it may be difficult to design the editor to incorporate this feature.

One elegant way to realize this feature is for the palette to hold prototypes of the new graphic objects and for these to be cloned when a user wants a new one. For this to work, all graphic objects in the editor must be clonable, allowing users to select any sort of graphic object as a prototype. The diagram in Figure 18-4-8 illustrates how this could work.

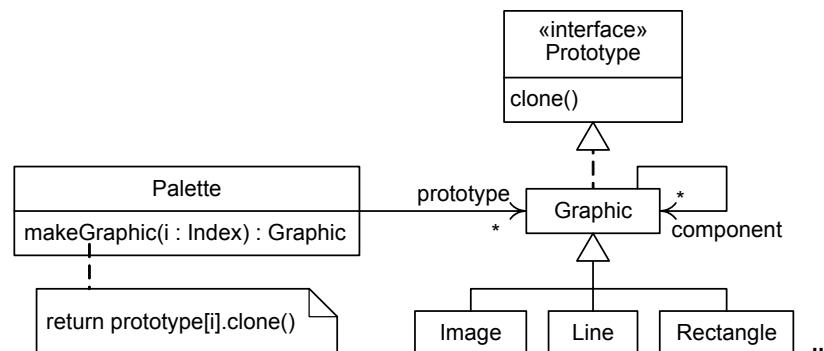


Figure 18-4-8 Architecture Editor Graphic Prototypes

The **Palette** maintains a collection of prototypes, each of which is a **Graphic** object. The **Graphic** object may be a primitive entity, such as a **Line** or a composite entity with components. All **Graphic** objects have a **clone()** operation because **Graphic** realizes the **Prototype** interface. The **makeGraphic()** operation is called when a user requests a new object from the **Palette**. This operation asks the requested object prototype to clone itself. New items can be added to the **Palette** or old ones removed at any time.

The **Palette** does not need to know the classes of the prototypes it manages because it does not use a constructor to make them. The **Palette** (a client) is thus decoupled from the products it uses by the **Prototype** instances it manages (the generators).

When to Use the Prototype Pattern

The **Prototype** pattern hides concrete product classes from clients, decoupling the clients from the product classes. Therefore, it can be used whenever clients need to be decoupled from products. This is the same advantage offered by the **Abstract Factory** and **Factory Method** patterns. The distinguishing feature of the **Prototype** pattern is that prototypes can be supplied and changed at runtime, in effect making it possible to set product classes during execution, which the **Factory** patterns cannot do. The **Prototype** pattern thus provides great flexibility in configuring and changing a program at runtime.

The main drawback of the **Prototype** pattern is that it relies on cloning, which presents problems concerning deep versus shallow copying. It may also be better in some cases to use a **Factory** pattern to exert more control over object creation—sometimes there can be too much flexibility.

Pattern Summary

Figure 18-4-9 summarizes the Prototype pattern.

Name: Prototype

Application: Decouple clients from product classes by producing new objects by cloning rather than by calling constructors.

Form: The static structure is a modified generator structure with the prototype class acting as both the generator and the product. Clients that need more instances of prototype objects call their `clone()` operations to obtain them.

Consequences: Clients need not know the classes of prototypes, making it possible to configure or alter clients at runtime by using different prototypes. The pattern depends on cloning, which raises issues about deep versus shallow copying of prototypes.

Figure 18-4-9 The Prototype Pattern

Section Summary

- A **clone** is a copy of an object. Copies can be made **shallow** or **deep**, depending on the object being copied.
- The **Prototype pattern** uses cloning to make new objects from prototypes.
- The Prototype pattern decouples clients from prototype product classes, greatly increasing configurability and changeability.

Review Quiz 18.4

1. Explain the difference between shallow and deep copying.
2. What is a copy constructor?
3. Give an analogy, different from the one in the text, for the Prototype pattern.
4. What advantage does the Prototype pattern have over the Abstract Factory pattern?

Chapter 18 Further Reading

All Sections

The pattern classification used in this book is based largely on Michael Norton's work ([Norton 2003]). All the patterns discussed in this chapter were first documented in [Gamma et al. 1995].

Section 18.3

Lea [2000] mentions the use of classes instead of singletons, especially in regards to their thread-safety.

Section 18.4

Bloch [2001] discusses cloning in Java in some detail.

Chapter 18 Exercises

Section 18.1

1. Suppose that a Java class has private constructors and no `clone` methods. Can such a class be instantiated? Explain how or why not.
2. The generator pattern allows a generator class to hide the classes of the product instances it creates. Explain why this is useful and illustrate your explanation with the Java `Collection` classes and `Iterator` class.

- Section 18.2**
3. *Fill in the blanks:* The _____ pattern is a generator as well as a broker pattern because the _____ class uses a _____ to create iterator objects on behalf of clients. The _____ class does this, in part, to hide the _____ class from the client, which is one of the uses of _____ patterns.
 4. Compare and contrast Java collection `Iterators` and `Enumerations` with respect to the way they employ the Factory Method pattern.
 5. The Factory Method pattern provides a factory method to create a new instance of a class. If instances of two or more classes are needed, then either more factory methods can be supplied (as in the Abstract Factory pattern), or a single factory method can accept one or more parameters that determine the class instantiated. Compare and contrast these alternatives.
 6. Elaborate the diagram in Figure 18-2-1 to illustrate the Abstract Factory pattern in analogy with automobile factories.
 7. The Java `Collections` class contains several factory methods. Is the Factory Method pattern used here?
 8. Consider the Abstract Factory pattern structure displayed in Figure 18-2-6. Suppose that a new product is needed, such as `ProductC`. How would this class diagram need to be changed to incorporate a new product? If this pattern were used in a program, would it be hard to make this change in the code?
 9. A program to play various sorts of card games uses an abstract factory to create instances of objects needed to play different games. Draw a UML diagram showing a draft design of this portion of the program. For example, different kinds of decks of cards are used for different games, as well as different kinds of players and hand evaluators.
- Section 18.3**
10. Write a “couple” class that can have exactly two instances. Have a `getInstance()` operation for each instance.
 11. Create an example of a situation where the Singleton pattern solves a problem that using a Java class with static methods and attributes cannot solve.
 12. Write a singleton class in which the single instance is held in a Java `static final public` variable. What advantages or disadvantages does this approach have?
 13. What levels of access to singleton classes can be provided in Java? Explain how to create singleton classes at each level of access you identify.
 14. Suppose that a media player program has a `SoundCard` super-class with sub-classes for different brands of sound cards. Evaluate a design alternative in which the `SoundCard` class is an abstract singleton class that generates a sub-class instance depending on the brand of sound card present.

15. How might the Singleton pattern be used in conjunction with the Abstract Factory pattern to ensure that a program uses only one concrete factory at a time?
- Section 18.4** 16. Rewrite the code in Figure 18-4-5 so that the `clone()` operation uses a public copy constructor instead of the regular constructor.
17. A Java `String` object is *immutable*—it cannot be changed. Does the `String` class implementation of `clone()` have to make a deep copy? Explain why or why not.
18. Make a class diagram of the Prototype pattern using stereotypes to make it more obvious that it has the static structure of a generator pattern.
19. Produce an alternative for AquaLush device configuration that uses the Prototype pattern rather than the Abstract Factory pattern, as in Figure 18-2-8.
20. The three main reasons for using a generator pattern are to control product creation, control product configuration, and decouple clients from product classes. Make a table in which you list the generator patterns in the rows and the reasons for using them in the columns. Check cells in the body of the table for reasons that apply to the corresponding patterns.

Research Project

21. Flyweight and Builder are patterns discussed by Gamma et al. [1995] that fit the generator model. Find out about one or more of these patterns and write an essay in which you explain how they fit the generator paradigm.

Team Project

22. Form a team of two or three. Choose a generator pattern, think of a simple example to illustrate it, and write a Java program implementing your example. Create a Web page in which you explain the pattern and illustrate it with your example program.

Chapter 18 Review Quiz Answers**Review Quiz 18.1**

1. The client in a broker pattern does not need to access the third party in the pattern (the supplier), while the client must access the third party (the product) in the generator pattern; thus, the static navigability relationships in broker and generator patterns are different. The third party in the broker pattern (the supplier) already exists when broker interaction takes place, but the third party in the generator pattern (the product) is created during the interaction.
2. The client in the generator pattern always calls the generator's factory method(s), so every generator must have a factory method.
3. The `VehicleFactory` generator class decouples the client from the individual product classes. The client only has to deal with classes that conform to the `Vehicle` interface and not with individual vehicle classes, such as `CompactCar` or `HeavyVan`.

**Review
Quiz 18.2**

1. A factory method is a non-constructor operation that creates and returns instances. Factory patterns involve several classes and interfaces that include factory methods. A class may have a factory method but not be associated with other classes and interfaces in a way that realizes a Factory pattern.
2. A factory class is a class that exists only to hold factory methods. Factory classes are central features of the Abstract Factory pattern.
3. The Abstract Factory pattern is a restriction of the Factory Method pattern because the generator class in the Abstract Factory pattern is a factory class, while the generator class in the Factory Method pattern is not. The generator class is more restricted in the Abstract Factory pattern than in the Factory Method pattern.
4. The Abstract Factory pattern is a generalization of the Factory Method pattern because the generator in a Factory Method pattern has a single factory method that creates an instance of a single concrete product class, while the generator in the Abstract Factory pattern has several factory methods that produce instances of several concrete product classes. The Abstract Factory pattern can be thought of as incorporating several Factory Method patterns.

**Review
Quiz 18.3**

1. The singleton class factory method must be called before there are any class instances. This is possible only with a class operation.
2. The attribute holding the single instance of the singleton class must be a class variable because it is manipulated by a class operation. This is possible only if it is a class variable.
3. The singleton class constructor must be private or protected to prevent clients from directly instantiating the class. Clients must be forced to get instances of the class from the singleton class factory method so that the factory method can control the number of instances that are created.

**Review
Quiz 18.4**

1. Shallow copying is making a field-by-field copy of some entity into another. Some of the copied values may be references, in which case both the original and the copy will hold references to the same objects. This often leads to problems. Deep copying, on the other hand, is making a copy with attention to whether a value is a reference or not. Non-reference values are reproduced, while references are treated by making copies of the referenced objects and placing references to the new objects in the copy. Copying referenced objects must be recursive, meaning that references in referenced objects are also treated by copying the referenced objects. This continues to some semantically sensible limit.
2. A copy constructor is a constructor that takes an instance of its class as an argument and creates a clone of its argument.
3. An analogy for the Prototype pattern is taking a broken part from some machine or furnishing, such as a faucet, a screw, or a furnace filter, to the hardware store and purchasing something like it. The broken part is like a prototype, and buying another item to replace it is like cloning the prototype.
4. The Prototype pattern allows new prototypes to be introduced for cloning during program execution. In effect, this provides for the addition of product classes at runtime, which the Abstract Factory pattern does not allow.

19 Reactor Design Patterns

Chapter Objectives

This chapter completes our discussion of mid-level design patterns with a final category containing two reactor patterns.

By the end of this chapter you will be able to

- Illustrate and explain the structure and behavior of reactor patterns and list their uses;
- Describe, illustrate, and explain two important reactor patterns (the Command and Observer patterns); and
- Use these patterns in mid-level design.

Chapter Contents

- 19.1 The Reactor Category
 - 19.2 The Command Pattern
 - 19.3 The Observer Pattern
-

19.1 The Reactor Category

Event-Driven Design

Event-driven design is an approach to program design that focuses on events to which programs must react. *Events* include occurrences in the program's environment to which it must respond—mainly input from an actor, such as a signal from a device, a message from a network, or a mouse click from a user. Events also include important occurrences within the program that demand a response, such as the completion of some process, the passing of time, or the throwing of an exception. In event-driven design, the designer thinks about the events to which the program must respond and designs the program to respond to these events.

Event-driven programming contrasts with other ways of approaching program design that we have mentioned before, particularly stepwise refinement. Recall that **stepwise refinement** is a top-down approach to program design that repeatedly decomposes procedures until programming-level operations are reached. This approach encourages designers to think of the program as a process that transforms inputs to outputs. In contrast, an event-driven design approach encourages programmers to think about handling events rather than transforming data.

Some programs and program components are best thought of as processes for transforming data, but many programs and program components are best thought of as event handlers. For example, graphical user interface code is best thought about in terms of handling events because there is no

way to predict what the user will do—programs must simply react to user input events.

Reactor patterns assist in designing event-driven programs or program components because they model ways to set up program components that react to other components when certain events occur. More specifically, in a reactor pattern a reactor is assigned to monitor events in a target class on behalf of a client and to take action in response to events in the target class. This is an excellent model for event-driven design.

Reactor Patterns

Reactor patterns have a **Client** that needs events in a **Target** class to be handled. The **Client** assigns this responsibility to a **Reactor** class. The **Target** class notifies the **Reactor** when events occur, and the **Reactor** responds on behalf of the **Client**. The structure of a reactor pattern is pictured in Figure 19-1-1.

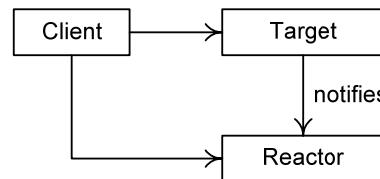


Figure 19-1-1 Reactor Pattern Structure

The **Client** must have access to the **Target** and the **Reactor** to register the **Reactor** with the **Target**. Often, the **Client** creates the **Reactor** as well as registering it with the **Target**. The **Target** needs to access the **Reactor** to notify it of events. The **Reactor** typically interacts with other objects in handling events; its collaborators may include the **Client** and the **Target** as well as other classes that depend on particular cases.

Reactor pattern behavior is more complex than broker or generator pattern behavior. Reactor patterns have two phases of activity:

Setup Phase—The **Client** registers the **Reactor** with the **Target**.

Operational Phase—The **Reactor** responds to event notifications from the **Target**.

These phases are shown in the sequence diagram in Figure 19-1-2.

The **Client** interacts with the **Target** but not the **Reactor** in the setup phase, and thereafter, the **Client** may not be involved at all. The **stimulus()** operation at the start of the operational phase is used to indicate something that causes an event to occur in the **Target**. An event may occur because of a message, but it may also occur because of something internal to the **Target**. The **Target** notifies the **Reactor** of the event, and the **Reactor** responds. The **Reactor**'s response often includes messages sent to arbitrary collaborators, which are not shown in this diagram.

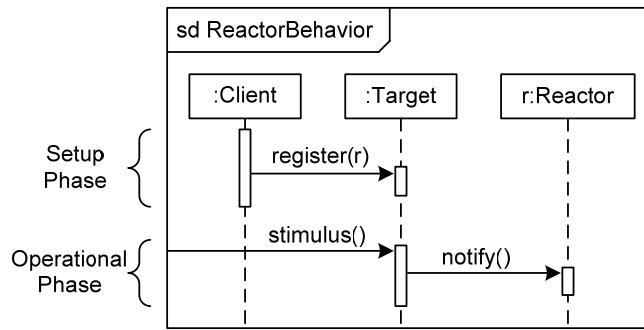


Figure 19-1-2 Reactor Pattern Behavior

Reasons for Using Reactor Patterns

The reactor patterns provide a good model for event-driven portions of a program for the following reasons:

Client and Target Decoupling—Once the client assigns a reactor to handle target events, the client and the target do not need to interact further. This makes it easy to change and reuse targets and clients.

Low Reactor and Target Coupling—The target class does not need to know anything about the reactor class except that it must be given event notifications. The reactor, on the other hand, must know how to react to events, which often includes knowing quite a lot about the target. Even so, the target does not need to be changed at all to replace a reactor.

Client Decomposition—The client delegates target event handling to the reactor. This simplifies the client and distributes control, making it less centralized. This pattern encourages a delegated control style.

Operation Encapsulation—The event handling centralized in a reactor is essentially an encapsulated operation. This operation can be passed between objects or stored, allowing other processing possibilities. For example, a series of actions encapsulated as reactor objects can be stored in a list and used as the basis for macros or undo functions.

Reactor Patterns and Event-Driven Architectures

We discussed the *Event-Driven* or *Implicit-Invocation* architectural style in Chapter 15. Recall that in this style, a central event dispatcher mediates interaction between components that generate events and those interested in them. Components interested in particular events register their interest with the event dispatcher. Components announce events to the event dispatcher, which then directs event notifications to components that have registered interest in the announced events.

Both Event-Driven architectures and reactor patterns support event-driven design, and they are similar in many ways. In particular, they both have a static structure that includes components that announce events and others that respond to them. Also, they both have a two-phase dynamic structure: There is a setup phase for component registration and an operational phase for raising and handling events.

Style Versus Design Pattern	<p>The Event-Driven architectural style operates at a higher level of abstraction, identifying major program components, while reactor design patterns are at a lower level of abstraction involving individual classes. Nevertheless, as noted in previous chapters, there is no strict boundary between architectural and detailed design, so this distinction may not be clear in individual cases.</p> <p>The Event-Driven style has a dedicated event dispatcher, but reactor patterns do not. This structural difference has many consequences:</p> <ul style="list-style-type: none"> ▪ The event dispatcher completely decouples components announcing events from components responding to them. The reactor and target still have some degree of coupling in reactor patterns. Using the Event-Driven style thus makes programs even more changeable and configurable than does using reactor patterns. ▪ An event dispatcher is a somewhat complex component. Making, maintaining, and using an event dispatcher involves considerably more effort than using a reactor pattern. Adopting an Event-Driven style in lieu of using reactor patterns is justified only if a significant portion of a program has an event-driven design. Otherwise, using reactor patterns is simpler and easier. ▪ A program with an event dispatcher may not perform well because the event dispatcher may not be able to send event notifications to interested components fast enough. Reactor patterns may be faster because reactors are directly connected to target objects that announce events. <p>In summary, the Event-Driven style provides more configurability and changeability at the expense of complexity and slower performance. Reactor patterns are simple and fast, but they couple target and reactor classes to some extent, making programs somewhat less configurable and changeable.</p>
------------------------------------	---

Pattern Summary

Figure 19-1-3 summarizes the reactor pattern category.

Name: Reactor Category

Application: Assign a reactor object to monitor and react to event notifications from a target on behalf of a client.

Form: Reactor patterns have three classes: a client, a reactor, and a target. The client registers the reactor with the target. The reactor then responds to event notifications from the target.

Consequences: Reactor patterns provide event-driven design models. They decouple clients and targets, decompose clients, and encapsulate reactions to events.

Figure 19-1-3 The Reactor Category

- Section Summary**
- **Event-driven design** focuses on how a program reacts to events.
 - **Reactor patterns** support event-driven design by assigning reactors to respond to events in a target on behalf of a client.
 - Reactor patterns decouple clients and targets, have low coupling between reactors and targets, decompose clients, and encapsulate event handling.
 - Reactor patterns and the Event-Driven architectural style both support event-driven design, but with trade-offs in configurability, changeability, simplicity, and performance.

Review Quiz 19.1

1. Give an example, different from the one in the text, of a program component or function that lends itself to event-driven design.
 2. What are the two phases of reactor pattern behavior?
 3. What are the relative advantages and disadvantages of using an Event-Driven architectural style compared to using reactor patterns?
-

19.2 The Command Pattern

Function Objects

A common exercise when studying sorting algorithms is to write a program that compares the behavior of several sorting algorithms. The program times how quickly the algorithms can sort arrays of various sizes. Pseudocode for the main loop of such a program appears in Figure 19-2-1.

```

print "Sort Time1 Time2 Time3 ... Timenk"
for each operation sort
    print sort.name
    for each array a
        startTime = now()
        sort(a)
        endTime = now()
        print( endTime - startTime )
    println

```

Figure 19-2-1 Comparing Sorting Algorithms

After printing a header line, this program goes through each sorting algorithm, prints its name, and then sorts arrays of several sizes and prints the time it took to sort each array.

This code will only work if operations can be treated like values—they must be assignable to variables, passed as parameters to procedures, and so forth. Some languages allow this. For example, this pseudocode could be refined into C or C++ code because these languages support pointers to functions. Other languages, such as Java, do not allow operations to be treated as values, so this approach will not work in many languages.

This sort comparison program is only one example among many where it is useful to treat operations as values. Chapter 16 discussed internal iteration control, which occurs when an iteration mechanism is passed an operation that it applies to each element of a collection. Internal iteration control requires a way to pass operations as values. The X Windows System, the standard graphical user interface toolkit for UNIX systems, uses callback functions extensively. *Callback functions* are operations passed to user interface widgets (such as buttons, menu items, and scrollbars) that are used to “call back” to the application when a user manipulates a widget (for example, pressing a button, selecting a menu item, or moving a scrollbar elevator). This mechanism relies on passing operations as values and storing them in variables.

Object-oriented languages have an idiom that effectively treats operations as values. The trick is to encapsulate an operation in an object. Objects are values in object-oriented languages, and as a result the encapsulating object can be stored in a variable, passed to operations, and so forth. When it is time to execute the operation, it is called through the object.

An object that encapsulates an operation is called a **function object** or a **functor**; we call the encapsulating class a **function class**. The encapsulated operation is declared as a public operation in the function class. The function class is instantiated to obtain a function object. For example, if we were writing the sort comparison program in a language like Java, we might design classes like those in Figure 19-2-2.

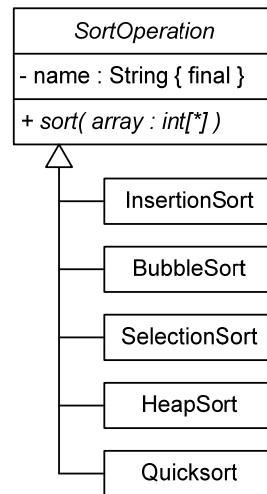


Figure 19-2-2 Sorting Function Classes

Each sub-class of `SortOperation` is a function class that encapsulates a sorting algorithm in its implementation of the abstract `sort()` operation. With this class hierarchy in mind, we can recast the sort comparison program with more Java-related detail, as shown in Figure 19-2-3.

```

Collection sortCollection = new ArrayList()
sortCollection.add( new InsertionSort() )
sortCollection.add( new BubbleSort() )
...
print "Sort Time1 Time2 Time3 ... Timenk"
for each element sorter of sortCollection
    print sorter.toString()
    for each array a
        startTime = now()
        sorter.sort(a)
        endTime = now()
        print( endTime - startTime )
    println

```

Figure 19-2-3 Using Function Objects

The SortOperation function objects are created and added to a collection. The collection is then traversed to apply each sorting operation to several arrays. Note how the sorting operations are applied through the function objects.

Function objects provide an elegant solution to the problem of treating operations as values in object-oriented languages that do not support this facility. They also provide a few additional advantages over treating operations as values:

- Additional features can be added to the function class, enhancing its capabilities. For example, the SortOperation function classes in the sort comparison program also record the name of the sorting algorithm, which is subsequently displayed in the output. A plain operation would not be able to reveal its name in this way.
- The function class can include other data and operations the encapsulated operation needs. For example, a quicksort operation might be implemented with a partition helper operation; a Shell sort operation might have additional parameters controlling its sorting passes recorded as attributes. Packaging everything an operation needs in the function class increases reusability, flexibility, and maintainability.

Function Objects and the Command Pattern

The **Command pattern** is a reactor pattern that uses a function object as its reactor. The client passes a function object to the target during the setup phase. During the operational phase, the target calls the function object's operation whenever an event of interest occurs in the target.

An Analogy

The Command pattern is analogous to giving an emergency phone number to a babysitter. The parents (the client) give the babysitter (the target) a piece of paper (the reactor) with an emergency phone number on it (the

encapsulated operation) before leaving the babysitter in charge. This is analogous to the setup phase. After the parents leave, an emergency may occur. If so, the babysitter (the target) consults the paper and calls the emergency phone number (invokes the encapsulated operation). This is analogous to the operational phase.

Command Pattern Structure

The structure of the Command pattern adheres closely to the form of a reactor pattern with the addition of an interface, as indicated by the class diagram shown in Figure 19-2-4.

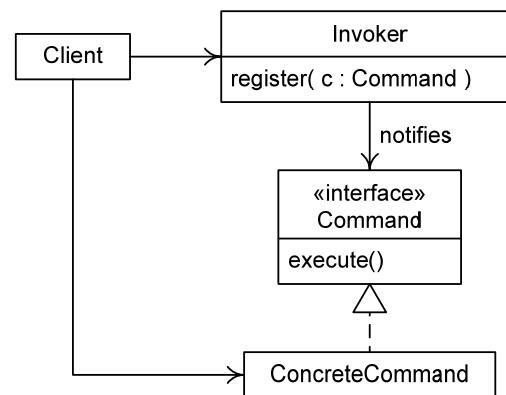


Figure 19-2-4 Command Pattern Structure

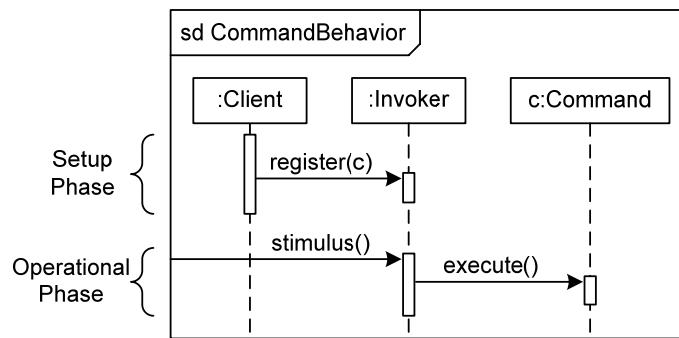
The **Invoker** (the target class) has a **register()** operation that accepts any instance of a class realizing the **Command** interface (the reactor class). This makes the **Invoker** flexible. The **Client** must have access to a **ConcreteCommand** so that it can register it with the **Invoker** during setup. The **Invoker** must store the **Command** so it can notify the **Command** instance whenever an event of interest occurs.

Note that the **ConcreteCommand** is a function class encapsulating the **execute()** operation. The use of function classes as reactors is a defining characteristic of the **Command** pattern.

The presence of an **Invoker** class is also an essential element of this pattern. The sort comparison example discussed previously has a **Client** (the main program class), an interface corresponding to **Command** (**SortOperation**), and several concrete command classes (the function classes), but nothing corresponding to the **Invoker** class. The sort program uses function objects, but it does not use the **Command** pattern. Similarly, an internal iteration mechanism might use function objects but not the **Command** pattern, again because it lacks an **Invoker** class.

Command Pattern Behavior

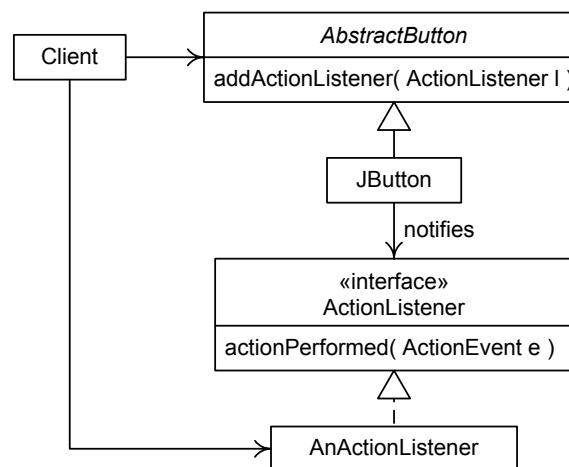
The **Command** pattern's behavior closely follows the reactor pattern paradigm, as the sequence diagram in Figure 19-2-5 shows.

**Figure 19-2-5 Command Pattern Behavior**

The Client registers a Command with the Invoker during the setup phase. The Invoker then calls the Command's `execute()` operation whenever an event of interest occurs in the Invoker, shown by the arrival of a `stimulus()` message at the Invoker.

Command Pattern Examples

The Java AWT and Swing graphical user interface toolkits both use the Command pattern extensively. In Java, command classes are conventionally called *listeners* or *handlers*. Dozens of AWT and Swing classes act as invokers, accepting listener and handler registrations for command classes responding to a broad range of events. As a simple example, consider the way that a Swing JButton notifies an application about button press events. A class diagram of the relevant classes in this interaction appears in Figure 19-2-6.

**Figure 19-2-6 Action Listeners and the Command Pattern**

The `AbstractButton`, `ActionEvent`, and `JButton` classes and the `ActionListener` interface are part of Swing; the other two classes are written by the programmer. `JButton` is a sub-class of `AbstractButton`, which includes the

listener registration operation `addActionListener()`. A JButton is therefore a command invoker. The ActionListener interface is a command interface. AnActionListener is the function class that responds to notifications of button presses from the JButton. The JButton calls `actionPerformed()` whenever it is pressed. The ActionEvent argument includes data, such as the object where the event occurred (the JButton object) and the time when the event occurred.

By using the Command pattern, Swing provides a way for designers to focus on responding to events, thus enabling an event-driven design approach. It is also easy to write code that implements responses to button presses; programmers only need to program a function object and register it with the button as a listener.

Using this pattern also helps structure the program: The code for responding to button presses is encapsulated in an action listener, making it easier to find, reuse, and change. This also helps decompose the program and decouples the interaction with button classes from the rest of the program.

Command Pattern Variations

Several aspects of the Command pattern may be varied or elaborated to help achieve particular design goals:

- An invoker may accept registration of more than one command. When an event of interest occurs in the invoker, it calls all the registered commands. Several operations may then be executed in response to a single event.
- The invoker may offer operations for unregistering one or more commands. This may be useful if the configuration or event-driven behavior of a program needs to change during execution.
- An invoker may offer several kinds of registrations for different kinds of events. This offers finer control over event handling and usually makes programming easier. It may also improve performance.
- The Command interface may include several operations to be called in response to different events. This is useful if responses to different events require distinctly different processing or different parameters. It may also simplify function classes and improve performance.

When to Use the Command Pattern

The Command pattern is useful in event-driven design, particularly for delegating a client's response to events from an invoker class to another class that encapsulates this response. As we have seen, this pattern is useful in designing mechanisms to respond to user inputs. It is also useful for designing mechanisms to respond to events from other actors, such as sensors, actuators, or other systems, and also to internal events, such as the expiration of a timer or the occurrence of an error.

The Command pattern helps decompose clients; lowers coupling among invoker classes, clients, and concrete command classes; and encapsulates event-handling code, making it easier to change, replace, or reuse.

Pattern Summary

Figure 19-2-7 summarizes the Command pattern.

Name: Command

Application: Enable clients to encapsulate responses to events in an invoker class and delegate event handling to a command class.

Form: A command class encapsulates an operation called by an invoker whenever an event occurs; the command class handles events on behalf of a client. The client registers the command with the invoker during a setup phase. The invoker calls the encapsulated operation during an operational phase.

Consequences: Event-driven design is facilitated. Event handling is isolated in a command class, helping to decompose the client, and making event-handling code easy to find, change, and reuse. The invoker is loosely coupled to both the command and client classes, making them easy to change.

Figure 19-2-7 The Command Pattern

Section Summary

- A **function object** is an object that encapsulates an operation; its class is a **function class**. Function objects allow operations to be treated like values.
- The **Command pattern** is a reactor pattern in which the reactor class is a function class. In use, the pattern is first set up and then enters an operational phase during which events are handled.
- The Command pattern helps decompose clients, lowers coupling, and encapsulates event handling, thus making programs more maintainable and their parts more reusable.

Review Quiz 19.2

1. What is a callback function?
2. What advantage do function objects provide?
3. What variations or elaborations can be made to the basic Command pattern?

19.3 The Observer Pattern

Reducing Coupling

Minimizing coupling is of primary importance in designing good modules, and it is one of the major factors that determines a module's degree of reusability and changeability. The **Observer pattern** is a simple, flexible, and widely applicable reactor pattern for reducing the coupling between classes while preserving their ability to interact. The Observer pattern can be used whenever one or more objects (*observers*) must track changes in another object (the *subject* or *observable*), and it is desirable to reduce the coupling between the observers and the subject. The Observer pattern is of long standing in software design; it is also sometimes called the **Publish-Subscribe pattern**.

Observer Patterns in the MVC Architecture

An excellent illustration of how the Observer pattern can reduce coupling is provided by programs that employ the Model-View-Controller architecture. Recall from Chapter 15 that in this architectural style all problem-domain data and operations reside in a *model*. One or more *controllers* handle user input, and one or more *views* display the state of the system to the user. The controller and the views comprise the user interface, and the model is the problem domain-portion of the system.

A non-event-driven way to implement the MVC architecture is for the model to keep track of all its views and to call particular view operations whenever it changes. For instance, in a media player the CD-playing model would need to know about the views displaying the track it was playing. It would also have to call particular view operations to change the track title and track duration whenever it started playing a new track.

This approach works, but it couples the model tightly to its views because the model must know about the views and which view operations to call when certain changes occur. This tight coupling makes the program harder to change and maintain. Changes to the view implementation often require changes to the model as well, and the model is not reusable without its views, or at least without modifying it to remove dependence on them.

The Observer pattern decouples the model from its views by taking an event-driven design approach. Rather than viewing the interaction between a model and its views as a dialog between peer objects, the Observer pattern regards the model as a target in which change events occur to which the views must react. The views respond by querying the model to find out what has changed and then updating themselves accordingly. The model needs to know little about its views, decoupling it almost completely from them.

The model, as an Observer pattern subject, only needs to keep a list of its views, which are Observer pattern observers. Each view registers with the model as an observer. When the model changes, it notifies its observers that it has changed, but that is all. It is up to the views to query the model for whatever additional data they need to update themselves. The diagram in Figure 19-3-1 on page 580 illustrates this arrangement.

The advantage of this approach is that the model is almost completely decoupled from its views: The model does not need to know anything about the views except that they must be notified of changes. The model does not even need to know in advance that there are any views because each view must register itself with the model to receive change notifications. The model is now isolated from modifications in the view portion of the system, making it easy to change the views and making the model more reusable.

Though we have focused on the model and its views, any controllers that need to know about changes in the model can also register themselves as observers.

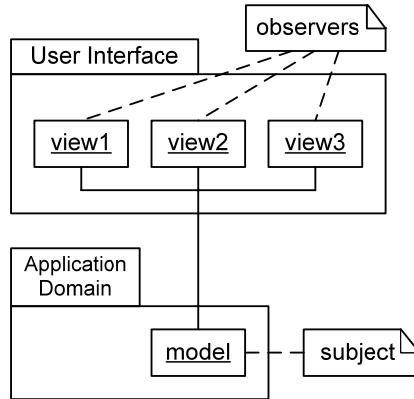


Figure 19-3-1 Observers in the MVC Architecture

An Analogy

The Observer pattern can be compared to a “current awareness” or “selective dissemination of information” service. In such services, people register profiles of their interests in current events, scholarly publications, or whatever type of information the service tracks. When something occurs that might be of interest to a subscriber, the service sends him or her a notification of the event. The subscriber can then seek additional information if he or she desires.

The subject in the Observer pattern corresponds to the current awareness service, and the observers correspond to its subscribers. The subscribers must register with the service, just as observers must register with a subject. The service keeps a list of its subscribers and notifies them when events of interest occur, just as a subject keeps a list of its observers and notifies them when it changes. Subscribers can cancel or reinstate their subscriptions at any time, just as observers can unregister or reregister themselves at any time.

Subject and Observer Operations

From the discussion so far, it should be clear that every observer must provide some standard public operation that a subject can call to notify it of changes, but that is all the subject needs to know about its observers.

The subject, on the other hand, must provide an observer registration operation, and it must notify its observers when it changes. These are the only fundamental requirement of the pattern. As a practical matter, the subject must have some query operations so observers can learn how the subject has changed. Observers need this information so that they can update themselves.

The subject may also provide operations for unregistering observers and controlling when it notifies its observers. Too frequent notifications, or notifications after trivial changes, may cause efficiency problems. Operations for controlling observer notifications can fix such problems.

Observer Pattern Structure

The Observer pattern has the form of a reactor pattern with the subject as the target and the observers as the reactors. The basic reactor form is elaborated with observer and subject interfaces. Furthermore, a subject must accept registrations from many observers. This form is shown in Figure 19-3-2.

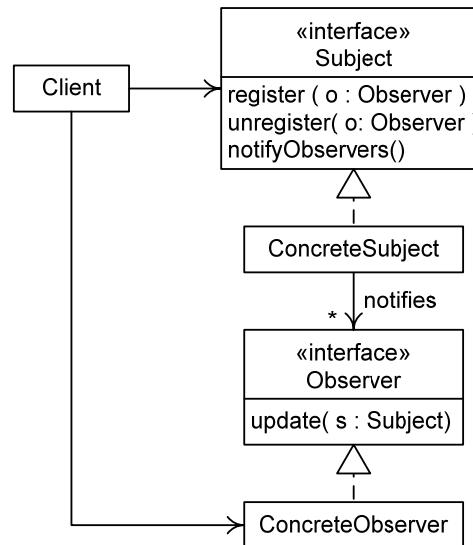


Figure 19-3-2 Observer Pattern Structure

The **Observer** interface guarantees that whatever object registers itself with a subject will implement an `update()` operation with the correct signature. Furthermore, any object of any class can register itself as an observer as long as it implements the **Observer** interface, and an observer can register itself with any object that implements the **Subject** interface. As a result, a subject may be completely ignorant of the nature of its observers, and an observer may observe arbitrary subjects. A single observer may register with several subjects because it can distinguish which one is notifying it by checking the parameter of the `update()` operation. The pattern is completely flexible in terms of the number and type of interacting observers and subjects.

Observer Pattern Behavior

Characteristic of a reactor pattern, Observer pattern behavior has two phases. In the setup phase, objects register with a subject as observers. In the operational phase, the subject notifies its observers whenever it changes in a way that may be of interest to them. Then observers usually need to query the subject to learn more about the nature of the change that prompted the notification. The sequence diagram in Figure 19-3-3 illustrates these two phases with a single `ConcreteSubject` and two `ConcreteObservers`. It includes `change()` and `getState()` operations in the `ConcreteSubject`. The `stimulus()` operation represents a computation that

alters the `ConcreteSubject` in a way that prompts notification of its observers. The `getState()` operation represents queries that observers may make to learn about the subject's new state.

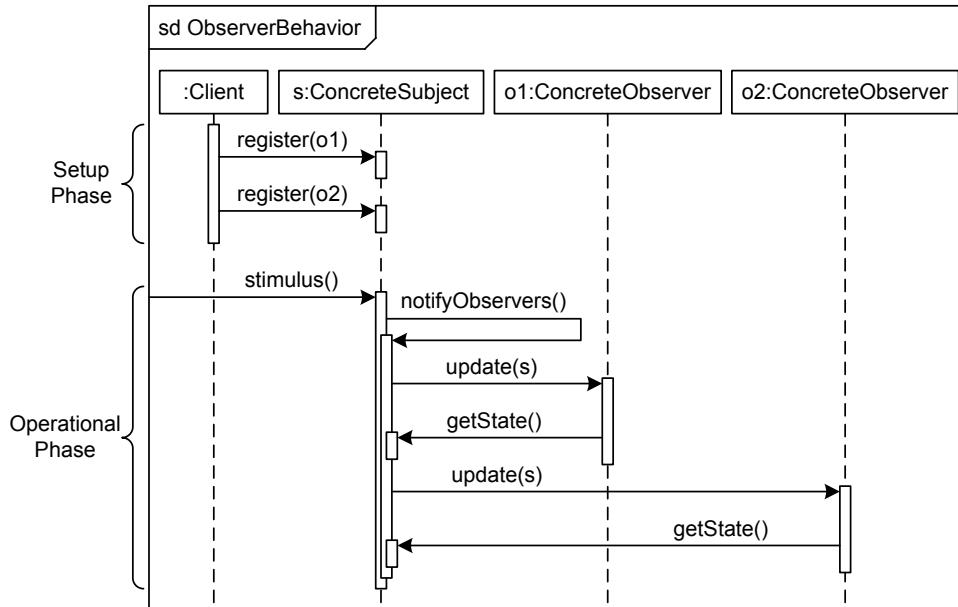


Figure 19-3-3 Observer Pattern Behavior

Although this sequence diagram is more complex than the reactor paradigm sequence diagram, it has the same form with more details. The setup phase shows two registrations of reactors (the concrete observers) with the target (the concrete subject). The operational phase begins with a `stimulus()` message indicating something that causes a change in the subject, as in the reactor paradigm. The `update()` operation is the Observer pattern version of the reactor pattern `notify()` message. The `notifyObservers()` message prompts observer notification, and the `getState()` messages are part of the reactor response to the notification.

An Observer Pattern Example: AquaLush

Several AquaLush modules are driven by time. Automatic irrigation occurs at certain times. During automatic irrigation, moisture sensors and water usage are checked every minute. The current time display in the control panel must be changed every minute.

All these modules could poll a clock from time to time, but an event-driven approach is clearly simpler, easier, and less error prone. Consequently, a central `Clock` object notifies modules that time has passed every minute. This could be implemented by having the `Clock` know about each time-driven module and calling special operations of each one to notify them that time has passed. However, this would couple the `Clock` tightly to other modules and make it harder to change the program. A much better

solution is to lower coupling with the Observer pattern. The diagram in Figure 19-3-4 illustrates application of the Observer pattern to the Aqualush clock.

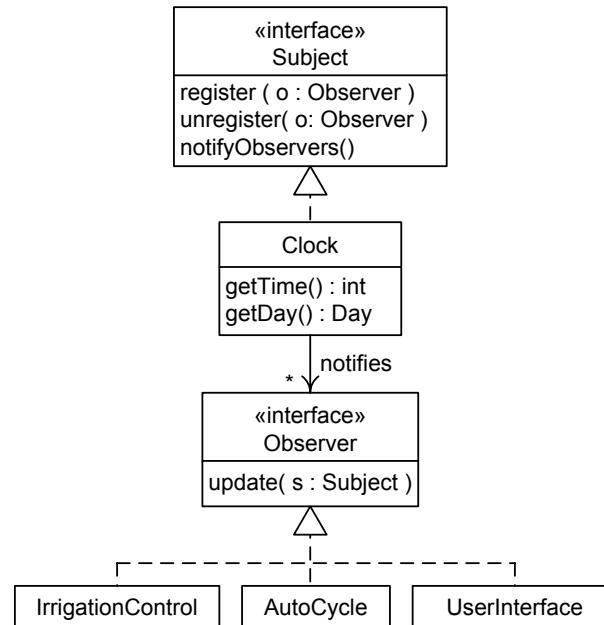


Figure 19-3-4 AquaLush Clock Using the Observer Pattern

The `Clock` class implements the `Subject` interface so it can be a target of observation. The time-driven modules implement the `Observer` interface so they can register as `Clock` observers. Among the observers are `IrrigationControl`, which decides when to start irrigation; `AutoCycle`, which controls automatic irrigation; and `UserInterface`, which controls user interaction, including time display. The `Clock` calls each observer's `update()` operation every minute. The observers can then use the `Clock`'s `getTime()` and `getDay()` operations to obtain the current time.

The client class has been left out of this diagram because each of the observers registers itself with the `Clock`, which also uses the Singleton pattern and hence is accessible to all of them. In this case, each observer plays the role of a client as well as a reactor.

When to Use the Observer Pattern

We have discussed the use of the Observer pattern in conjunction with the MVC architecture. The Observer pattern can be used in a similar fashion to decouple the user interface from the rest of the program even in programs that do not use the MVC architecture. We have also seen an example of using the Observer pattern in a program driven by a clock. Many programs that react to changes in time or the state of some device can also benefit from using the Observer pattern.

