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

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

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

Contents

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 *make* 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 *using* 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:

- 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.

BLANK PAGE

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?

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

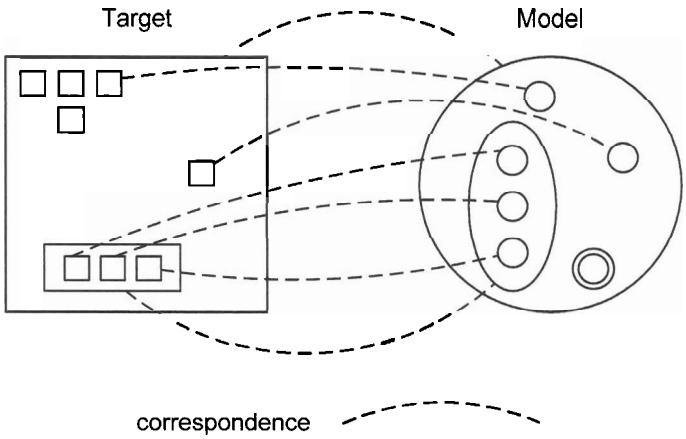

Figure 1-1-1 How Modeling Works

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 sub-programs.
- 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.

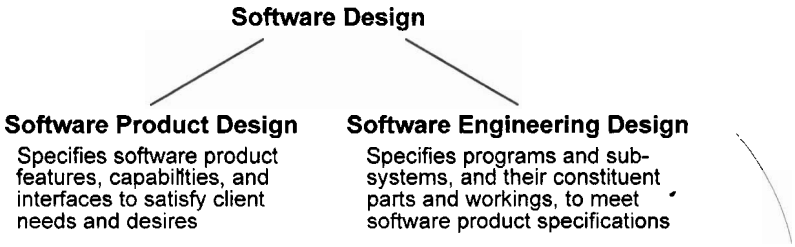

Figure 1-2-2 Software Design Sub-Fields

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.

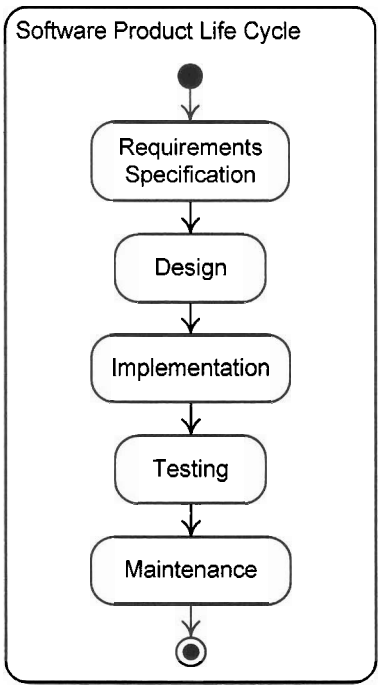

Figure 1-3-1 The Waterfall Life Cycle Model

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

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.

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.

Implementation Activity

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.

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.

Testing Activity

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.

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.

Maintenance Activity

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.

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.

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.

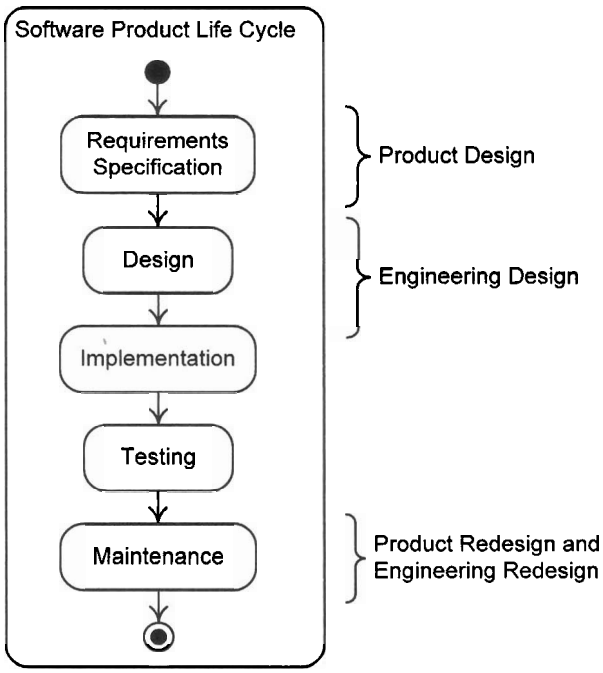

Figure 1-3-2 Design Across the Life Cycle

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.