But the Observer pattern can be used in many other circumstances as well. It is a candidate for use in any situation where

- One or more objects must keep track of and react to the state of another, and
- The objects must be loosely coupled.

Such situations are extremely common, so the Observer pattern is very useful.

Observer Pattern Disadvantages

The observer pattern has two main drawbacks:

- It may be hard for people unfamiliar with patterns and object-oriented programming to understand. There are stories of maintenance programmers who have completely reworked programs to remove perfectly good uses of the Observer pattern because they did not understand it.
- Observer notifications may be expensive. Frequent notifications may lead to a great deal of observer activity, especially if a subject has many observers. The problem may be made worse if each observer needs to query the subject extensively to determine how it has changed.

Only improved software design education can solve the first problem; there are several ways to reduce the cost of using the Observer pattern:

- The observers may coordinate their activities with one another.
- The subject can provide operations to control the frequency and timing of observer notification.
- The subject can provide additional information about the nature of its changes in the notifications it sends to observers.
- The subject can provide several kinds of registration, allowing observers to register for only the sort of changes they are interested in.

Judicious use of these techniques can usually resolve Observer pattern efficiency problems.

Java Support

The `java.util` package provides an `Observer` interface and an extendable `Observable` class, as indicated in Figure 19-3-5.

The `Observer` interface declares the observer notification method `update()`, which requires a reference to the notifying `Observable` and an additional Object argument containing arbitrary additional information. The `Observable` class has operations for managing its observer list and two `notifyObserver()` operations. It also keeps a changed flag that must be set using `setChanged()` before observers will actually be notified. Classes can sub-class the `Observable` class to become subjects of observation and implement the `Observer` interface to become observers.

Although it is very handy to have observer list management and notification ready-made in the `Observable` class, it often cannot be used because Java does not support multiple inheritance. A class that must

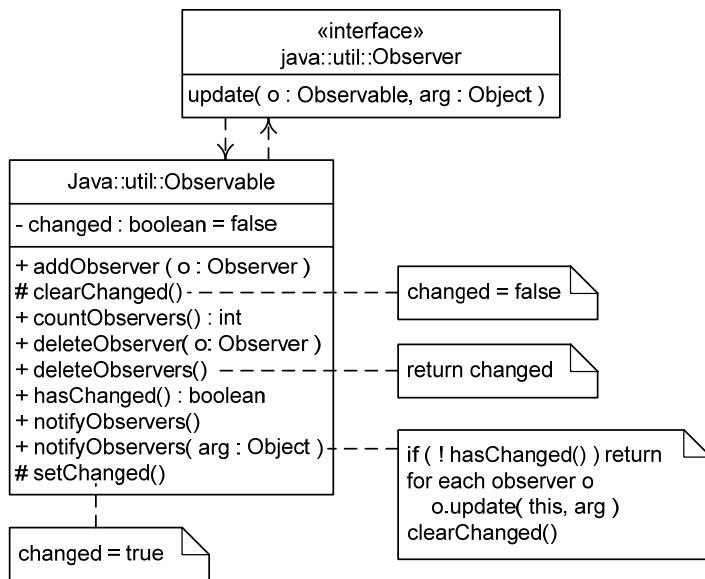


Figure 19-3-5 Java Observer Pattern Support

extend some other class cannot also extend `Observable`. Furthermore, if a class cannot extend `Observable`, it cannot use the `Observer` interface either because these two depend on one another: Either both must be used or neither. It would have been better to have declared an `Observable` interface used by `Observer` and then created an `AbstractObservable` class implementing the `Observable` interface with the implementation present in the current `Observable` class. This would allow use of the `Observable` and `Observer` interfaces even by subject classes unable to extend `AbstractObservable`.

Pattern Summary

Figure 19-3-6 summarizes the Observer pattern.

Name: Observer

Application: Decouple observer objects that must monitor and respond to changes in a subject object from the subject while maintaining their ability to monitor the subject.

Form: Structure and behavior both conform to the reactor paradigm, with the addition of observer and subject interfaces. A client registers observers with the subject in a setup phase. The subject notifies its observers of changes during an operational phase. The observers react to notifications by querying the subject to obtain data about the changes and then responding.

Consequences: Using an event-driven approach to decouple subjects from observers makes programs easier to change. Performance problems may arise from frequent notification or inefficient observer response, but these can be addressed by refining the notification and response strategies.

Figure 19-3-6 The Observer Pattern

Section Summary

- Frequently, observer objects must monitor and react to changes in a subject. The **Observer pattern** minimizes coupling between observers and subjects.
- The Observer pattern adheres closely to the reactor pattern form.
- The Observer pattern decreases coupling and therefore increases maintainability and reusability.
- Using the Observer pattern may lead to performance problems, but they can be addressed by refining use of the pattern.

Review Quiz 19.3

1. How can the Observer pattern be used to advantage in programs that employ the Module-View-Controller architectural style?
 2. Can an object be both a subject and an observer at the same time?
 3. What operations does the subject in an Observer pattern need to implement?
-

Chapter 19 Further Reading**All Sections**

The reactor pattern category is a specialization of the collaborator category proposed by Norton [2003]. All the patterns discussed in this chapter were first documented in [Gamma et al. 1995].

Chapter 19 Exercises**Section 19.1**

1. *Fill in the blanks:* Reactor design patterns have three classes: a client, a _____, and a _____. The client must have access to _____ so that the client can _____. The target must have access to the _____ so that it can _____.
2. Draw UML class diagrams to illustrate the differences in the static structures of the Event-Driven architectural style and reactor design patterns.
3. Could reactor patterns be used in a program with an Event-Driven architectural style? If not, why not? If so, what might be the advantages or disadvantages of doing this?

Section 19.2

4. Write a complete Java sort comparison program similar to the example discussed in this section.
5. Design an internal iterator for a sub-class of a Java collection class. Document your design in a class diagram and include comments containing pseudocode illustrating the details of how it would work.
6. Does the design you produced for the last exercise use the Command pattern? Explain why or why not.

7. A robot is controlled by several operations, including `move(d:Distance,s:speed)` and `turn(d:Degrees)`.
 - (a) How could function objects be used to compose robot scripts consisting of sequences of move and turn operations?
 - (b) The `move()` and `turn()` operations require different parameters. What problem does this present for using function objects to implement scripts? How can it be resolved?
 - (c) Design a mechanism to create and execute robot scripts and document it with a class diagram.
 - (d) Does your design use the Command pattern? Explain why or why not.
8. Examine the `java.util.Timer` and `java.util.TimerTask` classes. Do they use the Command pattern? Explain why or why not.
9. The `Zorister` class can experience two internal events to which clients should respond—namely, a crossing-lower-threshold event and a crossing-upper-threshold event. Use the Command pattern to design the following alternatives:
 - (a) A single command object that has a single operation with one parameter indicating the event type can be registered with the `Zorister`.
 - (b) A single command object can be registered with the `Zorister`, but it has two operations: one called by the `Zorister` when a crossing-upper-threshold event occurs, and the other called by the `Zorister` when a crossing-lower-threshold event occurs.
 - (c) Two command objects can be registered with the `Zorister`, one for each event, but the registered object classes must conform to a single command interface.
10. Compare and contrast the design alternatives you produced in the last exercise:
 - (a) What are the advantages and disadvantages of each one?
 - (b) Suppose that the response to each event is quite complex, with little processing overlap between the two event handlers. Which alternative would be best?
 - (c) Suppose that the response to each event is quite simple, with much processing overlap between the two event handlers. Which alternative would be best?

- Section 19.3**
10. Explain the differences between the Command and Observer patterns.
 11. Propose an implementation for the observer list maintained by a subject in the Observer pattern.
 12. Write Java code for `Observer` and `Subject` interfaces as indicated in the Observer pattern, and then write an `AbstractSubject` class that implements observer list management and observer notification. Why does your implementation provide better Observer pattern support than that provided by the `Java util` package?

13. Design a *Clock* class that refines use of the Observer pattern to improve performance. Specifically, the *Clock* class should offer an observer registration operation that allows observers to indicate the interval between notifications, specified in seconds. Express your design in a class diagram.
 14. Explain how the software you designed in the last exercise would improve performance.
- Research Project** 15. Consult the software engineering literature and find two examples of the use of reactor patterns different from those used as examples in this chapter. Write a paper in which you explain how the pattern is used and why the designers chose to use it.
- Team Project** 16. Form a team of two or three. Choose a reactor pattern, think of a simple example to illustrate it, and write a Java program implementing your example. Create a web page in which you explain the pattern and illustrate it with your example program.

Chapter 19 Review Quiz Answers

Review Quiz 19.1

1. Devices that report intermittent events, such as motion detectors or impact detectors, are good candidates for event-driven design treatment. These devices, or virtual devices, can notify reactors when the events they monitor occur.
2. The first phase of reactor pattern behavior is a setup phase during which a client registers a reactor with a target. The second phase is an operational phase during which the target notifies the reactor whenever an event of interest occurs, and the reactor responds.
3. An Event-Driven architectural style requires use of an event dispatcher, which is more complex, harder to build and maintain, and may not perform as well as a design using reactor patterns. However, an event dispatcher completely decouples targets and reactors, making programs very configurable and changeable. While reactor patterns decrease coupling between targets and reactors, they do not eliminate it.

Review Quiz 19.2

1. A callback function is an operation passed to a user interface component that allows it to invoke an application operation in response to a user action. Callback functions are standard mechanisms in non-object-oriented graphical user interface toolkits.
2. Function objects allow operations to be treated like values in object-oriented languages that do not directly support this facility. Function classes also provide a means for adding features to complement the encapsulated operation, and they can include data and operations needed to use the encapsulated operations, making them more reusable and flexible.
3. The Command pattern can be elaborated by adding more invoker registration options, allowing multiple command registration, adding registration management operations, and adding operations to the *Command* interface.

**Review
Quiz 19.3**

1. The MVC architectural style has views and controllers in a user interface component and models in a problem domain component. Ideally these components are layers, which means that the models should not use the views and controllers. If the Observer pattern is used to make the views and controllers into observers and the models into subjects, then the models do not use the views and controllers. This is because a subject in the Observer pattern does not use its observers. In other words, the subject does not depend on the presence of correct versions of the observers to achieve its goals. In fact, the subject does not even need any observers to be present. Hence, the Observer pattern is a model for the relationship between controllers, views, and models in the MVC style.
2. An object can be both a subject and an observer at the same time because a class can implement both **Subject** and **Observer** interfaces. This sometimes occurs. For example, a simulated vehicle object may be an observer of a clock because it needs to change its position over time, and also a subject observed by a user interface component that must redisplay it when it changes position.
3. The subject in an Observer pattern needs to implement operations to manage observer registration (at least a `register()` operation) and observer notification operations (at least one).



A Glossary

abstract class—a class that cannot be instantiated; contrast with **concrete class**.

abstract data type (ADT)—an ordered pair consisting of a set of values (the **carrier set** of the type) and operations for manipulating those values.

Abstract Factory pattern—a **generator pattern** in which clients using a special generator object create several kinds of product instances; the generator and product classes realize interfaces or specialize abstract classes.

abstract operation—an unimplemented operation; an operation signature without a body.

abstraction—suppressing or ignoring some properties of objects, events, or situations in favor of others; the opposite of **refinement**.

acceptor—a finite automaton that responds to events but generates no actions; also called a **recognizer**; see **transducer**.

accessible—a program entity is accessible at a point in a program text if it may be used at that point.

action—in UML, an atomic task or procedure.

active—an operation is active when it is either **executing** or **suspended**; an object is active when one or more of its operations are active.

active review—an examination of a work product by qualified individuals who are asked to answer questions about specific aspects of the work product.

activity—in UML, a non-atomic task or procedure decomposable into actions.

activity diagram—a UML notation showing actions and the flow of control and data between them.

actor—a type of agent that interacts with a product.

adapter—a component that provides a new interface for an existing component.

Adapter pattern—a **broker pattern** in which an adapter or wrapper object provides client objects with a modified supplier object interface; also called a **Wrapper pattern**.

address—the location of the storage cell in computer memory where a value is stored.

adequacy—the degree to which a design satisfies the specifications and conforms to the constraints of a design problem.

aesthetic principle—a statement that designs with certain characteristics are better than those without because they are more beautiful or pleasing to the mind or senses.

aggregation association—in UML, a representation of the part-whole relation between the instances of the associated classes; see **composition association**.

algorithm—a sequence of computational steps that transforms inputs into outputs.

alias—a variable with the same address as another variable.

analysis—the activity of breaking down a design problem for the purpose of understanding it.

analysis class model—see **conceptual model**.

analysis model—any representation of a design problem.

application domain—an area of knowledge or skill; also called a **problem domain**.

architectural design—the activity of specifying a program’s major parts; their responsibilities, properties, and interfaces; and the relationships and interactions among them. Also the specifications that result from this activity.

architectural style—a paradigm of program or system constituent types and their interactions.

architecture—see **software architecture**.

artifact—in UML, any physical representation of data used or produced during software development or software product operation.

assertion—a statement that must be true at a designated point in a program.

asset—any artifact that can be reused during software development.

association—in UML, a connection between classes representing a relation on the sets of instances of the connected classes.

association class—in UML, a class connecting other classes that represents a relation on the sets of the instances of the connected classes and also encapsulates attributes and operations pertaining to this relation.

association qualifier—a collection of one or more attributes that, in conjunction with an instance of the qualified class, partitions the instances of another class participating in a binary association with the qualified class.

assumption—a state of affairs taken for granted in product development.

attribute—a data item held by an object or a class.

atomic requirements statement—a natural-language requirements statement that specifies a single function, feature, characteristic, or property of a product that has a unique identifier.

atomizing requirements—splitting natural-language statements of requirements until each statement is atomic and associated with a unique identifier.

attribute—a datum held by an object.

audit—a review of a work product by a team of qualified individuals outside the organization that produced the work product.

availability—the readiness of a program for use.

basic design principle—a statement that certain design characteristics make a design better because it is better able to meet stakeholder needs and desires.

basic flow—in a use case description, a specification of a typical successful interaction between the product and its actors.

beauty—the aesthetic appeal of a design.

behavioral requirement—see **functional requirement**.

Blackboard architectural style—a Shared-Data architectural style in which shared-data accessors are activated by a shared-data store (called a *blackboard*) when it is changed; see **Shared-Data architectural style** and **Repository architectural style**.

bloated controller—a controller that is too large and complex; see **controller**.

bottom-up strategy—an approach to problem-solving that proceeds by solving pieces of a large problem in detail and then connecting the solved pieces to form a solution for the entire problem.

box-and-line diagram—a design notation that uses arbitrary symbols or icons (*boxes*), usually connected by various sorts of *lines* to model software.

branch point—a place in a use case description where the action flow may diverge.

brief—a short description of an actor or use case supplementing a use case diagram.

broker pattern—a category of mid-level design patterns in which instances of a broker class mediate the interaction between client and supplier class instances.

business requirement—a statement of a client or development organization goal that a product must meet.

callout—a note attached to a line or arrow picking out part of a graphic.

calls relation—module *A* *calls* module *B* when *A* triggers execution of *B*; also called **invokes relation**.

cardinality constraint—a specification of how many instances of a relationship a particular entity filling a relationship role can participate in.

carrier set—the set of values manipulated in an abstract data type.

centralized control style—a control style in which a few controllers make all significant decisions.

clarity—a document has clarity if it is **clear**.

class—an abstraction of a set of objects with common operations and attributes.

Class Adapter pattern—a **broker pattern** in which an adapter class provides client objects with a modified supplier interface by sub-classing the supplier class.

class diagram—a graphical representation of classes in a problem domain or a software solution.

class invariant—an assertion that must be true of any class instance between calls of its exported operations.

class model—a representation of classes in a problem or a software solution.

class operation—an operation encapsulated by a class and callable without recourse to any instance of the class; see **instance operation**.

class variable—an attribute whose value is shared by each instance of a class; see **instance variable**.

clear—a document is clear if it is easy for qualified readers to understand.

client—someone outside a development organization with an interest in a product, such as a customer or a user.

clone—a copy of an object.

cohesion—the degree to which a module's parts are related to one another.

collection—an object that holds or contains other objects.

collection iteration—the process of serial access of each element of a collection; also called **iteration over a collection**.

Command pattern—a reactor pattern that uses a function object as its reactor.

comment—in UML, an arbitrary text string attached to any UML model element.

complete—a document is complete if it contains all relevant material.

completeness—see **complete**.

component—a part of a program or system; in UML, a modular, replaceable unit with well-defined interfaces.

component and interaction co-design—the process of generating, evaluating, and improving designs of components and interactions together.

component diagram—a UML diagram that shows (UML) components, their relationships to their environments, and possibly their internal structures.

component-based development—a software development approach in which products are designed and built using commercially available or custom-built software components.

composite name—in UML, a sequence of one or more simple names separated by the composite name delimiter (usually the double colon ::); see **simple name**; also called a **qualified name**.

composition association—in UML, a representation of the part-whole relation between the instances of the associated classes in which each part is a member of a single whole; see **aggregation association**.

conceptual model—a static analysis model of the important entities in a problem, their responsibilities or attributes, the important relationships among them, and perhaps some of their behaviors.

concrete class—a class that may be instantiated; contrast with **abstract class**.

concrete operation—an operation that has an implementation; contrast with **abstract operation**.

concurrency—the simultaneous execution of more than one computational activity on a single computer system.

consistency—a specification is consistent if all of its assertions can be satisfied by a single entity.

consistent—see **consistency**.

constraint—any factor that limits developers and particularly one that limits design solutions; in UML, a property that a model element must satisfy or a relationship between model elements that must be maintained.

constructive design principle—a statement, based on design experience, that certain engineering design characteristics make a design better.

container class—a class whose instances hold a collection of objects.

continuous review—the practice of frequently evaluating work products during their creation to ensure their high quality upon completion; see **critical review**.

contract—a binding agreement between two or more parties.

control flow branch—a point in a control flow where alternative paths may be taken.

control style—a way that decision making is distributed among program components.

controller—a program component that makes decisions and directs other components.

copy constructor—a constructor that takes an instance of its class as an argument and creates a clone of its argument; see **clone**.

correct—a statement is correct if it is contingent (may be true or false) and accords with the facts.

correctness—a document has correctness if all its statements are **correct**.

coupling—the degree of the connection between pairs of modules.

critical review—an evaluation of a finished product to determine whether it is of acceptable quality; see **continuous review**.

customer—someone who pays for a product.

data requirement—a statement that certain data must be input to, output from, or stored in a product.

data responsibility—an obligation to maintain some data; see **responsibility**.

data store—a data storage mechanism under a system's control.

data structure—a scheme for storing values in computer memory.

data structure diagram—a notation depicting data structures that uses rectangles to represent memory cells and arrows to represent pointers or references to memory cells.

data type—a set of values (the **carrier set** of the type) and a collection of operations for manipulating and examining values in the carrier set.

deadlock—a set of activities is *deadlocked* when each member of the set can proceed only if some other member of the set causes some event to occur.

declarative specification—a statement about what an entity is or how it behaves that does *not* use an algorithm to describe it; see **algorithm** and **procedural specification**.

deep copy—a copy operation in which non-referenced values stored in an original entity are reproduced in the copy, but referenced entities in the original composite entity are copied and references to the new entities are placed in the copy; contrast with **shallow copy**.

defensive copy—a copy of a variable’s value that a module provides to other modules as a means of hiding information.

delegated control style—a control style in which decision making is distributed through a program.

delegation—a tactic wherein one module (the *delegator*) entrusts another module (the *delegate*) with a responsibility.

deliverable—any item produced for someone else, such as a customer, client, or coworker.

dependency relation—a binary relation that holds between two entities *I* and *D* when a change to *I* (the *independent entity*) may affect *D* (the *dependent entity*); contrast with **association**.

deployment diagram—a UML diagram that models computational resources, the communication paths between them, and the artifacts that reside in and execute on them.

DeSCRIPTR—an acronym for “decomposition, states, collaborations, responsibilities, interfaces, properties, transitions, and relations,” which are aspects of engineering design specifications; see also **PAID**.

design—the activity of specifying a product or service satisfying client needs and desires, subject to constraints; also the specifications that result from this activity.

design by contract—using pre- and postconditions to specify an operation by stating the rights and obligations of the operation and its callers.

design class model—a representation of the classes in a software system and their attributes, operations, and associations in abstraction from language and implementation details.

design document—a complete engineering design specification comprised of a **software architecture document** and a **detailed design document**.

design heuristic—a rule of thumb or a procedure that helps generate and select good designs and design documentation; see **heuristic**.

design method—an orderly procedure for generating a precise and complete software design solution that meets client needs and constraints.

design notation—a symbol system for expressing designs; see **notation**.

design pattern—a design, part of a design, or a template for a design that can be imitated in generating new designs; see **pattern** and **software design pattern**.

design principle—a statement that certain characteristics make a design better, used as a quality criterion in generating and evaluating designs; also called a **design tenet**.

design process—a sequence of steps for understanding a design problem and formulating, evaluating, and improving designs until a satisfactory design is created and documented; see **process**.

design rationale—an explanation of why a design is the way that it is, usually consisting of a justification of important design decisions.

design story—a very brief (several paragraphs) description of an application stressing its most important aspects.

design task—a small job done in the design process.

design team—a group of individuals with the skills and talents necessary to completely specify the features, behavior, form, structure, and workings of a product.

design tenet—see **design principle**.

design theme—an important problem, concern, or issue that must be addressed in a design.

desk check—an examination of a work product by an individual.

detailed design—the activity of specifying the internal elements of all major program constituents; their structures, relationships, and processing; and often their algorithms and data structures. Also the specifications that result from this activity.

detailed design document (DDD)—a document that specifies a program's detailed design.

deterministic finite automaton—a finite automaton that has no spontaneous transitions and has a single transition that it must make in response to every event in each of its states; see **finite automaton**.

device interface module—a portion of a program containing program units responsible for communicating with external systems or machines and for hiding the details involved in such communications.

dialog map—a state diagram whose nodes represent user interface states.

dispersed control style—a control style in which decision making is dispersed widely through a program.

distributed systems—systems formed by several programs running on different machines that communicate with one another and coordinate their activities.

divide and conquer—a problem-solving heuristic in which a large problem is solved by decomposing it into several smaller problems, solving those, and then combining the solutions into a solution for the large problem.

dynamic design model—a representation of the way various aspects of a software system change during execution.

economy—the degree to which a design can be built cheaply, quickly, and with minimal risk.

electronic prototype—a program that behaves like the product it models in some way.

element property—in UML, a characteristic of a model element.

elicitation workshop—a facilitated and directed discussion aimed at describing a product design problem and establishing stakeholder needs and desires.

encapsulation—packaging program elements, such as data structures, constants, and executable statements, in a program unit.

engineering design—the activity of specifying the technical mechanisms and workings of a product; also the specifications that result from this activity.

engineering judgment—the opinion of a designer about the quality of a design based on experience and careful consideration of the advantages and disadvantages of the alternatives.

estimation—calculation of the approximate cost, effort, time, or resources required to achieve some end.

event—a noteworthy occurrence at a particular time.

event list—a list of all the occurrences, both internal and external, to which a product must respond in some way.

Event-Driven architectural style—an architectural style in which program constituents announce events to an event dispatcher that then invokes operations of constituents that have registered interest in those events; also called the **Implicit-Invocation architectural style**.

event-driven design—an approach to program design that focuses on **events** to which programs must react.

evolutionary prototype—a prototype developed a little at a time until it becomes the final product.

executing—an operation is executing when some process is running its code; see **active**.

exploratory prototype—a prototype developed merely as a design aid and then discarded once the design is complete; also called a **throwaway prototype**.

exported—an entity is exported from the module where it is defined when it is visible outside the module; see **visible**.

extension—in a use case description, a specification of an alternative interaction between the product and its actors documented elsewhere in the description.

external iteration control—a form of iteration mechanism in which a client controls access to collection elements; see **iteration mechanism**; contrast with **internal iteration control**.

external iterator—an iterator that provides access to collection elements under the control of the client; see **external iteration control** and **iterator**.

Façade pattern—a **broker pattern** in which a façade object provides a client object with a simplified interface to a complex sub-system of supplier objects.

factory class—a class that exists only to provide one or more **factory methods**.

factory method—a non-constructor operation that creates and returns class instances.

Factory Method pattern—a **generator pattern** in which clients using a generator object to create product instances are decoupled from the generator and product classes by interfaces or abstract super-classes.

Factory pattern—a **generator pattern** in which the interfaces and super-classes are used to generalize the factory and product classes.

feasibility—whether a design can be built.

feature—an **attribute** or **operation** of a class.

fidelity—how closely a prototype represents the final product it models.

field—a variable implementing an **attribute**.

filter—a program component that transforms a stream of input data into a stream of output data.

filtering iterator—an iterator that provides access to elements of a collection meeting some criterion; see **iterator**.

finite automaton—an abstract computational device consisting of a finite number of states, one of which is distinguished as the *initial state* (where state transitions begin), and transitions between states that occur spontaneously or are triggered by events; also called a **finite state machine**; see **deterministic** and **non-deterministic finite automaton**.

finite state machine—see **finite automaton**.

focus group—an informal discussion among a small group of people (typically six to nine) led by a facilitator who keeps the group on topic.

fork—a point where a single flow of control is broken into two or more concurrent flows of control.

function class—a class whose instances are **function objects**.

function object—an object that encapsulates an operation; also called a **functor**.

functional requirement—a statement of how a software product must map program inputs to program outputs; also called a **behavioral requirement**.

functor—see **function object**.

generalization—in UML, the relation that obtains between one model element (the parent) and another (the child) when the child is a special type of the parent.

generator pattern—a category of mid-level design patterns in which generator class instances produce instances of a product class for use by client class instances.

global visibility—the visibility of an entity when that entity is accessible by name throughout a program.

guard—in UML, a Boolean expression between square brackets.

helper operation—an operation needed to implement an exported operation.

heterogeneous architecture—an architecture that employs two or more architectural styles.

heuristic—a rule providing guidance, but no guarantee, for achieving some end; see **design heuristic**.

horizontal prototype—a working model of some or all of a product's user interface.

implementability principle—a statement that certain engineering design characteristics make a design better because it is more likely to be realized quickly, cheaply, and successfully.

implementation—the creation of executable artifacts for delivery to customers in a software product; the relation between an abstract operation and a concrete operation that overrides it; the relation between an interface type and a concrete class that provides implementations for all its abstract operations.

implementation class model—a representation of the classes in a software system showing implementation and language details left out of design class models.

Implicit-Invocation architectural style—see **Event-Driven architectural style**.

information hiding—shielding the internal details of a module's structure and processing from other modules.

inheritance—a declared relation between classes that causes every attribute and operation from the inherited class or classes (the *super-class* or *super-classes*) to be an attribute or operation of the inheriting class (the *sub-class*).

inspection—a formal review of a work product by a trained team of inspectors, with assigned roles, using a checklist to find defects.

instance operation—an operation encapsulated in class instances and callable only using instances of the class; see **class operation**.

instance variable—an attribute whose values are held separately by each instance of a class; see **class variable**.

interaction—in UML, the communication behavior of individuals exchanging information to accomplish a task.

interaction design—the activity of specifying products that people are able to use effectively and enjoyably.

interaction diagrams—in UML, notations for modeling the communication behavior of individuals exchanging information to accomplish a task, including sequence, communication, interaction overview, and timing diagrams.

interface—a boundary across which two entities communicate; in Java, a named collection of abstract operations and named constants that is also called an **interface type**; in UML, a named collection of public attributes and abstract operations.

interface specification—a description of the mechanism an entity uses to communicate with its environment.

interface type—in Java, a named collection of abstract operations and named constants.

internal iteration control—a form of iteration mechanism in which the iteration mechanism applies an operation to each element of a collection on behalf of a client, thus retaining control of access to collection elements; see **iteration mechanism**; contrast with **external iteration control**.

internal iterator—an iterator that accepts an operation and applies it to each element of a collection; see **internal iteration control** and **iterator**.

interview—a question and answer session.

invokes relation—see **calls relation**.

iteration mechanism—a language feature or a set of operations that allows clients to access each element of a collection.

iteration over a collection—see **collection iteration**.

iterative planning and tracking—making a rough base or initial project plan, and refining it at fixed periods during a project in light of tracking data and completed work products.

iterator—an entity that provides serial access to each element of an associated collection.

Iterator pattern—a **mid-level design pattern** for implementing external iterators.

join—a point where two or more concurrent flows of control are merged into a single flow of control.

Law of Demeter—the rule that an operation of an object *obj* should send messages only to the following entities: object *obj*; the attributes of *obj*; the arguments of the operation; the elements of a collection that is an argument of the operation or an attribute of *obj*; objects created by the operation; and global classes and objects.

layer—a program module in a **Layered architectural style**.

Layered architectural style—an architectural style in which a program is partitioned into cohesive modules with well-defined interfaces (**layers**) that are arranged in an array such that each layer can use only the layer directly below it (*Strict*) or any of the layers below it (*Relaxed*).

leadership—directing and helping people doing work.

leading—see **leadership**.

lexical analyzer—a program component that transforms a stream of characters into a stream of *tokens*, or symbols recognized by the program.

link—in UML, a connection between objects indicating that the *n*-tuple of objects is a member of a relation.

literal—a string denoting a particular constant value.

local visibility—the visibility of an entity when that entity is accessible by name only within the module where it is defined.

logical architecture—the configuration of a product’s major constituents and their relationships to one another in abstraction from the product’s implementation as code and its execution on actual machines.

low-level design—the specification of small details at the lowest levels of abstraction.

Maintainability—the ease with which a product can be corrected, improved, or ported.

management—the activity of assembling, directing, and supporting human and other resources in accomplishing tasks beyond the capacity of an individual.

market—a set of actual or prospective customers who need or want a product, have the resources to exchange something of value for it, and are willing to do so.

marketing—the process of conceiving products and planning and executing their promotion, distribution, and exchange with customers.

Mediator pattern—a **broker pattern** in which a mediator object facilitates interactions among several other objects. Each object is regarded as a client that interacts with the rest of the objects as suppliers through the brokering mediator.

message—a request that an object perform one of its operations or provide access to one of its attributes.

method—an implementation of an operation; see **design method**.

mid-level design—the activity of specifying software at the level of medium-sized components, such as compilation units or classes, and their properties, relationships, and interactions; also the specifications that result from this activity.

mid-level design pattern—software design patterns modeling collaborations between several modules (usually classes); see **software design pattern**.

milestone—any significant event in a project.

minispec—a step-by-step description of how a process or operation transforms its inputs to outputs.

mock-up—an executable model of a program’s user interface.

model—an entity used to represent another entity, called the *target*, by establishing (a) a correspondence between the parts or elements of the target and the parts or elements of the model, and (b) a correspondence between relationships among the parts or elements of the target and relationships among the parts or elements of the model.

Model-View-Controller (MVC) architectural style—an architectural style in which user interface view components display the state of problem domain model components and both the views and the models are altered by user interface controller components.

modular—a program or system composed of well-defined, conceptually simple, and independent units that communicate through well-defined interfaces.

modularity principle—a statement that modules with certain characteristics are better than those without them.

module—a program unit with parts.

multiplicity—in UML, a constraint on the size of a collection or a property of a relation from a set A to a set B indicating how many elements of set B may be related to a single element of set A .

name—an identifier bound to a program entity; in UML, a string identifying a model element.

navigable—an association between classes A and B is navigable from class A to class B if an instance of A can access an instance of B .

need statement—a sentence that documents a single product feature, function, or property needed or desired by some stakeholder.

needs elicitation—see **requirements elicitation**.

needs identification—see **requirements elicitation**.

needs list—a catalog of need statements; see **need statement**.

node—in UML, a computational device having at least some memory and usually also processing capabilities.

non-behavioral requirement—see **non-functional requirement**.

non-deterministic finite automaton—a finite automaton that has spontaneous transitions or more than one transition that it may make in response to some event in some of its states; see **finite automaton**.

non-functional requirement—a statement that a software product must have certain properties; also called a **non-behavioral requirement**.

non-local visibility—the visibility of an entity when that entity is accessible by name outside the module where it is defined, but not accessible by name in an entire program.

notation—a symbolic representational system; see **design notation**.

note—in UML, a dog-eared rectangle containing arbitrary commentary text attached to zero or more diagram elements by dashed lines.

notification—an interaction style in which a subject notifies an observer when some subject condition becomes true.

object—a distinct encapsulation of data and behavior.

Object Adapter pattern—a **broker pattern** in which an adapter object provides client objects with a modified supplier interface by keeping a reference to the supplier class and delegating work to it.

object diagram—a graphical representation of objects in a problem domain or in a software solution.

object model—a representation of objects in a problem or a software solution.

object-oriented paradigm—an approach to software development that organizes software systems as collections of interacting encapsulations of data and behavior (objects).

observation—a requirements elicitation technique in which designers study stakeholders as they go about their work.

Observer pattern—a **reactor pattern** in which the reactor and target classes are decoupled by depending on subject and observer interfaces or abstract classes rather than on concrete reactor classes; also called the **Publish-Subscribe pattern**.

operation—an object or class behavior; a unit of activity; a sub-program.

operation specification—a structured text specifying an operation’s interface and responsibilities, and possibly its implementation.

operational responsibility—an obligation to perform a task; see **responsibility**.

operational-level requirement—a statement about individual inputs, outputs, computations, operations, calculations, characteristics, and so forth that a product must have or provide without reference to physical realization.

operations—standardized activities that occur continuously or at regular intervals.

opportunity funnel—a mechanism for collecting product ideas from diverse sources.

opportunity statement—a brief description of a product development idea.

organization chart—a tree or other hierarchical display of the positions in an organization and the reporting relationships among them, often including the names of the individuals filling each position.

organizing—see **project organizing**.

outside-in design—the process of designing collaborations between program components beginning with a program’s interactions with its environment.

package—in UML, a collection of model elements.

package diagram—a UML diagram whose primary symbols are package symbols.

package visibility—an entity has package visibility if it is visible in the class where it is defined as well as all classes in the same package or namespace.

PAID—an acronym for “packaging, algorithms, implementation, and data structures,” which are aspects of low-level detailed design specifications; see also **DeSCRIPTR**.

paper prototype—a rough model of a product constructed from paper, note cards, whiteboards, or cardboard boxes.

participatory design—a product design technique in which stakeholders work alongside designers in generating, refining, and evaluating design solutions.

pattern—a model proposed for imitation; see **design pattern**.

performance—the ability of a program to accomplish its function within limits of time or computational resources.

persistence—the ability of data to survive the process that creates it.

physical architecture—the realization of a product as code and data files residing and executing on computational resources.

physical-level requirement—a statement about the details of the physical form of a product, its physical interface to its environment, or its data formats.

pipe—a conduit over which a stream of data flows.

Pipe-and-Filter architectural style—an architectural style in which program constituents are **filters** connected by **pipes**.

planning—see **project planning**.

pointer—a location in a computer's memory.

polling—an interaction style in which an observer repeatedly queries a subject to find out when some subject condition becomes true.

polymorphism—the ability to resolve references to identically named entities based on distinctions of class membership, object encapsulation, and operation signature.

postcondition—an assertion that must be true upon completion of an activity or operation.

pragmatics—a specification of how legitimate messages of a communications medium are used in context to accomplish certain tasks.

precondition—an assertion that must be true at the initiation of an activity or operation.

Principle of Adequacy—the statement that designs that meet more stakeholder needs and desires, subject to constraints, are better.

Principle of Beauty—the statement that beautiful (simple and powerful) designs are better.

Principle of Changeability—the statement that designs that make a program easier to change are better.

Principle of Cohesion—the statement that module cohesion should be maximized.

Principle of Coupling—the statement that module coupling should be minimized.

Principle of Design for Reuse—the statement that designs that produce reusable assets are better.

Principle of Design with Reuse—the statement that designs that reuse existing assets are better.

Principle of Economy—the statement that designs that can be built for less money, in less time, with less risk, are better.

Principle of Feasibility—the statement that a design is acceptable only if it can be realized.

Principle of Information Hiding—the statement that each module should shield the details of its internal processing from other modules.

Principle of Least Privilege—the statement that modules should not have access to unneeded resources.

Principle of Simplicity—the statement that simpler designs are better.

Principle of Small Modules—the statement that designs with small modules are better.

private—something intended for the exclusive use of some individual or group; something that is not part of an entity’s interface; an access specifier that restricts access to an enclosing entity.

private visibility—an attribute or operation has private visibility if it is visible only within the class where it is defined.

problem concept—an important entity in a problem.

problem domain—an area of knowledge and expertise that must be understood in designing a product.

problem domain glossary—an alphabetical listing of terms in a problem domain with definitions, explanations, and cross references.

procedural specification—a statement of an algorithm to describe how an entity carries out its behavior; see **algorithm** and **declarative specification**.

procedure—a collection of instructions directing computation; a sub-program.

process—a collection of related tasks that transforms a set of inputs into a set of outputs (see **design process**); a computational activity with its own set of resources (see **thread**).

process specification—see **minispec**.

product—a raw material, artifact, service, or idea satisfying client needs and desires that a customer is willing to exchange something of value to obtain.

product category—a dimension along which products may differ; see **product type**.

product design—the activity of specifying the functions, capabilities, behavior, and properties of a product in response to client needs and desires, including its aesthetics, styling, and the way that it interacts with its environment; also the specifications that result from this activity.

product plan—a list of approved development projects, with start and delivery dates.

product scope—the features and capabilities of a product; see **scope**.

product type—a collection of products that have the same value in a particular product category; see **product category**.

product vision statement—a general description of a product’s purpose and form.

profile—a set of scenarios used to evaluate whether a product is likely to meet a set of requirements.

program entity—anything in a program that is treated as a unit.

programming idiom—a standard way to do something in a particular programming language.

project—a one-time effort to achieve a particular, current goal of an organization, subject to specific time or cost constraints.

project mission statement—a document defining a development project’s goals and limits.

project organizing—structuring the organizational entities involved in a project and assigning responsibility and authority to these agencies.

project planning—formulating a scheme for doing a project.

project scope—the work to be done in a project; see **scope**.

project staffing—filling the positions in an organizational structure and keeping them filled with appropriate individuals.

project tracking—observing the progress of work and adjusting work and plans accordingly.

proof of concept prototype—see **vertical prototype**.

property—in UML, a named value attached to a model element; a characteristic of a thing.

protected visibility—an attribute or operation has protected visibility if it is visible in the class where it is defined as well as in all its sub-classes.

protocol—a specification of the ordering or timing of relationships between events or messages passed between entities.

prototype—a working model of part or all of a final product.

Prototype pattern—a **generator pattern** in which a client obtains correctly configured product instances by asking an already configured product to clone itself; the client is decoupled from the product class by an interface declaring the cloning operation.

provided interface—an interface realized by a component; see **interface**; contrast with **required interface**.

Proxy pattern—a **broker pattern** in which a proxy object interacts with client objects in lieu of a supplier object.

pseudocode—English augmented with programming language constructs.

public—something intended for use by anyone; something that is part of an entity’s interface; an access specifier that allows access by any object with access to an enclosing entity.

public visibility—an attribute or operation has public visibility if it is visible in the class where it is defined and anywhere its class is visible.

Publish-Subscribe pattern—see **Observer pattern**.

qualified name—see **composite name**.

quality attribute—a characteristic or property of a software product independent of its function that is important in satisfying stakeholder needs and desires.

quality gate—a mechanism that keeps poor-quality work products from moving on to the next step of a process.

reactor pattern—a category of mid-level design patterns in which reactor class instances respond to events generated by target class instances on behalf of client class instances.

realization—in UML, the relation that holds between an interface and a class or component when the class or component includes the interface’s attributes and implements its operations.

recognizer—see **acceptor**.

reference—an expression that evaluates to an address where a value is stored.

refinement—adding details about the properties of objects, events, or situations in the course of problem solving; the opposite of **abstraction**.

relation—a set of tuples; a relation R on sets A_1, A_2, \dots, A_n is a subset of $A_1 \times A_2 \times \dots \times A_n$.

relationship—a connection between entity-types whose elements bear a relation to one another; a single connection between particular entities is an *instance* of the relationship.

relationship role—a way that an entity-type can participate in a relationship.

reliability—the ability of a program to behave in accord with its requirements under normal operating conditions.

Repository architectural style—a Shared-Data architectural style in which the shared-data stores (called *repositories*) are passive and do not activate shared-data accessors; see **Shared-Data architectural style** and **Blackboard architectural style**.

required interface—an interface on which a component depends; see **interface**; contrast with **provided interface**.

requirement—a statement that a product must have a certain feature, function, capability, or property.

requirements consistency—a set of requirements is **consistent** if a single product can satisfy them all.

requirements development—the activity of creating product requirements.

requirements elicitation—the activity of determining stakeholders' needs and desires for a product.

requirements engineering—the activity of creating, modifying, and managing requirements over the life of a product.

requirements management—the activity of ensuring that requirements are realized in products and controlling requirements changes.

requirements traceability—the ability to track requirements from their expression in an SRS to their realization in design documentation, source code, and user documentation and their verification in reviews and tests.

requirements validation—the activity of confirming with stakeholders that a product design satisfies their needs and desires.

resolution—the activity of solving a design problem.

responsibility—an obligation to perform a task (an **operational responsibility**) or to maintain some data (a **data responsibility**).

responsibility-driven decomposition—a technique for program decomposition in which component responsibilities are decomposed and used to generate sub-components.

reusability—the degree to which the parts of a software product can be used in other products.

review—an examination and evaluation of a work product or process by qualified individuals or teams.

risk—any occurrence with negative consequences.

risk analysis—an orderly process of identifying, understanding, and assessing **risks**.

risk management—an orderly **risk analysis** followed by efforts to contain, control, or mitigate **risks**.

robust iteration mechanism—an iteration mechanism that conforms to some coherent specification of behavior when its associated collection is changed during iteration; see **iteration mechanism**.

rolename—in UML, a name for the role played by an object in a relation represented by an **association** or **link**.

scenario—an interaction between a product and particular individuals.

scenario description—a statement of a scenario.

schedule—a specification of the start and duration of work tasks.

scope—extent of control, influence, or responsibility; see **project scope** and **product scope**.

scoring matrix—a table showing design alternatives in the columns and weighted selection criteria in the rows that is used to select a design alternative.

secret—something kept from knowledge or view.

security—the ability of a program to resist being harmed or causing harm by hostile acts or influences.

semantics—a specification of the meanings of legitimate messages in a communications medium.

shallow copy—a copy operation in which values (including references) stored in an original entity are reproduced in the copy; contrast with **deep copy**.

Shared-Data architectural style—an architectural style in which one or more shared-data stores are used by one or more independent shared-data accessors; see **Blackboard architectural style** and **Repository architectural style**.

signature—a specification of the name, parameters (including types and possibly names), and return type of an operation, method, procedure, function, or sub-program. A signature may also include specification of thrown exceptions, visibility, and other operation characteristics, depending on the language.

simple name—in UML, a string of letters, digits, or punctuation characters, not including the path delimiter character (usually the double colon ::).

simplicity—the degree to which a design is easy to understand and free of complexity and multiplicity of parts.

singleton class—a class that can have only one instance.