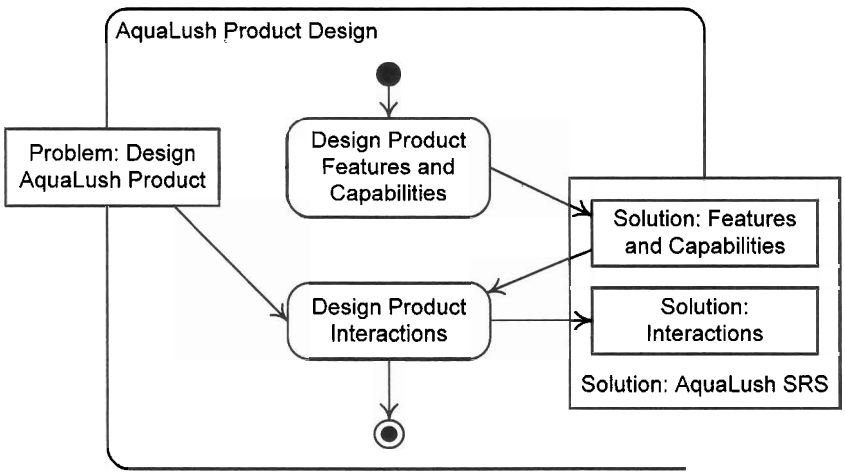

Figure 1-3-3 AquaLush Product Design Problems and Solutions

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.

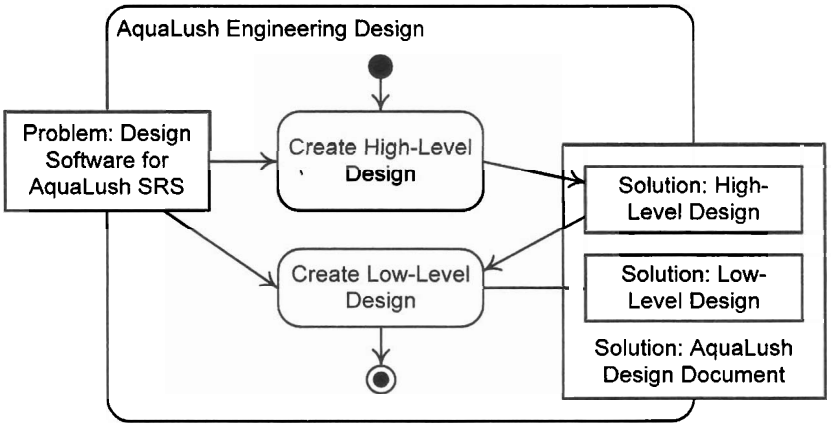

Figure 1-3-4 AquaLush Engineering Design Problems and Solutions

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.