Singleton pattern—a **generator pattern** in which a product class (acting as a generator) allows itself to be instantiated only once or only a small, fixed number of times.

software—any executable entity, such as a program, or its parts, such as sub-programs.

software architecture—the structure of a program comprised by its major constituents, their responsibilities and properties, and the relationships and interactions between them; called an **architecture** when there is no chance of confusion with hardware architecture.

software architecture document (SAD)—a document that specifies a program's **software architecture**.

software component—a reusable, replaceable piece of **software**.

software design—the activity of specifying the nature and composition of software products that satisfy client needs and desires, subject to constraints; also the specifications that result from this activity.

software design method—an orderly procedure for generating a precise and complete design solution that meets client needs and constraints.

software design pattern—a model proposed for imitation in solving a software design problem; see **design pattern**.

software engineering design—the activity of specifying programs, sub-systems, and their constituent parts and workings to meet software product specifications; also the specifications that result from this activity.

software life cycle—the sequence of activities through which a software product passes from initial conception through retirement from service.

software product—one or more programs, data, and supporting materials and services satisfying client needs and desires either as an independent artifact or as an essential ingredient in some other artifact.

software product design—the activity of specifying software product features, capabilities, and interfaces to satisfy client needs and desires; also the specifications that result from this activity.

software requirement—a statement that a software product must have a certain feature, function, capability, or property.

software requirements specification (SRS)—a document cataloging all the requirements for a software product.

software reuse—the use of existing artifacts to build new software products.

specification—a statement that must be true of a product or service.

staffing—see **project staffing**.

stakeholder—anyone affected by a product or involved in or influencing its development.

stakeholders-goals list—a catalog of important stakeholder categories and their goals.

state—a mode or condition of being.

state diagram—a UML diagram for describing finite state automata.

state transition model—a model that shows the states of a computational entity and the transitions between its states.

static design model—a representation of aspects of a program that do not change during execution.

stepwise refinement—a top-down approach to program design that repeatedly decomposes procedures into smaller procedures until programming-level operations are reached.

stereotype—in UML, a string written between guillemots (« ») categorizing a model element.

string—a sequence of symbols; in UML, a sequence of characters in a format not specified by UML.

structured design—a once-dominant software design method that emphasizes procedural decomposition.

suspended—an operation is suspended when it sends a synchronous message and is waiting for the message to return; see **active**.

syntax—a specification of the elements of a communications medium and the ways they may be combined to form legitimate messages.

tagged value—in UML, a name-value pair written in the format *name = value*; the tag is the name.

target market—a market for which an organization decides to develop products.

technical requirement—a statement of a feature, function, capability, or property that a product must have that is not a direct client or development organization goal.

terminological consistency—using words with same meaning and always using the same words to refer to a particular thing.

testable specification—see **verifiable specification**.

thread—an activity inside a process; threads in the same process share its resources.

three-tiered architecture—a Layered architectural style with an interface layer, an application domain layer, and a persistent storage layer.

throwaway prototype—see **exploratory prototype**.

top-down strategy—an approach to problem solving that begins by solving the general problem resulting from abstracting most details of the original problem and gradually adding solution details to handle more and more of the details of the original problem.

tracking—see **project tracking**.

transaction—an operation on persistent data that usually must conform to requirements of atomicity, consistency, isolation, and durability.

transducer—a finite automaton that responds to events and generates actions; see **acceptor**.

transition—a change from one state to another.

trigger—an event that initiates a use case.

two-tiered architecture—a Layered architectural style with an interface layer and an application domain layer.

type-safe—a language is type-safe if it does not allow an operation to be applied to values for which it is not well defined.

uniformity—a description has uniformity when it treats similar items in similar ways.

use case—a type of complete interaction between a product and its environment.

use case description—a specification of the interaction between a product and the actors in a use case.

use case diagram—a UML notation representing a product’s use cases and the actors involved in each use case.

use case model—a use case diagram together with use case descriptions for each use case in the diagram.

user—someone who operates a product on a regular basis.

user interface design—the activity of specifying the physical details of a product’s form and behavior; also the specifications that result from this activity.

user interface diagram—a drawing of (part of) a product’s visual display when it is in a particular state.

user interface module—a portion of a program responsible for communication between the program and human users.

user interface state—a distinct configuration or mode of a user interface.

user-centered design—a design process that is stakeholder focused, is iterative, and relies on empirical evaluations.

user-level requirement—a statement about how a product must support stakeholders in achieving their goals or tasks.

uses relation—module *D* uses module *I* when a correct version of *I* must be present for *D* to execute correctly.

utility tree—a tree whose sub-trees are **profiles** and whose leaves are **scenarios**.

validation—see **requirements validation**.

variable—a programming language device for storing values.

verifiability—a document has verifiability if all its specifications are **verifiable**.

verifiable specification—a specification for which there is a definitive procedure to determine whether it is met.

vertical prototype—a working model of a product that does some processing aside from that required for presenting the product’s user interface.

virtual device—a software simulation of, or interface to, a real hardware device or system.

visibility—the portion of a program text over which a program entity may be referred to by its (original) name.

visible—the property of a program entity at every point in a program’s text where it may be accessed by its original name, not an alias.

walkthrough—an informal examination of a work product by a team of reviewers.

well formed—a document is well-formed if its notations are used correctly.

well formedness—see **well-formed**.

wrapper—see **adapter**.

Wrapper pattern—see **Adapter pattern**.

B AquaLush Case Study

Appendix Contents

- B.1 AquaLush Irrigation System Overview
 - B.2 AquaLush Project Mission Statement
 - B.3 AquaLush Stakeholders-Goals List
 - B.4 AquaLush Needs List
 - B.5 AquaLush User-Level Requirements
 - B.6 AquaLush Use Case Model
 - B.7 AquaLush Software Requirements Specification
 - B.8 AquaLush Conceptual Model
 - B.9 AquaLush Profiles and Scenarios
 - B.10 AquaLush Software Architecture Document
 - B.11 AquaLush Detailed Design Document
-

B.1 AquaLush Irrigation System Overview

Introduction

MacDougal Electronic Sensor Corporation (MESC), an electronic sensor manufacturer, has decided to start a new company to exploit a newly perfected soil moisture sensor. The company, called Verdant Irrigation Systems (VIS), will develop and market lawn and garden irrigation systems.

Timers regulate most irrigation or sprinkler systems: They release water for a fixed period on a regular basis. This may waste water if the soil is already wet or not provide enough water if the soil is very dry. VIS products will use the new soil moisture sensors to control irrigation. Irrigation will still take place on a regular basis, but now it may be skipped if the soil is already wet or continued until the soil is sufficiently moist.

VIS's first product is the AquaLush Irrigation System. It is targeted at high-end residential or small commercial properties.

A small team is charged with developing the software driving this product.

Opportunity Statement

Create an irrigation system that uses soil moisture sensors to control the amount of water used.

B.2 AquaLush Project Mission Statement

1. Introduction

Timers regulate most non-agricultural irrigation or sprinkler systems: They release water for a fixed period on a regular basis. This may waste water if the soil is already wet or not provide enough water if the soil is very dry. MacDougal Electronic Sensor Corporation has developed a new soil moisture sensor that can be used to fundamentally change the way that irrigation systems work. Irrigation can be controlled by the soil moisture sensors so that irrigation is skipped or suspended if the soil is

sufficiently moist or extended if the soil is too dry. It is expected that this will make more efficient use of water resources as well as making irrigation more effective.

MESC hopes that this change in irrigation systems will spur the sale of its new sensor. MESC would also like to sell moisture-controlled irrigation systems.

To take advantage of these two opportunities, MESC has created a new company called Verdant Irrigation Systems. VIS will develop and market moisture-controlled irrigation systems. The first product to be fielded by VIS is the AquaLush Irrigation System, a demonstration product establishing the viability of moisture-controlled irrigation systems.

2. Product Vision and Project Scope

Product Vision Statement—The AquaLush Irrigation System will use soil moisture sensors to control irrigation, thus saving money for customers and making better use of water resources.

Major Features—AquaLush will

- Monitor water usage and limit usage to amounts set by users.
- Allow users to specify times when irrigation occurs.
- Be operated from a simple central control panel.
- Have a Web-based simulator.

Project Scope—The current project will create the minimal hardware and software necessary to field a viable product, along with a Web-based product simulator for marketing the product.

3. Target Markets

The first version of AquaLush is for high-end residential and small commercial users needing automated irrigation systems for plots ranging from about half an acre up to about five acres. The product is aimed at both first-time buyers and customers ready to replace their current timer-controlled systems with a more economical system.

4. Stakeholders

Management—The Board of Directors of MacDougal Electronic Sensor Corporation, as the controlling interest in Verdant Irrigation Systems, and the CEO of Verdant Irrigation Systems.

Developers—The four-member AquaLush development team, which includes three software engineers and a mechanical engineer specializing in non-agricultural irrigation systems. Besides developing the product, this team will also support it in the field.

Marketers—The two-person VIS marketing department. There is also a contractor paid by the marketers to develop and maintain the VIS Web site, which will host the AquaLush simulator.

Purchasers—Homeowners, groundskeepers, lawn and garden service professionals, irrigation system professionals, and small business purchasing agents.

Users—Homeowners, groundskeepers, lawn and garden service professionals, irrigation system professionals, and small business maintenance personnel.

5. Assumptions and Constraints

Assumptions—The developers may take the following for granted:

- AquaLush will use the new moister sensor developed by MESC.
- AquaLush may use standard irrigation valves, pipes, fittings, and control hardware.

Constraints—The product must conform to the following restrictions:

- AquaLush will support only the MESC moisture sensor.
- The fielded product and Web simulator will use the same core irrigation control software.

- The product will implement moisture-controlled irrigation, but its design will not preclude eventual incorporation of timer-controlled irrigation.

6. Business Requirements

AquaLush must establish moisture-controlled irrigation as a viable technology.

The first version of AquaLush must be brought to market within one year of the development project launch.

AquaLush's retail price must be no more than 10% greater than the average cost of timer-controlled irrigation products for the same market segment.

AquaLush must achieve at least 5% target market share within one year of introduction.

AquaLush must establish base irrigation control technology for use in later products.

AquaLush must provide a Web-based simulation that at least 70% of users agree provides an accurate representation of the actual product and its use.

AquaLush must demonstrate irrigation cost savings of at least 15% per year over competitive products.

B.3 AquaLush Stakeholders-Goals List

Stakeholder Category	Goals
Management	Achieve business requirements
Developers	Achieve business requirements
	Create a high-quality, maintainable product
	Create a code base reusable in later products
Marketers	Achieve business requirements
Purchasers	Pay the least for a product that meets irrigation needs
	Purchase a product that is cheap to operate
	Purchase a product that is cheap to maintain
Installers	Have a product that is easy and fast to install
Operators	Irrigation can be scheduled to occur at certain times
	Irrigation schedules can be set up and changed quickly
	Irrigation schedules can be set up and changed without having to consult instructions
Maintainers	It is quick and easy to tell when the product is not working properly
	It is quick and easy to track down problems
	It is quick and easy to fix problems
	The product is able to recover from routine failures (such as loss of power or water pressure) by itself
	One sort of failure (such as loss of power or water pressure) does not lead to other failures (such as broken valves or sensors)
Webmaster	The product and its parts have a low rate of failure
	The simulation is easy to install and maintain

Table B-3-1 AquaLush Stakeholders-Goals List

B.4 AquaLush Needs List

1. Introduction

This needs list does not contain any detailed needs. Instead, it specifies only user-level needs having to do with broad stakeholder goals and tasks. Needs are prioritized using a five-point scale, with one being the highest priority. Priorities are noted in parentheses at the end of the needs statement.

2. Constraints

Management, Developers, and Marketers need the first version of AquaLush to be brought to market within one year of the development project launch. (2)

Management, Developers, and Marketers need AquaLush to contain the base irrigation control software for use in later products. (1)

Installers need AquaLush software to be configurable using either standard tools or tools supplied with the product. (1)

3. Functional Needs

Management, Developers, and Marketers need AquaLush to be mainly a moisture-controlled irrigation product. (1)

Management, Developers, Marketers, and Purchasers need AquaLush to have a Web-based simulation that at least 70% of users agree provides an accurate representation of the actual product and its use. (2)

Installers and Maintainers need AquaLush to allow the current time to be set. (1)

Operators need AquaLush to allow them to schedule irrigation for certain times. (1)

Operators need AquaLush to allow them to set the moisture levels that control irrigation. (1)

Management, Developers, Marketers, Purchasers, and Operators need AquaLush to irrigate only until the desired moisture level is achieved. (1)

Purchasers and Operators need AquaLush to allow them to set the maximum amount of water used in irrigation. (3)

Purchasers and Operators need AquaLush to monitor the amount of water used in irrigation and stop irrigation when the maximum allocated amount of water is reached. (3)

Maintainers need AquaLush to detect valve and sensor failures. (2)

Operators need AquaLush to continue operating as normally as possible in the face of valve and sensor failures. (2)

Maintainers need AquaLush to report on whether the system has detected failed components. (3)

Operators and Maintainers need AquaLush to report what is wrong with the system when it is not working properly. (4)

Maintainers need AquaLush to report any failed hardware components along with their locations. (3)

Maintainers need AquaLush to recover from routine failures (such as power failure or water pressure failures) without Maintainer intervention. (2)

4. Non-Functional Needs

Management, Purchasers, and Marketers need AquaLush to be able to irrigate up to five acres. (1)

Installers need AquaLush to be configurable so that they may vary the type of irrigation valves, the number of irrigation valves, the number of moisture sensors, and the association between valves and sensors. (3)

Developers need AquaLush to be developed to be maintainable. (1)

Developers need AquaLush to contain irrigation software components reusable in later products. (1)

Maintainers need AquaLush to not fail as a consequence of other failures; in other words, to prevent cascades of failures. (2)

Marketers and Maintainers need AquaLush software to be highly reliable (no more than one failure per month of normal operation). (1)

The company Web site's Webmaster needs the AquaLush simulation to be easy to install and maintain. (3)

5. Data Needs

Installers need AquaLush to record the system configuration in a persistent way so that it can be restored after power failures. (2)

Maintainers need AquaLush to record the locations of failed hardware components. (3)

6. Interface Needs

Installers need AquaLush software configuration time to be no more than 1 minute per irrigation valve after no more than 30 minutes of reading the instruction manual. (4)

Operators need AquaLush to have an interface that allows them to set up or make typical changes to irrigation schedules in less than five minutes without consulting a manual. (2)

Operators and Maintainers need AquaLush to be operated from a single central control panel. (2)

Managers, Marketers, Purchasers, and Maintainers need AquaLush to control a variety of irrigation valves. (5)

B.5 AquaLush User-Level Requirements

1. Introduction

1.1 The AquaLush Irrigation System must be a software-controlled irrigation system that automatically adjusts irrigation output based on user-supplied criteria and moisture sensor readings.

1.2 The software, as delivered, must consist of two parts:

- A Web-based simulation; and
- Software to control the hardware.

2. Functional Requirements

2.1 Setting Parameters

2.1.1 AquaLush must allow its mode (manual or automatic) to be set.

2.1.2 AquaLush must allow the current time to be set.

2.1.3 AquaLush must allow the days and times during the day when irrigation occurs, called *irrigation times*, to be set.

2.1.4 AquaLush must allow moisture levels that control irrigation, called *critical moisture levels*, to be set.

2.1.5 AquaLush must allow the maximum amount of water used in irrigation, called the *water allocation*, to be set.

2.2 General Operation

- 2.2.1 AquaLush must allow operation in either an *automatic* or a *manual mode*.
- 2.2.2 AquaLush must monitor the amount of water used in irrigation.
- 2.2.3 AquaLush must detect valve and sensor failures.
- 2.2.4 AquaLush must continue operating as normally as possible in the face of valve and sensor failures.
- 2.2.5 AquaLush must report failed components, and their locations, when asked to do so.

2.3 Manual-Mode Operation

- 2.3.1 AquaLush must allow individual valves to be opened or closed.
- 2.3.2 AquaLush must provide data about manual irrigation.

2.4 Automatic-Mode Operation

- 2.4.1 AquaLush must irrigate only during irrigation times.
- 2.4.2 AquaLush must irrigate only until the set critical moisture level is achieved.
- 2.4.3 AquaLush must irrigate only until the water allocation is reached.

2.5 Simulation

- 2.5.1 The Web-based AquaLush simulation must represent all operational and maintenance features of the delivered product.
- 2.5.2 The Web-based AquaLush simulation must provide means to control the simulation itself.

3. Non-Functional Requirements

3.1 Installation

- 3.1.1 AquaLush must support enough sensors and valves to irrigate up to five acres.
- 3.1.2 AquaLush software must be configurable using standard tools (such as a text editor).
- 3.1.3 Once configured, AquaLush must not require reconfiguration unless the installed hardware changes.
- 3.1.4 AquaLush software must be configurable in no more than 1 minute per irrigation valve after no more than 30 minutes of reading the instruction manual.

3.2 Operation

- 3.2.1 AquaLush must have an interface that allows irrigation to be set up or altered in less than five minutes without consulting a manual.
- 3.2.2 AquaLush must be operated from a single central control panel.

3.3 Failure

- 3.3.1 AquaLush must recover from power failures without human intervention.
- 3.3.2 AquaLush must not fail when a sensor or valve fails.
- 3.3.3 AquaLush software must fail no more than once per month of normal operation.

3.4 Simulation

- 3.4.1 The AquaLush simulation must be installable on a Web site in no more than one hour.
- 3.4.2 The AquaLush simulation must require no more than one hour per month of maintenance.
- 3.4.3 At least 70% of users must agree that the AquaLush Web-based simulation provides an accurate representation of the actual product and its use.

3.5 Evolution

- 3.5.1 The main AquaLush irrigation software components must be reusable in later products.
- 3.5.2 AquaLush must eventually control a variety of irrigation valves.
- 3.5.3 AquaLush must accommodate implementation of traditional timer-controlled irrigation.

4. Data Requirements

- 4.1 AquaLush must record the following data in a persistent store:

- Type of each irrigation valve;
 - Location of each irrigation valve;
 - Flow rate of each irrigation valve;
 - Status of each irrigation valve;
 - Location of each moisture sensor;
 - Status of each moisture sensor;
 - Association between valves and sensors;
 - Current mode of operation;
 - Irrigation times;
 - Critical moisture levels; and
 - Water allocation.
-

B.6 AquaLush Use Case Model

Actors

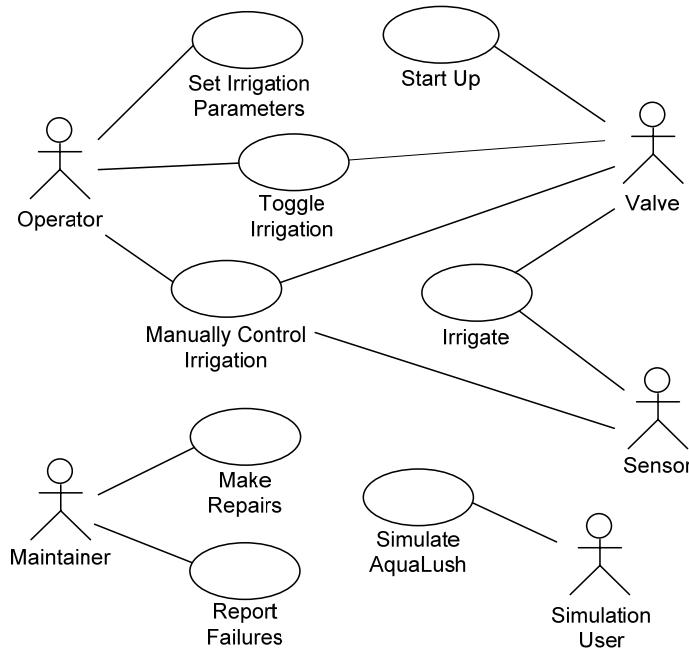
Sensor—A moisture sensor device.

Valve—An irrigation valve device.

Operator—A user operating AquaLush on a day-to-day basis.

Maintainer—A user performing maintenance tasks on AquaLush.

Simulation User—A person using the Web-based AquaLush simulation.

Use Case Diagram**Figure B-6-1 AquaLush Use Case Diagram****Use Case Descriptions****Use Case 1: Toggle Irrigation****Actors:** Operator, Valve**Stakeholders and Needs:**

Operators—To control irrigation times, to continue operating as normally as possible in the face of Valve and Sensor failures.

Managers, Marketers—To achieve 5% market share.

Maintainers—To detect and record Valve failures, to recover from power failures without Maintainer intervention.

Preconditions: None.

Postconditions: AquaLush is set to irrigate either automatically or manually, and this setting is recorded in a persistent store.

All Valve failures are recorded.

Persistent store failures are reported.

Trigger: Operator sets the mode.

Basic Flow:

1. Operator sets the mode from manual to automatic.
2. AquaLush turns off any Valves that may be on.
3. AquaLush records the mode as automatic, confirms the change, and waits for the next irrigation time.

Extensions:

- 1a Operator sets the mode from automatic to manual:
 - 1a1. AquaLush closes all Valves.
 - 1a2. AquaLush records the mode as manual and confirms the change, and the use case ends.
- 1b Operator sets the mode to its current setting:
 - 1b1. AquaLush confirms the current mode and the use case ends.
- 2a A Valve fails:
 - 2a1. AquaLush tries twice more to close the Valve, and if it succeeds the use case continues.
 - 2a2. AquaLush alerts the Operator of the failure and records that this Valve failed in its persistent store; then the use case continues.
- 1a2a, 3a, 2a2a AquaLush cannot write to its persistent store:
 - 3a1. AquaLush alerts the Operator of the failure and the use case continues.

Use Case 2: Set Irrigation Parameters**Actors:** Operator**Stakeholders and Needs:**

Operators, Purchasers—To schedule irrigation for certain times, to set the moisture levels that control irrigation, to set the maximum amount of water used in irrigation.

Preconditions: None.

Postconditions: AquaLush automatic irrigation parameters are set, and these settings are recorded in a persistent store.

All Valve failures are recorded.

Persistent store failures are reported.

Trigger: Operator sets automatic irrigation parameters.

Basic Flow:

1. Operator sets the irrigation time, water allocation, or critical moisture levels.
2. AquaLush validates the new setting(s).
3. AquaLush records and confirms the new setting(s).

Extensions:

- 2a A setting is invalid:
 - 2a1. AquaLush notifies the Operator of the invalid setting.
 - 2a2. Operator adjusts the setting.
 - 2a3 AquaLush checks the setting and either resumes the basic flow or returns to step 2a1.
- 3a AquaLush cannot write to its persistent store:
 - 3a1. AquaLush alerts the Operator of the failure and the use case continues.
- 3b Irrigation is in progress when the irrigation time is set:
 - 3b1. AquaLush completes the current irrigation cycle, and the new irrigation time takes effect upon completion of the cycle.
- 3c Irrigation is in progress when the water allocation is set:
 - 3c1. AquaLush recomputes the allocations for each irrigation zone, and they take effect immediately.
- 3d Irrigation is in progress when the critical moisture levels are set:
 - 3d1. AquaLush uses the new critical moisture levels to control irrigation immediately.

Use Case 3: Manually Control Irrigation**Actors:** Operator, Valve, Sensor**Stakeholders and Needs:**

Operators—To schedule irrigation for certain times, to continue operating as normally as possible in the face of Valve and Sensor failures, to irrigate the site.

Maintainers—To detect and record Valve failures, to recover from power failures without Maintainer intervention.

Preconditions: AquaLush is in manual mode.**Postconditions:** All Valve and Sensor failures are recorded.

Persistent store failures are reported.

Trigger: Operator selects a non-empty set of closed Valves and directs that they be opened.**Basic Flow:**

1. Operator selects a non-empty set of closed Valves and directs that they be opened.
2. AquaLush opens each Valve in the set and displays to the Operator for each open Valve: its location, how long it has been open, how much water it has used, and the moisture level of its associated Sensor. AquaLush also shows the total water used since the start of the use case.

The following Operator action and AquaLush responses may be done in any order, and they may be done repeatedly.

3. Operator selects a set of open Valves and directs that they be closed.
 4. AquaLush closes the indicated Valves and removes them from the Valve status display.
 5. Operator selects a set of closed Valves and directs that they be opened.
 6. AquaLush opens the selected Valves and adds them to the Valve status display.
- The following steps are taken to end the use case.
7. The Operator indicates that he or she is finished.
 8. AquaLush acknowledges that manual irrigation is finished and closes all Valves.

Extensions:

2a,4a,6a,8a A Valve fails:

2a1. AquaLush tries twice more to manipulate the Valve, and if it succeeds, the use case continues.

2a2. AquaLush alerts the Operator of the failure and records that this Valve failed in its persistent store; then the use case continues.

2b A Sensor fails:

2b1. AquaLush tries twice more to read the Sensor, and if it succeeds, the use case continues.

2b2. AquaLush alerts the Operator of the failure and records that this Sensor failed in its persistent store; then the use case continues.

2*2a AquaLush cannot write to its persistent store:

2a2a1. AquaLush alerts the Operator of the failure, and the use case continues.

Use Case 4: Make Repairs**Actors:** Maintainer**Stakeholders and Needs:**

Maintainers—To fix problems easily.

Preconditions: None.**Postconditions:** The clock is reset or AquaLush is notified of repaired Valves and Sensors.

The persistent store is updated when hardware is repaired.

Persistent store failures are reported.

Trigger: Maintainer begins a repair session.

Basic Flow:

1. Maintainer begins a repair session.
2. AquaLush displays the set of failed Valves and Sensors.

The following Maintainer actions and AquaLush responses may be done (as pairs) in any order, and they may be done repeatedly.

3. Maintainer sets the current time.
4. AquaLush resets its clock and confirms it to the Maintainer. The current time is used to control automatic irrigation immediately.
5. Maintainer indicates which of the failed Valves and Sensors are now repaired.
6. AquaLush adjusts its persistent store to record the repairs. AquaLush begins using the repaired Sensors and Valves for irrigation immediately.
7. The use case ends when the Maintainer indicates that repairs are complete.

Extensions:

- 6a AquaLush cannot write to its persistent store:

6a1. AquaLush alerts the Maintainer of the failure and the use case continues.

Use Case 5: Report Failures

Actors: Maintainer

Stakeholders and Needs:

Maintainers—To fix problems easily.

Preconditions: None.

Postconditions: Failed Valves and Sensors are reported.

Trigger: Maintainer requests a failure report.

Basic Flow:

1. Maintainer requests a failure report.
2. AquaLush displays the failed Valves and Sensors and their locations, or indicates that there are no failures.
3. Maintainer ends the use case.

Extensions:

None.

Use Case 6: Start Up

Actors: Valve

Stakeholders and Needs:

Maintainers—To recover from power failures without Maintainer intervention.

Operators—To continue operating as normally as possible in the face of Valve and Sensor failures.

Preconditions: AquaLush was powered down.

Postconditions: If persistent storage is intact and readable, then AquaLush is restored to its state prior to shutdown, except that all Valves are closed.

All Valve failures are recorded or persistent store failure is reported.

Trigger: The system is powered up and AquaLush is restarted.

Basic Flow:

1. AquaLush reads its persistent store and establishes its configuration.
2. AquaLush reads its persistent store and restores all its parameters.
3. AquaLush reads its persistent store and closes all working Valves. If AquaLush is in automatic mode, it waits for the irrigation time.

Extensions:

- *a AquaLush cannot read its persistent store:
- *a1. AquaLush displays a failure message and suspends operation.

3a A Valve fails:

- 3a1. AquaLush tries twice more to close the Valve, and if it succeeds the use case continues.
- 3a2. AquaLush records that this Valve failed in its persistent store.
- 3a3. AquaLush goes on with its processing.

3a2a AquaLush cannot write to its persistent store: The use case continues.

Use Case 7: Irrigate

Actors: Valve, Sensor

Stakeholders and Needs:

Management, Developers, Marketers, Operators, Purchasers—To irrigate only until the desired moisture level is achieved.

Operators, Purchasers—To monitor the amount of water used in irrigation and to stop irrigation when the maximum allocated amount of water is reached.

Maintainers—To detect Valve and Sensor failures, to recover from power failures without Maintainer intervention.

Operators—To continue operating as normally as possible in the face of Valve and Sensor failures.

Preconditions: AquaLush is in automatic irrigation mode.

Postconditions: All irrigation zones have been irrigated until either their Sensors indicate that the critical moisture level has been reached or the water allocated for the zone is used up.

All Valve and Sensor failures are recorded.

All persistent store failures are reported.

Trigger: The current time is the irrigation time.

Basic Flow:

1. AquaLush reads each (working) Sensor. If a Sensor reports moisture below its zone's critical moisture level, then AquaLush places that zone on an active zone list.
2. AquaLush counts the (working) Valves in the active zones and divides the water allocation by this count to get a Valve allocation. Each zone is assigned a zone allocation, which is the Valve allocation multiplied by the number of Valves in the zone.

For each active zone:

3. AquaLush opens all the (working) Valves in the zone, and marks the time when it does so.
4. Every minute, AquaLush reads the zone Sensor and also computes the zone water usage by multiplying the total flow rate for all Valves in the zone by the time that the Valves have been open.
5. If the zone's critical moisture level is reached, or the zone allocation is reached, then AquaLush closes all the Valves in the zone.

The use case ends when all zones in the active zone list have been irrigated.

Extensions:

- 1a A Sensor fails:
 - 1a1. AquaLush tries twice more to read the Sensor, and if it succeeds goes on as before.
 - 1a2. AquaLush records that this Sensor failed in its persistent store.
 - 1a3. AquaLush continues with the next Sensor.

- 3a,4a2a,5a A Valve fails:
 - 3a1. AquaLush tries twice more to manipulate the Valve, and if it succeeds the use case continues.
 - 3a2. AquaLush records that this Valve failed in its persistent store.
 - 3a3. AquaLush goes on with its processing.

- 4a A Sensor fails:
 - 4a1. AquaLush tries twice more to read the Sensor, and if it succeeds the use case continues.
 - 4a2. AquaLush closes all Valves in the zone.
 - 4a3. AquaLush records that this Sensor failed in its persistent store.
 - 4a4. AquaLush continues with step 5 as if the Sensor had reached the zone's critical moisture level.

- 5b The zone allocation is not exhausted:
 - 5b1. AquaLush adds the remainder of the zone allocation to the zone allocations of the unirrigated active zones and divides this sum by the number of Valves in the unirrigated active zones to compute a new Valve allocation.
 - 5b2. AquaLush recomputes the zone allocations for each active unirrigated zone by multiplying the new Valve allocation by the number of Valves in the zone.

- 1a2a,3a2a,4a3a AquaLush cannot write to its persistent store: AquaLush alerts the Operator of the failure and the use case continues.

Use Case 8: Simulate AquaLush

Actors: Simulation User

Stakeholders and Needs:

Purchasers, Management, Developers, Marketers—To have a Web-based simulation providing an accurate representation of the actual product and its use.

Preconditions: None.

Postconditions: Simulators have interacted with an accurate simulation.

Trigger: The simulation Web page is loaded.

Basic Flow:

Any of the following Simulator actions may be done in any order, and they may be done repeatedly.

1. Simulator uses the simulated control panel to Toggle Irrigation, Set Irrigation Parameters, Manually Control Irrigation, Make Repairs, or Report Failures.
2. AquaLush Simulator displays the simulated effects on the representation of a site with irrigation zones containing Sensors and Valves. AquaLush Simulator also displays the simulated time, simulation speed, Sensor readings, Valve flows, water evaporation rate, and failed Valves and Sensors.
3. AquaLush Simulator speeds up or slows down the simulation, sets the simulated time, sets the simulated water evaporation rate, or makes Valves or Sensors fail or be repaired.
4. AquaLush Simulator alters the simulation in response to the new parameter settings.

The use case ends when the simulation Web page is exited.

B.7 AquaLush Software Requirements Specification

1. Product Description

1.1 Please see section B.2 *AquaLush Project Mission Statement* for an overview of the product.

1.2 The software, as delivered, must consist of two parts:

- A Web-based simulation; and
- Software to control the hardware.

2. Functional Requirements

2.1 Setting Parameters

2.1.1 All parameters set by users must remain in effect until changed by users.

2.1.2 AquaLush must run in one of two modes: *manual* or *automatic*.

2.1.2.1 Users must be able to set the mode.

2.1.2.2 Setting the mode to its current value must have no effect on irrigation.

2.1.2.3 Setting the mode to a different value must result in AquaLush closing all valves.

2.1.2.4 AquaLush must display the current mode to the user after the user sets the mode.

2.1.3 AquaLush must allow users to set the *current time*.

2.1.3.1 The current time consists of the current day and the current time of day.

2.1.3.2 AquaLush must either not allow an invalid time setting or validate the new time setting.

2.1.3.3 If a new time setting fails validation, then AquaLush must notify the user of the problem and not accept the new setting.

2.1.3.4 AquaLush must display a reset current time to the user.

2.1.3.5 If the current time is set so that an irrigation time is skipped, then AquaLush must wait for the next irrigation time. In other words, AquaLush makes no attempt to detect and react to irrigation times skipped as a result of a time setting.

2.1.4 AquaLush must allow the days and time of day when irrigation occurs, called the *irrigation time*, to be set.

2.1.4.1 AquaLush must either not allow an invalid irrigation time setting or validate the new irrigation time setting.

2.1.4.2 If a new irrigation time setting fails validation, then AquaLush must notify the user of the problem and not accept the new setting.

2.1.4.3. If the irrigation time is set when automatic irrigation is in progress, then AquaLush must finish the current irrigation cycle and the new irrigation time must take effect upon completion of the cycle.

2.1.4.4 AquaLush must display a reset irrigation time to the user.

2.1.5 AquaLush must allow moisture levels that control irrigation, called *critical moisture levels*, to be set for each irrigation zone.

2.1.5.1 AquaLush must either not allow an invalid critical moisture level setting or validate the new critical moisture level setting.

2.1.5.2 If a new critical moisture level setting fails validation, then AquaLush must notify the user of the problem and not accept the new setting.

2.1.5.3. If the critical moisture level is set when automatic irrigation is in progress, then AquaLush must use the new critical moisture level to control irrigation immediately.

2.1.5.4 AquaLush must display a reset critical moisture level to the user.

2.1.6 AquaLush must allow the maximum amount of water used in each irrigation cycle, called the *water allocation*, to be set.

2.1.6.1 AquaLush must either not allow an invalid water allocation setting or validate the new water allocation setting.

2.1.6.2 If a new water allocation setting fails validation, then AquaLush must notify the user of the problem and not accept the new setting.

2.1.6.3. If the water allocation is set when automatic irrigation is in progress, then AquaLush must recompute the allocations for each unirrigated irrigation zone and use them immediately to control irrigation.

2.1.6.4 AquaLush must display a reset water allocation to the user.

2.2 General Operation

2.2.1 A configuration file must be prepared at installation so that AquaLush can read this file to obtain its configuration at startup.

2.2.2 AquaLush must allow operation in either automatic or manual mode.

2.2.3 AquaLush must monitor the amount of water used in irrigation.

2.2.4 AquaLush must detect valve and sensor failures.

2.3 Manual-Mode Operation

2.3.1 In manual-mode operation, AquaLush must allow users to select non-empty sets of (working) valves and direct that they be opened or closed.

2.3.2 AquaLush must display the following data for each manually opened valve while it is open:

- (a) Its identifier
- (b) Its location
- (c) How long it has been open
- (d) How much water it has used
- (e) The moisture level reported by its associated sensor

2.3.3 AquaLush must display the total water used in manual irrigation.

2.3.3.1 AquaLush must set the total water used in manual irrigation to zero when it starts up in manual mode or when it is switched to manual mode from automatic mode.

2.3.3.2 When no valve is open in manual irrigation mode, AquaLush must set the total water used in manual irrigation to zero.

2.4 Automatic-Mode Operation

2.4.1 In automatic-mode operation, AquaLush must begin an *irrigation cycle* when the current time reaches the irrigation time.

2.4.1.1 In automatic-mode operation, AquaLush must conduct an irrigation cycle by irrigating one irrigation zone at a time and successively irrigating each zone until all have been irrigated.

2.4.2 AquaLush must do the following during an irrigation cycle:

2.4.2.1 AquaLush must read the working sensor in each zone and place all zones whose sensors are below that zone's critical moisture level on an *active zone list*.

2.4.2.2 AquaLush must count the working valves in the active zones and divide the water allocation by this count to get a *value allocation*.

2.4.2.3 AquaLush must assign each zone a *zone allocation*, which is the valve allocation multiplied by the number of working valves in the zone.

2.4.2.4 For each active zone, AquaLush must open all the working valves in the zone.

2.4.2.5 Every minute, AquaLush must read the zone sensor.

2.4.2.6 Every minute, AquaLush must compute the zone water usage by multiplying the total flow rate for all open valves in the zone by the time that the valves have been open.

2.4.2.7 If a zone's critical moisture level is reached, then AquaLush must close all valves in the zone and go to the next zone.

2.4.2.8 If a zone's zone allocation is reached, then AquaLush must close all valves in the zone and go to the next zone.

2.4.2.9 When all valves in a zone are closed, if the zone allocation is not exhausted, then AquaLush must add the remainder of the zone allocation to the total of the zone allocations of the unirrigated zones, divide this sum by the number of working valves in the unirrigated zones to produce a new valve allocation, and multiply this value by the number of working valves in each unirrigated zone to produce a new zone allocation for each unirrigated zone.

2.4.2.10 An irrigation cycle must end when every zone on the active zone list has been irrigated.

2.5 Failures

2.5.1 If a sensor cannot be read, then AquaLush must try to read it twice more.

2.5.1.1 If a sensor can be read within three tries, then AquaLush must ignore the error.

2.5.1.2 If a sensor cannot be read after three tries, then AquaLush must mark that sensor as failed in its persistent store and stop using it in irrigation.

2.5.2.3 If a sensor cannot be read after three tries in manual-mode operation, then AquaLush must alert the user that the sensor has failed.

2.5.2.4 If a sensor cannot be read after three tries when AquaLush is irrigating that sensor's zone, then AquaLush must close all valves in that zone and continue the irrigation cycle as if that sensor had indicated that the zone had reached its critical moisture level.

2.5.2 If a valve cannot be manipulated (opened or closed), then AquaLush must try to manipulate it twice more.

2.5.2.1 If a valve can be manipulated within three tries, then AquaLush must ignore the error.

2.5.2.2 If a valve cannot be manipulated after three tries, then AquaLush must mark that valve as failed in its persistent store and stop using it in irrigation.

2.5.2.3 If a valve cannot be manipulated after three tries in manual-mode operation, then AquaLush must alert the user that the valve has failed.

2.5.3 If AquaLush cannot write to its persistent store, then it must alert the user of the problem and continue operation.

2.5.4 If AquaLush cannot read its persistent store it must alert the user of the problem.

2.5.4.1 If AquaLush cannot read its persistent store during operation, then it must continue operation.

2.5.4.2 If AquaLush cannot read its persistent store when starting up, then it must display an error message and suspend operation.

2.5.5 AquaLush must provide reports of failed valves and sensors when requested to do so by the user.

2.5.5.1 If no valves have failed, then AquaLush must report that there are no valve failures.

2.5.5.2 If no sensors have failed, then AquaLush must report that there are no sensor failures.

2.5.5.3 If valves or sensors have failed, then AquaLush must report for each failed valve and sensor:

- (a) The valve or sensor identifier
- (b) The valve or sensor location

2.5.6 AquaLush must allow users to indicate that failed valves and sensors have been repaired.

2.5.6.1 When told that a failed valve or sensor is repaired, AquaLush must update its persistent store with this information.

2.5.6.2 When told that a failed valve or sensor is repaired, AquaLush must begin using the repaired valve or sensor in irrigation immediately.

2.6 Startup

2.6.1 When power is applied to the system, the AquaLush software must load and execute.

2.6.2 When it starts, AquaLush must first read its configuration file from persistent store to establish its configuration.

2.6.3 After establishing its configuration, AquaLush must read its persistent store to restore its state.

2.6.4 After starting, AquaLush must close all valves.

2.6.5 AquaLush must make no attempt to resume an irrigation operation that may have been in progress when it was shut down.

2.7 Simulation

2.7.1 The Web-based AquaLush simulation must represent all operational and maintenance features of the delivered product.

2.7.2 The simulation must provide a realistic representation of the AquaLush control panel.

2.7.3 The simulation must provide a representation of a site with irrigation zones.

2.7.3.1 Each simulated irrigation zone must simulate a sensor and display the simulated sensor's moisture level.

2.7.3.2 Each simulated irrigation zone must simulate regions watered by irrigation valves and display the states of the simulated valves and their flow rates.

2.7.4 The simulation must provide means for controlling the simulation.

2.7.4.1 The simulation must display the *simulated time* and provide a means for setting the simulated time.

2.7.4.1.1 The simulation must provide controls to change the *simulation speed* (that is, speed up or slow down the simulated time).

2.7.4.2 The simulation must simulate water evaporation, reflected in the measured moisture levels at the simulated sensors.

2.7.4.2.1 The simulation must provide controls to alter the simulated *water evaporation rate*.

2.7.4.3 The simulation must simulate valve and sensor failure and repair.

2.7.4.3.1 The simulation must provide controls to alter the failure status of valves and sensors.

3. Data Requirements

3.1 Irrigation Zones

3.1.1 The irrigated site must be divided into 1 to 32 *irrigation zones*.

3.1.2. Each irrigation zone must have exactly one moisture sensor.

3.1.3 Each irrigation zone must have 1 to 32 irrigation valves.

3.1.4 An *irrigation zone identifier* must consist of the letter “Z” (upper- or lowercase) followed by a unique integer in the range 0..999.

3.1.5 An *irrigation zone location* must be a string of 0 to 24 characters not containing angle brackets.

3.1.6. The zone’s *critical moisture level* must be an integer in the range 0..100 interpreted as a percent of saturation.

3.2 Sensor Data

3.2.1 The following data must be recorded for each sensor:

- (a) Identifier
- (b) Location
- (c) Operational status

3.2.2 A *sensor identifier* must consist of the letter “S” (upper- or lowercase) followed by a unique integer in the range 0..999.

3.2.3 A *sensor location* must be a string of 0 to 24 characters not containing angle brackets.

3.2.4 The *operational status* must be a Boolean value interpreted as the truth value of the statement “This sensor is working.”

3.3 Valve Data

3.3.1 The following data must be recorded for each valve:

- (a) Identifier
- (b) Type
- (c) Location
- (d) Flow rate
- (e) Operational status

3.3.2 A *valve identifier* must consist of the letter “V” (upper- or lowercase) followed by a unique integer in the range 0..999.

3.3.3 A *valve type* must be a string of 0 to 16 characters not containing angle brackets.

3.3.4 A *valve location* must be a string of 0 to 24 characters not containing angle brackets

3.3.5 A *flow rate* must be an integer in the range 1.. $2^{31}-1$ interpreted as gallons per minute of water flow through the valve.

3.3.6 The *operational status* must be a Boolean value interpreted as the truth value of the statement “This valve is working.”

3.4 Configuration File

3.4.1 The *configuration file* must be a text file editable with a standard text editor.

3.4.2 A configuration file must have the form (specified using data definition notation) shown in Figure B-7-1.

3.4.2.1 Each elementary data item must be separated from the others by white space with the exception of the following:

3.4.2.1.1 The zone, sensor, and valve letters must not be separated from their identifier numbers.

3.4.2.1.2 All characters between angle brackets must be considered part of the description.

3.4.2.1.3 The curly braces and semicolon do not need to be separated from other tokens by white space.

3.4.3 The configuration file must be named “config.txt” and must be placed in the same directory as the AquaLush executable program.