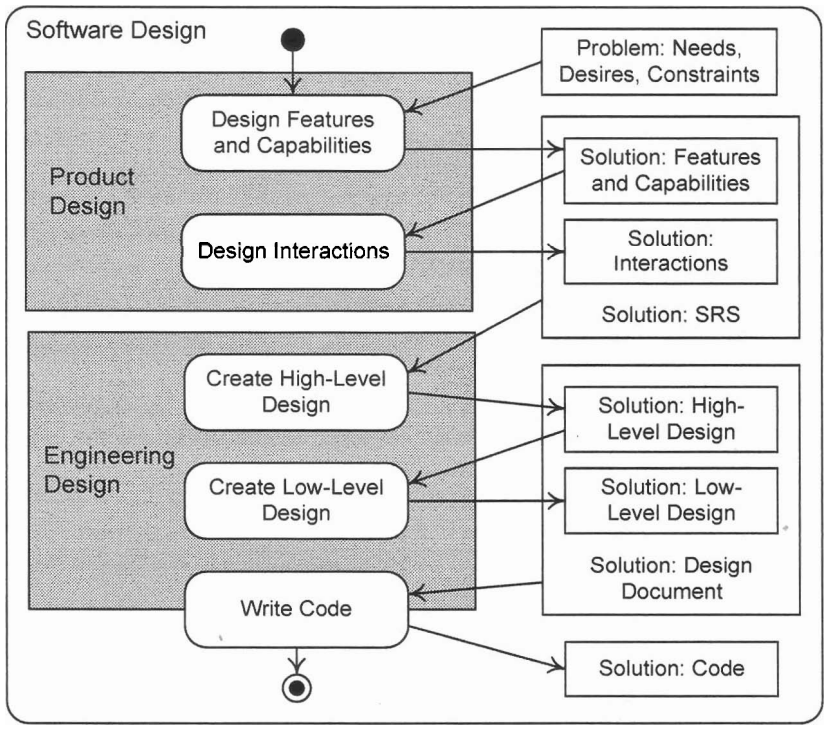

Figure 1-3-5 Software Design Problems and Solutions

"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 the than 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 sub-systems 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

- 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

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.

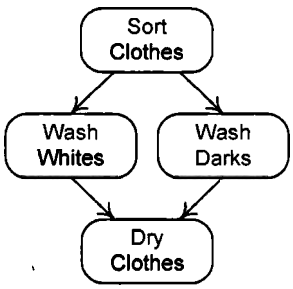

Figure 2-1-1 Action Nodes and Activity Edges

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 *activity symbols*. 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.

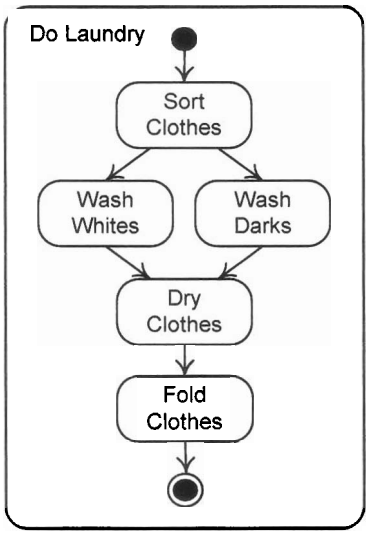

Figure 2-1-2 A Laundry Process

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.

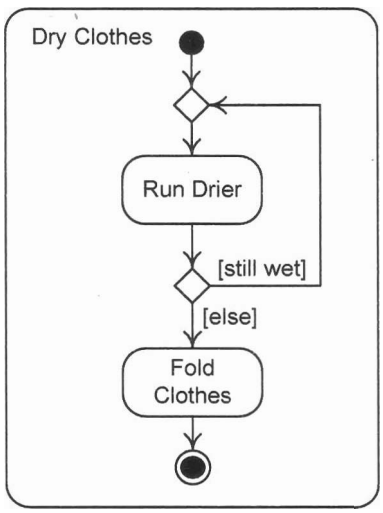

Figure 2-1-3 Drying Clothes in a Loop

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.

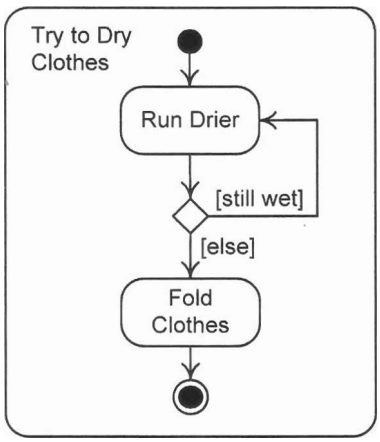

Figure 2-1-4 A Deadlocked Activity

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.

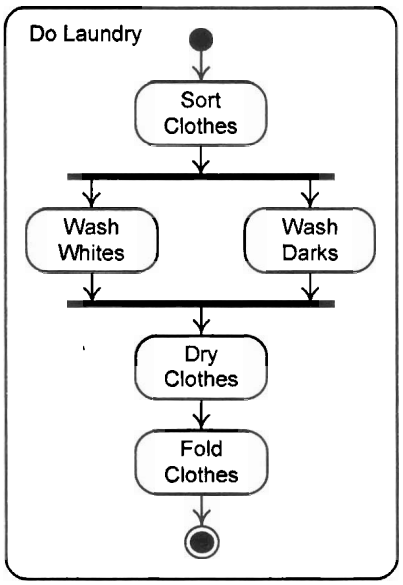

Figure 2-1-5 Laundry Process with Forks and Joins

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.

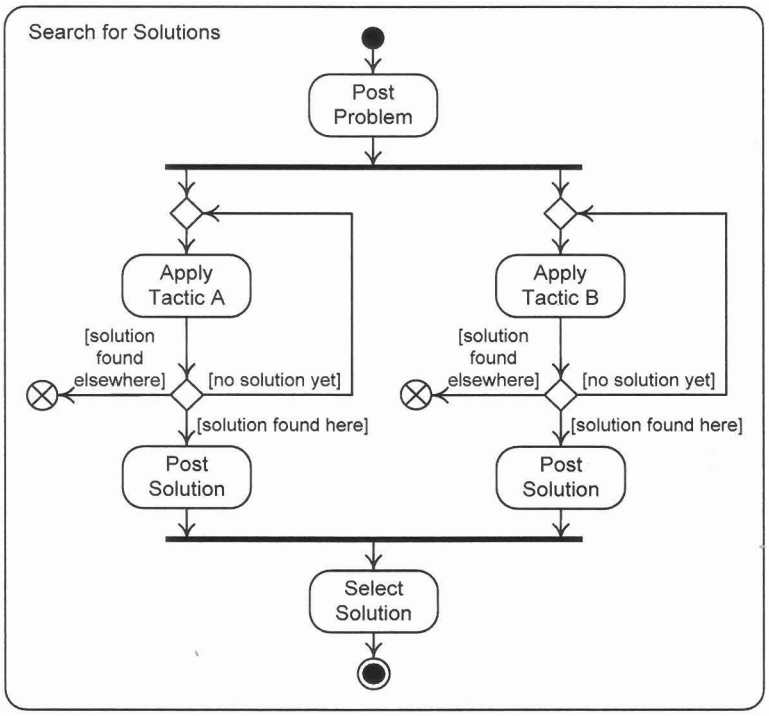

Figure 2-1-6 Concurrent Problem Solution

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.