3.4.4 A configuration file that cannot be found, opened, read, or parsed must be treated as a persistent store failure.

```

configFile = 1:32{ zoneSpec }
zoneSpec = "zone" + zoneIdentifier + location + "{" + zoneBody + "}"
zoneBody = sensorSpec + 1:32{ valveSpec }
sensorSpec = "sensor" sensorIdentifier location ","
valveSpec = "valve" valveIdentifier valveType flowRate location ","
valveType = "<" valveDescription ">"
valveDescription = * 0 to 16 characters not containing angle brackets *
flowRate = * integer in range 0..231-1 *
zoneIdentifier = zoneLetter + idNumber
sensorIdentifier = sensorLetter + idNumber
valveIdentifier = valveLetter + idNumber
zoneLetter = "Z" | "z"
sensorLetter = "S" | "s"
valveLetter = "V" | "v"
idNumber = * integer in range 0..999 *
location = "<" location description ">"
locationDescription = * 0 to 24 characters not containing angle brackets *

```

Figure B-7-1 AquaLush Configuration File

3.5 System Parameters

3.5.1. AquaLush must record its system parameters in persistent store so that its state can be restored at startup.

3.5.2. System parameters include the following items:

- (a) Mode of operation
- (b) Irrigation time
- (c) Water allocation

3.5.3 The *mode* must be either *manual* or *automatic*.

3.5.4 The *irrigation time* must specify both the days and the time of day when irrigation is to occur.

3.5.4.1 The irrigation days must be a subset of {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}.

3.5.4.2 The irrigation time of day must be a 24-hour-clock value in the range 0000 to 2359 (accurate to one minute).

3.5.5 The *water allocation* must be an integer in the range 0..2³¹-1 interpreted as gallons of water.

3.5.6. The *current time* is recorded in the system clock and may be set by the user; it consists of both the current day and the current time of day.

3.5.6.1 The current day must be an element of {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}.

3.5.6.2 The current time of day must be a 24-hour-clock value in the range 0000 to 2359 (accurate to one minute).

3.6 Manual Mode Data Display

3.6.1 The following data must be displayed for each valve in manual mode:

- (a) Valve identifier
- (b) Valve location
- (c) How long the valve has been open
- (d) How much water has been used
- (e) Associated sensor moisture level

3.6.1.1 The *valve identifier* and *location* must be the valve data recorded in the system. The *location* may be truncated to fit user interface display constraints.

3.6.1.2 The *length of time the valve has been open* must be a military time specification in the form *hhmm*, where *hh* must be hours in the range 0..23 and *mm* must be minutes in the range 0..59. All four digits must always be displayed.

3.6.1.3 The *amount of water used by the valve* must be an integer in the range 0.. $2^{31}-1$ describing the water used in gallons.

3.6.1.4 The *associated sensor moisture level* must be a moisture sensor reading in the range 0..100.

3.6.2 The total water used during manual irrigation must also be displayed. This value must be an integer in the range 0.. $2^{31}-1$ describing the water used in gallons.

3.6.3 Each displayed value must be updated every minute or when a valve is opened.

3.7 Failure Reports

3.7.1 Failure reports must display the following data:

- (a) The failed sensor or valve identifier
- (b) The failed sensor or valve location

3.7.1.1 The *failed sensor or valve identifier* must be the sensor or valve identifier recorded in the system.

3.7.1.2 The *failed sensor or valve location* must be the sensor or valve location recorded in the system. It may not be truncated on display.

3.9 Simulation Parameters

3.8.1 The simulated time consists of the simulated day and the simulated time of day.

3.8.1.1 The simulated day must be an element of {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}.

3.8.1.2 The simulated time of day must be a 24-hour-clock value in the range 0000 to 2359 (accurate to one minute).

3.8.2 The simulation speed must be a multiple of the real rate of time specified as an integer multiplier in the range 1..1000.

3.8.3 The simulated water evaporation rate must be an integer in the range 0..100 interpreted as the percent change in moisture level per day (as reflected in the simulated moisture sensor reading).

3.9 Defaults

3.9.1 The default mode must be *automatic*.

3.9.2 The default irrigation day must be the set {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} (every day).

3.9.3 The default irrigation time must be 0200.

3.9.4. The default water allocation must be 10000.

3.9.5. The default zone critical moisture level must be 50.

3.9.6. The default sensor and valve operational status must be *true*.

- 3.9.7. The default simulation speed must be *1* (real time).
- 3.9.8. The default simulated rate of water evaporation must be *10*.

4. Non-Functional Requirements

4.1 Installation

- 4.1.1 AquaLush software must be configurable in no more than 1 minute per irrigation valve after no more than 30 minutes of reading the instruction manual.

4.2 Operation

- 4.2.1 AquaLush must have an interface that allows irrigation to be set up or altered in less than five minutes without consulting a manual.
- 4.2.2 AquaLush must be operated from a single central control panel.

4.3 Failure

- 4.3.1 AquaLush must recover from power failures without human intervention.
- 4.3.2 AquaLush software must fail no more than once per month of normal operation.

4.4 Simulation

- 4.4.1 The AquaLush simulation must be installable on a Web site in no more than one hour.
- 4.4.2 The AquaLush simulation must require no more than one hour per month of maintenance.
- 4.4.3 At least 70% of users must agree that the AquaLush Web-based simulation provides an accurate representation of the actual product and its use.

4.5 Evolution

- 4.5.1 The main AquaLush irrigation software components must be reusable in later products.
- 4.5.2. AquaLush must eventually control a variety of irrigation valves.
- 4.5.3. AquaLush must accommodate implementation of traditional timer-controlled irrigation.

5. Hardware Interface

Were AquaLush a real product, this section would be quite large and detailed because it would present the data formats and interaction protocols for communicating with the actual moisture sensors, irrigation valves, and perhaps user interface devices. Since AquaLush is only an exercise, much of this information is missing.

In our later discussion and in our simulation, we will assume the following abstract requirements.

5.1 Sensors and Valves

- 5.1.1 Sensors must be readable and provide data convertible to a percent of saturation value.
- 5.1.2 Sensor failures must be detectable when a sensor is read.
- 5.1.3. Valves must be openable and closable.
- 5.1.4. Valve failures must be detectable when a valve is manipulated.

5.2 Control Panel

- 5.2.1 The AquaLush central control panel must be based on ATM machine technology.
- 5.2.1.1 It must include a monochrome screen that displays 16 lines of 40 ASCII characters each.
- 5.2.1.2 It must include eight push buttons adjacent to the screen, called *screen buttons*.
- 5.2.1.2.1 There must be four buttons on each side of the screen, arranged so that their centers align with the 9th, 11th, 13th, and 15th lines of the display.

5.2.1.3 It must include a 12-key keypad.

5.2.1.3.1 The keypad must have 10 numeric keys, a DEL (delete) key, and an ESC (escape) key.

6. Control Panel User Interface

6.1 General Control Panel Conventions

6.1.1 The control panel must have a menu-based textual user interface.

6.1.1.1 Most screens must present a menu selection or ask the user to fill in values as a result of a menu selection.

6.1.1.1 Pressing one of the screen buttons must take the indicated action.

6.1.1.2 Each child screen must have a menu item for returning to the parent screen from the bottom-right screen button.

6.1.1.3 Pressing the ESC button on a child screen must cancel any changes the user may have made and return to the parent screen.

6.1.1.4 Pressing the ESC button on the main menu screen must have no effect.

6.1.1.5 Pressing a screen button that does not have a function indicated in the display must have no effect.

6.1.1.6 Pressing a keypad digit button or the DEL button when the screen does not prompt for numeric input must have no effect.

6.1.1.7 Many screens have centered text. Text must be centered by dividing the length of the text in two, rounding down, and subtracting this value from 20 to reach the display column where the text begins. The first column must be zero.

6.2 Control Panel Dialog Maps

6.2.1 The control panel must conduct a dialog with the user that begins in a main menu with AquaLush in automatic mode.

6.2.2 The user may change to a menu screen with AquaLush in manual mode.

6.2.3 The menu screens in the two modes must be identical except that

- (a) Their mode indicators must be different; and
- (b) The manual mode menu must have an extra menu item for controlling irrigation manually.

6.2.4 The control panel must conform to the dialog map in Figure B-7-2, which shows the main menu states and the manual irrigation control state.

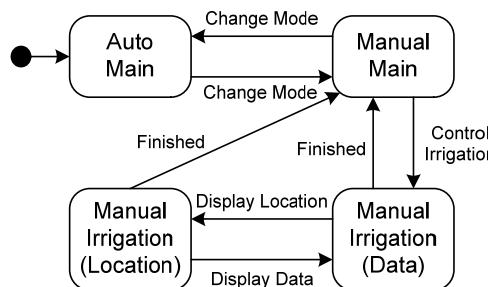


Figure B-7-2 AquaLush Dialog Map

Note that all states corresponding to the common main menu items are suppressed in this dialog map.

- 6.2.4.1 The **Change Mode** action must be a “Change Mode” screen button press.
- 6.2.4.2 The **Control Irrigation** action must be a “Control Irrigation” screen button press.
- 6.2.4.3 The **Finished** action must be either a “Finished” screen button press or a keypad ESC button press.
- 6.2.4.4 The **Display Location** action must be a “Display Location” screen button press.
- 6.2.4.5 The **Display Data** action must be a “Display Data” screen button press.
- 6.2.5 The control panel must conform to the dialog map in Figure B-7-3, which shows the common main menu item states suppressed in Figure B-7-2.

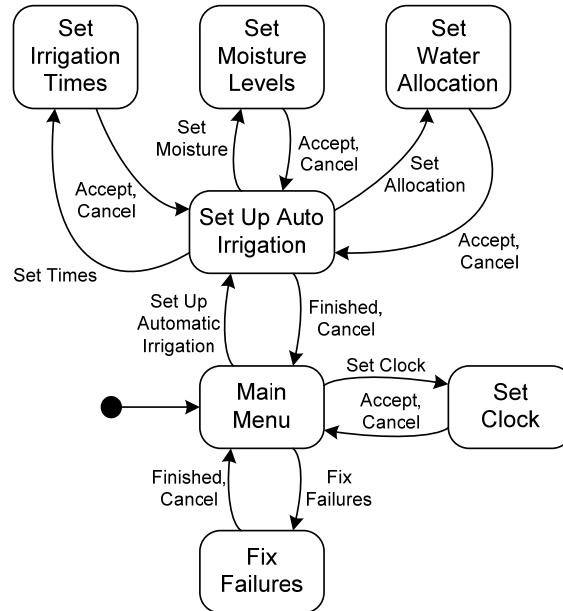


Figure B-7-3 Expanded AquaLush Dialog Map

- 6.2.5.1 The **Main Menu** state in this dialog map is a placeholder for the **Auto Main** state or the **Manual Main** state.
- 6.2.5.2 The **Accept** action must be an “Accept New Settings” screen button press.
- 6.2.5.3 The **Finished** action must be a “Finished” screen button press.
- 6.2.5.4 The **Cancel** action must be a keypad ESC button press.
- 6.2.5.5 The **Set Clock** action must be a “Set the Clock” screen button press.
- 6.2.5.6 The **Fix Failures** action must be a “Fix Failures” screen button press.
- 6.2.5.7 The **Set Up Automatic Irrigation** action must be a “Set Up Automatic Irrigation” screen button press.
- 6.2.5.8 The **Set Times** action must be a “Set Irrigation Times” screen button press.
- 6.2.5.9 The **Set Moisture** action must be a “Set Critical Moisture Levels” screen button press.
- 6.2.5.10 The **Set Allocation** action must be a “Set Water Allocation” screen button press.

6.2.6. The control panel must conform to the dialog map in Figure B-7-4 in the special circumstance that a device or persistent store error occurs during user interaction.

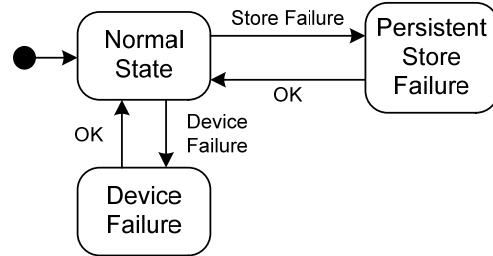


Figure B-7-4 AquaLush Dialog Map with Failures

6.2.6.1 Note that the **Normal State** is a placeholder for any of the states in the previous dialog maps.

6.2.6.2 The **Device Failure** action must be a detected sensor or valve failure.

6.2.6.3 The Store Failure action must be an inability to read from or write to the persistent data store.

6.2.6.4 The OK action must be an “OK” screen button press.

6.3 Control Panel User Interface Diagrams

6.3.1. The **Auto Main** state must be the initial state of the user interface when AquaLush is installed.

6.3.1.1 The Auto Main state must display the main menu screen shown in Figure B-7-5.

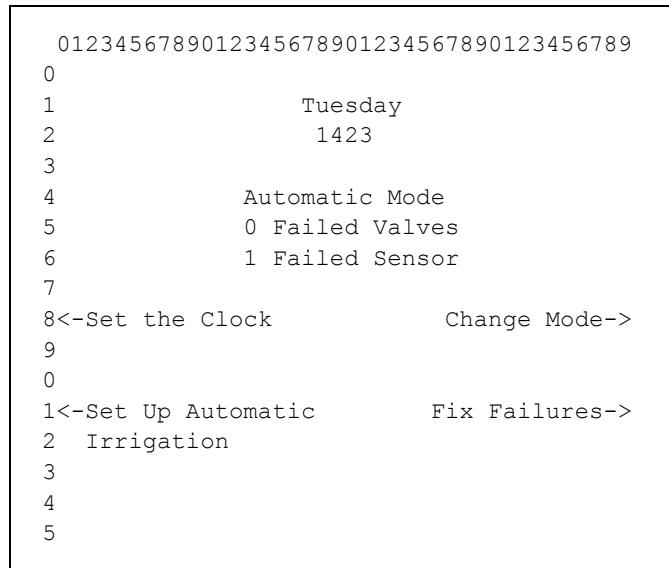


Figure B-7-5 AquaLush Automatic Main Menu Screen

6.3.1.2 Note that the digits in the borders of Figures B-7-5 through B-7-17 are for ease in interpreting the diagram: They must not appear on the screen.

6.3.1.3 Note that the day, time, number of failed valves, and number of failed sensors must vary. This data must always be centered on the screen.

6.3.2. The **Manual Main** state must be an alternative main menu state in manual mode.

6.3.2.1 The **Manual Main** state must display the main menu screen shown in Figure B-7-6.

```
0123456789012345678901234567890123456789
0
1           Tuesday
2           1423
3
4           Manual Mode
5           0 Failed Valves
6           1 Failed Sensor
7
8<-Set the Clock           Change Mode->
9
0
1<-Set Up Automatic         Fix Failures->
2 Irrigation
3
4           Control->
5           Irrigation
```

Figure B-7-6 AquaLush Manual Main Menu Screen

6.3.2.2 Note that the day, time, number of failed valves, and number of failed sensors must vary. This data must always be centered on the screen.

6.3.3. The **Manual Irrigation (Data)** state must allow users to directly open and close valves and must display irrigation data about each valve.

6.3.3.1 The **Manual Irrigation (Data)** state must display the screen shown in Figure B-7-7.

```
0123456789012345678901234567890123456789
0   Valve Zone Wet% Time      Water Used
1   -----
2   +V24   Z3     89  0012      221
3   +V25   Z3     89  0012      221
4   -V26   Z3     89  0000      0
5   -----
6           21345 Gallons Used
7
8<-Open/Close Valve           Scroll Up->
9
0
1<-Propagate to Zone         Scroll Down->
2
3
4<-Show Locations            Finished->
5
```

Figure B-7-7 AquaLush Manual Irrigation Data Screen

6.3.3.2 The region between the dashed lines must be a scrollable display of all working valves and data about their states or locations.

6.3.3.2.1 The valves must be listed in zone identifier order and then in valve identifier order within a zone.

6.3.3.2.2 The middle, highlighted element is the *current valve*.

6.3.3.2.2.1 The list must scroll up to or down to the middle element so that any valve can be made current.

6.3.3.2.2.2 There must always be a current valve.

6.3.3.2.3 The two middle screen buttons on the right must control scrolling.

6.3.3.3 If the user presses the “Open/Close Valve” screen button, then the current valve must be opened (if it is closed) or closed (if it is open).

6.3.3.4 If the user presses the “Propagate to Zone” screen button, then all valves in the current valve’s zone must be opened (if the current valve is open) or closed (if the current valve is closed).

6.3.3.5 The Valve field must show the identifier and status of each valve.

6.3.3.5.1 The status must be indicated by a single character in column three in the format “+” for open and “-” for closed.

6.3.3.5.2 The valve identifier must be left-aligned and occupy columns four through seven.

6.3.3.6 The Zone field must display the zone identifier left-aligned in columns 9 through 12.

6.3.3.7 The Wet% field must display the percent saturation of the moisture sensor in the valve’s zone right-aligned in columns 14 through 16 with no leading zero.

6.3.3.7.1 If the valve’s zone moisture sensor has failed, then this column must be filled with two dashes and be right-aligned.

6.3.3.8 The Time field must display the time that the valve has been open in four-digit military time notation in columns 19 through 22.

6.3.3.9 The Water Used field must show the number of gallons of water released by the valve during this manual irrigation cycle right-aligned in columns 24 through 36 with no leading zeros.

6.3.3.10 The Gallons Used display in row eight must show the number of gallons released by all valves during this irrigation cycle with no leading zeros.

6.3.3.10.1 The Gallons Used display must be centered.

6.3.4. The **Manual Irrigation (Location)** state must allow users to directly open and close valves and must display the location of each valve.

6.3.4.1 The **Manual Irrigation (Location)** state must display the screen shown in Figure B-7-8.

6.3.4.2 The region between the dashed lines must be a scrollable display that works just as in the **Manual Irrigation (Data)** state.

6.3.4.3 The screen buttons must work just as in the **Manual Irrigation (Data)** state.

6.3.4.4 The Valve field must display data just as in the **Manual Irrigation (Data)** state.

6.3.4.5 The Zone field must display data just as in the **Manual Irrigation (Data)** state.

6.3.4.6 The Location field must display the valve’s location string left-aligned in columns 14 through 37.

6.3.4.7 The Gallons Used display must work just as in the **Manual Irrigation (Data)** state.

6.3.5. The **Set Clock** state must allow users to set the AquaLush hardware clock.

```

0123456789012345678901234567890123456789
0   Valve Zone Location
1   -----
2   +V24  Z3   Front lawn N edge
3   +V25  Z3   Front lawn NE corner
4   -V26  Z3   Front lawn SE corner
5   -----
6           21345 Gallons Used
7
8<-Open/Close Valve          Scroll Up->
9
0
1<-Propagate to Zone        Scroll Down->
2
3
4<-Show Data                Finished->
5

```

Figure B-7-8 AquaLush Manual Irrigation Location Screen

6.3.5.1 The Set Clock state must display the screen shown in Figure B-7-9.

```

0123456789012345678901234567890123456789
0           Current Day: Tuesday
1           Current Time: 1423
2
3   Buttons change day; keys change time.
4
5<-Monday          Friday->
6
7
8<-Tuesday         Saturday->
9
0
1<-Wednesday       Sunday->
2
3
4<-Thursday        Accept New Settings->
5

```

Figure B-7-9 AquaLush Set Clock Screen

6.3.5.2 The current day and time displays must change as time progresses.

6.3.5.2.1 If the user alters the current time, then the display must no longer progress.

6.3.5.3 If the user presses the DEL key, then characters must be erased from the Current Time display from right to left.

6.3.5.4 If the user presses keypad digit keys, then digits must be added to the Current Time display from left to right.

6.3.5.4.1 If the user presses keypad digit keys when there are already four digits in the Current Time display, then the display must not be changed and the terminal must beep.

6.3.5.4.2 Users must be constrained in entering digits in the following ways:

- (a) The first digit must be in the range zero to two.
- (b) If the first digit is zero or one, then the second digit must be in the range zero to nine.
- (c) If the first digit is two, then the second digit must be in the range zero to three.
- (d) The third digit must be in the range zero to five.
- (e) The fourth digit must be in the range zero to nine.

6.3.5.4.2.1 If the user attempts to enter an illegal digit, the digit must not be displayed and the terminal must beep.

6.3.5.5 If the user presses a screen key for a day, the day must be shown in the Current Day display immediately.

6.3.5.6 If the user presses the ESC key, the clock must not be changed.

6.3.5.7 If the user presses the “Accept New Settings” button, the current settings (possibly altered by the user) must be used to reset the clock.

6.3.5.7.1 If a new time is only partially entered when the user presses the “Accept New Settings” button, the time must not be reset.

6.3.6. The Fix Failures state must allow users to see which devices have failed and to tell AquaLush which have been repaired.

6.3.6.1 The Fix Failures state must display the screen shown in Figure B-7-10.

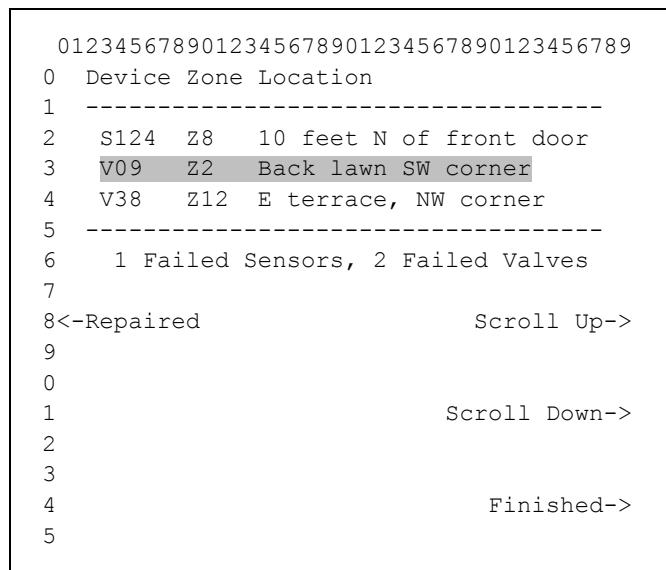


Figure B-7-10 AquaLush Fix Failures Screen

6.3.6.2 The region between the dashed lines must be a scrollable display of all failed sensors and valves, their zones, and their locations.

6.3.6.2.1 The failed sensors must be listed first, followed by failed valves in zone identifier order and then in valve identifier order within a zone.

6.3.6.2.2 The middle, highlighted element is the *current device*.

6.3.6.2.1 If the list is empty, there must be no current device.

6.3.6.2.2 A non-empty list must scroll up to or down to the middle element so that any device can be made current and there is always a current device.

6.3.6.2.3 The two middle screen buttons on the right must control scrolling.

6.3.6.3 If the user presses the “Repaired” screen button, then the current device must be recorded as repaired and removed from the list.

6.3.6.3.1 When a device is removed from the list and there is a device below the removed device, the device below must become the current device.

6.3.6.3.2 When a device is removed from the list and there is no device below the removed device but there is one above it, then device above must become the current device.

6.3.6.3.3 When a device is removed from the list and the list becomes empty, there must be no current device.

6.3.6.4 The Device field must show the identifier of each failed device left-aligned in columns three through six.

6.3.6.5 The Zone field must show the failed device’s zone identifier left-aligned in columns 9 through 12.

6.3.6.6 The Location field must display the failed device’s location string left-aligned in columns 14 through 37.

6.3.6.7 The failed devices summary in row eight must show counts of failed sensors and valves with no leading zeros in a centered display.

6.3.7. The **Set Up Auto Irrigation** state must allow users to configure automatic moisture-controlled irrigation.

6.3.7.1 The **Set Up Auto Irrigation** state must display the screen shown in Figure B-7-11.

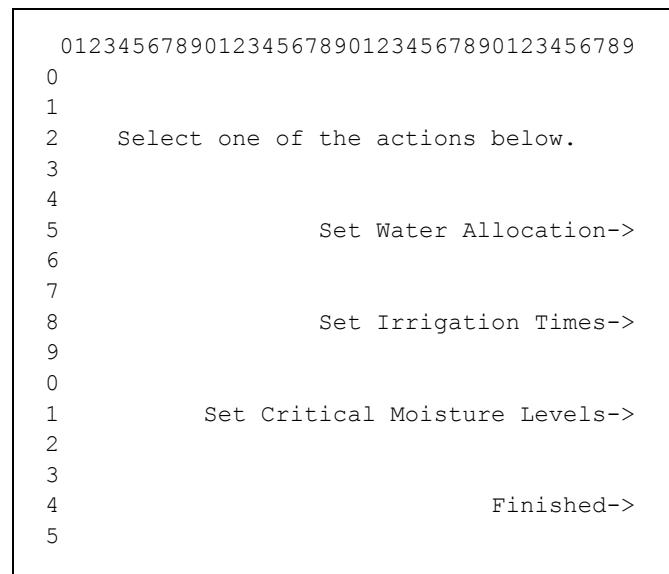


Figure B-7-11 AquaLush Set Up Auto Irrigation Screen

6.3.8. The Set Irrigation Times state must allow users to set automatic irrigation times.

6.3.8.1 The Set Irrigation Times state must display the screen shown in Figure B-7-12.

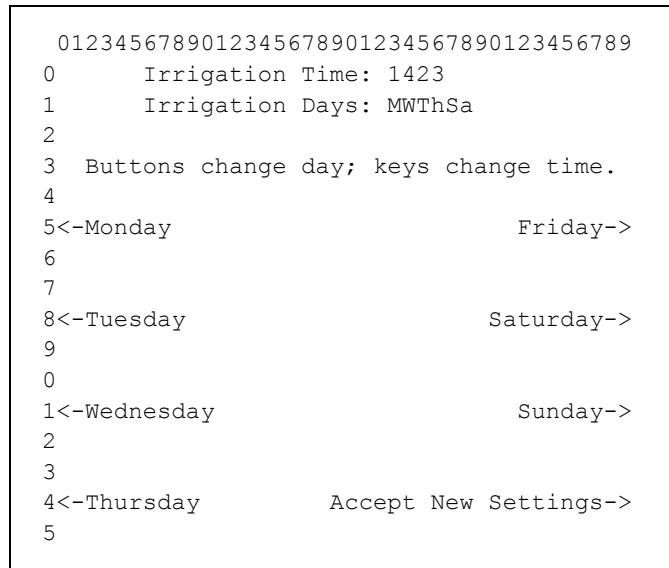


Figure B-7-12 AquaLush Set Irrigation Times Screen

6.3.8.2 If the user presses the DEL key, then characters must be erased from the Irrigation Time display from right to left.

6.3.8.3 If the user presses keypad digit keys, then digits must be added to the Irrigation Time display from left to right.

6.3.8.3.1 If the user presses keypad digit keys when there are already four digits in the Irrigation Time display, then the display must not be changed and the terminal must beep.

6.3.8.3.2 Users must be constrained in entering digits in the following ways:

- (a) The first digit must be in the range zero to two.
- (b) If the first digit is zero or one, then the second digit must be in the range zero to nine.
- (c) If the first digit is two, then the second digit must be in the range zero to three.
- (d) The third digit must be in the range zero to five.
- (e) The fourth digit must be in the range zero to nine.

6.3.8.3.2.1 If the user attempts to enter an illegal digit, the digit must not be displayed and the terminal must beep.

6.3.8.4 The Irrigation Days display must show the current set of irrigation days as a day set string.

6.3.8.4.1 The day set string must show the currently selected irrigation days in order, starting from Monday using, the following abbreviations:

- (a) Monday must be abbreviated as "M."
- (b) Tuesday must be abbreviated as "Tu."
- (c) Wednesday must be abbreviated as "W."
- (d) Thursday must be abbreviated as "Th."
- (e) Friday must be abbreviated as "F."
- (f) Saturday must be abbreviated as "Sa."
- (g) Sunday must be abbreviated as "Su."

6.3.8.5 If the user presses a screen key for a day and that day is present in the Irrigation Days display, then that day must be removed from the Irrigation Days display.

6.3.8.6 If the user presses a screen key for a day and that day is not present in the Irrigation Days display, then that day must be added to the Irrigation Days display.

6.3.8.7 If the user presses the ESC key, the irrigation days and time must not be changed.

6.3.8.8 If the user presses the “Accept New Settings” button, the current settings (possibly altered by the user) must be used for irrigation times.

6.3.8.8.1 If a new irrigation time is only partially entered when the user presses the “Accept New Settings” button, the irrigation time must not be changed, but any changes to the irrigation days must be accepted.

6.3.9. The **Set Moisture Levels** state must allow users to alter each zone’s critical moisture level.

6.3.9.1 The **Set Moisture Levels** state must display the screen shown in Figure B-7-13.

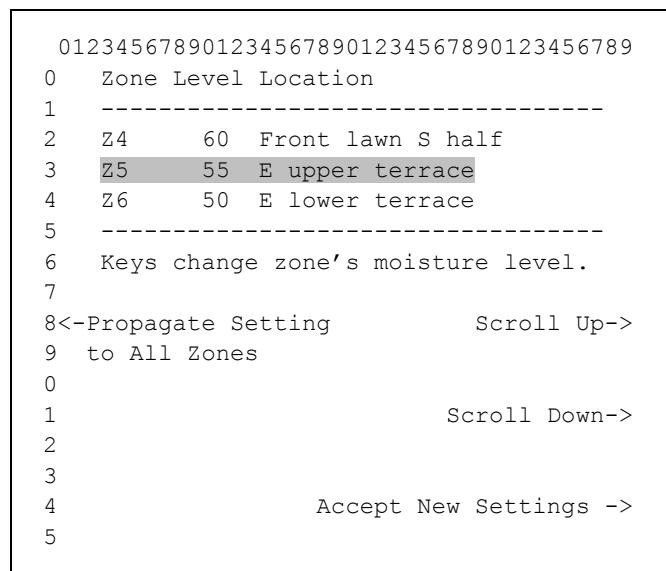


Figure B-7-13 AquaLush Set Moisture Levels Screen

6.3.9.2 The region between the dashed lines must be a scrollable display of all zones, their critical moisture levels, and their locations.

6.3.9.2.1 The zones must be listed in zone identifier order.

6.3.9.2.2 The middle, highlighted element is the *current zone*.

6.3.9.2.2.1 The list must scroll up to or down to the middle element so that any zone can be made current.

6.3.9.2.2.2 There must always be a current zone.

6.3.9.2.3 The two middle screen buttons on the right must control scrolling.

6.3.9.3 If the user presses the “Propagate Settings to All Zones” screen button, then the current zone’s critical moisture setting must be applied to all zones.

6.3.9.4 If the user presses the DEL key, then characters must be erased from the current zone’s moisture level display from right to left.

6.3.9.5 If the user presses keypad digit keys, then digits must be added to the current zone's moisture level display from left to right.

6.3.9.5.1 If the user presses keypad digit keys when there are already three digits in the current zone's moisture level display, then the display must not be changed and the terminal must beep.

6.3.9.5.2 Users must be constrained in entering digits in following ways:

(a) The first two digits must be in the range zero to nine.

(b) The third digit must be a zero, and it must be accepted only if the first two digits are "10."

6.3.9.6 The Zone field must show the zone identifier left-aligned in columns three through six.

6.3.9.7 The Level field must show the zone's critical moisture level right-aligned in columns 9 through 11 with no leading zeros.

6.3.9.8 The Location field must display the zone's location string left-aligned in columns 14 through 37.

6.3.9.9 If the user presses the ESC key, then no changes made to the critical moisture levels must take effect.

6.3.9.10 If the user presses the "Accept New Settings" button, then all changes to critical moisture levels must take effect.

6.3.10. The Set Water Allocation state must allow users to set the total amount of water to be used in an automatic irrigation cycle.

6.3.10.1 The Set Water Allocation state must display the screen shown in Figure B-7-14.

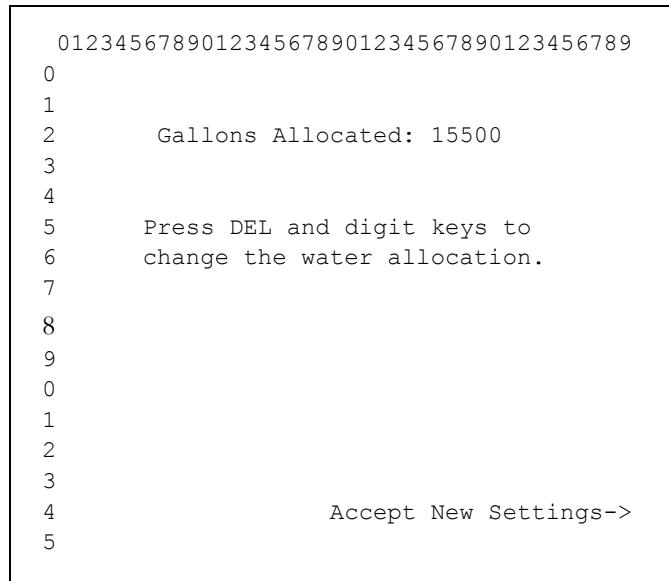


Figure B-7-14 AquaLush Set Water Allocation Screen

6.3.10.2 If the user presses the DEL key, then characters must be erased from the Gallons Allocated display from right to left.

6.3.10.3 If the user presses keypad digit keys, then digits must be added to the Gallons Allocated display from left to right.

6.3.10.3.1 If the user presses keypad digit keys when there are already 14 digits in the Gallons Allocated display, then the display must not be changed and the terminal must beep.

- 6.3.10.4 If the user presses the ESC key, the water allocation must not be changed.
- 6.3.10.5 If the user presses the “Accept New Settings” button, the current water allocation must be used to regulate irrigation.
- 6.3.11. The **Device Failure** state must be entered to notify the user that a sensor or valve has failed and possibly that AquaLush cannot write to its persistent store.
- 6.3.11.1 The **Device Failure** state must display the screen shown in Figure B-7-15.

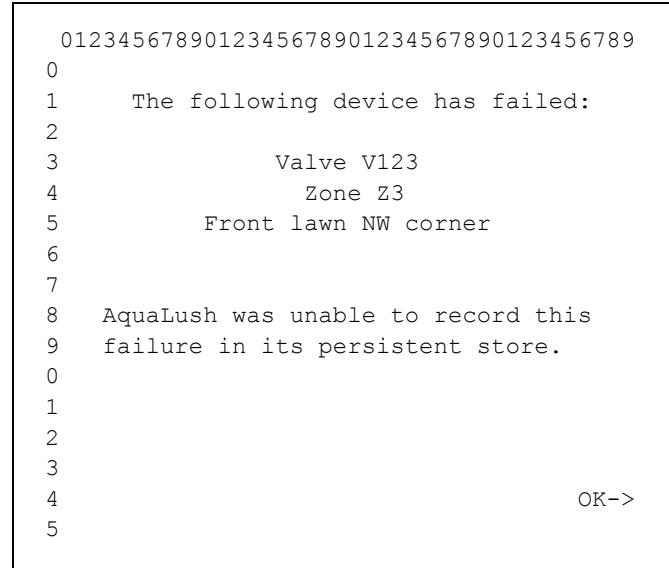


Figure B-7-15 AquaLush Device Failure Screen

- 6.3.11.2 The device identifier, zone, and location must vary.
- 6.3.11.3 If the failed device is a valve, then the display must show the valve identifier, the valve’s zone, and the valve’s location string.
- 6.3.11.4 If the failed device is a sensor, then the display must show the sensor identifier, the sensor’s zone, and the sensor’s location string.
- 6.3.11.5 The data displayed in rows three through five must be centered.
- 6.3.11.6 The message about failure to write to persistent store must appear only if this failure actually occurs.
- 6.3.11.7 When the user presses the “OK” button, the program must return to the state from which it came when the failure was detected.
- 6.3.12. The **Persistent Store Failure** state must be entered to notify the user that AquaLush cannot read from or write to its persistent store.
- 6.3.12.1 The **Persistent Store Failure** state must display the screen shown in Figure B-7-16.
- 6.3.12.2 When the user presses the “OK” button, the program must return to the state from which it came when the failure was detected.

```
0123456789012345678901234567890123456789  
0  
1  
2  
3  
4  
5     AquaLush was unable to read from or  
6     write to its persistent data store.  
7  
8  
9  
0  
1  
2  
3  
4                                         OK->  
5
```

Figure B-7-16 AquaLush Persistent Store Failure Screen

6.3.13 If AquaLush cannot read its persistent store during startup, then it must display the screen shown in Figure B-7-17 and suspend further processing.

```
0123456789012345678901234567890123456789  
0  
1  
2  
3  
4     *****  
5     * AquaLush is unable to start. *  
6     * Please call your installer    *  
7     * or supplier for help.      *  
8     *****  
9  
0  
1  
2  
3  
4  
5
```

Figure B-7-17 AquaLush Unable to Start Screen

7. Simulation User Interface

7.1 User Interface Appearance

7.1.1. The simulation user interface must appear as indicated in Figure B-7-18.

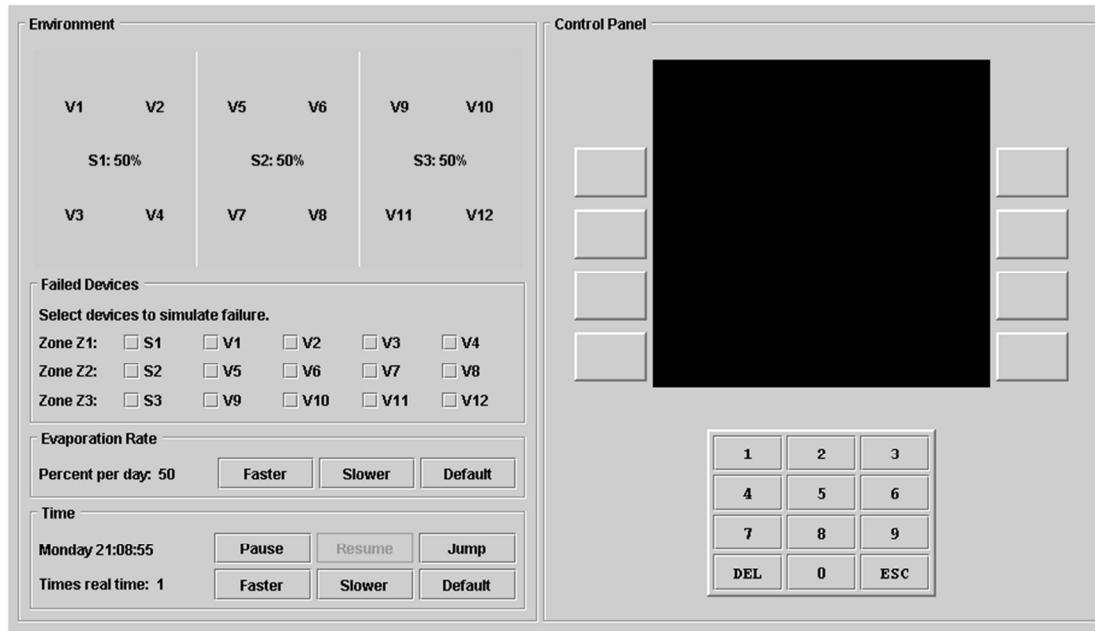


Figure B-7-18 AquaLush Simulation User Interface

7.1.1.1 The right-hand side of the user interface must simulate the AquaLush control panel.

7.1.1.2 The left-hand side of the user interface must control and display the state of the simulated irrigation site.

7.2 User Interface Behavior

7.2.1. The failed devices controls must behave as specified in Figure B-7-19.

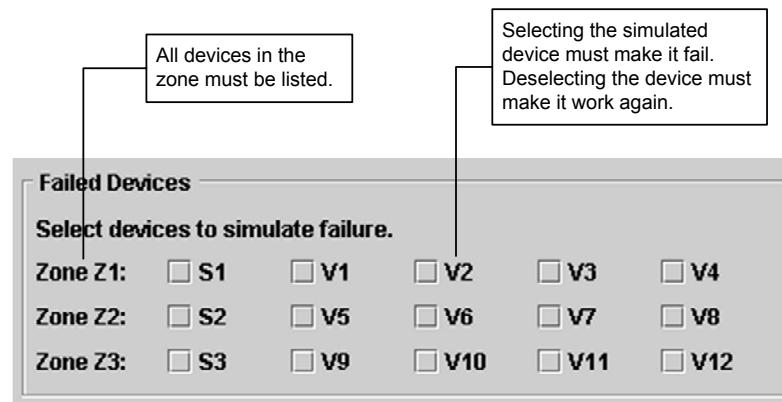


Figure B-7-19 Simulated Failed Devices Controls

7.2.2. The irrigation site display must behave as indicated in Figure B-7-20.

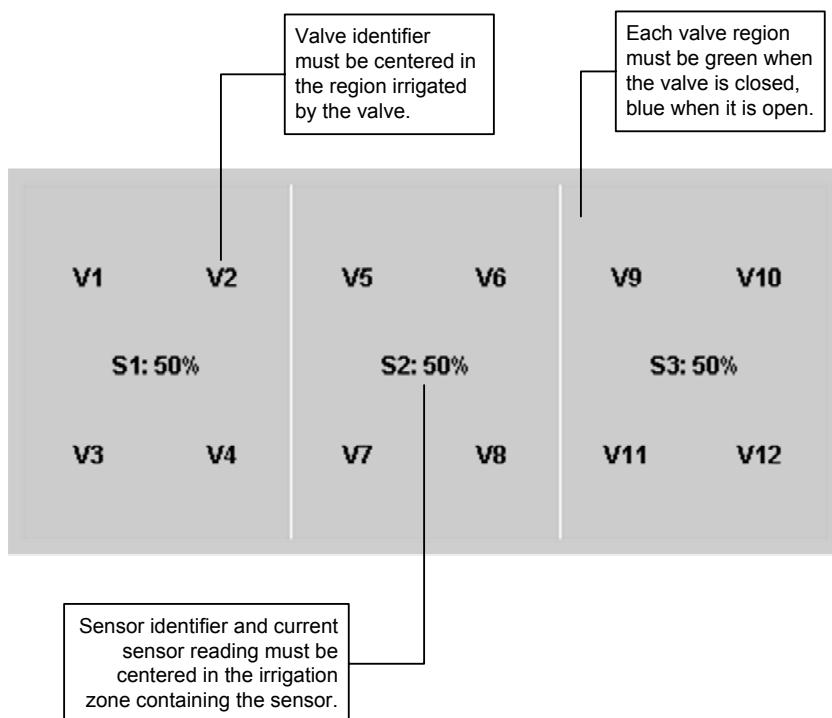


Figure B-7-20 Simulated Irrigation Site Display

7.2.3. The simulated time controls must behave as specified in Figure B-7-21.

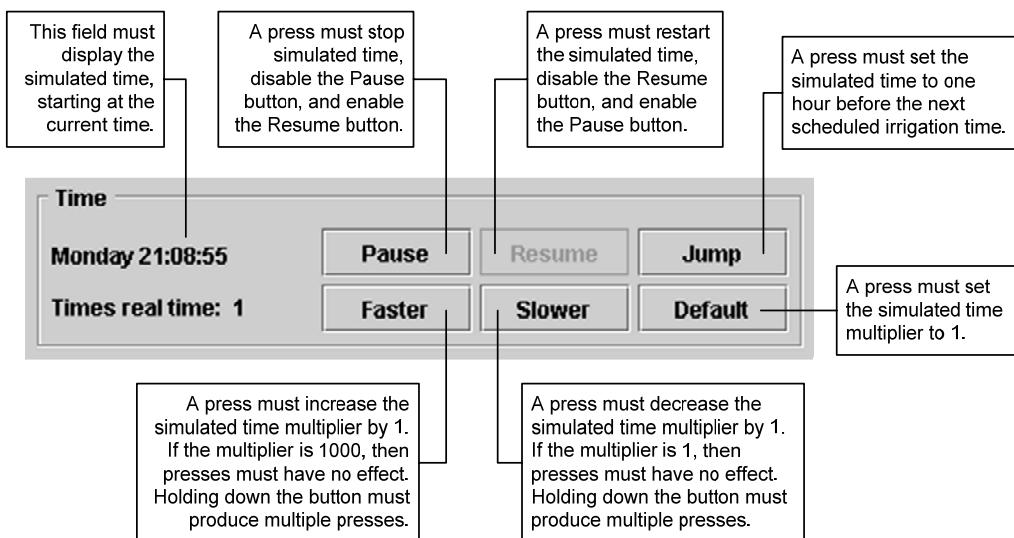


Figure B-7-21 Simulated Time Controls

7.2.4. The evaporation controls must behave as specified in Figure B-7-22.

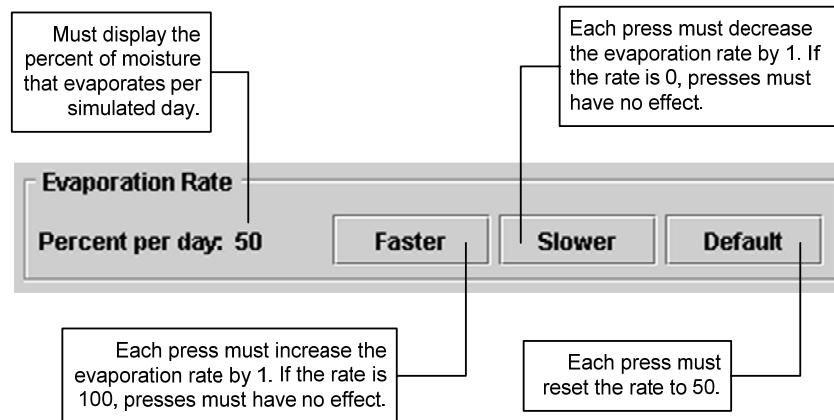


Figure B-7-22 Simulated Evaporation Controls

B.8 AquaLush Conceptual Model

Overview

Although no manual irrigation cycle is mentioned explicitly in the SRS or the use case model, it seems appropriate to have one to go along with the automatic irrigation cycle that is central to the way that automatic irrigation works.

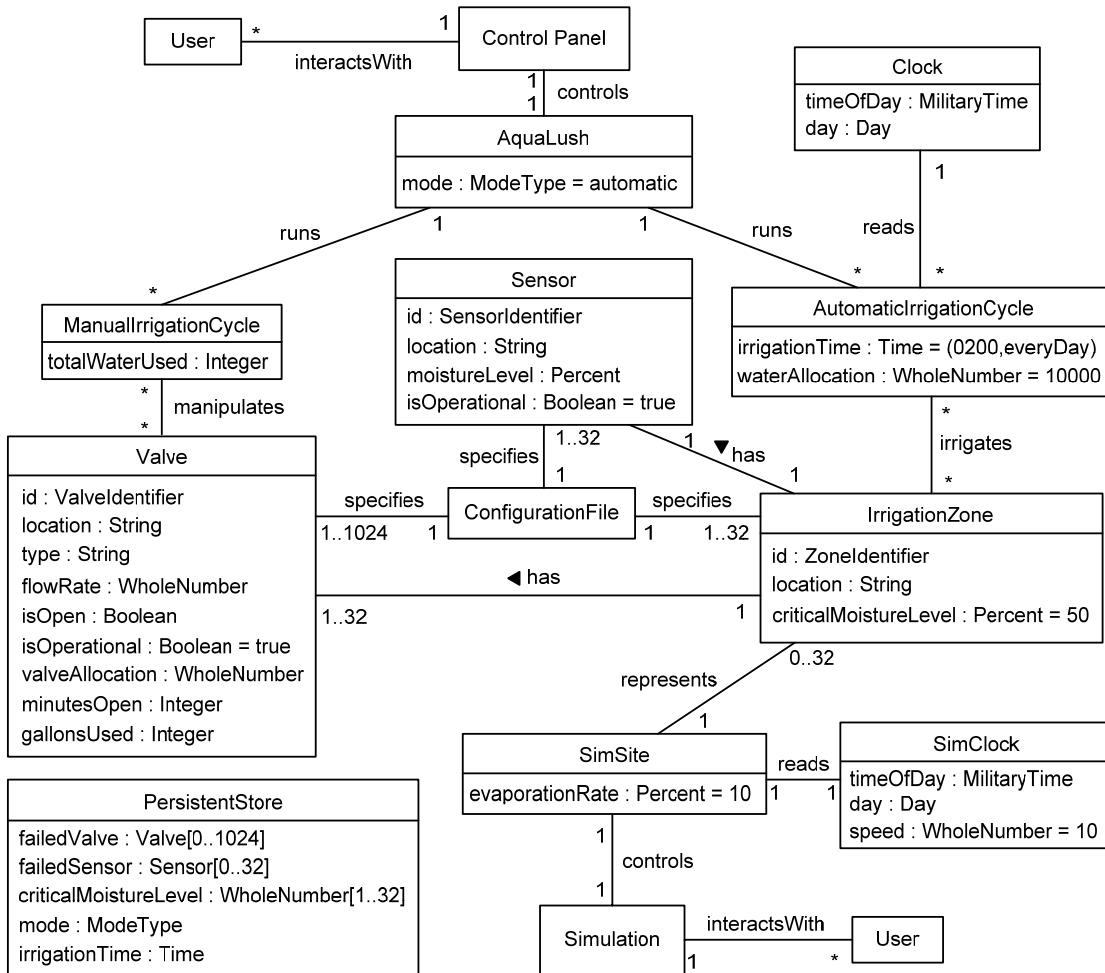


Figure B-8-1 AquaLush Conceptual Model

B.9 AquaLush Profiles and Scenarios

Utility Tree

The numbers following the scenario names in Figure B-9-1 are scenario weights based on importance and likelihood of occurring. The weights are either High (H), Medium (M), or Low (L).

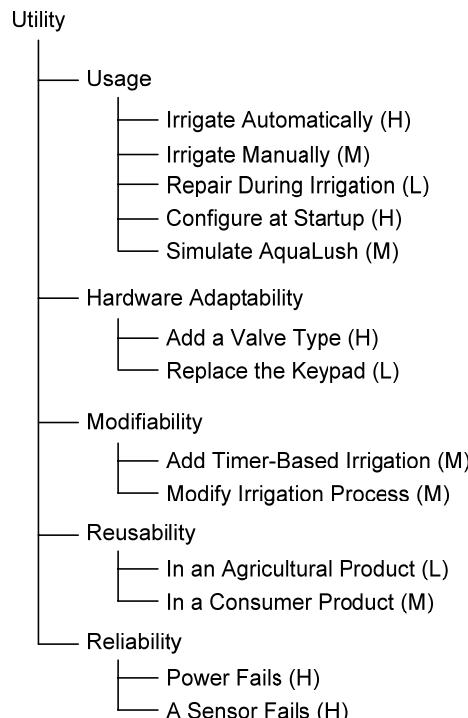


Figure B-9-1 AquaLush Utility Tree

Scenario Descriptions

The following descriptions are arranged by profile.

Usage Profile

Irrigate Automatically—AquaLush is in automatic mode. An irrigation time arrives and AquaLush begins an automatic irrigation cycle. When all zones are irrigated the irrigation cycle ends and all valves are closed.

Irrigate Manually—AquaLush is in manual mode and a user begins a manual irrigation cycle. The user directs AquaLush to open or close valves. AquaLush responds by opening or closing the valves and displaying irrigation information. The user eventually directs AquaLush to end the irrigation cycle. AquaLush closes any open valves and ends the cycle.

Repair During Irrigation—AquaLush is in the midst of an automatic irrigation cycle. A valve in the zone currently being irrigated is broken, and hence AquaLush is not using it. The user informs AquaLush

that the broken valve is repaired. AquaLush begins using it within one minute of being informed that it is repaired.

Configure at Startup—The AquaLush program loads and executes. AquaLush is able to communicate with all valves and sensors.

Simulate AquaLush—A Web user brings up the AquaLush simulation. The user manipulates the simulation. The simulation works just the way the fielded AquaLush product does.

Hardware Adaptability Profile

Add a Valve Type—A developer modifies the AquaLush source code and design documents so that AquaLush supports a new type of valve. The developer completes the process in two days.

Replace the Keypad—A new brand of keypad is used in the control panel. A developer is able to modify the source code to use the new keypad within two days.

Modifiability Profile

Add Timer-Based Irrigation—A new timer-based mode is added to AquaLush. When in timer-based mode, AquaLush starts an automatic irrigation cycle at the preset irrigation time and irrigates each zone for a set period of time or until the zone's water allocation is exceeded. A developer is able to modify the design and source code to add this mode in two weeks.

Modify Irrigation Process—The automatic irrigation process is changed to allow users to specify concurrent irrigation of two or more zones. A developer is able to modify the design and source code to make this change in three weeks.

Reusability Profile

In an Agricultural Product—VIS develops a new product especially for an agricultural market. Developers are able to reuse at least 60% of the design and code from AquaLush.

In a Consumer Product—VIS develops a new product especially for the home. Developers are able to reuse at least 90% of the design and code from AquaLush.

Reliability Profile

Power Fails—Power is applied to AquaLush after a power failure. AquaLush returns to its state at the time of the power failure, except that AquaLush is not able to set its clock and, if an irrigation cycle was underway when the power failed, it is abandoned. AquaLush closes all irrigation valves.

A Sensor Fails—A sensor fails during automatic irrigation. AquaLush closes all valves in that zone within one minute and goes to the next zone. AquaLush marks the sensor as broken and does not attempt to use it again until told that it is repaired by a user.

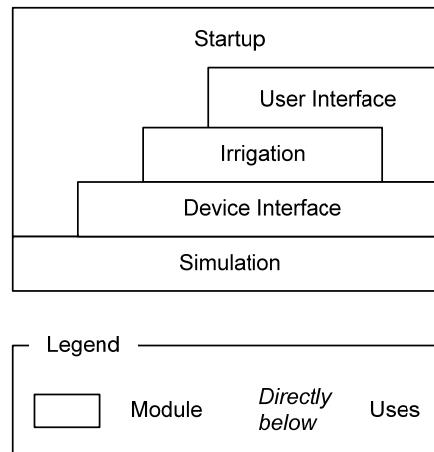
B.10 AquaLush Software Architecture Document

1. Product Overview

Please see section B.2 *AquaLush Project Mission Statement* for an overview of the product.

2. Architectural Models

AquaLush is a Relaxed Layered system, as shown in the layer diagram in Figure B-10-1.

**Figure B-10-1 AquaLush Layers Overview**

Modules use only layers below them that they touch directly. A fielded system does not include the bottom Simulation layer, which is needed only as a stand-in for the hardware that would be present in a fielded system.

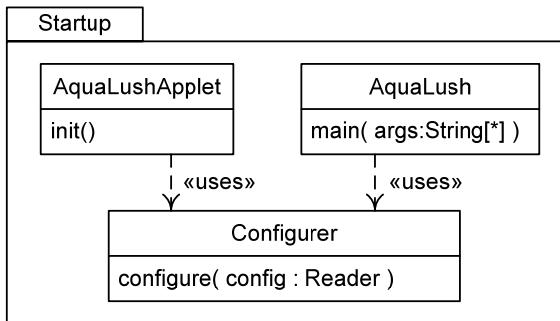
2.1 Layer Responsibilities

Layer	Responsibilities
Startup	Create and connect all runtime components based on a configuration specification and whether the program is a simulation or a fielded product. Restore the state of the system from persistent store.
User Interface	Implement a device-independent AquaLush text-based user interface.
Irrigation	Control both manual and automatic irrigation.
Device Interface	Implement virtual devices providing interfaces to all external entities, including hardware devices, a clock, and persistent storage.
Simulation	Implement components simulating hardware devices, simulate an irrigation site, and provide views and controls for manipulating the simulation.

Table B-10-2 AquaLush Layer Responsibilities

2.2 Startup Layer Decomposition

The Startup layer contains the main program class, which may be either an application or an applet, assuming the program is written in Java. The main program uses a Configurer, which sets up the program based on the available hardware (real or simulated) and whether it is a simulation or a fielded program, and restores the previous program state. The Configurer reads a configuration specification to determine what sort of hardware (or simulated hardware) the program must use. It reads a persistent store to reset the program state. Figure B-10-3 shows the decomposition of the Startup layer.

**Figure B-10-3 Startup Layer Structure****2.2.1 Startup Layer Module Responsibilities**

Module	Responsibilities
AquaLushApplet	Ask the Simulation layer to create all GUI components and simulated entities, and create a Configurer that uses these simulation objects to configure the program. Obtain the configuration specification. Call the Configurer to configure the program.
AquaLush	Create a Configurer that uses real hardware. Obtain the configuration specifications. Call the Configurer to configure the program.
Configurer	Create and connect runtime components based on configuration specifications and whether the program is a simulation or a fielded product. Restore the state of the system from persistent store.

Table B-10-4 Startup Layer Module Responsibilities**2.2.2 Startup Layer Interface Specifications**

Services Provided

Applet initialization	Syntax:	init()
	Pre:	None.
	Post:	The applet simulating AquaLush is initialized.
Program execution	Syntax:	main(args : String[])
	Pre:	None.
	Post:	The AquaLush program is set up according to its local configuration and its state is reset to what it was when it was last executing.

Table B-10-5 Startup Layer Interface Specifications

Services Required

`Simulation.create()`—Create the AquaLush simulation, including GUI components simulating the control panel display, keypad, and screen buttons; the simulated irrigation site with valves and sensors; and simulated time and persistent store objects.

`DeviceFactory.createDeviceX()`—Create various virtual devices already connected to real or simulated hardware.

`DeviceX.setListener()`—Add a listener to a keypad or screen button virtual device.

`Irrigator.create()`—Create irrigation control objects.

`Irrigator.addX()`—Configure the Irrigator with zones, sensors, and valves, and configure the virtual devices controlling valve and sensor hardware.

`Irrigator.restoreState()`—Restore the state of the program from persistent store.

`UserInterface.create()`—Create the user interface.

`UserInterface.setDisplay()`—Configure the user interface with the virtual display.

`Reader.read()`—Obtain an input character from the configuration specification.

2.2.3 Startup Layer Design Rationale

The configuration task could have been placed in the applet and application classes, but then there would have been two versions of very similar code. Separating this task into a separate component (the Configurer) that can be reused makes it easier to change and correct as well as faster to implement, though it takes more work to design originally. Note: This component will make use of the Abstract Factory pattern for virtual devices, which we discuss in the next section.

2.3 Simulation Layer Decomposition

The Simulation layer contains components that simulate the entire external environment during a simulation. Five of these are externally visible: a time simulator, a persistent storage simulator, a display device simulator, and sensor and valve simulators. In addition this layer has a façade class that hides other components (such as a simulated keypad, simulated screen buttons, the simulated site, and so forth). These components are pictured in Figure B-10-6.

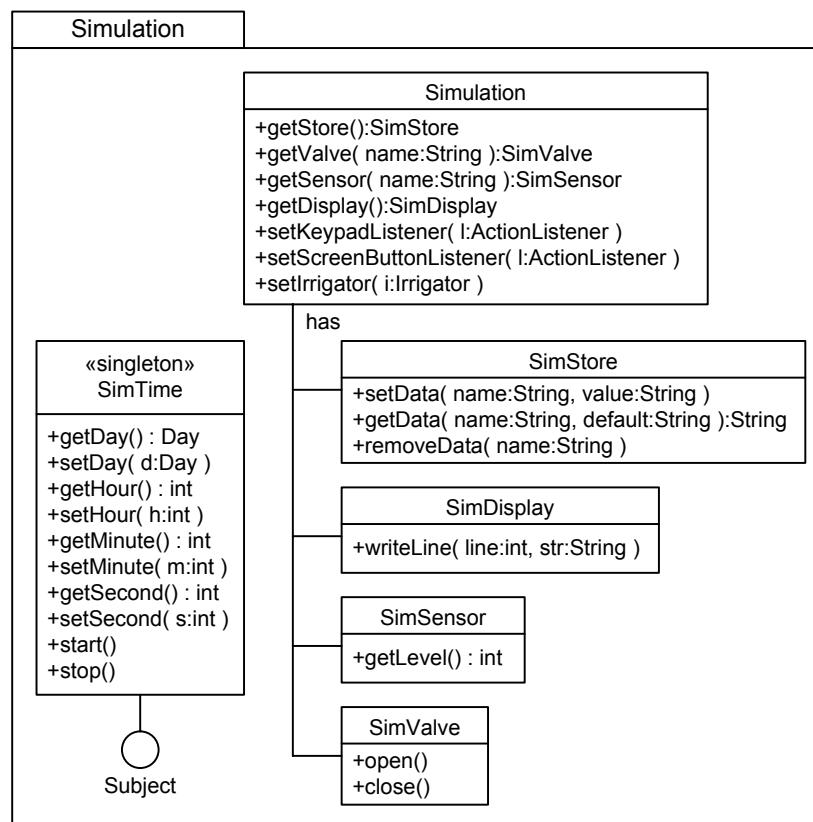


Figure B-10-6 Simulation Layer Structure

2.3.1 Simulation Layer Module Responsibilities

Module	Responsibilities
Simulation	Simulate hardware devices and the real world, and display the simulation using GUI components.
SimStore	Simulate a persistent store. A real file will be used if possible; if not, no real persistence will be simulated.
SimDisplay	Simulate a 16-line-by-40-character monochrome textual display as a GUI component.
SimSensor	Simulate a moisture sensor.
SimValve	Simulate an irrigation valve.
SimTime	Simulate the passage of time. Notify other components of the passage of simulated time.

Table B-10-7 Simulation Layer Module Responsibilities

2.3.2 Simulation Layer Interface Specifications

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Create the simulation object	<i>Syntax:</i>	<code>create()</code>
	<i>Pre:</i>	None.
	<i>Post:</i>	All simulated entities are created.
Provide simulated persistent store	<i>Syntax:</i>	<code>getStore() : SimStore</code>
	<i>Pre:</i>	None.
	<i>Post:</i>	The simulated storage object is returned.
Provide a simulated valve	<i>Syntax:</i>	<code>getValve(name : String) : SimValve</code>
	<i>Pre:</i>	name is not null.
	<i>Post:</i>	The simulated valve whose identifier is name is returned, or null is returned if no such valve exists.
Provide a simulated sensor	<i>Syntax:</i>	<code>getSensor(name : String) : SimSensor</code>
	<i>Pre:</i>	name is not null.
	<i>Post:</i>	The simulated sensor whose identifier is name is returned, or null is returned if no such sensor exists.
Provide the simulated display	<i>Syntax:</i>	<code>getDisplay() : SimDisplay</code>
	<i>Pre:</i>	None.
	<i>Post:</i>	The simulated display object is returned.
Register a keypad listener	<i>Syntax:</i>	<code>setKeypadListener(I : KeypadListener)</code>
	<i>Pre:</i>	None.
	<i>Post:</i>	KeypadListener I will start to receive notifications of simulated keypad key presses.
Register a screen button listener	<i>Syntax:</i>	<code>setScreenButtonListener(I : ScreenButtonListener)</code>
	<i>Pre:</i>	None.
	<i>Post:</i>	ScreenButtonListener I will start to receive notifications of simulated screen button presses.

Record a value in simulated persistent store	Syntax: setData(name : String, value : String) Pre: name and value are not null. Post: The name-value pair is recorded in persistent store. Throws DeviceFailureException if persistent store cannot be read.
Fetch a value from simulated persistent store	Syntax: getData(name : String, default : String) : String Pre: name is not null. Post: Returns the value associated with the name in persistent store. If no value is associated with name and default is not null, then associates the default with the name and returns it. If no value is associated with the name and the default is null, then returns null. Throws DeviceFailureException if persistent store cannot be read.
Remove data from simulated persistent store	Syntax: removeData(name : String) Pre: name is not null. Post: The name and associated value are removed from persistent store. Throws DeviceFailureException if persistent store cannot be read.
Write to the simulated display	Syntax: write(line : int, str : String) Pre: 0 <= line < 16 and str is not null. Post: String str is written to line number line of the simulated display, starting at position 0.
Read a simulated sensor	Syntax: getLevel() : int throws DeviceFailureException Pre: None. Post: The simulated moisture level for the sensor is returned. Throws DeviceFailureException if the sensor has a simulated failure.
Open a simulated valve	Syntax: open() throws DeviceFailureException Pre: None. Post: Opens a simulated valve. Throws DeviceFailureException if the valve has a simulated failure.
Close a simulated valve	Syntax: close() throws DeviceFailureException Pre: None. Post: Closes a simulated valve. Throws DeviceFailureException if the valve has a simulated failure.
Fetch the instance of SimTime	Syntax: SimTime.instance() : SimTime Pre: None. Post: A single instance of the SimTime object will always be returned. The SimTime object always starts off with the current real time, but is stopped.
Get the simulated time	Syntax: getDay() : Day getHour() : int getMinute() : int getSecond() : int Pre: None. Post: Returns the requested component of the simulated time.

Set the simulated time	Syntax:	setDay(d : Day) setHour(h : int) setMinute(h : int) setSecond(s : int)
	Pre:	0 <= h < 24 and 0 <= m, s < 60.
	Post:	The simulated time is adjusted as indicated.
Start and stop the simulated time	Syntax:	start() stop()
	Pre:	None.
	Post:	The simulated time is paused or resumed. Starting simulated time when it is already going or stopping it when it is already stopped has no effect.
Time notification	Syntax:	addObserver(o : Observer)
	Pre:	None.
	Post:	Observer o will be notified when simulated time is not stopped and a simulated second has passed.

Table B-10-8 Simulation Layer Interface Specifications**Services Required**

The **Simulation** layer requires the services of a GUI toolkit (such as Java Swing). It requires a **Day** enumeration type. It also requires a mechanism for regular notification of the passage of time occurring at least once per second and ideally much faster so that simulated time can be sped up.

Usage Guide

The **Simulation** layer is needed only when **AquaLush** is being simulated. Create it first because it is used in every other layer and in constructing the objects in almost every other layer.

2.3.3 Simulation Layer Design Rationale

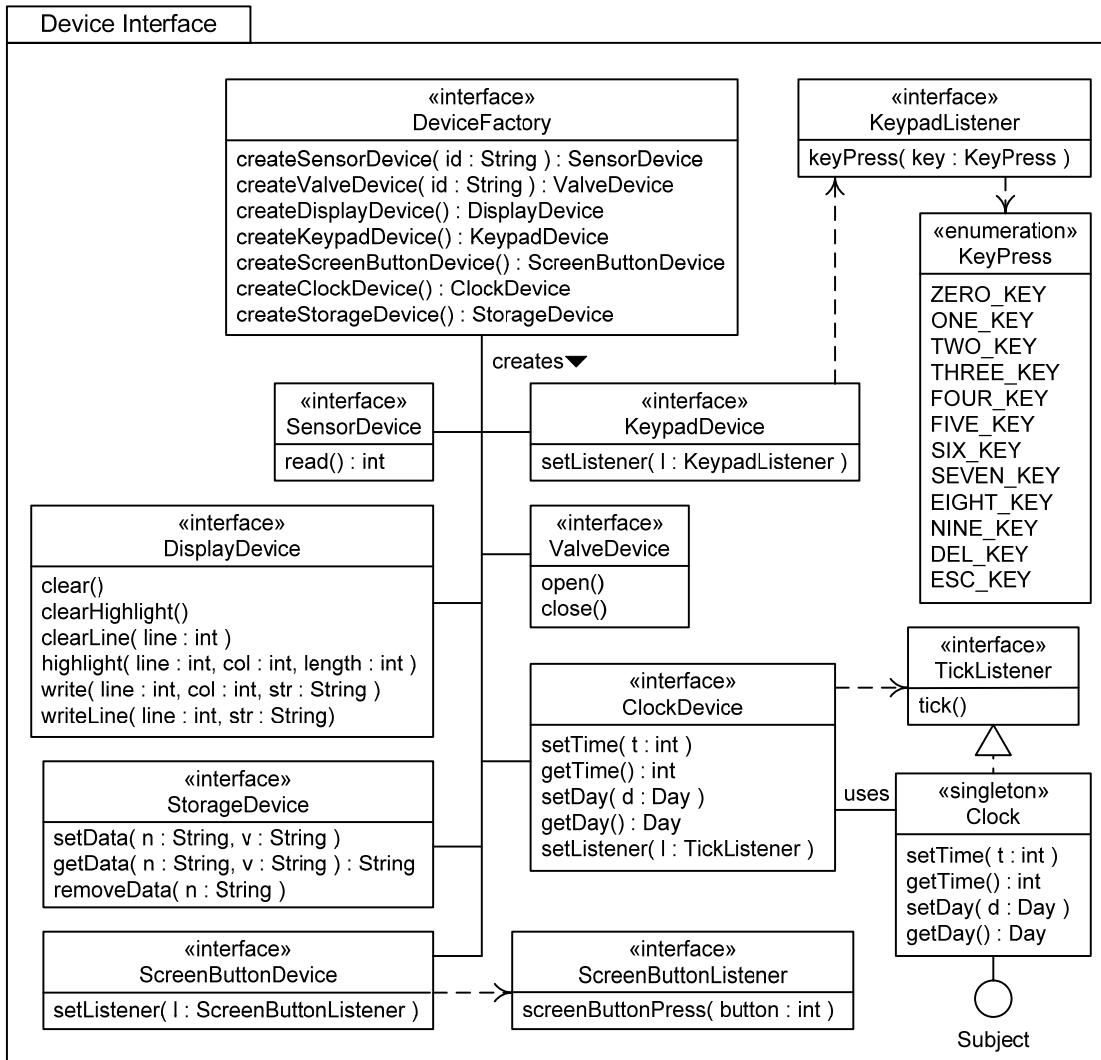
Many alternatives considered for the **Simulation** layer lacked various components or had functionality present in other layers. For example, the persistent store was not initially in this layer until it was realized that an applet may not be allowed to access a host operating system because of security concerns. Eventually it was realized that this layer needs to simulate *everything* in the environment of the software. As another example, in one alternative the simulation GUI construction process was left partly up to the applet, so that the applet requested a simulated control panel and a simulated environment from the **Simulation** layer, then combined them to form the applet. The present alternative is more cohesive.

The **SimTime** class is a singleton class. This was deemed necessary because having more than one simulated time object would certainly cause the program to fail. Once **SimTime** is made a singleton there is no need for an operation to retrieve it from the **Simulation** layer façade, the role played by the **Simulation** class.

SimTime is also a subject in the Observer pattern. If it were an invoker in the Command pattern, then it could have only a single reactor. However, several components in the simulation as well as the **ClockDevice** in the **Device Interface** layer need to be notified of the passage of simulated time.

2.4 Device Interface Layer Decomposition

The **Device Interface** layer provides virtual devices to hide the real or simulated external entities manipulated by the program. Writing the program to interact only with virtual devices makes **AquaLush** highly configurable. The virtual devices provided in this layer are pictured in Figure B-10-9.

**Figure B-10-9 Device Interface Layer Structure**

Note that these entities are all interface types except for the **Clock** and the **KeyPress** enumeration. This layer also includes private realizations of these interfaces that actually do the work, but these are hidden at the architectural level of detail.

2.4.1 Device Interface Layer Module Responsibilities

Module	Responsibilities
ClockDevice Interface	Provide virtual clock hardware that keeps track of the day of the week and the time of the day, accurate to the minute.
TickListener Interface	Guarantee that a module listening to the ClockDevice implements a tick() operation, which is needed for the Command pattern.
StoreDevice Interface	Provide virtual persistent storage of name-value pairs.
DisplayDevice Interface	Provide a virtual 16-line-by-40-character monochrome textual display device.
SensorDevice Interface	Provide virtual moisture sensor hardware.
ValveDevice Interface	Provide virtual irrigation valve hardware.
KeypadDevice Interface	Provide a virtual 12-key keypad hardware device.
KeypadListener Interface	Guarantee that a module registered as a Keypad listener implements the keyPress() operation, which is needed for the Command pattern.
ScreenButtonDevice Interface	Provide a virtual hardware device with eight push buttons.
ScreenButtonListener Interface	Guarantee that a module registered as a ScreenButton listener implements the screenButtonPress() operation, which is needed for the Command pattern.
Clock	Provide the day of the week and the time of day, accurate to one minute. Notify observers of the passage of time every minute. The Clock should be used by the rest of the system; it relies on the ClockDevice.
DeviceFactory Interface	Provide abstract factory methods for creating all virtual devices. This interface is needed as part of the Factory Method pattern.

Table B-10-10 Device Interface Layer Module Responsibilities

2.4.2 Device Interface Layer Interface Specifications

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the precondition is violated.

Set the time of the day (Clock and ClockDevice)	Syntax:	<code>setTime(milTime : int)</code>
	Pre:	milTime is a legitimate military time specification.
	Post:	The clock device is reset to milTime.
Get the time of the day (Clock and ClockDevice)	Syntax:	<code>getTime() : int</code>
	Pre:	None.
	Post:	The current time is returned, accurate to the minute, in military time format.
Set the day of the week (Clock and ClockDevice)	Syntax:	<code>setDay(d : Day)</code>
	Pre:	None.
	Post:	The clock device is reset to day d.

Get the day of the week (Clock and ClockDevice)	Syntax:	getDay() : Day
	Pre:	None.
	Post:	The current day of the week is returned.
Set the ClockDevice listener	Syntax:	setListener(I : TickListener)
	Pre:	None.
	Post:	TickListener I will start to receive notifications of the passage of time every minute. Any previous TickListener is replaced.
Register a Clock observer	Syntax:	addObserver(o : Observer)
	Pre:	o is not null.
	Post:	Observer o will start to receive notifications every minute. Throws an exception if the precondition is violated.
Register a ScreenButton listener	Syntax:	setScreenButtonListener(I : ScreenButtonListener)
	Pre:	None.
	Post:	ScreenButtonListener I will start to receive notifications of screen button presses.
Register a Keypad listener	Syntax:	setKeypadListener(I : KeypadListener)
	Pre:	None.
	Post:	KeypadListener I will start to receive notifications of keypad button presses.
Record a value in persistent store	Syntax:	setData(n : String, v : String)
	Pre:	n and v are not null.
	Post:	The name-value pair is recorded in persistent store.
Fetch a value from persistent store	Syntax:	getData(n : String, v : String) : String
	Pre:	n is not null.
	Post:	Return the value associated with n in persistent store. If no value is associated with n and v is not null, then associate v with n and return v. If no value is associated with n and v is null, return null.
Remove data from persistent store	Syntax:	removeData(n : String)
	Pre:	n is not null.
	Post:	Remove the association between n and a value (if any) from persistent store.
Clear the display	Syntax:	clear()
	Pre:	None.
	Post:	The display is blanked.
Clear all display highlighting	Syntax:	clearHighlight()
	Pre:	None.
	Post:	All highlighted lines are returned to regular display; the text is unaltered.
Clear a line of the display	Syntax:	clearLine(line : int)
	Pre:	0 <= line < 16.
	Post:	The text (and highlighting) are removed from the designated line.

Highlight a display line	Syntax:	highlight(line : int, col : int, length : int)
	Pre:	0 <= line < 16 and 0 <= col < 40 and 0 <= length.
	Post:	The portion of line starting at the column and extending for the length of the characters (or until the end of the line) is highlighted.
Write a string to the display	Syntax:	write(line : int, col : int, str : String)
	Pre:	0 <= line < 16 and 0 <= col <= 40 and str is not null.
	Post:	String str is written to line number line and column col of the simulated display. If str is too long to fit on the line it is truncated to fit. The line is not highlighted.
Write a line to the display	Syntax:	writeln(line : int, str : String)
	Pre:	0 <= line < 16 and str is not null.
	Post:	String str is written to line number line of the simulated display, starting at position 0. If str is more than 40 characters it is truncated to 40 characters. If it is less than 40 characters, then it is padded with blanks until it is 40 characters. The line is not highlighted.
Read a sensor	Syntax:	read() : int throws DeviceFailureException
	Pre:	None.
	Post:	Returns the moisture level. Throws DeviceFailureException if the sensor fails.
Open a valve	Syntax:	open() throws DeviceFailureException
	Pre:	None.
	Post:	Opens a valve. Throws DeviceFailureException if the valve fails.
Close a valve	Syntax:	close() throws DeviceFailureException
	Pre:	None.
	Post:	Closes a valve. Throws DeviceFailureException if the valve fails.
Create virtual devices	Syntax:	createSensorDevice(id : String) : SensorDevice createValveDevice(id : String, rate) : ValveDevice createDisplayDevice() : DisplayDevice createKeypadDevice() : KeypadDevice createScreenButtonDevice() : ScreenButtonDevice createClockDevice() : ClockDevice createStorageDevice() : StorageDevice
	Pre:	id is not null.
	Post:	The desired virtual device is created. The virtual device is fully configured and connected to real or simulated hardware.

Table B-10-11 Device Interface Layer Interface Specifications**Services Required**

The Device Interface layer uses actual or simulated hardware. The details of these services vary: That is the point of having a device interface layer. This layer also requires a Day enumeration type and a DeviceFailureException class. The ClockDevice requires some sort of real or simulated device that notifies it when one minute has passed.

Usage Guide

The Clock, not the ClockDevice, is the main time and time notification service provider. The Clock is a singleton, so it is globally visible. It is also an observable, so it can be accessed from any layer and arbitrarily many observers can register with it for time notifications.

The extra parameter in `getData()` is there to make it easy to initialize the persistent store when the program is run for the first time. Default persistent values are supplied, and they will be written to the persistent store even if the store did not previously exist.

All sensors, valves, and zones are identified across layers using `String` identifiers. Hence these must match up and be hooked into whatever mechanism is used to associate virtual devices with real or simulated hardware.

2.4.3 Device Interface Layer Design Rationale

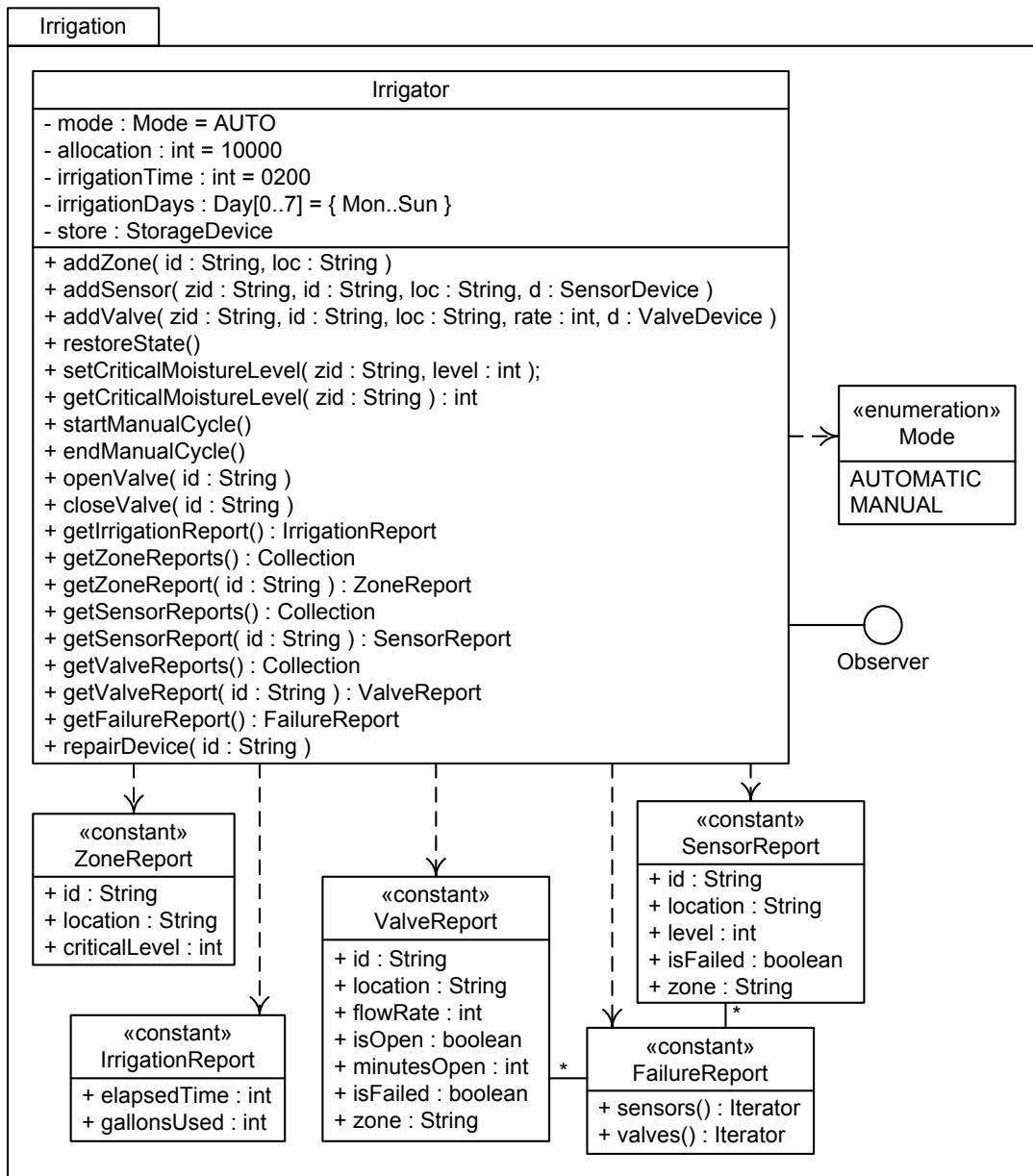
Other versions of this layer did not include the `DeviceFactory`. The `DeviceFactory` is part of the Abstract Factory pattern, which allows the details of virtual device configuration to be hidden inside concrete factories and makes it very easy to change the configuration. Design alternatives that include the Abstract Factory pattern are clearly superior.

A question that was revisited several times was how the clock should be handled. Does there need to be a `Clock` distinct from a `ClockDevice`? On the one hand, the `ClockDevice` is supposed to provide a virtual device as an interface to real or simulated hardware, while a `Clock` provides time and notification services to clients. These are distinct responsibilities, so it appears that there ought to be two entities. On the other hand, in practice the services provided by a `Clock` are almost identical to those of a `ClockDevice`, and it is not good to multiply entities beyond necessity. In the end the alternative with a distinct `Clock` and `ClockDevice` was chosen to maintain uniformity with the rest of the modules in the Device Interface layer.

Having resolved the issue of whether the `Clock` is distinct from the `ClockDevice`, the question of where the `Clock` should reside arises. On the one hand, the `Clock` is responsible for providing time and notification services rather than a uniform interface to hardware, suggesting that it should not be in the Device Interface layer. On the other hand, a `Clock` is a kind of device and it does not fit particularly well in any other layer, so perhaps the best home for it is the Device Interface layer. The latter argument seems slightly more persuasive, so the `Clock` is in the Device Interface layer. It could be moved easily, however.

2.5 Irrigation Layer Decomposition

The Irrigation layer is the core of AquaLush in the sense that it realizes the program's main function of controlling irrigation. Its architectural structure is quite simple: It contains a façade module that provides operations for configuration, setting irrigation parameters, and controlling manual irrigation. These operations depend on an enumeration type and several classes that merely package data about the state of the program for return from query functions. The class diagram in Figure B-10-12 illustrates this structure.

**Figure B-10-12 Irrigation Layer Structure**

The classes with the «constant» stereotype are immutable data containers that simply hold values returned by query operations.

2.5.1 Irrigation Layer Module Responsibilities

Module	Responsibilities
Irrigator	Hold irrigation parameters, oversee irrigation cycles, and provide a façade to the rest of the irrigation-layer facilities.
Mode	Enumeration values for program modes.
ZoneReport	Record providing data about irrigation zones.
ValveReport	Record providing data about the state of a valve.
SensorReport	Record providing data about the state of a sensor.
IrrigationReport	Record providing data about the state of an irrigation cycle.
FailureReport	Record containing ValveReport and SensorReport instances for failed valves and sensors.

Table B-10-13 Irrigation Layer Module Responsibilities

2.5.2 Irrigation Layer Interface Specifications

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Create irrigation objects	Syntax:	<code>create(s : StorageDevice)</code>
	Pre:	<code>s</code> is not null.
	Post:	Irrigation-layer objects are created and initialized.
Set and get irrigation parameters	Syntax:	<code>setMode(m : Mode)</code> <code>getMode() : Mode</code> <code>setAllocation(a : int)</code> <code>getAllocation() : int</code> <code>setIrrigationTime(t : int)</code> <code>getIrrigationTime() : t</code> <code>setIrrigationDays(s : Set)</code> <code>getIrrigationDays() : Set</code>
	Pre:	$0 < a$ and t is a valid military time specification and s is a set of Day.
	Post:	The irrigation parameters are set or fetched. Changes are recorded in the persistent store.
Add a zone	Syntax:	<code>addZone(id : String, loc : String)</code>
	Pre:	<code>id</code> and <code>loc</code> are not null; <code>id</code> is unique.
	Post:	The layer is configured with a new zone.
Add a sensor	Syntax:	<code>addSensor(zid: String, id : String, loc : String, d : SensorDevice)</code>
	Pre:	No parameter is null and the designated zone does not have a sensor.
	Post:	Adds a sensor to the designated zone. Throws an <code>IllegalStateException</code> if the zone already has a sensor.

Add a valve	Syntax:	addValve(zid: String, id : String, loc : String, rate : int, d : ValveDevice)
	Pre:	No parameter is null and 0 < rate and the designated zone does not have 32 valves.
	Post:	Adds a sensor to the designated zone. Throws an IllegalStateException if the zone already has 32 valves.
Restore program state	Syntax:	restoreState()
	Pre:	Layer configuration is complete.
	Post:	Persistent store is read and the previous state of the program is restored.
Set or get a zone's critical moisture level	Syntax:	setCriticalMoistureLevel(zid : String, level : int) getCriticalMoistureLevel(zid : String) : int
	Pre:	zid is not null and names a registered zone and 0 <= level <= 100.
	Post:	The designated zone's critical moisture level is set or returned and 0 <= result <= 100. The change is recorded in persistent store.
Start and end manual irrigation cycles	Syntax:	startManualCycle() endManualCycle()
	Pre:	mode is MANUAL
	Post:	A manual irrigation cycle is started or ended and waterUsed is set to zero when a cycle starts. Throws an IllegalStateException if mode is not MANUAL.
Manually open or close a valve	Syntax:	openValve(id : String) closeValve(id : String)
	Pre:	id is not null and id names a registered valve and mode is MANUAL.
	Post:	A valve is opened or closed. Attempts to open or close failed valves do nothing. Throws an IllegalStateException if mode is not MANUAL.
Obtain irrigation cycle data	Syntax:	getIrrigationReport() : IrrigationReport
	Pre:	None.
	Post:	The IrrigationReport is populated and returned.
Get zone data	Syntax:	getZoneReports() : Collection getZoneReport(id : String) : ZoneReport
	Pre:	id is not null and names a registered zone.
	Post:	If no zone is identified, a collection of ZoneReports, one for each registered zone, is populated and returned. If a zone is identified, a report for that zone is populated and returned.
Get sensor data	Syntax:	getSensorReports() : Collection getSensorReport(id : String) : SensorReport
	Pre:	id is not null and names a registered sensor.
	Post:	If no sensor is identified, a collection of SensorReports, one for each registered sensor, is populated and returned. If a sensor is identified, a report for that sensor is populated and returned.
Get valve data	Syntax:	getValveReports() : Collection getValveReport(id : String) : ValveReport
	Pre:	id is not null and names a registered valve.
	Post:	If no valve is identified, a collection of ValveReports, one for each registered valve, is populated and returned. If a valve is identified, a report for that valve is populated and returned.