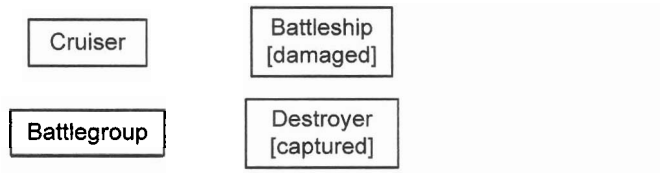

Figure 2-1-7 Object Nodes

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

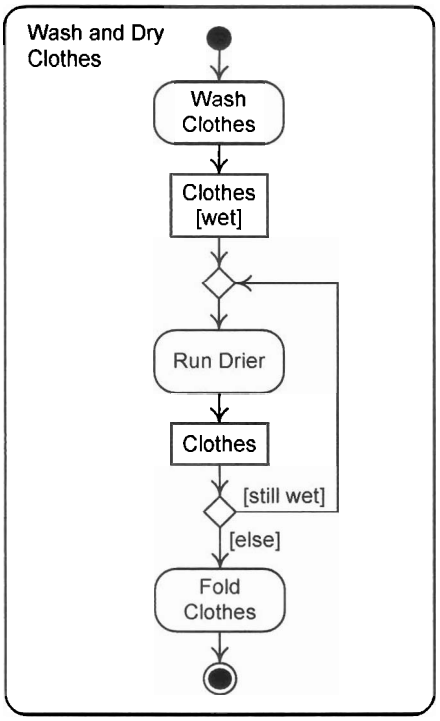

Figure 2-1-8 Data and Control Flows

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.

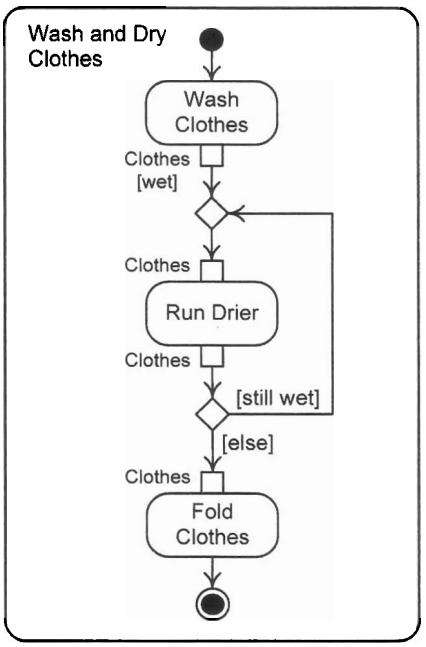

Figure 2-1-9 Pins and Data Flows

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.

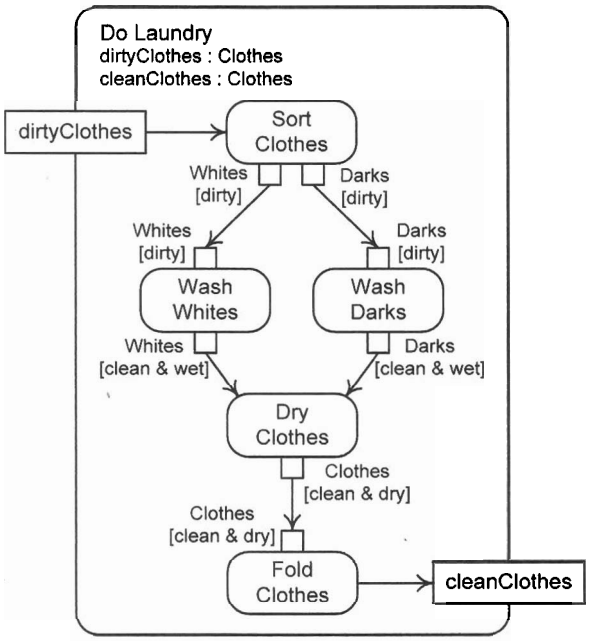

Figure 2-1-10 Activity Parameters

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.

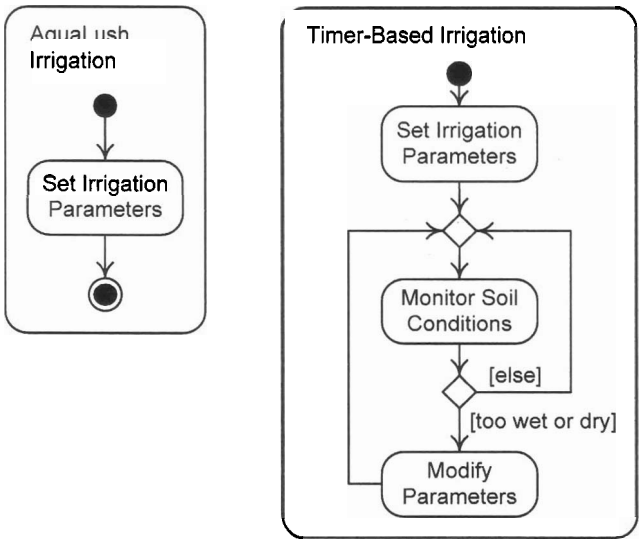

Figure 2-1-11 AquaLush Versus Conventional Irrigation

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.