Get a failed hardware report	Syntax:	getFailureReport() : FailureReport
	Pre:	None.
	Post:	A new FailureReport is created and populated. It will contain ValveReports and SensorReports only for failed devices.
Mark a device as repaired	Syntax:	repairDevice(id : String)
	Pre:	id is not null and names a registered valve or sensor.
	Post:	The indicated valve or sensor is marked as repaired. The change is recorded in persistent store.

Table B-10-14 Irrigation Layer Interface Specifications**Services Required**

The Irrigation layer uses the following operations from the Device Interface layer: ValveDevice.open(), ValveDevice.close(), SensorDevice.read(), StorageDevice.setData(), StorageDevice.getData(), Clock.getTime(), Clock.setTime(), Clock.getDay(), Clock.setDay(), Clock.addObserver(). The Irrigator is the Clock observer.

Usage Guide

The Irrigation layer should be completely configured using the `addX()` operations before the `restoreState()` operation is called. The `restoreState()` operation uses the Irrigation layer configuration to obtain system state values such as device failure states; if the configuration is not complete, these state values will not be set correctly.

The Irrigator registers itself with the Clock, so a client should not do this.

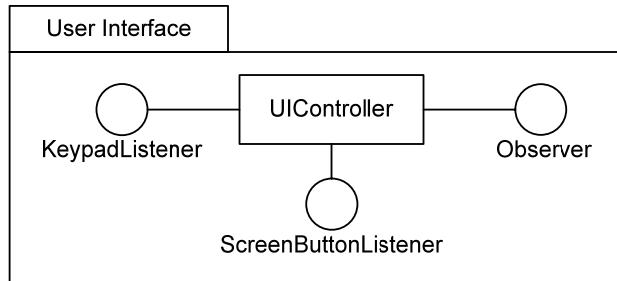
The various report classes are merely containers for groups of values. They are intended to be used by the User Interface layer to populate screens with data about the state of the system, especially during manual irrigation. They are supposed to provide a wide range of information so they will not have to be changed even if the user interface is changed. Note that because the system is changing during irrigation, the values in the report become stale, so they need be retrieved anew every minute or after every change.

2.5.3 Irrigation Layer Design Rationale

There are many design alternatives that expose the internals of the Irrigation layer by making the Zone, Sensor, and Valve classes part of the layer interface. This makes the Irrigator class much simpler and obviates the need for the many report classes used to return data about the configuration, the state of irrigation, and device states. Such alternatives have two problems: By exposing the internal structure of this layer they make it much harder to change in the future because they do not hide information, and they make it much harder to enforce constraints on irrigation behavior. To illustrate this last point, the alternative previously documented does not allow valves to be manually opened or closed unless the mode is *automatic*. Allowing direct access to valves would make it harder to enforce this constraint. It seems clear that the previous alternative is the best choice because it favors information hiding and control over simplicity.

2.6 User Interface Layer Decomposition

The User Interface layer is responsible for coordinating user interaction with output on a monochrome textual display and input from a 12-key keypad and 8 screen buttons. This layer has a UIController that executes a state machine derived from the user interface dialog map. The controller dispatches input from the keypad, screen buttons, and the clock to the currently active screen.

**Figure B-10-15 User Interface Layer Structure****2.6.1 User Interface Layer Module Responsibilities**

Module	Responsibilities
UIController	Execute a state machine whose states are screens from the dialog map. Dispatch user input and clock notifications to the current screen.

Table B-10-16 User Interface Layer Module Responsibilities**2.6.2 User Interface Layer Interface Specifications**

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Create the user interface	<i>Syntax:</i>	<code>create(d : DisplayDevice, w : Irrigator)</code>
	<i>Pre:</i>	<code>d</code> and <code>w</code> are not null.
	<i>Post:</i>	The user interface controller is created and the initial screen in the program is activated.

Table B-10-17 User Interface Layer Interface Specifications

Services Required

The **User Interface** layer uses almost all the operations from the **Irrigation** layer. From the **Device Interface** layer, it uses all the services provided by the **DisplayDevice** and the **Clock**, which it observes. This layer also uses the **KeyPress** enumeration from the **Device Interface** layer.

Usage Guide

The **UIController** must be registered as the listener for the **KeypadDevice** and the **ScreenButtonDevice** to receive notifications from them. It registers itself with the **Clock**.

2.6.3 User Interface Layer Design Rationale

Although the intent is for this layer to implement a state machine realizing the user interface dialog map, the architecture description is so general that any sort of implementation is possible: The **UIController** can be treated as a façade hiding all sorts of alternative implementations. This is perhaps the best reason to prefer this architectural alternative. Other alternatives would provide more detail that would constrain the detailed design alternatives.

2.7 Runtime Components

Figure B-10-18 depicts the runtime configuration of AquaLush.

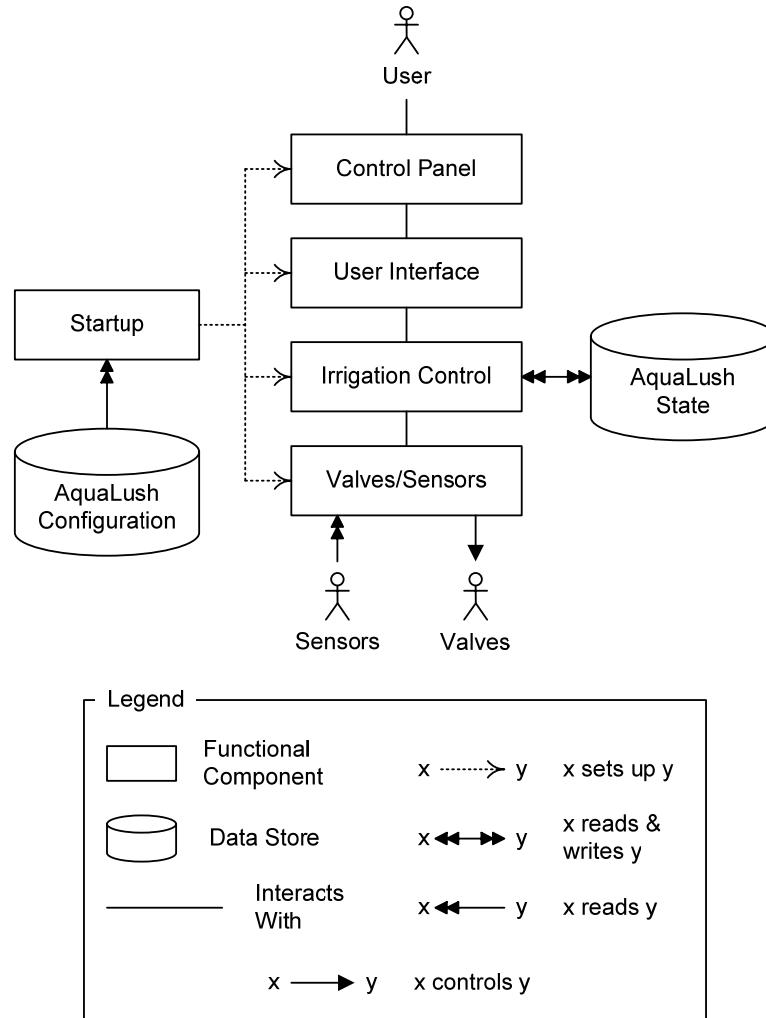


Figure B-10-18 AquaLush Runtime Structure

The **Startup** component executes first and configures the remaining parts of the program using the data it reads from the **AquaLush Configuration** data store. The **User** interacts with the **AquaLush Control Panel**, which interacts with the **User Interface**. The **User Interface** translates the **User's** desires into interactions with the **Irrigation Control** component to set the mode of the program, set program parameters, and control manual irrigation. The **Irrigation Control** component maintains a copy of its state in the **AquaLush State** data store and uses it to recover its state when it starts up after loss of power. The **Irrigation Control** component interacts with the **Valves/Sensors** component, which reads hardware (or simulated) **Sensors** and controls hardware (or simulated) **Valves**.

3. Mapping Between Models

Table B-10-19 indicates the uses relations between layers and their constituents. The layers in the rows use the classes in the columns.

	User Interface	Irrigation	Device Interface	Simulation
Startup	UIController	Irrigator	DeviceFactory KeypadDevice ScreenButtonDevice	Simulation
User Interface		Irrigator	Clock DisplayDevice	
Irrigation			Clock ValveDevice SensorDevice StorageDevice	
Device Interface				SimTime SimDisplay SimValve SimSensor SimStore

Table B-10-19 Use Relations in AquaLush

The runtime components shown in Figure B-10-18 are comprised of the modules and data stores shown in Table B-10-20.

Runtime Component	Modules
Startup	Startup layer configuration module(s), plus DeviceInterface.DeviceFactory
AquaLush Configuration	Configuration specification (file or applet parameters)
Control Panel	DeviceInterface.DisplayDevice, DeviceInterface.KeypadDevice, and DeviceInterface.ScreenButtonDevice; display, keypad, and screen button hardware or Simulation.SimDisplay and a simulated keypad
User Interface	UserInterface.UIController
Irrigation Control	The objects in the Irrigation layer, plus DeviceInterface.StorageDevice, DeviceInterface.Clock, DeviceInterface.ClockDevice, Simulation.SimStore, or a configuration file, and Simulation.SimTime or real clock hardware
Valves/Sensors	DeviceInterface.ValveDevice, DeviceInterface.SensorDevice
Valves and Sensors	Simulation.SimValve, Simulation.SimSensor, or actual hardware sensors and valves

Table B-10-20 AquaLush Modules and Data Stores

4. Architectural Design Rationale

The layers in this program are rather unusual, but they are arranged as they are to satisfy one of the most important AquaLush quality attributes: configurability. Because the core software, principally the contents of the Irrigation layer, must run unchanged in a simulation or in a fielded product, the program must be configurable so that *all* interactions with its environment can be changed. This includes things that are usually not considered part of the environment, such as time (because simulated time may be faster, slower, or even stop), and things that are usually taken for granted (such as the persistent store, which may not be accessible in an applet).

There is no reasonable alternative except to have a **Device Interface** layer. It uses the **Simulation** layer to provide the entire external environment during a simulation and hardware devices, a file system, and the system clock in a fielded program. In either case, the **Device Interface** layer shields the rest of the program from having to deal with the differences occasioned by these radically different environments.

User interfaces are usually coded using the user interface toolkit provided by a programming environment. However, since the AquaLush simulation uses a GUI and the fielded program uses ATM-like hardware, the user interface must either be coded in several versions or coded once in a device-independent component that uses virtual devices. The latter alternative is clearly preferable. Furthermore, it makes sense to separate the **User Interface** layer from the **Irrigation** layer so that the latter can be unchanged even if the user interface is altered.

Finally, the complexity of the configuration problem suggests the need for a layer responsible for configuring the program at startup. Isolating this task in one place makes it easier to code, test, and modify. The **Startup** layer is responsible for this task.

B.11 AquaLush Detailed Design Document

1. Mid-Level Design Models

The mid-level design models elaborate each architectural layer and are organized in this section by layer.

1.1 Startup Layer Static Structure

The Startup layer is responsible for initializing the program as part of a fielded product or as a Web simulation, configuring the program according to the hardware available to it, and restoring the previous program state. The Startup layer has the static structure shown in Figure B-11-1.

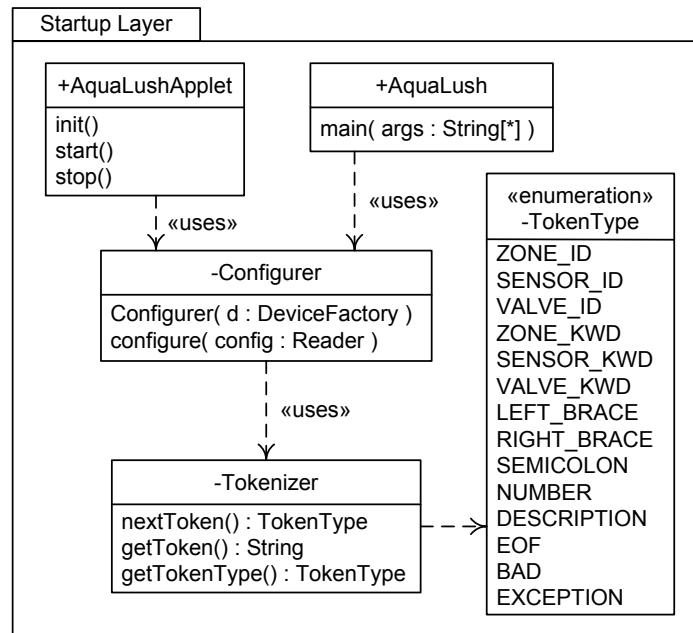


Figure B-11-1 Startup Layer Mid-Level Static Structure

This diagram adds only a few details to the architectural design structure—principally, the `Tokenizer` and `TokenType` classes and a few operations in the `Configurer` and `AquaLushApplet` classes. These additions are documented in Table B-11-2.

1.1.1 Startup Layer Local Module Responsibilities

Module	Responsibilities
Tokenizer	Process configuration reader input character by character to produce a stream of tokens that the Configurer can parse to interpret the configuration specification.
TokenType	Provide a type-safe token type enumeration for communication between the Tokenizer and the Configurer.

Table B-11-2 Startup Layer Local Module Responsibilities

1.1.2 Startup Layer Local Module Interface Specifications

Services Provided

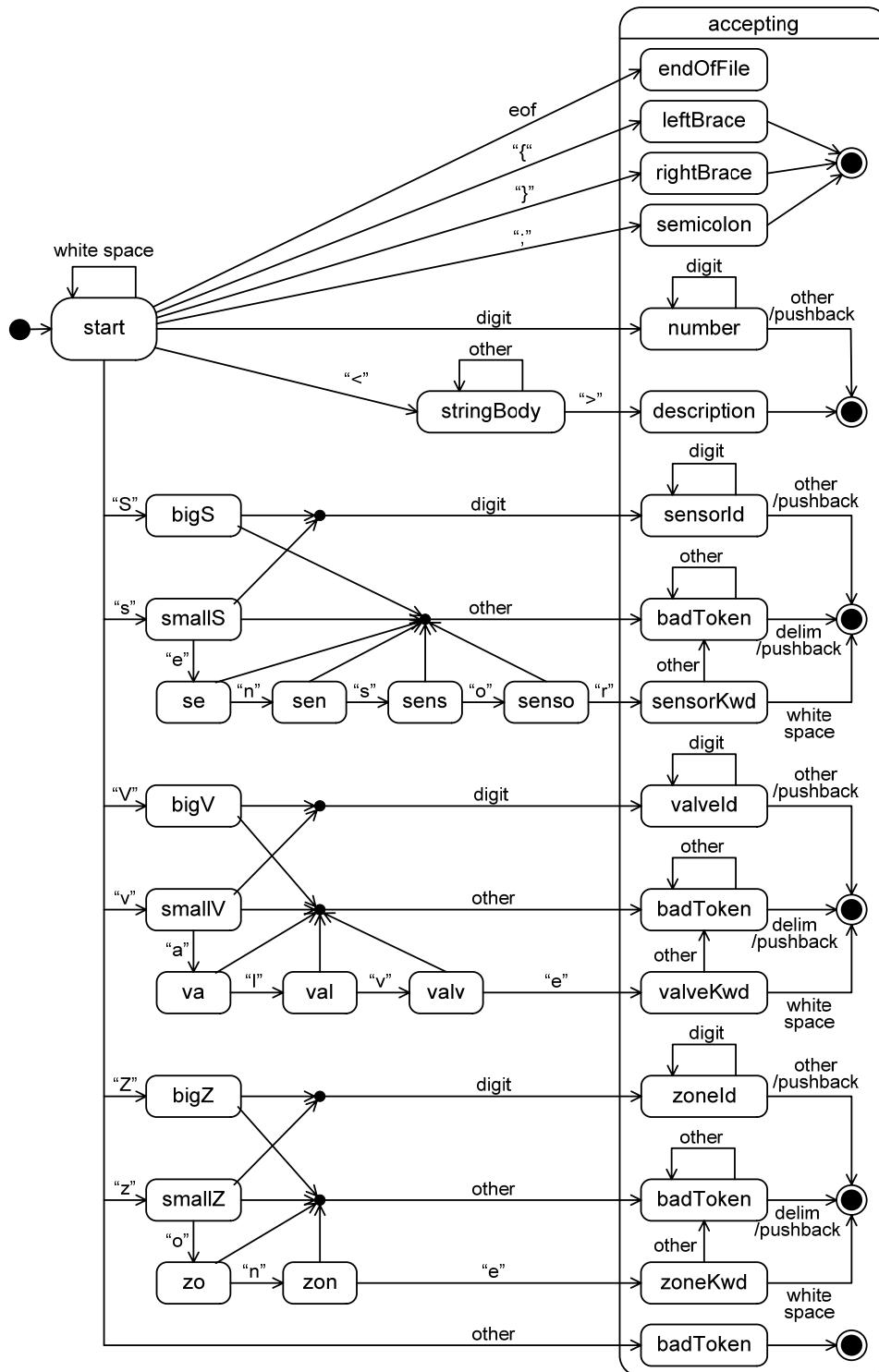
All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Applet start	<i>Syntax:</i>	start()
	<i>Pre:</i>	None.
	<i>Post:</i>	The AquaLush simulated clock is started.
Applet stop	<i>Syntax:</i>	stop()
	<i>Pre:</i>	None.
	<i>Post:</i>	The AquaLush simulated clock is stopped.
Configurer construction	<i>Syntax:</i>	Configure(d : DeviceFactory)
	<i>Pre:</i>	d is not null.
	<i>Post:</i>	The Configurer is ready to configure the program.
Get next token	<i>Syntax:</i>	nextToken() : TokenType
	<i>Pre:</i>	None.
	<i>Post:</i>	Gets the next token from the configuration input stream and returns its type.
Get the text of the current token	<i>Syntax:</i>	getToken() : String
	<i>Pre:</i>	nextToken() has been called at least once.
	<i>Post:</i>	The text of the current token is returned. Returns null if the precondition is violated.
Get the type of the current token	<i>Syntax:</i>	getTokenType() : TokenType
	<i>Pre:</i>	nextToken() has been called at least once.
	<i>Post:</i>	The type of the current token is returned. Returns TokenType.UNKNOWN if the precondition is violated.

Table B-11-3 Startup Layer Local Module Interface Specifications

1.2 Startup Layer Behavior

The `Tokenizer` can execute a state machine to recognize configuration file tokens. The diagram in Figure B-11-4 models the tokenizing state machine.

**Figure B-11-4 Tokenizer State Machine**

In this diagram all events are either the end of the input file, and therefore treated as a character, or single-character inputs. Labels with a single character designate that character; other labels mean the following:

eof—The end of the input file.

white space—A blank, tab, newline, or carriage return character.

digit—The characters “0” through “9.”

other—Any character not labeling another arrow emanating from the same state.

The **pushback** action means that the character last consumed is placed back on the input stream.

1.3 Simulation Layer Static Structure

The **Simulation** layer includes components that simulate the environment of the program, including system hardware. The **Simulation** layer contains almost all the user interface code implementing the GUI for the simulation, written using Java Swing components. Most of these entities are not visible at the architectural level. The mid-level design also adds devices that are not explicitly present in the architectural view. Only the details added to the architectural specification are documented in Figure B-11-5, which depicts the detailed structure of the **Simulation** layer.

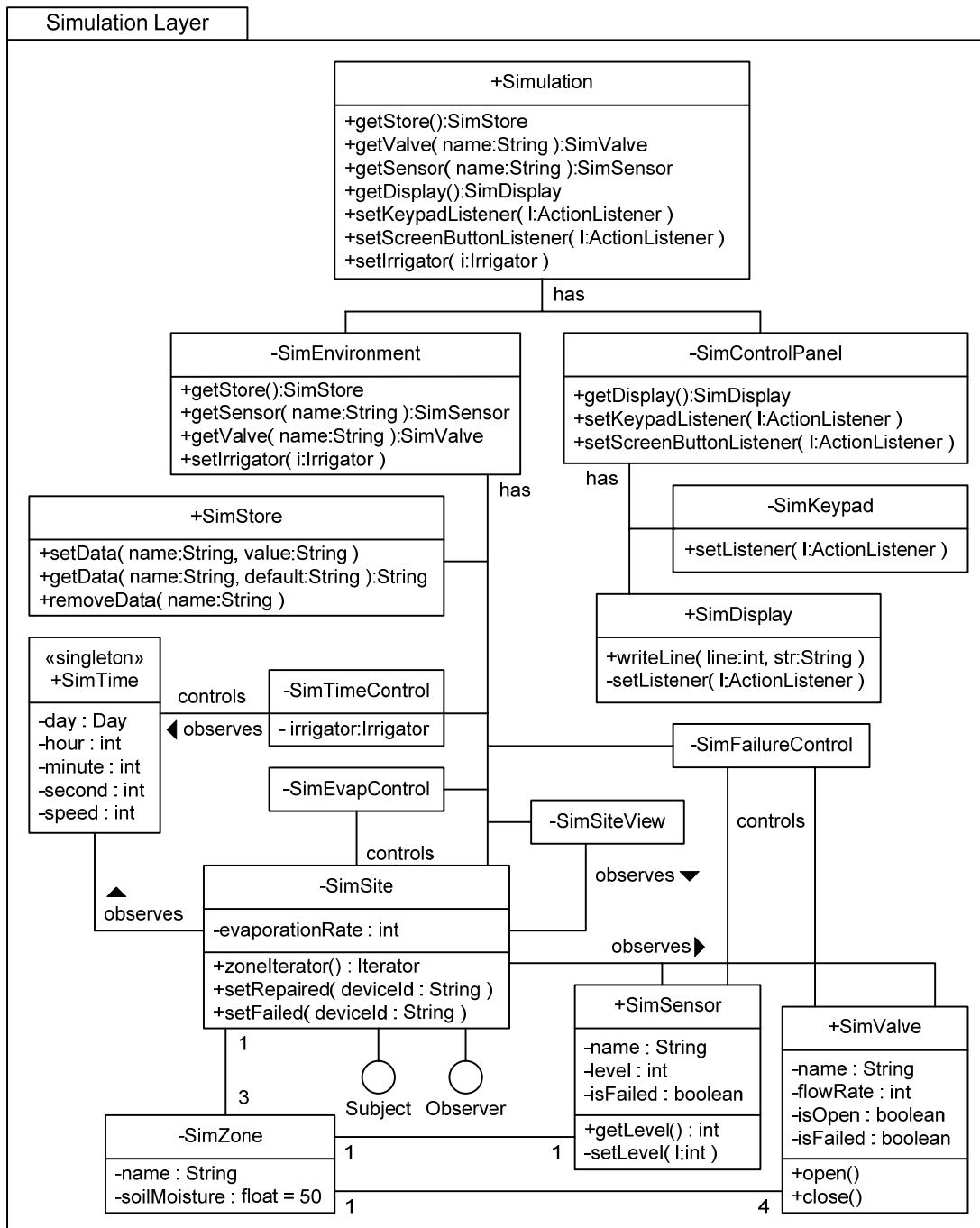


Figure B-11-5 Simulation Layer Mid-Level Static Structure

1.3.1 Simulation Layer Local Module Responsibilities

Module	Responsibilities
SimEnvironment	A Swing panel containing all the displays and controls for the simulated environment, including display of the irrigation site and controls for the simulated time, evaporation rate, and hardware failures and repairs.
SimControlPanel	A Swing panel containing all the widgets for the AquaLush control panel, including the simulated display, screen buttons, and keypad.
SimKeypad	A Swing panel simulating a keypad with 12 buttons.
SimTimeControl	A Swing panel displaying and controlling the simulated time.
SimEvapControl	A Swing panel displaying and controlling the simulated water evaporation rate.
SimSiteView	A Swing panel displaying the irrigation site.
SimFailureControl	A Swing panel displaying and controlling the failure status (failed or running) of the simulated valves and sensors.
SimSite	A collection keeping track of the simulated site evaporation rate and irrigation zones. It is responsible for adjusting the moisture levels in each SimZone.
SimZone	A collection holding a SimSensor and the four SimValves in a portion of the simulated irrigation site.

Table B-11-6 Simulation Layer Local Module Responsibilities

1.3.2 Simulation Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Provide simulated persistent store (SimEnvironment)	Syntax: <code>getStore() : SimStore</code>
	Pre: None.
	Post: The simulated storage object is returned.
Provide a simulated valve (SimEnvironment, SimSite, SimZone)	Syntax: <code>SimEnvironment.getValve(name : String) : SimValve</code>
	Pre: name is not null.
	Post: The simulated valve whose identifier is named is returned, or null is returned if no such valve exists.
Provide a simulated sensor (SimEnvironment, SimSite, SimZone)	Syntax: <code>getSensor(name : String) : SimSensor</code>
	Pre: name is not null.
	Post: The simulated sensor whose identifier is named is returned, or null is returned if no such sensor exists.
Provide the simulated display (SimControlPanel)	Syntax: <code>getDisplay() : SimDisplay</code>
	Pre: None.
	Post: The simulated display object is returned.
Register a Keypad listener (SimControlPanel)	Syntax: <code>setKeypadListener(I : KeypadListener)</code>
	Pre: None..
	Post: KeypadListener I will start to receive notifications of simulated keypad key presses.

Register a Keypad listener (SimKeypad)	Syntax:	setListener(I : KeypadListener)
	Pre:	None.
	Post:	KeypadListener I will start to receive notifications of simulated keypad key presses.
Register a ScreenButton listener (SimControlPanel)	Syntax:	setScreenButtonListener(I : ScreenButtonListener)
	Pre:	None.
	Post:	ScreenButtonListener I will start to receive notifications of simulated screen button presses.
Register a ScreenButton listener (SimDisplay)	Syntax:	setListener(I : ScreenButtonListener)
	Pre:	None.
	Post:	ScreenButtonListener I will start to receive notifications of simulated screen button presses.
Set a sensor's moisture level (SimSensor)	Syntax:	setLevel(I : int)
	Pre:	$0 \leq I \leq 100$.
	Post:	The SimSensor's moisture level is changed.
Set the failure status of a sensor (SimSensor)	Syntax:	setIsFailed(value : Boolean)
	Pre:	None.
	Post:	The SimSensor's failure status is set as indicated by value.
Set the failure status of a sensor (SimValve)	Syntax:	setIsFailed(value : Boolean)
	Pre:	None.
	Post:	The SimValve's failure status is set as indicated by value.

Table B-11-7 Simulation Layer Local Module Interface Specifications**Services Required**

The SimStore class will use Java services for persistent storage provided by the `java.util.Properties` class. Properties will be stored in a file called “AquaLushState.xml.”

1.3.4 Implementation Notes

The SimSite is supposed to represent reality, so the **SimZones** and their **SimSensors** and **SimValves** are set up explicitly with code in the various class constructors.

1.4 Simulation Layer Behavior

The SimSite observes **SimTime** using the Observer pattern. When a minute passes, the **SimSite** passes its **evaporationRate** attribute (which is in percent per hour) to each **SimZone**. The **SimZones** determine how to change the moisture level of each **SimSensor**. The **SimSite** notifies its observer (the **SimSiteView**) when the sensors or valves change. As an optimization, the **SimSiteView** is not updated on the minute because the **SimZones** may be in the process of changing. The sequence diagram in Figure B-11-8 models this interaction.

One additional feature must be noted about the **Simulation** layer’s structure and behavior. The **SimTimeControl** has a button that jumps the simulated time to one hour before the next scheduled irrigation time. This feature does not fit into the architecture because it requires the **Simulation** layer to use the **Irrigation** layer, in violation of the layering constraint. This single violation of the Layered style cannot be avoided. However, its effects can be minimized as follows:

- The **SimTimeControl** is passed a reference to the **Irrigator** object. It can interrogate the **Irrigator** when its jump button is pressed to obtain the next irrigation time and day. Thus, the **Irrigator** does not use the **Simulation** layer as required in the architecture.

- The AquaLush applet gets a reference to the Irrigator from the Configurer and passes it to the Simulation object, which passes it to SimEnvironment, which passes it to SimTimeControl. Thus, no parts of the program are involved in this violation of architectural constraints except for the applet and the Simulation layer, which causes the problem and is never part of a fielded product.

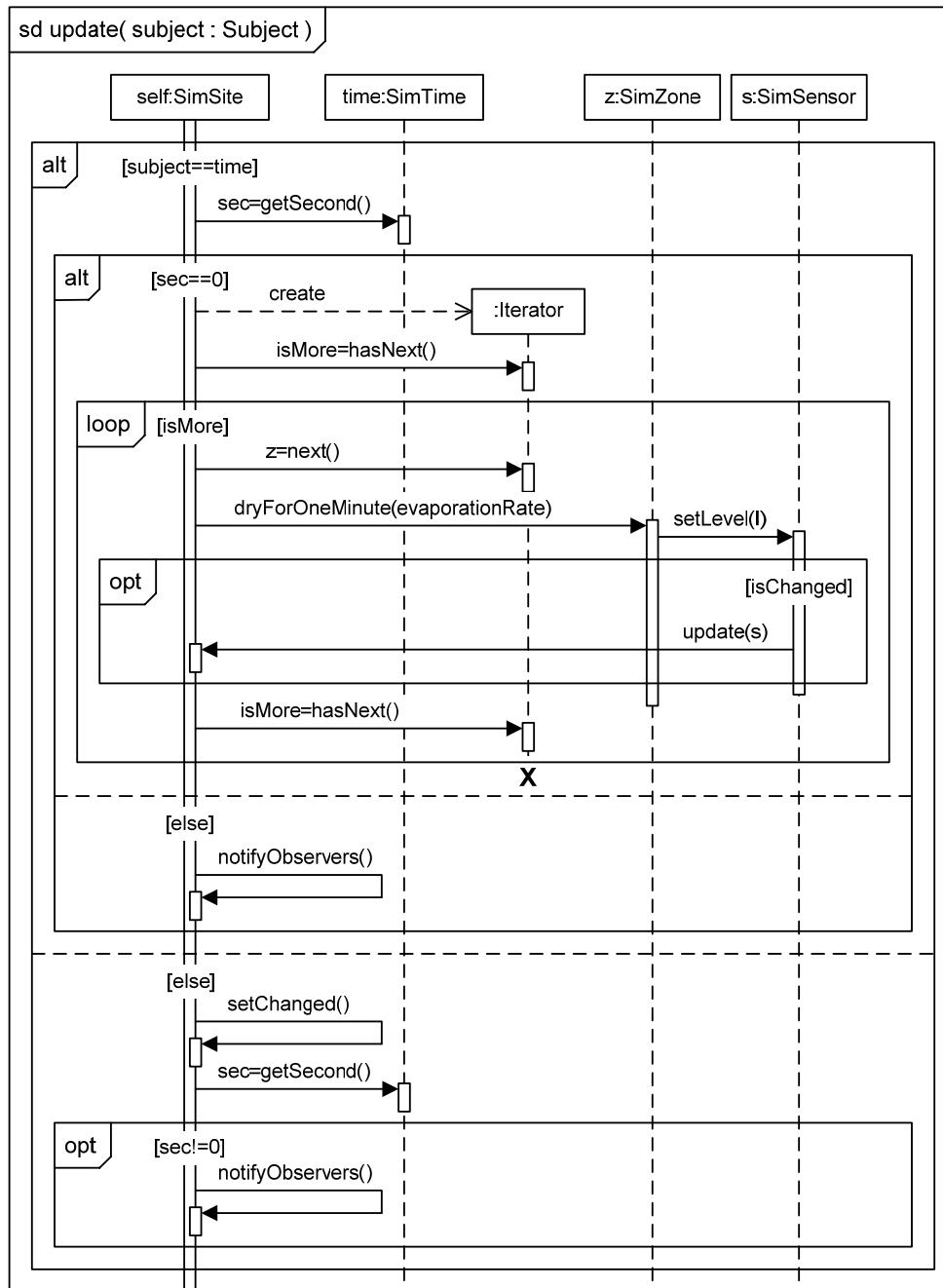


Figure B-11-8 SimSite.update() Behavior

1.5 Device Interface Layer Static Structure

The Device Interface layer provides virtual devices to hide the real or simulated devices used by the program. The mid-level design of this layer merely adds the devices conforming to the interfaces specified at the architectural level of detail. The diagram in Figure B-11-9 includes various “real” device classes. These are placeholders for one or more device drivers for actual hardware devices.

The mid-level static structure of the Device Interface layer is shown in Figure B-11-9.

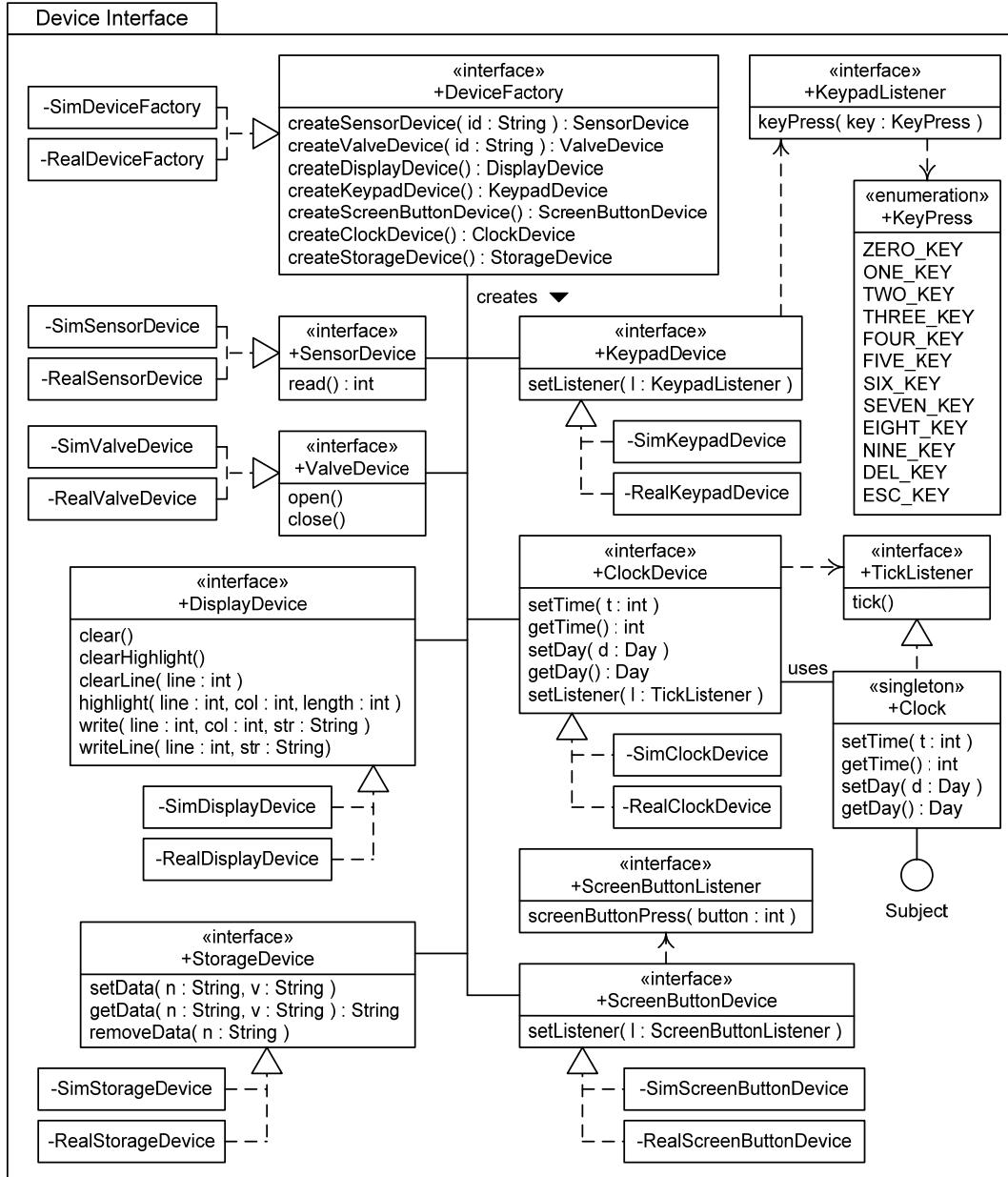


Figure B-11-9 Device Interface Layer Mid-Level Static Structure

1.6 Irrigation Layer Static Structure

The Irrigation layer is the central module of the application. It controls automatic irrigation and acts as the user's agent during manual irrigation. The architectural view of this module presents a façade for controlling manual and automatic irrigation and for retrieving reports about the state of irrigation for display to the user. The mid-level design view adds the classes and operations necessary to realize the system configuration and control irrigation. The mid-level design structure is illustrated in Figure B-11-10.

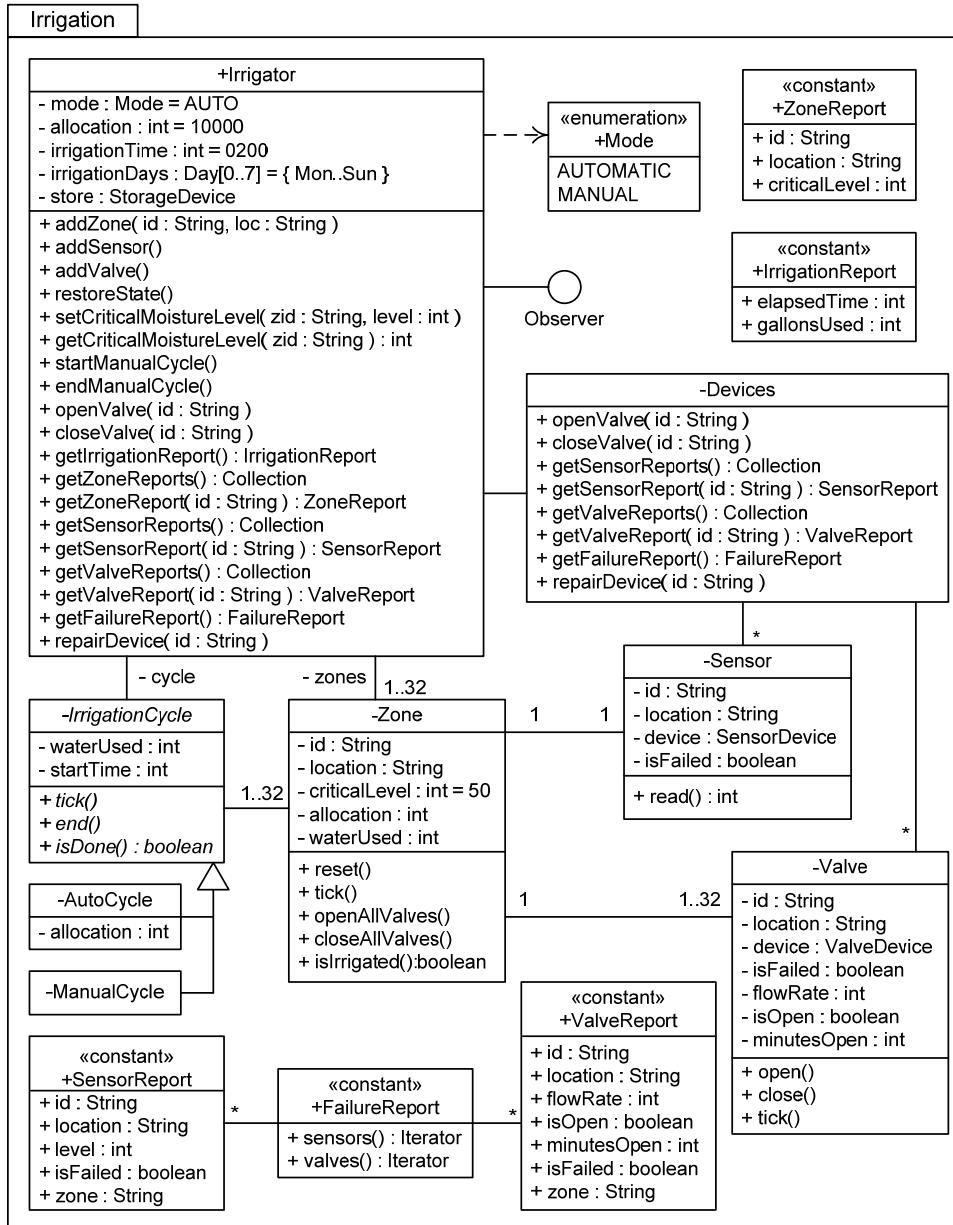


Figure B-11-10 Irrigation Layer Mid-Level Static Structure

1.6.1 Irrigation Layer Local Module Responsibilities

Module	Responsibilities
Devices	Keep track of all devices to make it easier to obtain reports about them and change their failure statuses.
IrrigationCycle	Abstract super-class for all irrigation cycles.
AutoCycle	Control an automatic irrigation cycle.
ManualCycle	Control a manual irrigation cycle.
Zone	Hold zone data and manage automatic irrigation of the zone.
Sensor	Hold sensor data and read a SensorDevice.
Valve	Hold valve data, keep track of how much water a valve uses during irrigation, and control a ValveDevice.

Table B-11-11 Irrigation Layer Local Module Responsibilities

1.6.2 Irrigation Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Create an automatic irrigation cycle (AutoCycle)	Syntax:	create(allocation:int, zones:Collection)
	Pre:	0 < allocation and no irrigation cycle is in progress.
	Post:	A new automatic irrigation cycle is created and immediately starts.
Create a manual irrigation cycle (ManualCycle)	Syntax:	create(zones : Collection)
	Pre:	No irrigation cycle is in progress.
	Post:	A new manual irrigation cycle is created.
One minute passes (IrrigationCycle, Zone, Valve)	Syntax:	tick()
	Pre:	None.
	Post:	Time dependent actions are completed: AutoCycle ticks the current zone and checks if zone irrigation is complete. ManualCycle ticks all zones. Zone ticks its valves and then updates water used. Valve (if open) updates minutes open.
See if a cycle is ended	Syntax:	isDone() : Boolean
	Pre:	None.
	Post:	Returns true if automatic irrigation is complete, false otherwise.
A cycle is ended (IrrigationCycle)	Syntax:	end()
	Pre:	None.
	Post:	All valves are closed.
Check for auto irrigation completion (Zone)	Syntax:	isIrrigated() : boolean
	Pre:	None.
	Post:	Zone returns true if auto irrigation is completed.

Open or close all valves in a zone	Syntax:	closeAllValves() openAllValves()
	Pre:	None.
	Post:	All the valves in the zone are opened or closed.
Manually open or close a valve (Irrigator)	Syntax:	openValve(id : String) closeValve(id : String)
	Pre:	id is not null and names a registered valve.
	Post:	A valve is opened or closed. Attempts to open or close failed valves do nothing.
Get sensor data (Irrigator)	Syntax:	getSensorReports() : Collection getSensorReport(id : String) : SensorReport
	Pre:	id is not null and names a registered sensor.
	Post:	If no sensor is identified, a collection of SensorReports, one for each registered sensor, is populated and returned. If a sensor is identified, a report for that sensor is populated and returned.
Get valve data (Irrigator)	Syntax:	getValveReports() : Collection getValveReport(id : String) : ValveReport
	Pre:	id is not null and names a registered valve.
	Post:	If no valve is identified, a collection of ValveReports, one for each registered valve, is populated and returned. If a valve is identified, a report for that valve is populated and returned.
Get a failed hardware report (Irrigator)	Syntax:	getFailureReport() : FailureReport
	Pre:	None.
	Post:	A new FailureReport is created and populated. It will contain ValveReports and SensorReports only for failed devices.
Mark a device as repaired (Irrigator)	Syntax:	repairDevice(id : String)
	Pre:	id is not null and names a registered valve or sensor.
	Post:	The indicated valve or sensor is marked as repaired. The change is recorded in persistent store.

Table B-11-12 Irrigation Layer Local Module Interface Specifications**1.6.3 Irrigation Layer Design Rationale**

The Devices collection is not strictly necessary because each Zone can handle the Devices module responsibilities. However, the Zone class already has several responsibilities, so the reporting and repair status modification responsibilities are passed off to the Devices class to simplify the Zone class.

The AutoCycle class originally had full control over automatic irrigation. For example, the AutoCycle class would query each Zone to obtain its Sensor and critical moisture level, and then read the Sensor to compare it with the critical moisture level. The AutoCycle class would query the Zone to obtain its valve set; go through the set and query each Valve about its flow rate, its allocation, and how long the valve had been open to compute its water usage; and then compare this with its allocation to decide whether the allocation was exhausted. This design made for a very bloated and complex AutoCycle class. An alternative with distributed control lets each Zone decide whether irrigation is complete and lets each Valve decide whether it has used up its allocation. This simplifies the design and was chosen as the better design alternative.

1.7 Irrigation Layer Behavior

The central behavior of the Irrigation layer is its response time. The **Irrigator** object is an observer of the **Clock**. The behavior of the **Irrigator.update()** operation is pictured in Figure B-11-13.

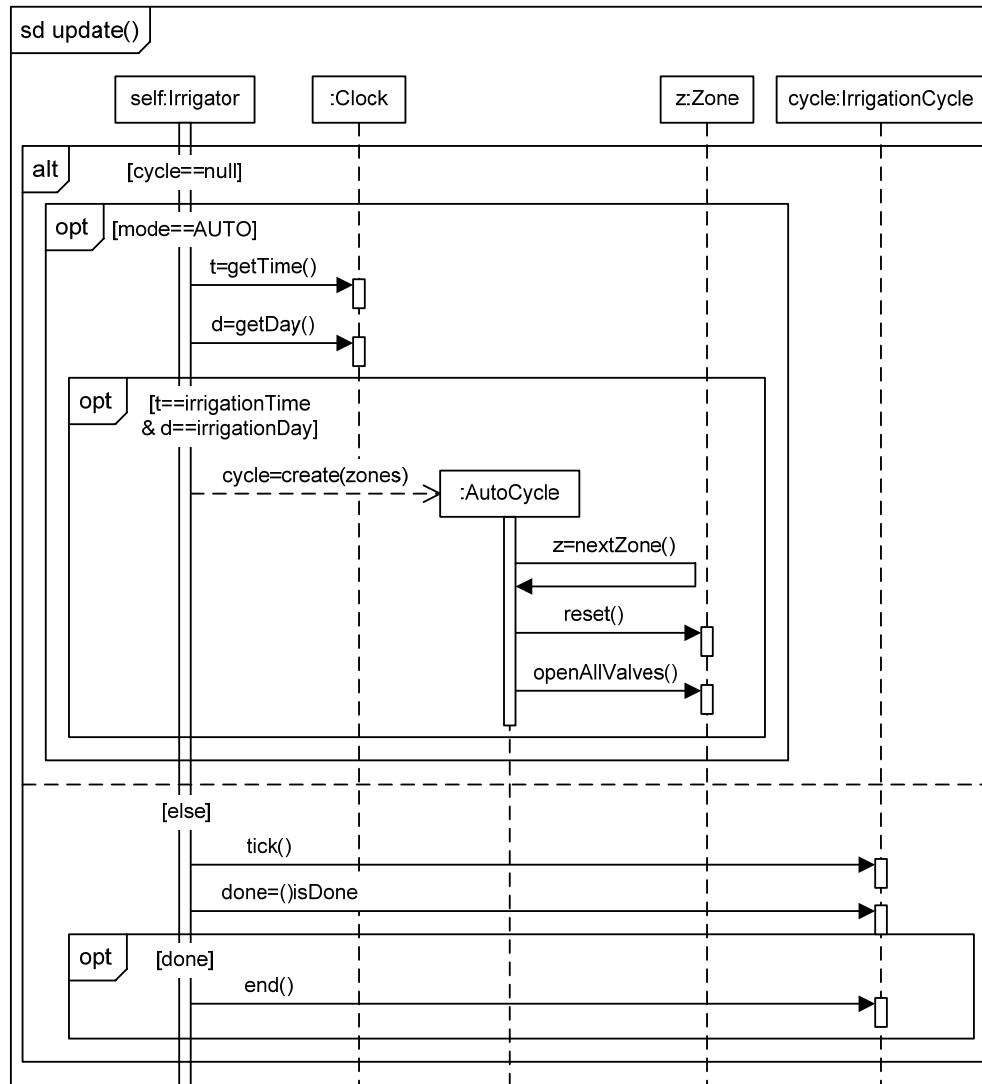
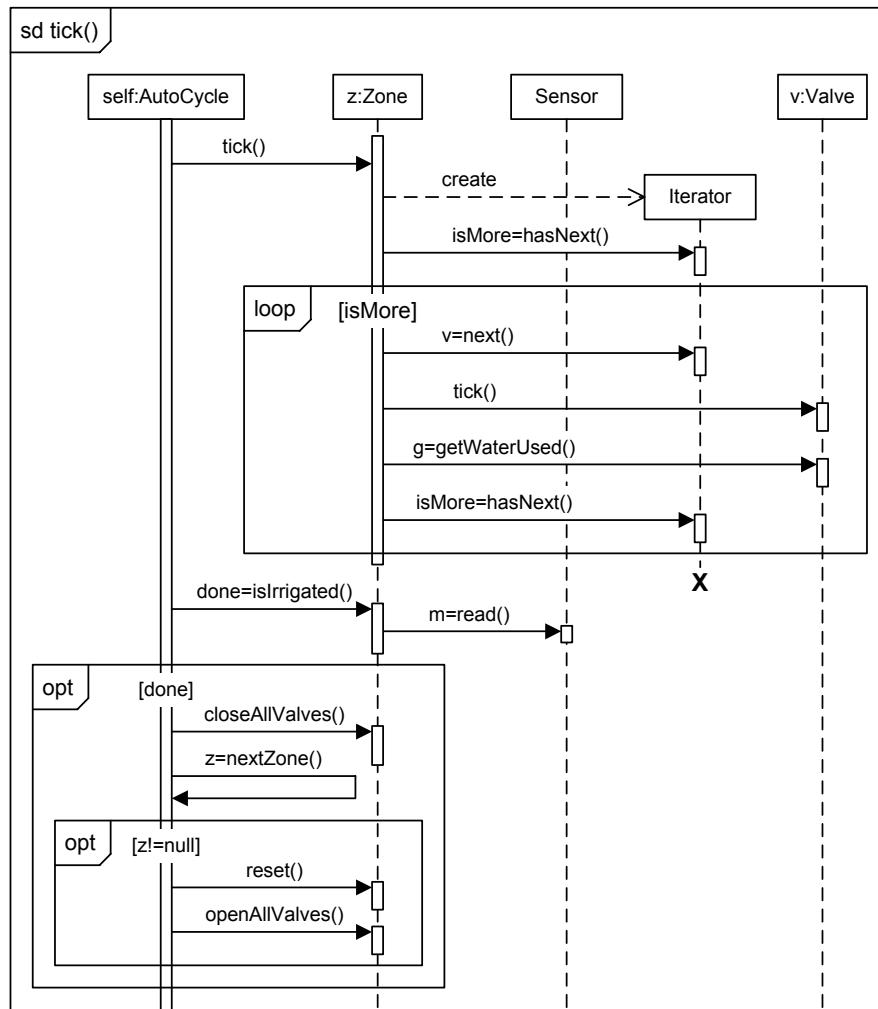


Figure B-11-13 Irrigator.update() Behavior

The **Irrigator** calls **IrrigationCycle.tick()** whenever a cycle is in process. The **AutoCycle.tick()** operation is modeled in Figure B-11-14.

**Figure B-11-14 AutoCycle.tick() Behavior**

The `ManualCycle.tick()` operation is pictured in Figure B-11-15.

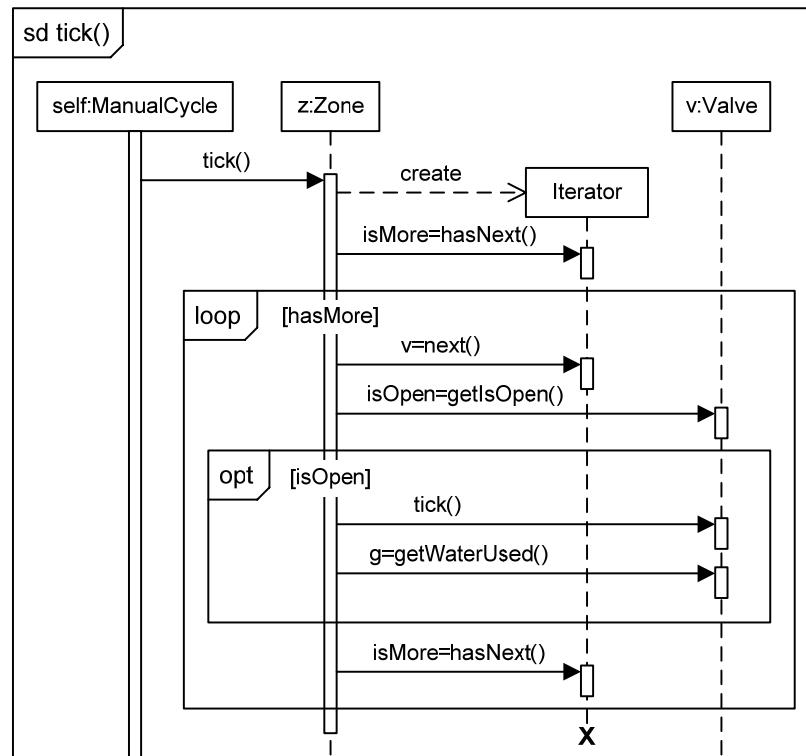
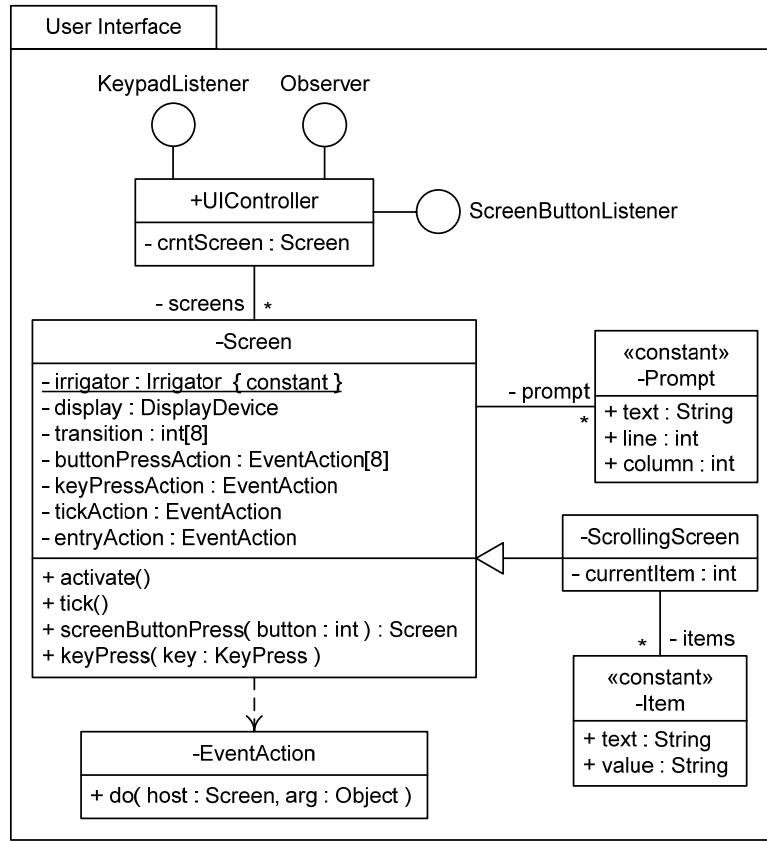


Figure B-11-15 ManualCycle.tick() Behavior

1.8 User Interface Layer Static Structure

The User Interface layer coordinates interaction with a user by consuming input provided via a 12-key keypad and 8 screen buttons and providing output on a monochrome screen of 16 lines with 40 characters per line. This layer has a main **UIController** class that executes a state machine. Each state corresponds to a screen. Transitions occur when users press screen buttons. Internal actions are prompted by screen button and keypad key presses.

The details added in mid-level design are the classes representing the screen states and auxiliary classes. The static structure of the **User Interface** layer is shown in Figure B-11-16.

**Figure B-11-16 User Interface Layer Mid-Level Static Structure**

The Screen super-class is for “plain” screens without scrollable regions. It uses the Command pattern to assign actions for screen button presses and keypad key presses.

1.8.1 User Interface Layer Local Module Responsibilities

Module	Responsibilities
Screen	Control the display and process user input and clock ticks (routed to it from UIController). Each Screen object is a state in a state machine.
ScrollingScreen	Display a scrollable list in the center of the display and handle scrolling commands.
Item	An immutable class holding text and values for the scrollable list.
Prompt	An immutable class holding text written to screens.
EventAction	Command pattern command class whose operation is called when events occur.

Table B-11-17 User Interface Layer Local Module Responsibilities

1.8.2 User Interface Layer Local Module Interface Specifications

Services Provided

All operations with preconditions on parameters throw `IllegalArgumentExceptions` if the preconditions are violated.

Activate a screen when it becomes current	Syntax:	activate()
	Pre:	None.
	Post:	The display screen is cleared and all prompts are written to it. If activationAction is not null, its do() operation is called with arg set to null.
Notify a screen that time has passed	Syntax:	tick()
	Pre:	None.
	Post:	If tickAction is not null, its do() operation is called with arg set to null.
Notify a screen that a screen button has been pressed	Syntax:	screenButtonPress(button : int) : Screen
	Pre:	None.
	Post:	If buttonPressAction[button] is not null, its do() operation is called with arg set to the button value. If transition[button] is -1, the result is the current screen; otherwise, the result is screen with number transition[button].
Notify a screen that a keypad key has been pressed	Syntax:	keyPress(key : KeyPress)
	Pre:	None.
	Post:	If keyPressAction is not null, its do() operation is called with arg set to the key value.

Table B-11-18 User Interface Layer Local Module Interface Specifications

Usage Guide

The `UIController` must create and initialize all the `Screen` and `EventAction` objects. It must then execute the state machine by keeping track of the current screen. The `UIController` calls a screen's `activate()` operation when it becomes current. The `UIController` must also direct one-minute clock ticks, screen button presses, and keypad key presses to the current screen whenever they occur. Note that the `UIController` is an observer of the `Clock`, but the screens are not. This way only the current screen gets time notifications.

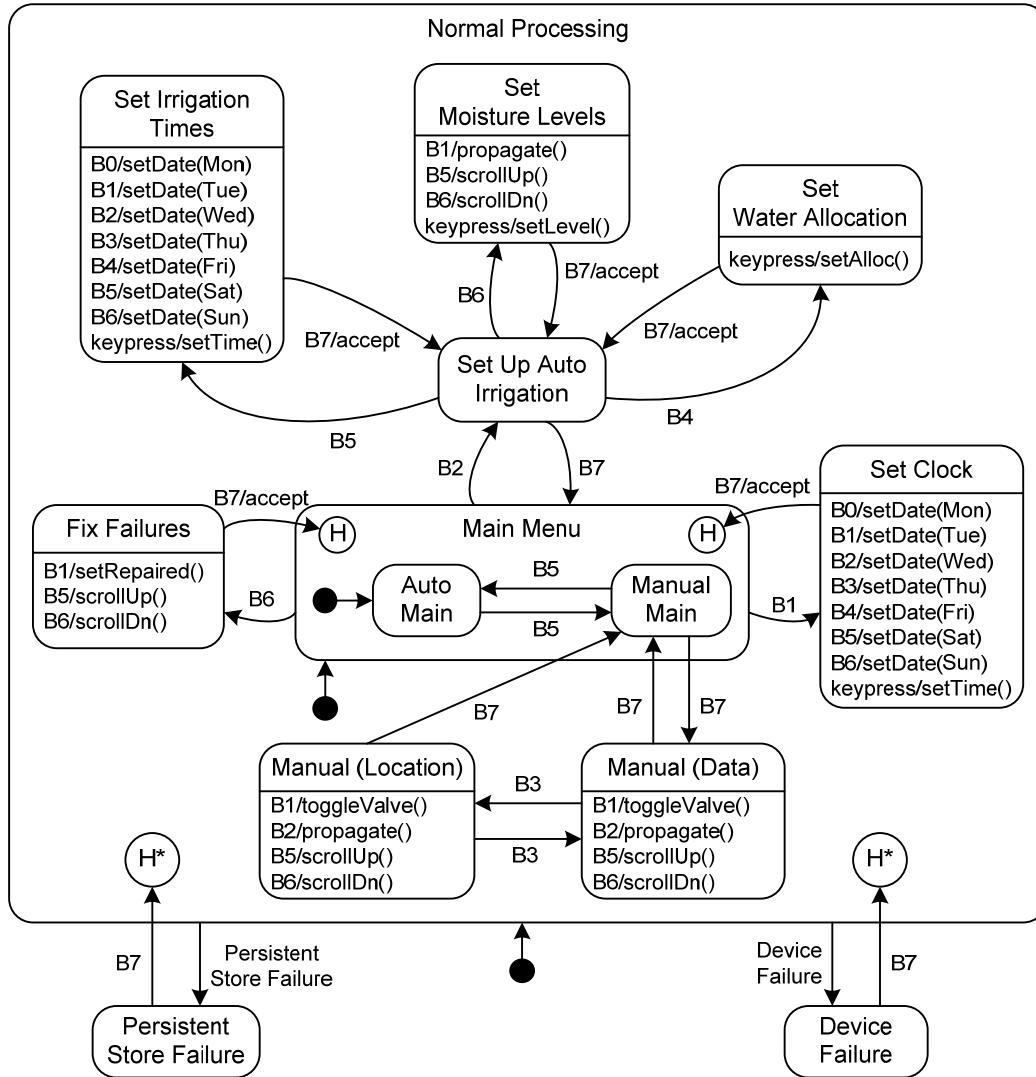
1.8.3 User Interface Layer Design Rationale

There are many alternative ways to implement the state machine that the `UIController` must realize. Among the alternatives considered were those with many specialized `Screen` sub-classes. For example, there might be a `TimeSettingScreen` class and an `AllocationSettingScreen` class. This alternative is more complicated but less flexible, so it was decided to give the `Screen` super-class a very general mechanism for handling events.

1.9 User Interface Layer Behavior

The user interface changes state based on user input. If each screen represents a state, then user interface states are captured by the dialog map in the SRS. The state diagram in Figure B-11-19 brings the dialog maps in the SRS together and adds internal actions to them.

Events B0 through B7 are presses of screen buttons 0 through 7. States in this diagram are represented by instances of the `Screen` class. The internal actions are handled either by the screen object or by command objects registered with the screen object.

**Figure B-11-19 User Interface States**

The UIController simply passes keypad presses and time notifications on to the current screen. However, a screen button press may cause a state change, meaning the interaction is more complex. Figure B-11-20 models the interaction that occurs when a screen button is pressed.

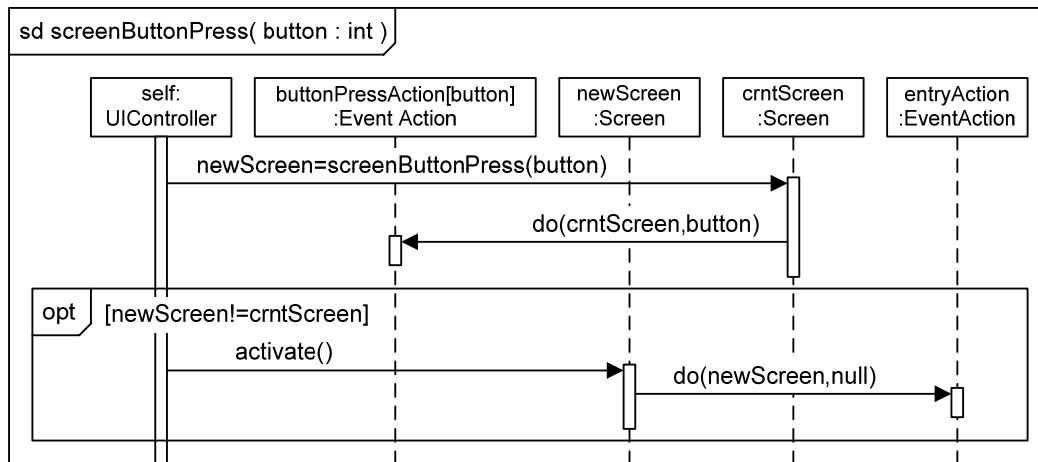


Figure B-11-20 UIController Screen Button Press Handling

2. Low-Level Design Models

There are few tricky low-level details about this program: The data structures and algorithms are all quite straightforward, and no advanced language features are required, though some can be used to make the code smaller. Consequently, only the module packaging and a few operation specifications are supplied in this section.

2.1 Packaging

Java packages are used to implement the design packages for each layer. This keeps the code structure very similar to the static design structure, making it easy to find the code that corresponds to each part of the design. Specifically, the following Java packages are used:

- startup—Startup-layer code modules.
- ui—User Interface-layer code modules.
- irrigation—Irrigation-layer code modules.
- device—Device Interface-layer code modules.
- device.sim—Code for virtual devices for simulated hardware.
- device.real—Code for virtual devices for real hardware.
- simulation—Simulation-layer code modules.
- util—Utility code modules (such as widely used exceptions and enumerations).

3. Mapping Between Design Models

The virtual devices in the Device Interface layer use simulated hardware in the Simulation layer. Table B-11-21 indicates the mapping between virtual devices and simulated devices.

Device Interface Layer (device)	Virtual Device (device.sim)	Simulated Device (simulation)
DisplayDevice	SimDisplayDevice	SimDisplay
ScreenButtonDevice	SimScreenButtonDevice	SimDisplay
KeypadDevice	SimKeypadDevice	SimKeypad
StorageDevice	SimStorageDevice	SimStore
ClockDevice	SimClockDevice	SimTime
SensorDevice	SimSensorDevice	SimSensor
ValveDevice	SimValveDevice	SimValve

Table B-11-21 Virtual Device to Simulated Device Mapping

There are two items to be noted about this mapping:

1. Both the `SimDisplayDevice` and the `SimScreenButtonDevice` virtual devices use the `SimDisplay` simulated device. It is easier to implement the GUI realizing the display and the screen buttons together as a single class, so this single simulated device provides two logically distinct services.
2. The `SimClockDevice` relies on `SimTime`, which is not a simulated piece of hardware. `SimTime` keeps track of time in the simulated world. It would have been possible to create a `SimClock` entity, but it would only have mediated the communication between `SimTime` and `SimClockDevice`, so it was not used.

4. Detailed Design Rationale

The prevailing principles driving this design are the following:

- Adhere to the access constraints imposed by the layered architecture to maintain its integrity.
- Hide information, make coherent modules, and decouple modules as much as possible to make the program as changeable as possible.
- Take advantage of the conceptual model to make design models reflect the problem domain as closely as possible.
- Design to interfaces as much as possible.
- Take advantage of standard mid-level design patterns to solve design problems and make the design easy to understand.

These principles are used to help generate, evaluate, and select design alternatives. For example, the `Irrigation` layer's design is based largely on the `AquaLush` conceptual model. The notification-driven behavior of the entire program is based on the `Observer` pattern, though it is implicit in the architecture as well.

A different set of guiding principles might have produced a different detailed design. For example, reuse is not emphasized in this design. Had it been, the `Tokenizer` class in the `Startup` layer would probably be based on the `java.util.StreamTokenizer` class rather than being written from scratch.

The only violation of architectural constraints is that the `SimTimeControl` uses the `Irrigator` to obtain the irrigation time and day. Much thought was given to ways to implement this feature without violating architectural principles, but it is impossible to do so: The `SimTimeControl` cannot work properly without data generated in higher layers of the program. Recognizing this, the next question was how to minimize the impact of the violation. One alternative considered was to make the `SimTimeControl` an observer of a higher-layer object such as a UI `Screen` or the `Irrigator` object. However, this would have required making these objects into subjects for no other reason.

It was realized that the `SimTimeControl` only needs a reference to the `Irrigator`, which already provides query functions for the data it needs, and that the `AquaLush` applet can query the `Configurer` to get an `Irrigator` object reference to pass to the `Simulation` object. This requires only a few lines of code in the applet, an extra query function in the `Configurer`, and extra functions in `Simulation` and `SimEnvironment`. This appears to be the simplest and least intrusive way to provide the data that `SimTimeControl` needs to realize its requirements.



C References

- Abbot, Russell. "Program Design by Informal English Description." *Communications of the ACM* 26 (11), November 1983, 882–894.
- Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1979.
- Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.
- Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.
- Beck, Kent. *Extreme Programming Explained*. Addison-Wesley, 2000.
- Bennett, Simon, John Skelton, and Ken Lunn. *Schaum's Outline of UML*. McGraw-Hill, 2001.
- Bloch, Joshua. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.
- Boehm, Barry. *Software Engineering Economics*. Prentice-Hall, 1981.
- Booch, Grady. *Object-Oriented Design*. Benjamin/Cummings, 1991.
- Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide, 2nd Edition*. Addison-Wesley, 2005.
- Bosch, Jan. *Design and Use of Software Architectures*. Addison-Wesley, 2000.
- Braude, Eric. *Software Design: From Programming to Architecture*. John Wiley and Sons, 2003.
- Britton, Kathy Heninger, R. Alan Parker, and David L. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules." *Proceedings of the 5th International Conference on Software Engineering*, 1981, 195–204.
- Budgen, David. *Software Design*. Addison-Wesley, 1994.
- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley and Sons, 1996.
- Chonoles, Michael Jesse, and James A. Schardt. *UML 2 For Dummies*. Wiley Publishing, Inc., 2003.
- Clark, Robert G., and Leslie B. Wilson. *Comparative Programming Languages, 3rd Edition*. Addison-Wesley, 2001.
- Clements, Paul, Felix Bachman, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures*. Addison-Wesley, 2003.
- Clements, Paul, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. Addison-Wesley, 2002.
- Coad, Peter. "Object-Oriented Patterns." *Communications of the ACM* 35 (9), September 1992, 152–159.
- Coad, Peter, and Edward Yourdon. *OOA—Object-Oriented Analysis*. Prentice-Hall, 1991.
- Cockburn, Alistair. *Writing Effective Use Cases*. Addison-Wesley, 2001.
- Coleman, Derek, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayese, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.
- Cooper, Alan, and Robert Reimann. *About Face 2.0: Essentials of Interaction Design*. Wiley Publishing, Inc., 2003.
- Coplien, James. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
- Coplien, James, and Douglas C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

- Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*, 2nd Edition. McGraw-Hill, 2001.
- Daniels, John. "Modeling with a Sense of Purpose." *IEEE Software* 19 (1), January 2002, 8–10.
- Davis, Alan M. *Software Requirements: Objects, Functions, and States*. Prentice-Hall, 1993.
- Demarco, Tom. *Structured Analysis and System Specification*. Yourdon, 1978.
- Epp, Susanna S. *Discrete Mathematics with Applications*. Brooks/Cole, 1995.
- Ertas, Atila, and Jesse C. Jones. *The Engineering Design Process*, 2nd Edition. John Wiley and Sons, 1996.
- Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- Fowler, Martin, and Kendall Scott. *UML Distilled*. Addison-Wesley, 1997.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- Gelernter, David. *Mirror Worlds*. Basic Books, 1998.
- Gilb, Tom, and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.
- Gould, John, and Clayton Lewis. "Designing for Usability: Key Principles and What Designers Think." *Communications of the ACM* 28 (3), March 1985, 300–311.
- Graham, Ian. *Object-Oriented Methods*, 3rd Edition. Addison-Wesley, 2001.
- Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8, 1987, 231–274.
- Hauffe, Thomas. *Design: An Illustrated Historical Overview*. Barron's, 1996.
- Heskett, John. *Toothpicks and Logos*. Oxford University Press, 2002.
- Hopcroft, John, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- Humphrey, Watts S. *Introduction to the Personal Software Process*. Addison-Wesley, 1997.
- International Standards Organization. *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995E, 1995.
- Jacobson, Ivar, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.
- Jones, T. Capers. *Estimating Software Costs*. McGraw-Hill, 1998.
- Kiel, Mark, and Erran Carmel. "Customer-Developer Links in Software Development." *Communications of the ACM* 38 (5), May 1995, 33–44.
- Kifer, Michael, Arthur Bernstein, and Philip Lewis. *Database Systems: An Application-Oriented Approach*, 2nd Edition. Pearson Education, 2005.
- Kotler, Philip. *A Framework for Marketing Management*. Prentice-Hall, 2001.
- Krasner, Glenn, and Stephen Pope. "A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming* 1 (3), August/September 1988, 26–49.
- Kruse, Robert, and Alexander Ryba. *Data Structures and Program Design in C++*. Prentice-Hall, 1999.
- Larman, Craig. *Applying UML and Patterns*, 3rd Edition. Prentice-Hall, 2005.
- Lauesen, Soren. *Software Requirements: Styles and Techniques*. Addison-Wesley, 2002.
- Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns*, 2nd Edition. Addison-Wesley, 2000.
- Lieberherr, Karl, and Ian Holland. "Assuring Good Style for Object-Oriented Programs." *IEEE Software* 6 (5), September 1989, 38–48.

- McGrath, Joseph E. *Groups: Interaction and Performance*. Prentice-Hall, 1984.
- Meyer, Bertrand. "Applying 'Design By Contract'." *IEEE Computer* 25 (10), October 1992, 40–51.
- Meyer, Bertrand. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall International, 1997.
- Myers, Glenford J. *Composite Structured Design*. Van Nostrand Reinhold, 1978.
- Norman, Donald. *The Design of Everyday Things*. Currency/Doubleday, 1990.
- Norton, Michael. *Composing Software Design Patterns*. Computer Science Department Masters Thesis, James Madison University, 2003.
- Object Management Group (OMG). *UML 2.0 Superstructure Specification*. <http://www.omg.org>. April 2004.
- Object Management Group (OMG). *Unified Modeling Language, v1.5*. <http://www.omg.org>. March 2003.
- Orr, Kenneth. *Structured Systems Development*. Yourdon Press, 1977.
- Parnas, David L. "Designing Software for Ease of Extension and Contraction." *IEEE Transactions on Software Engineering* SE-5 (3), March 1979, 128–138.
- Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12), 1972, 1053–1058.
- Parnas, David L. and David M. Weiss. "Active Design Reviews: Principles and Practices." *Proceedings of the 8th International Conference on Software Engineering*, 1985, 132–136.
- Polya, George. *How To Solve It*. Princeton University Press, 1971.
- Pree, Wolfgang. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
- Preece, Jennifer, Yvonne Rogers, and Helen Sharp. *Interaction Design*. John Wiley and Sons, 2002.
- Preece, Jennifer, Yvonne Rogers, Helen Sharp, David Benyon, Simon Holland, and Tom Carey. *Human-Computer Interaction*. Addison-Wesley, 1994.
- Pressman, Roger. *Software Engineering: A Practitioner's Approach, 5th Edition*. McGraw-Hill, 2001.
- Riel, Arthur J. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- Rising, Linda. *The Patterns Almanac 2000*. Addison-Wesley, 2000.
- Robertson, Suzanne, and James Robertson. *Mastering the Requirements Process*. ACM Press, 1999.
- Royce, Walker. *Software Project Management: A Unified Framework*. Addison-Wesley, 1998.
- Rubin, Jeffrey. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. John Wiley and Sons, 1994.
- Rumbaugh, James, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, 2nd Edition*. Addison-Wesley, 2004.
- Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- Shneiderman, Ben, and Catherine Plaisant. *Designing the User Interface, 4th Edition*. Pearson Education, 2005.
- Sebesta, Robert W. *Concepts of Programming Languages, 6th Edition*. Pearson Education, 2004.
- Shaw, Mary, and David Garlan. *Software Architecture: Perspectives on An Emerging Discipline*. Prentice-Hall, 1996.
- Shlaer, Sally, and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, 1988.
- Sommerville, Ian. *Software Engineering, 7th Edition*. Addison-Wesley, 2004.
- Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

- Standish, Thomas A. *Data Structures in Java*. Addison-Wesley, 1998.
- Stevens, W. P., G. J. Myers, and L. Constantine. “Structured Design.” *IBM Systems Journal* 13 (4), 1974, 115–139.
- Stevens, Wayne. *Software Design: Concepts and Methods*. Prentice-Hall, 1991.
- Thayer, Richard, ed. *Software Engineering Project Management, 2nd Edition*. IEEE Computer Society Press, 1997.
- Thayer, Richard, and Merlin Dorfman, eds. *Software Requirements Engineering, 2nd Edition*. IEEE Computer Society Press, 1997.
- Ullman, David G. *The Mechanical Design Process*. McGraw-Hill, 1992.
- Ulrich, Karl, and Steven Eppinger. *Product Design and Development, 2nd Edition*. McGraw-Hill, 2000.
- Wiegers, Karl. *Software Requirements, 2nd Edition*. Microsoft Press, 2003.
- Winograd, Terry, ed. *Bringing Design to Software*. Addison-Wesley, 1996.
- Wirfs-Brock, Rebecca, and Alan McKean. *Object Design*. Addison-Wesley, 2003.
- Wirth, Niklaus. “Program Development by Stepwise Refinement.” *Communications of the ACM* 14 (4), 1971, 221–227.
- Wood, Jane, and Denise Silver. *Joint Application Design*. John Wiley and Sons, 1989.
- Yourdon, Edward and Larry Constantine. *Structured Design*. Prentice-Hall, 1979.
- Yourdon, Edward. *Modern Structured Analysis*. Prentice-Hall, 1989.

Index

- Abstract classes, 325, 335–336, 547
- Abstract data types (ADTs), 449
- Abstract Factory pattern, 545, 548–552, 564
 - when to use, 551
- Abstract operations, 273, 325–326, 335–336
- Abstraction, 7–10, 159, 161, 404
 - levels of, 44–45, 88–90, 100, 105, 158, 212, 235, 242–243, 319, 371–372, 395, 436, 454, 466, 469–470, 571
- Acceptors, 415–416, 418–419
- Access
 - accessibility, 197, 235, 246, 320–321, 328, 364, 433–438, 553, 556–557, 582
 - extending, 435–438
 - proxies, 530, 533
- Action nodes, 34–36, 38, 40, 44–46
 - naming, 44
- Actions
 - execution order, 402
 - in activity diagrams, 34–46
 - in state diagrams, 397–398, 402, 404, 406, 415, 418
 - in use case descriptions, 169–173
 - naming, 268, 405
- Active design review, 308–312, 454–456
 - completion phase, 309, 311
 - performance phase, 309, 311
 - preparation phase, 310
- Active operations, 130, 140, 365, 371, 373
- Activities, 34–36, 38–39, 44, 56, 176–177
 - coordinating, 381, 476, 499, 516, 584
 - deadlocked, 37
 - delegating, 514
 - design, 18, 24
 - management of, *see* Projects, managing
 - naming, 44, 268
- Activity diagrams, 47–48, 50–54, 79, 98, 110, 138, 162, 168, 175, 194, 212, 255, 259, 309, 321, 337, 359, 454
 - notation, 33–46
 - when to use, 44, 260
- Activity edges, 34, 40, 46
- Activity final nodes, 34–35, 38, 40, 42
- Activity graphs, 34–35
- Activity nodes, 34, 359
 - Activity parameters, 42–44
 - Activity symbols, 35, 42, 46
 - Actor domain classes, 342, 344
 - Actors, 159, 162, 164–177, 341, 344
 - naming, 165
 - Acyclic graphs, 475
 - Adaptee classes and objects, 523–529
 - Adapter pattern, 522–529
 - when to use, 527
 - Addresses, of variables, 433–435
 - Adequacy, 307, 312, 453, 454
 - Principle of, 131, 232–233, 246, 305
 - Adornments, *see* Association adornments
 - ADT, *see* Abstract data type
 - Aesthetic design principles, 245–248
 - Aggregation associations, 330–331, 335
 - Agile methods, 60, 64
 - Alexander, Christopher, 464, 467
 - Algorithms, 45–46, 246, 321, 430, 439–441, 448–449, 466–467, 491
 - Aliases, 435–438
 - Alternative fragments, 368
 - Analogy
 - automobile factories, 545
 - babysitter, 574–575
 - current awareness service, 580
 - electrical adapters, 523
 - Greek, Roman, and Norse gods, 553
 - interior designer, 505
 - scheduling a meeting, 516
 - stockbroker, 505
 - subordinate attending meetings, 530
 - travel agent, 514
 - warehouse, 499
 - watch batteries, 561
 - Analysis, 48–56, 61, 98, 100–104, 115, 121, 148, 162, 172, 184, 194, 196–200, 208, 215, 242, 289, 321, 336, 345, 348
 - class models, 196, 197–198
 - models, 196, 198–199, 289, 336, 345
 - Application domain, 342, 375, 470, 481–483, 485, 514–515
 - module, 481
 - Arboretum management system example, 221

- Architectural design, 54–55, 61, 67, 226–230, 253–259
 evaluation, 305
 finalization, 307–312
 notation, 265–268
 resolution, 287–288
- Architectural style, 288, 298–300, 466–467, 470, 473, 483
- Arrays, 434, 441, 446, 449–451
 of records, 451
- Artifacts, 277–280
- Assembly connectors, 274–275, 285, 328
- Assertions, 169, 261, 442–447
- Assets, 245
- Association adornments, 271, 331, 334–336
- Association attributes, 333, 335
- Association class connectors, 332, 335
- Association classes, 332–333, 335
 naming, 332
- Association lines, 159–166, 204–205, 217–218, 279, 324, 330–334
- Association qualifiers, 333–336
- Associations, 159–166, 204–219, 270, 279 324, 330, 332–336, 361
 aggregation, 330–331, 335
 naming, 205, 207, 217
- Assumptions, 83–85, 104
- Asynchronous messages, 362–367, 373
- ATM example, 161–162
- Atomic requirements statements, 128–129
- Atomizing requirements, 128–131
- Attributes, 196, 202–213, 216, 219, 328–329, 333–335, 339, 348, 386, 430, 432, 436, 443, 446
 compartment, 200–202, 206, 208–209, 326
 development, 257, 298
 naming, 207
 operational, 257
 quality, 257–258, 263, 293–296, 298, 302, 306, 338, 470
- Audits, 308, 312, 454
- Author role, 138, 140, 454
- Availability, 257, 265
- Ball and socket symbol, 275, 326, 328, 335
- Ball symbol, 274, 326, 335
- Basic design principles, 26, 131, 229, 231–233, 246–248, 305
- Basic flow, 169–177
 Principle of, 131, 246–248
- Beauty, 179, 245–248
 Principle of, 131, 246–248
- Behavioral requirements, 86
- Binary relations, 207–208, 217, 270, 333
- Binary search, 246, 448–449, 466
- Black boxes, 227, 229
- Black holes, 404
- Blackboard style, 477, 484
- Bloated controllers, 382, 386, 521
- Boggle example, 518, 519, 520
- Bottom-up strategy, 8, 100
- Box-and-line diagrams, 259, 289–291, 293–298, 469–470, 472, 474–475, 477, 479
 naming elements in, 267
 notation, 265–268, 277
- Brainstorming, 7, 51–52, 102, 173–174, 288, 302, 338
 team and individual, 122, 142
- Branch points, 36, 173, 177
- Branching, 36, 44, 46, 173, 177, 366
- Break fragments, 368–369
- Bricklin, Dan, 75
- Broker patterns, 505–506, 510–513, 517, 522–524, 529–530, 533, 544, 547
- Building complex example, 7
- Building monitoring system example, 478
- Business case documents, 80
- Business requirements, 84, 86–87, 92, 98, 102–104, 179, 256
- C++, 203, 207, 432–435, 542, 572
- Caldera example, 214–218, 374–376, 381
- Callback functions, 572
- Callouts, 420
- Calls relation, 468
- Car wash control program example, 424
- CardShark example, 534
- CAS, *see* Computer Assignment System example
- CASE, *see* Computer Aided Software Engineering
- Catalog Sales Support System example, 160
- Centralized control style, 381–384, 386–388, 521, 570
- Changeability, 196, 273, 453, 469–473, 477, 482–485, 500, 504, 513, 522, 564, 571–572, 578
 Principle of, 232–233, 305
- Channels, active and passive, 76
- Checklists, 114, 137–138, 140, 308–309, 312, 454
- Citation management system example, 471–472
- Clarity, 136, 141, 307, 312, 453–454
- Class Adapter pattern, 523–529

- Class diagrams, 26, 126, 196–198, 211–215, 217, 260, 262, 340–341, 343–345, 350, 360, 378–379, 430, 432, 439, 443, 481, 494, 500, 512, 523–525, 530, 547, 550, 554–555, 562, 575–576
 notation, 200–208, 324–336
 when to use, 208, 260, 430
- Class fields, 329
- Class invariants, 443, 445, 447
- Class libraries, 197, 273, 503–504, 525
- Class models, 196–200, 207–208, 213, 215–216, 218
 drafting, 336–345, 376
- Class name compartment, 200, 206
- Class operations, 329, 553, 556–557
- Class schedules, 48
- Class variables, 329, 336, 554–557
- Classes, naming, 207, 213–214
- Clients, 5, 24
- Clones, 531, 557, 560–564
- Cloning, 362, 560–564
 Cloning a stack example, 557, 560
- Cockburn, Alistair, 170
- Code, 5, 23, 56, 197, 229, 235, 237, 241, 245, 277, 280, 300, 321, 347, 350, 365, 431, 434–435, 440, 448, 473, 547, 572
- Cohesion, 241–243, 294, 298, 347, 351, 376, 382–384, 386, 440, 470–473, 501, 520, 522
 Principle of, 241–243, 347
- Collaboration, 228–229, 260, 320, 346, 359–360, 381, 437–438, 453, 466, 471, 483, 490, 505, 515–519
- Colleague classes, 517
- Collection iteration, 490–492, 496
- Command pattern, 574–578
 when to use, 577
- Communication diagrams, 260, 359
- Communication paths, 279–280
- Compartments, in UML diagrams, 197, 200–201, 206–209, 211, 275, 326, 360, 362, 366, 371, 398–400
 suppression of, 200–203, 206, 208–209
- Competitive analysis, 73
- Competitive strategy, 77–78
- Compilation dependency, 270
- Compiler example, 473–474
- Completeness, 113–114, 136, 141, 307, 312, 453–454
- Completion events, 397–398, 404, 407
- Component and interaction co-design, 374, 380–381
- Component diagrams, 259, 326
 notation, 273–277
- when to use, 276
- Component symbols, 273, 275
- Component-based development, 273
- Components, 269, 273–277, 289, 292, 303, 307, 309, 326, 335, 346, 374, 380–384, 386, 471, 473, 476, 480, 482, 522, 570–572
- Composite names, 200, 211
- Composite states, 400–418
- Composition associations, 330–331, 335
- Compound transitions, 411
- Compression example, 273, 275
- Computer Aided Software Engineering (CASE), 26
- Computer Assignment System (CAS) example, 87, 116, 144, 170, 185, 312, 386
- Conceptual models, 196–203, 208, 212–220, 288, 299, 341, 344, 376
- Concrete classes, 325
- Concrete operations, 325, 336
- Concurrency, *see* Execution, concurrent
- Concurrent composite states, 400, 407–409, 411–412, 414
- Concurrent regions, 408, 412, 414
 boundary lines, 400, 407
- Concurrent sub-states, 408–409, 414
- Configurability, 350–351
- Consistency, 113, 136, 141, 307, 312, 453–454
 terminological, 113
- Constraints, 83–85, 269, 309, 467, 473, 481
see also Design constraints
- Constructive design principles, 229, 233, 247–248, 305, 347
- Container classes, 344
- Continuous review, 141, 312, 455–456
- Contracts, 168, 441, 447
- Control flow, 267, 320, 359, 384, 40
- Control functions, 499
- Control styles, 381–389, 519, 521–522, 570
- Controllers, 341, 344, 381–384, 386, 480–485, 579
 bloated, 382, 386, 521
- Coordination components, 304
- Copy constructors, 560, 564
- Copy-on-write, 531
- Correctness, 113–114, 137, 179, 483
- Cost, 3, 17, 56, 63, 77, 84, 131, 138, 148, 184, 229, 453, 584
- Coupling, 239–243, 246, 294, 298, 347, 351, 382–384, 386, 440, 470, 473, 515, 519, 520–522, 543, 552, 571–572, 577–579, 582, 586
 Principle of, 239–243, 347
- CRC cards, 196–197
- Creational techniques, 336, 345

- Creativity, 10, 121, 259
 Critical reviews, 141, 454
 Crucial experiments, 133, 136
 Cruise control example, 400, 416
 Customer support, 77
- Data flow, 40, 42, 168, 440, 473
 diagrams, 168
 Data requirements, 86–89, 92, 100, 102, 140, 182, 199, 208, 288
 Data responsibility, 346–347, 351
 Data structure diagrams, 557
 notation, 449–452
 Data structures, 18, 216, 229, 237, 240, 242, 244, 321–322, 430, 439, 449, 451–452, 465–467, 557
 Data types, 18, 44, 196–197, 202–203, 212–213, 229, 237, 240, 242–243, 246, 321, 341, 344, 436, 449
 DDD, *see* Detailed design document
 Deadlock, 44, 414, 475
 Decision nodes, 36, 40
 Declarative sentences, 111, 114, 175
 Declarative specifications, 440, 447
 Decomposition, 18, 21, 25, 27, 48, 61–62, 182, 227, 229, 234–247, 276, 289, 292–293, 295, 298, 306, 312, 320, 336–337, 346, 350–351, 383, 404, 406, 430, 468, 513, 571–572, 576–578
 Deep copy, 557, 560–564
 Deep history pseudo-states, 412
 Deep history state indicators, 412
 Default values, 211, 369
 Defects, 138
 Defensive copy, 436
 Delegated control style, 381–384, 386–388, 521, 570
 Delegation, 350–351, 381–386, 514, 522–527, 530
 connectors, 275, 277
 Deliverables, 229
 Dependency arrow, 271–272, 278–279, 327
 Dependency relations, 270–272
 Deployment diagrams, 260
 notation, 277–280
 when to use, 280
 DeSCRIPTR, 229–230, 257, 260, 320–323, 430
 DeSCRIPTR-PAID, 230, 430, 454
 Design
 activity, 18, 24
 as a noun and a verb, 23
 as problem solving, 6–7
 by contract, 443, 447
- class models, 198, 340, 342, 344, 376, 384
 constraints, 17, 179, 194, 199, 229, 245
 document, 18, 21, 24, 48, 62, 128, 300, 322–323, 452–467
 event-driven, 568–569, 571, 577–579
 for reuse, 244–245, 247, 258, 551
 heuristics, 25, 27
 importance of, 3–4
 methods, 24–27, 179, 261, 322, 542
 notations, 25–27, 256, 259, 404, 453
 participatory, 101, 132
 principles, 24, 26, 136, 229, 233–248
 problems, 6, 7, 10, 18, 20, 47–52, 55, 61, 80, 87, 91, 98, 100–106, 114, 194, 196, 199, 229, 254
 processes, 20, 24–27, 33, 47–53, 55, 61, 73, 79–80, 84, 100, 102, 132, 142, 178, 254–255, 312
 rationales, 257, 309, 323, 453
 reviews, 308–312, 454–456
 solutions, 17, 21, 24, 27, 49, 87, 92, 100, 103–104, 109, 136, 178, 198, 341
 stories, 337, 339, 345
 structured, 25
 tasks, 26, 415
 teams, 13–15, 62, 122, 179, 288, 308–309, 312
 themes, 337–339
 tools, 7, 122, 147
 user-centered, 100–101, 131–132
 with reuse, 244–245, 247, 522
see also Software engineering design,
 Software product design
 Design-time code units, 298
 Desk checks, 137–138, 308, 454–455
 Detailed design, 45, 54–55, 61, 67, 226–227, 229–230, 254, 257, 280, 319–323, 429–430, 452–453
 Detailed design document (DDD), 322–324, 452, 456
 Deterministic finite automata, 396, 400–401, 405–406, 414
 Development attributes, 257, 298
 Devices, 262, 265, 277, 293–297, 344, 550, 568
 Device interface modules, 294–297
 Diagrams
 handwritten, 325
 suppressing elements of, 200–203, 206, 208–209, 328, 333–334, 364, 372, 403–404, 449–451
 Dialog maps, 419–422, 630–632
 Dispersed control style, 382, 384, 521
 Distributed systems, 280

- Document studies, 106, 109, 110
 Dorm rooms example, 205
 Dynamic models, 10–11, 24, 33, 44, 46, 142,
 167–168, 196, 200, 227, 267, 319, 336, 344,
 359, 374, 466, 473, 478, 510–511
- Ease of use, 12, 15, 196, 293
 Economy, 453–454
 Principle of, 131, 232–233, 305
 Editor palette example, 562
 Efficiency, 244
 Effort, 3, 56, 59, 61, 182, 245–256, 263, 312,
 344, 446, 571
 Elicitation techniques, 104, 106–111
 Elicitation workshops, 106, 109, 146
 [else] guards, 36, 44, 367, 406
 Empirical evaluation, 100, 103, 131
 Empirical studies, 100
 Engineering design, 12, 14–15, 18–19, 21, 24,
 47–48, 53, 61–62, 79, 127–128, 182, 194,
 196, 198–200, 208, 211, 213, 216, 227, 229,
 233, 247, 254–255, 305, 308
 English, 126, 130, 208, 257, 263, 285, 442, 448
 Entities
 dependent and independent, 270–271
 exported and imported, 272, 432
 program, 341–342, 430–438
 Enumeration classes, 342, 376
 Enumeration types, 264
 Established technology, 73–74
 Estimation, 56–57, 132
 Evaluation criteria, 133
 Events, 56, 163, 166, 169, 171–173, 175, 177,
 395–407, 409, 412, 414, 420, 424–425, 428,
 478–485, 568–572, 576–580, 582, 585
 naming, 405
 Event dispatchers, 478–480, 484, 570–571
 Event lists, 163–164
 Event-Driven architectural style, 478–480,
 483–484
 advantages and disadvantages, 480
 Event-driven design, 568–569, 571, 577–578,
 579
 Examples
 arboretum management system, 221
 ATM, 161–162
 Boggle, 518–520
 building complex, 7
 building monitoring system, 478
 Caldera, 214–218, 374–376, 381
 car wash control program, 424
 CardShark, 534
 catalog sales support system, 159
 citation management system, 471
 cloning a stack, 557, 560
 compiler, 473–474
 compression, 273, 275
 Computer Assignment System (CAS), 87,
 116, 144, 170, 185, 312, 386
 cruise control, 400, 416
 dorm rooms, 205
 editor palette, 562
 Fingerprint Access System (FAS), 172–175,
 186, 257, 265, 313
 gaming over the Internet, 279
 gas pump, 416, 420
 grain elevator tracking system, 353
 image proxy, 531
 indexing sub-system, 534
 Java Iterator, 416
 laundry, 34, 38
 Locker Assignment System, 150, 443–445
 model railroad, 8
 Open Systems Interconnection (OSI)
 model, 468
 parking garage controller, 149
 Pigeonhole Box Office System, 150
 Prescription Archive Program (PAP), 352
 Software Development Environment
 (SDE), 476
 Simply Postage Kiosk System, 151
 small engine repair business, 525
 stud finder, 423
 tape recorder, 397, 416
 traffic light, 86, 407, 410, 416
 Exceptions, 237, 346, 362–363, 395, 442, 568
 Execution, 211, 262, 363, 365, 443, 468, 479,
 482
 concurrent, 38, 44, 400, 407–411, 474–475,
 525–526
 environments, 278–280
 occurrences, 365–366, 371–373
 of action nodes, 34–44
 of prototypes, 143–145
 of operations, 365, 373
 order (actions), 402, 404–405
 parallel, 366, 407, 483
 program, 10–11, 211, 277
 synchronous and asynchronous, 363, 365
 Experts, 122, 199, 309, 463
 Exported entities, 432
 Exported operations, 436, 443, 445
 Extension mechanisms, 269–270
 Extensions, 169–172
 writing, 173–175
 External iterators, 493, 494–501, 503

- Façade pattern, 513–515, 521–522, 527
 when to use, 515
- Factory classes, 544, 548, 551–552
- Factory Method pattern, 544–548, 551–552, 563
 when to use, 548
- Factory methods, 525, 541–557, 560, 562
- Feasibility, 132, 137, 143, 147, 231, 254, 307, 312, 453
 Principle of, 131, 147, 233, 305
- Features, 326, 328–330, 335
- Fidelity, 144–148
- Fields, 87, 171–172, 556, 560
- Filters, 473–476, 485
- Filtering iterators, 502, 504
- Final states, 300, 307, 397, 404, 406–407, 412, 418
- Finalization of designs, 50, 136–141, 148, 307–312, 452–456
- Fingerprint Access System (FAS) example, 172–175, 186, 257, 265, 313
- Finite automata, 396–397, 399, 402, 404–405, 407, 414, 415–416
- Finite state machines, *see* Finite automata
- First-order logic, 126
- Flexibility, 350–351, 482, 485, 492, 494–495, 496, 498–499, 542, 544, 548–549, 556, 563, 574–575, 581, 588
- Flow final nodes, 38–39, 40
- Focus groups, 75–76, 102, 106, 109–111, 146
- Forks, 38, 40, 407, 411, 414–415
- Fork nodes, 38
- Formal notations, 126, 130
- Formats
- activity parameter, 43
 - class symbol attribute specification, 201–202, 204
 - class symbol operation specification, 202–204
 - converting, 474, 476
 - data, 87–89, 240, 319
 - fully dressed, 170
 - interaction fragment, 366–367
 - lifeline identifier, 361
 - loop operator, 369
 - object symbol attribute specification, 209
 - object symbol name compartment, 209
 - operation specification, 443–444
 - parameter specification, 43
 - sequence diagram frame, 360
 - sequence diagram message specification, 364–365
 - state diagram transition string, 398–399
 - use case description, 170–171
- Fragments, *see* Interaction fragments
- Frames, sequence diagram, 360, 366
- Frankston, Bob, 75
- Friends, C++, 432–433, 542
- Fully dressed format, 170
- Function classes, 573–575, 577
- Function objects, 573–575
- Functional components, 289–299, 303
- Functional organization, 57
- Functional requirements, 86–87, 92, 140, 148, 158, 167, 177, 182, 184, 198, 257, 263, 288, 293, 295, 301
- Functors, 573
- Gaming over the Internet example, 279
- Gantt chart, 78
- Gas pump example, 416, 420
- Gelernter, David, 245
- Generalization, 324–325, 344, 348
- Generator patterns, 505–506, 511, 540–549, 552, 554, 557, 562, 564
- Gerunds, 405
- Global visibility, 432, 436–438, 553–554
- Glossary
- in the DDD, 323, 453
 - problem domain, 110, 114
- Grain elevator tracking system example, 353
- Granularity, 165
- Graphical notations, 126–127, 196, 449
- Graphical user interfaces, 480, 514–515, 568–569, 573, 576
- Graphs, 34, 348, 350
- Guards, 36–37, 44, 367–370, 398, 406, 410, 412
- [else], 36, 44, 367, 406
 - exhaustive, 406
 - mutually exclusive, 368, 406, 412
- Hardware adaptability, 293, 295, 298
- Hash tables, 450–451, 466, 491
- HCI, *see* Human-Computer Interaction
- Heapsort algorithm, 252, 466
- Helper operations, 436, 438, 572
- Heterogeneous-style architectures, 483, 485
- Heuristics, 25–27, 236, 241
- activity diagram, 44
 - assertions, 445–446
 - atomization, 130
 - box-and-line diagram, 267–268
 - class diagram, 207–208, 325
 - class diagram generation, 342, 344
 - data structure diagram, 451–452

- elicitation, 105–106
- information hiding, 436–437
- interaction modeling, 381–388
- object diagram, 211
- requirements statement, 127–128
- scenario description, 301
- sequence diagram, 371–372
- state diagram, 404–406, 414
- state modeling, 345–351
- summary tables, 46, 108, 130, 167, 177, 211, 219, 268, 276, 307, 335, 344–345, 351, 373, 388, 406, 415, 438, 447, 452
- theme-based decomposition, 339–340
- use case description, 175–177
- use case diagram, 165–166
- History states, 412d–415
 - indicators, 412
- Human-Computer Interaction (HCI), 90
- IDE, *see* Integrated Development Environment
- Identifier, 128, 130, 170, 216, 300, 328, 333, 360–361, 373, 430, 434–435, 438, 475, 495
- Idioms, *see* Programming idioms
- Image proxy example, 531–532
- Immediate parts, 235, 243
- Implementability design principles, 244–245, 247–248
- Implementation activity, 18, 24
- Implementation class model, 197, 199–200
- Implicit Invocation architectural style, 478–480, 483, 485
- Indexing sub-system example, 534
- Individuals in interaction diagrams, 359–363, 371–372
 - anonymous, 361, 371
 - primary, 371, 374
- Industrial Revolution, 12, 15
- Information hiding, 236, 238, 240, 242–243, 246, 248, 273, 293–295, 297–299, 377, 382–384, 386–388, 430–440, 470–473, 491, 493–494, 496, 501–502, 504, 526, 530, 533, 542–543, 561
 - Principle of, 236–238, 243, 436, 437–438
- Inheritance, 320, 324, 348, 350–351, 522–523, 525
 - multiple, 324
- Initial nodes, 34–36, 40, 42
- Initial pseudo-states, 397
- Initial states, 300, 307, 396–397, 400, 402, 404, 406, 407, 414–415
- Initial values, 201, 203, 211, 216, 333
- Inner states, 400
- Input activity parameters, 42–44
- Input pins, 40
- Inspections, 56, 137–142, 308–309, 312, 454
 - meeting, 138, 140
 - overview, 138
 - process, 138
- Instance creation, 511, 543
- Instance operations, 329, 440
- Instance variables, 329, 336
- Institute of Electrical and Electronics Engineers IEEE), 91
- Integrated Development Environment (IDE), 57
- Interaction design, 14–15, 19, 62, 89–92, 106, 146, 169, 184, 295, 359–361, 365–367, 371, 373–374, 376, 379, 381–384, 386, 430, 442, 471, 480, 511, 521
- Interaction diagrams, 168, 260
 - notation, 359–373
 - when to use, 260, 430
- Interaction fragments, 366–370
- Interaction overview diagrams, 359–360
- Interfaces, 5, 228–229, 233–234, 261–263, 294–295, 320, 344, 430
 - adapters/wrappers and, 522–523, 527–528
 - brokers and, 511
 - collection, 492, 494–497, 498, 500
 - command, 575, 577
 - façades and, 513–515
 - Factory patterns and, 544–549, 551
 - Java Clonable interface, 560
 - Java Iterator interface, 503
 - observer and subject, 581, 583–585
 - operation, 430, 439–441
 - proxies and, 530, 533
 - provided and required, 274–275, 326–327, 335
 - specifying, *see* Interface specifications
 - symbols, 273–275, 326–328, 335
 - types, 500, 542
 - UML, 273–276, 326–328
 - user, *see* User interfaces
- Interface classes, 342, 521
- Interface specifications, 89, 228, 260–262
 - template, 262
- Internal iterators, 493–499
- Internal transitions, 398, 403–407, 412
- International Standards Organization (ISO), 469
- Interpolation search, 448
- Interviews, 75–76, 106, 109–111, 146
- ISO, *see* International Standards Organization
 - Italicizing in UML, 325–326, 335

- Iteration, 366, 369–370
 - control, 492–498, 505, 573
 - in design, 50, 52, 55, 60, 100–101, 184, 319
 - through a collection, 490–498, 501, 503–504
- Iteration mechanisms, 491–498, 573, 575
 - Built-in, 492–497
 - internally and externally controlled, 492–499, 573
 - robustness of, 497–498
- Iterative planning and tracking, 59–62
- Iterator pattern, 496, 498, 500–505, 511, 513, 547
- Iterators, 492, 496, 498–505, 511, 513, 547
 - external, 493, 494–501, 503
 - internal, 493–496, 498–499
 - robustness of, 501, 503–504
- Java, 45, 207, 230, 261, 273, 330, 368, 370, 416, 430–434, 436, 440–442, 464, 492, 501–503, 525–528, 546, 548, 558, 572–573, 576–577, 584–585
 - AWT, 551, 576
 - cloning in, 560–561
 - Collections, 261, 416, 527, 546
 - Enumerator interface, 503
 - Iterators, 416, 503
 - JVM, 278
 - singletons, implementing in, 556–557
 - Swing, 551, 576–577
- Join nodes, 38, 40
- Joins, 38, 40, 407, 411–412, 475
- Kind-of relation, 348
- Law of Demeter, 387–388
- Laundry example, 34, 38
- Layered architectural style, 298, 384, 467–472, 481, 485, 515
 - Relaxed, 468–469, 471–473
 - Strict, 468, 470–473, 485
 - when to use, 470
- Layers, 227–229, 266–267, 272–273, 298, 384, 467–472, 481, 483, 485, 515
- Leadership, 59, 62
- Leading-edge technology, 73–74, 107
- Least Privilege, Principle of, 238, 243
- Lego, 122
- Levels of abstraction, *see* Abstraction
- Lexical analyzer, 418–419, 474
- Life cycle, *see* Software life cycle
- Lifelines, 360–367, 371, 373
- Link lines, 210–211, 324
- Links, 210–211, 279, 324, 332
- Literals, 202–203, 209
- Local visibility, 432
- Locker Assignment System example, 150, 443–445
- Logical architecture, 277
- Lollipop symbol, 273, 275, 326
- Loop fragments, 369–370
- Low-level design, 21, 44, 62, 319, 321–324, 436, 444, 453, 466
- Maintainability, 258, 263, 302, 471–472, 477, 485, 500, 574, 586
- Maintenance activities, 18–19
- Maintenance profiles, 302
- Maintenance releases, 73, 76, 107–108
- Management, 17, 56–63, 74, 77, 80, 82
- Markets, 71–78, 106
 - segments of, 77, 82
 - studies of, 108
 - target, 71–74, 77, 82–83
- Marketing, 14, 75, 82–83
- Mediator pattern, 515–522
 - when to use, 520
- Merge nodes, 36–37, 40–42, 44
- Merge sort algorithm, 466
- Message arrows, 362–363, 371, 373
- Message specifications, 363, 365
- Messages, 359, 362–368, 371, 373, 382, 386, 395, 404, 515, 519, 568–569, 575, 582
- Metaphors, 122, 125, 147
- Micro-management, 381
- Middleware, 255
- Mid-level design, 319–323, 336, 344–345, 359, 361, 372–374, 380–381, 430, 446, 453, 490
 - patterns, 299, 466–467, 490, 504–505, 510, 529, 542
- Milestones, 56, 63
- Minispecs, 448, 452
- Mock-ups, 143
- Modeling, 125, 142, 167, 178, 196, 200, 208, 212, 242, 259, 267–268, 272–273, 275, 277, 280, 307, 321, 336, 351, 359, 372, 384, 396, 416 430
- Model railroad example, 8
- Model-View-Controller (MVC) architectural style, 480–483, 485, 579, 583, 586
 - advantages and disadvantages, 482
- Moderator role, 138–140
- Modifiability, 258, 293, 320, 513, 521
- Modularity, 233–235, 273, 293–294, 305–306
 - design principles, 233, 235–243, 382

- Modules, 227–229, 233–243, 247, 261, 270–271, 328, 347–348, 350–351, 432, 436–438, 467–468, 470–472, 478, 481, 578
 application domain, 481
 device interface, 294–297
 interface specification template, 261–262
 observer and subject, 582–583
 small, Principle of, 235–236, 243, 382
 user interface, 299, 471–472, 481
- Multi-dimensional ranking, 133–136, 305–307
- Multiplicity, 202, 205, 207, 210–212, 216, 218, 324, 330, 333–334
- “Must” and “shall,” 127
- MVC, *see* Model-View-Controller architectural style
- Name compartment, 360, 362, 371, 398
 class, 200, 206
 object, 208–209
- Name policy, UML, 200
- Name-direction arrow, 204
- Namespaces, 432
- Natural language, *see* English
- Navigability, 334–336
- Need statements, 111, 113–115, 123
- Needs analysis, 100, 105, 142
- Needs documents, 111, 114
- Needs elicitation, 100, 105–106, 114, 115, 142, 146
- Needs identification, 100
- Needs lists, 110–114, 138, 158, 162, 166, 172, 179
- Nested diagram compartment, 399
- Nested state diagrams, 399–401, 404–405, 413–414
 suppression of, 403
- Niche-market products, 72, 75, 106–107
- Nodes, 277, 279–280, 421
- Non-behavioral requirements, 87
- Non-functional requirements, 87, 92, 100, 184, 254, 257, 259, 288, 293, 295, 300, 346
- Non-local visibility, 432–433
- Notations, formal and semi-formal, 126–127, 130
- Note cards, 196
- Notes, 269, 420
- Notifications, 298, 376, 381, 481, 483, 580–585
- Nouns and noun phrases, 23, 44, 165, 207, 211, 213–215, 219, 268, 332, 405
- Novelty
 product line, 73–74
 technological, 73–74
- Null pointer, 450
- Object Adapter pattern, 523–529
- Object diagrams, 196, 531
 notation, 208–211
 when to use, 211
- Object model, 196, 199
- Object name compartment, 208–209
- Object nodes, 40, 42, 44, 46
 naming, 44
- Object-oriented design, 24–27, 207, 239, 241, 336, 439–440, 496
- Object-oriented paradigm, 324, 341
- Objects, naming, 211
- Observables, *see* Subjects
- Observation, 106, 109
- Observer pattern, 298, 578–586
 when to use, 583
- Observers, 299, 376–377, 381, 468, 578–585
- Onion diagrams, 468
- Open Systems Interconnection (OSI) model, 468
- Operands, 366–369, 373
- Operation compartment, 366
- Operation contracts, 441–445, 447
- Operation interfaces, 430, 439–441
- Operation specifications, 261, 322, 328–329, 360, 439–440, 443, 446–449, 452
- Operational attributes, 257
- Operational phase, 569–570, 574, 578, 581–582, 585
- Operational responsibilities, 351, 346–347
- Operational-level requirements, 89–92, 100, 102, 123, 158, 167, 177, 182, 212
- Operations
 abstract 273, 325–326, 335–336
 active, 130, 140, 365, 371, 373
 class, 329, 553, 556–557
 compartment, 200–201, 206, 326
 concrete, 325, 336
 executing, 365, 373
 exported, 436, 443, 445
 helper, 436, 438, 572
 instance, 329, 440
 naming, 207
 suspended, 362, 365, 373
- Opportunity funnel, 76–79
- Opportunity statements, 76–81
- Optional fragments, 367–368
- Organization charts, 110, 114
- Organizational structure, 56–59, 63
- Outer states, 400
- Output activity parameters, 42–44
- Output pins, 41–42
- Outside-in design, 374, 381

- Package diagrams, 260, 272, 276–277
 Package members, 271
 Package visibility, 328–329, 432, 436, 502
 Packages, 200, 229, 269–272, 276–277, 328, 432, 501, 504, 556, 584
 Packaging, 229, 321
 PAID, 230, 321, 323, 430, 454
 Parallel execution, 366, 407, 483
 Parameters, 203, 211, 237, 239, 364, 372, 398, 432–435, 437, 440, 445–446, 499, 549, 556
 activity, 42–44
 loop, 369–370
 output activity, 42–44
 passing, 435, 437
 reference, 433, 437
 Parking garage controller example, 149
 Part-whole relation, 330–331
 Pascal, 435
 Pattern catalogs, 466–467
 Patterns movement, 464, 466–467
 Performance, 87, 143, 234, 244, 257, 265, 437, 470, 472–474, 476–477, 480, 482–483, 511, 513, 520, 556, 571–572, 577, 586
 Physical architecture, 277, 279–280
 Physical-level requirements, 88–92, 100, 102, 106
 Pigeonhole Box Office System example, 150
 Pins, 41–42
 naming, 44
 Pipe-and-Filter style, 473–476, 483, 485
 advantages and disadvantages, 476
 Pipelines, 475, 489
 Pipes, 473–476, 483
 Planning, 56–57, 59–62, 74–78
 and tracking, iterative, 59–60, 61–62
 Pointers, 334, 449, 451–452, 572
 Polling, 376, 381
 Polymorphism, 440
 Portability, 87, 257, 473, 482
 Ports, 275
 Postconditions, 169–173, 262, 268, 442–447
 Pragmatics, 261, 268–269, 320, 430, 439–440, 523
 Preconditions, 169–173, 177, 262, 264, 442–447
 Prescription Archive Program (PAP) example, 352
 Prioritizing, 78, 112, 131, 135, 137, 140, 184
 Private visibility, 238–239, 329–330, 432, 436, 502, 556
 Problem concept, 212, 219
 Problem domain, 104–106, 108–110, 114, 122, 125, 142, 165, 167, 198–199, 208, 216, 341, 463, 579
 Problem domain glossary, 110, 114
 Problem solving, 6, 10–11, 38, 47, 48, 52, 321
 Problem statements, 49
 Procedural specifications, 325, 440, 447–448
 Processes
 engineering design, 21, 79, 85, 255, 455
 product design, 88–89, 92, 98, 102–103, 121, 136, 182
 Product categories, 72–74
 Product design, 12–16, 19, 21, 44, 47–48, 52, 61–62, 72–74, 84–85, 88–92, 98, 100, 102–104, 106, 109
 Product development ideas, 76–78
 Product lines, 72–78
 novelty, 73–74
 Product plans, 75, 78, 96
 Product types, 72
 Product vision statements, 81, 84, 98
 Productivity, 3, 104, 245, 463
 Products
 competitive, studies of 75–76, 108, 122
 consumer, 12, 72, 107, 108
 custom, 72, 75
 derivative, 73, 75–76, 107, 108
 existing, studies of, 108, 122
 maintenance releases, 73, 76, 107, 108
 new, 52, 73, 75–78, 81–84, 105, 107, 110
 niche-market, 72, 75, 106–107
 similar, 122, 125, 288,
 software, *see* Software products
 work, *see* Work products
 Profiles, 301–307
 Profitability, 77–78
 Program entities, 341–342, 430–438
 Program units, 18, 227, 229, 235, 257, 259, 294, 374, 467
 Programmers, 23, 234
 Programming idioms, 321, 466–467, 542
 Programming languages, 33, 207, 261–262, 364, 368, 430, 432–433, 435–436, 448, 465–466, 474, 476, 492, 553
 see also C++, Pascal, Java
 Project brief, 80
 Project charter, 80
 Project mission statement, 78–84, 91, 99, 162, 166
 Project plan, 56–58, 60–61, 96
 Project prioritization, 78

- Project scope, 81, 84, 113
- Project vision and scope document, 80
- Projects, 56
 - leading, 56, 59, 62
 - managing, 56–63
 - organizing, 56–58, 62
 - planning, 56–57
 - staffing, 56, 58, 62
 - tracking, 56, 58–61
- Properties, 227, 229, 309, 320
- Props, 122, 125
- Pros and cons, evaluating, 133, 179, 182, 305
- Protected visibility, 329, 433, 436, 438, 556, 560
- Protection proxies, 530, 533
- Prototype pattern, 561–564
 - when to use, 563
- Prototypes, 10, 61–62, 103, 108, 142–148, 304–305
 - electronic, 145, 147
 - evolutionary, 144, 148
 - exploratory, 144
 - high-fidelity, 145–148
 - horizontal, 143–144, 147–148
 - low-fidelity, 144–145, 148
 - paper, 144–147
 - proof-of-concept, 143
 - throwaway, 144, 146–148
 - vertical, 143–144, 147
- Provided interfaces, 274–275, 326–327, 335
- Proxies, 529–533
 - Proxy pattern, 529–534
 - when to use, 533
- Pseudocode, 269, 448, 452
- Public visibility, 236–238, 241, 273, 326, 328, 329–330, 335, 348, 433, 526, 554, 573, 580
- Publish-Subscribe pattern, 578
- Qualified classes, 333, 335
- Quality
 - assurance, 13, 62, 75, 128
 - attributes, 257–258, 263, 293–296, 298, 302, 306, 338, 470
 - characteristics, 136–137, 307–308, 453–454
 - gates, 141
 - of products and designs, 3–4, 90, 101, 131, 133, 141, 194, 231, 233, 245, 307, 452–454
- Quicksort algorithm, 246, 466, 574
- Rational Unified Process (RUP), 16, 60
- Rationalizing, 173, 177, 213, 302
- Reactor classes, 569–570, 575
- Reactor patterns, 505, 568–578, 581–582, 586
- Reader role, 138, 140
- Readiness check, 138
- Realization connector, 327, 335
- Recognizers, 415–416
- Recorder role, 138, 140
- Records, 449–451
- References, 261, 334, 362, 433–438, 446, 449–450, 500, 502, 512, 519–520, 523, 530–532, 542, 554, 558–559
- Reference alternatives, 134–135
- Refinement, 8, 59–62, 100, 123, 158, 167, 177–178, 181, 336, 339–340, 474, 568
 - stepwise, 25, 568
- Reified associations, 333, 335
- Relations, 204–208, 210, 332, 335, 348
 - calls, 468
 - dependency, 270, 272, 285
 - generalization, *see* Generalization
 - inheritance, *see* Inheritance
 - kind-of, 348
 - part-whole, 330–331
 - use, 270, 468
- Relationships, 126, 212, 217–218, 227–229, 266–268, 288, 320, 324, 344, 348, 445–446, 468
 - modeling, 8–10, 26
 - specifying, 260–261
- Relaxed Layered architectural style, 468–469, 471–473
- Reliability, 87, 227, 257, 293, 295, 320, 470, 472–473, 476, 483
- Remote Method Invocation, 279
- Remote proxies, 530, 533
- Repository style, 477, 484
- Required interfaces, 274–275, 326–327, 335
- Requirements activity, 24
- Requirements analysis, 100
- Requirements development, 85, 90, 92, 178
- Requirements elicitation, 100, 103–104, 106, 109
- Requirements engineering, 85
- Requirements generation, alternatives, 121–125
- Requirements management, 85, 92
- Requirements specification activity, 17
- Requirements traceability, 128–129, 131
- Requirements validation, 100, 103, 137, 142
- Resolution, 47–55, 61, 98, 100, 103, 121–123, 141, 148, 194, 198–199, 208, 211, 216, 226–227, 229, 254, 288, 422, 452

- Response time, 87
- Responsibilities, 227, 229, 241–242, 260, 289, 295, 309, 320, 339, 344–351, 374, 376, 381–386, 430, 439–440, 447, 452, 468–469, 471, 548, 552, 569
- Responsibility-driven decomposition, 346
- Return arrow, 362–363, 371, 373
- Reusability, 87, 234, 244–247, 257, 273, 276, 293, 295, 298, 302, 469–470, 472–473, 476, 480, 483, 511, 513, 520–522, 527, 572, 578–579, 586
- Reuse, 77, 244–245, 246–247, 348–351, 464, 471, 515, 522–523, 528–529, 551, 570, 578 design for, 244–245, 247, 258, 551 design with, 244–245, 247, 522
- Reviews, 101, 114, 128, 136–141, 166, 217–218, 308–312, 454–456 continuous, 141, 312, 455–456 critical, 141, 454
- Risk, 56, 58–59, 61, 63, 131, 179, 229, 232–233, 455–456 analysis of, 56, 58–59, 61, 63
- RMI, *see* Remote Method Invocation
- Robustness, 477, 480, 497
- Rolenames, 205, 207–208, 210, 325, 334–335 visibility, 334–335
- Roles, 12, 58, 138–140, 159, 165, 207–208, 211, 325, 381, 482–483, 511–512, 517, 530, 554, 562, 583 naming, 207 stakeholders', 101–102, 110
- Rolodex, 122
- Runtime components, 298
- SAD, *see* Software architecture document
- Safety, 3, 5, 83, 476, 478, 483, 529
- Scenario descriptions, 300–301, 303, 307
- Scenarios, 161–162, 172–173, 263, 300–307
- Schedules, 57–59, 60, 113, 254
- Scope, 52, 61, 81, 90, 113–114, 227, 229, 310, 321, 339, 344, 436, 438
- Scoring matrices, 133–136, 179, 182, 305–307, 322
- Security, 87, 239, 257, 265, 320, 478, 483, 505, 529
- Selectors, 361, 373
- self special name, 361–362, 371, 373
- Semantics, 261, 268–269, 320, 430, 439–440, 523
- Semi-formal notations, 126–127, 130
- Sequence diagrams, 359–360, 374, 439, 446, 517–520, 581–582 centralization in, 382–383
- notation, 360–373 when to use, 260, 372, 430
- Sequential composite states, 400, 404, 406, 412, 414, 415
- Setup phase, 569–570, 574–575, 578, 581–582, 585
- “Shall,” *see* “Must” and “shall”
- Shallow copy, 557, 560, 563–564
- Shared-data accessors, 476–477, 483, 485
- Shared-Data architectural style, 476–485 advantages and disadvantages, 477
- Shared-data stores, 476–477, 483, 485
- Signals, 235, 362–363, 395, 568
- Signatures, 273, 325, 371, 397–398, 439–440, 442, 581
- Simple names, 200, 202–203, 209, 211, 360–361, 364, 398
- Simple states, 400, 402, 404
- Simplicity, 179, 196, 245–246, 294, 470, 472, 572 Principle of, 131, 244–247
- Simply Postage Kiosk System example, 151
- Singleton classes, 553–557
- Singleton pattern, 553–557, 562, 582 when to use, 556
- Small engine repair business example, 525
- Small Modules, Principle of, 235–236, 243, 382
- Socket symbol, 273, 275, 326, 335
- Software architecture, 227, 254–259, 268–269, 288, 298, 300, 301, 305, 307, 310–311, 319–320, 322, 465
- Software architecture document (SAD), 255–260, 262, 307–312, 321–322, 452, 456
- Software components, 273, 276, 293, 345–346, 416
- Software design method, 24, 27
- Software Development Environment (SDE) example, 476
- Software engineering design, 14–16, 19, 25, 33, 47, 53, 55, 194, 229, 319–321, 345, 404, 416, 463, 466
- Software life cycle, 16, 19, 21, 24, 62, 79, 85, 194, 232, 452
- Software product design, 13–15, 19, 21, 47, 52, 55, 79, 84–85, 98, 103, 121, 142, 194, 229, 254, 416
- Software products, 4–5, 11, 13–17, 21, 24, 52, 55, 79, 81, 84–87, 98, 103–104
- Software reuse, *see* Reuse
- Sorting, 246, 476, 572, 575
- Source classes, 205–206, 218
- Source states, 396–398, 405
- Spreadsheets, 3, 5, 122

- Staffing, 56, 58, 62
- Stakeholders, 83–84, 88, 98, 100, 102–114, 121–123, 131–137, 140, 142, 158, 165, 167, 169, 171–172, 177–179, 184, 208, 257, 259, 422
focus on, 100, 103, 131
goals of, 105, 108–109, 113, 135, 158, 165
roles of, 101–102, 110
surveys of, 136
- Stakeholders-goals list, 110, 113–114, 138, 162, 166
- Stand-ins, *see* Proxies
- Startup classes, 342, 344
- State diagrams, 126,
 notation 395–422
 when to use, 260, 430
- State transitions, 309, 320, 396–397
- States, 11, 40, 42, 211, 309, 320, 359, 371, 395–422, 430, 443, 445, 557
 labeling, 405
- Static design models, 10, 167–168, 196, 200, 208, 265, 267, 272, 319, 336, 345, 351, 374, 420, 430, 466, 478, 510
- static keyword, 330
- Stepwise refinement, 25, 568
- Stereotypes, 270–271, 276, 325–326, 335
- Storage cells, 433–435
- Strict Layered architectural style, 468, 470–473, 485
- Strings, 202–203, 230, 431, 434–435, 574
 as names, 200, 360, 430
 manipulating, 274–275
 transition, 397–399, 403–405, 408, 411
- Structure charts, 25
- Structured design, 25
- Stub symbols, 403
- Stubbed states, 403
- Stud finder example, 423
- Sub-programs, 5, 57, 235–236, 240, 430, 432, 435–437
- Sub-states, 395, 400, 402–407, 412, 414–415, 418
- Subjects, 299, 377, 578–585
- Subordinate attending meetings analogy, 530
- Supplier classes, 505–506, 511, 513–514, 517, 521–523, 527, 531, 533–534
- Suppression, *see* Diagrams, suppressing elements of
- Surrogates, *see* Proxies
- Surveys, 75–76, 133
- Suspended operations, 362, 365, 373
- Synch states, 410–411, 414
- Synchronization, 38, 46, 320, 328, 362–363, 365, 367, 373, 407, 410, 415, 440, 474, 476, 479, 483, 525, 527
- Synchronous messages, 362, 365, 373
- Syntax, 207, 261–262, 320, 364, 439
 trees, 474
- System boxes, 160
- System sequence diagrams, 372
- Tagged values, 269
- Tape recorder example, 397, 416
- Target classes, 205–206, 218, 334, 569–570, 575
- Target markets, 71–74, 77, 82–83
 size, 72, 74
- Target states, 405–406, 409, 396–398, 411–412
- Teams, structure of, 57–58, 382
 see also Design teams
- Technical requirements, 86–87, 89, 100, 102, 104
- Technological novelty, 73–74
- Technology
 leaders, 77
 trajectories, 77
 types of, 73–74
 see also Established, Leading-edge, and Visionary technology
- Templates
 detailed design document (DDD), 322–323
 interface specification, 261–262
 project mission statement, 80
 software architecture document (SAD), 256–257, 308
 software requirements specification (SRS), 91–92
 use case description, 168–173
- Terminological consistency, 113
- Testability, 87, 127
- Testing activity, 18, 24
- Threads, 362, 525, 533
- Thread-safety, 522, 525, 527
- Time (as a resource), 3, 17, 27, 56, 58, 61, 63, 72, 84, 131, 179, 182, 184, 256–257, 360, 376, 379, 453
- Timing diagrams, 359–360
- Tokens, 34–44, 418–419, 474
- Top-down approach, 8, 21, 25, 27, 55, 100, 105, 178, 321, 374, 568
- Traceability, 128
- Tracking, 56, 58–61

- Trade-offs, 112, 131, 135, 254, 572
Traffic light example, 86, 407, 410, 416
Transactions, 87, 172
Transducers, 416–418
Transformational techniques, 336, 344–345
Transition junction point symbols, 411–412
Transition junction points, 412, 414–415
Transition strings, 397–399, 403–405, 411
Transitions, 229, 230, 320, 396–415, 430, 469
Transitivity, 330, 335–336
Traversals, 490–504, 511
Trial and error, 7, 52
Triggers, 169, 172–174, 396–398, 409, 468
- Underlining in UML, 211, 230, 286
Uniformity, 113–114, 136, 141
UNIX, 476, 572
Usability studies, 133, 136, 143, 147
Usage profiles, 301–302
Use case briefs, 164–165, 167
Use case descriptions, 167, 178, 181, 183, 198, 372
designing with, 181–182
notation, 168–177
when to use, 260
Use case diagrams, 167–168, 172, 175
designing with, 178–181
notation, 158–167
when to use, 167, 260
Use case models, 165–166, 178–184, 196, 198, 372
Use-case-driven iterative development, 184
Use cases, 158–184, 213, 263, 300–301
naming, 165, 301
User interfaces, 5, 14–15, 89–90, 92, 102, 104, 106, 133, 140, 143–144, 146–148, 175, 265, 289, 292–293, 297, 419–423, 468–469, 471, 478, 480–485, 551, 572, 579, 583
User interface design, 24, 90, 92, 102, 133, 143, 293, 422, 423
User interface diagrams, 420–423
User interface modules, 299, 471–472, 481
User interface states, 419, 422
User observation, 75
User tasks, 106
User-centered design, 100–101, 131–132
- User-level requirements, 88, 92, 100, 102, 106, 123, 133, 158, 182, 184
Use relation, 270, 468
Utility, 302
Utility tree, 302–303, 305
- Validation, 100, 173
Variables, 261, 321, 328, 329, 364–365, 371, 395, 398, 406, 411, 430–431, 433–435, 440, 443
Verbs and verb phrases, 23, 44, 165, 205, 207, 217, 332, 405
auxiliary, 127
Verifiability, 136, 141
Views, 256, 298, 322, 480–485, 579
Virtual devices, 227–228, 262, 294–297, 310–311, 550
ideal, 295
Virtual machines, 34–35, 278, 280
Virtual proxies, 530, 532–533
Visibility, 230, 320–321, 328–329, 334, 430–438, 440, 553–554, 556
limiting, 436–437
VisiCalc, 75
Visionary technology, 73–74, 105, 107, 108
Visual Basic, 147, 478
- Walkthroughs, 138, 303–304, 308, 312, 454
Waterfall software life cycle, 16–17, 19, 85
Wedding cake diagram, 468
Well-formedness, 136, 141, 307, 312, 404, 453–454, 456
“What” versus “how” distinction, 19–20
White holes, 404
Wildcards, 170
Wirth, Niklaus, 25
Word processors, 3, 5, 126
Work products, 56, 61–63, 137–138, 141, 308
Workflows, 16
Wrapper pattern, 522
Wrappers, 522
- X Windows System, 469, 573
- Z, 126