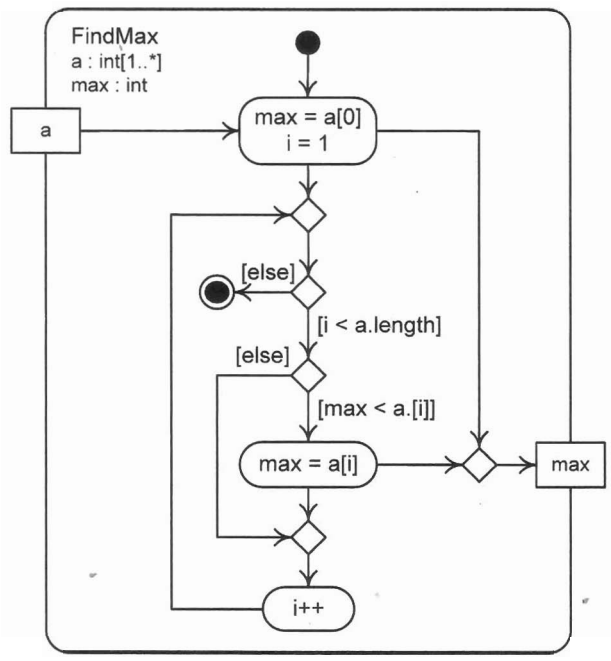

Figure 2-1-12 Maximum Finding Algorithm

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.

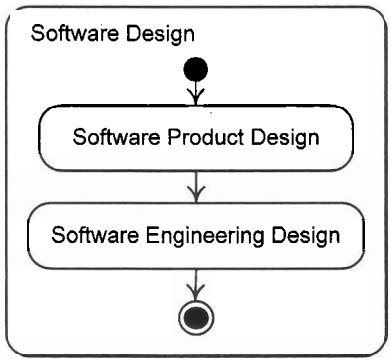

Figure 2-2-1 Software Design Process

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.

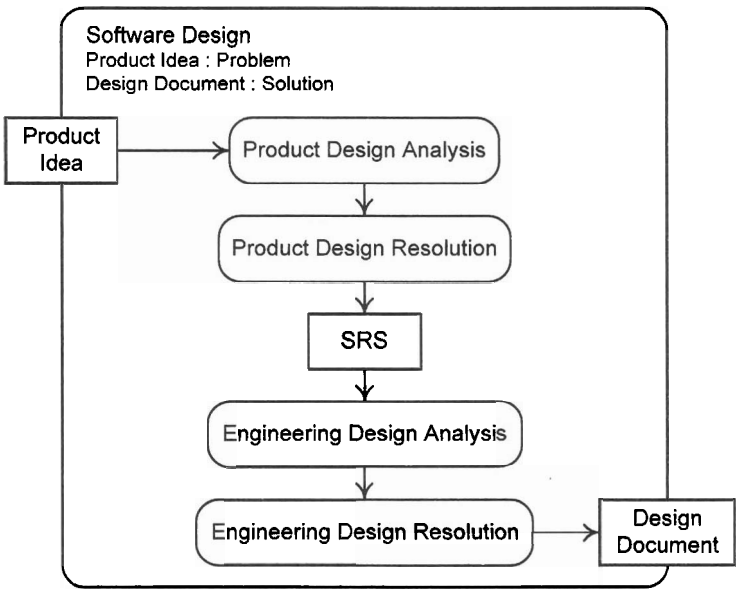

Figure 2-2-2 Analysis and Resolution in Software Design

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.

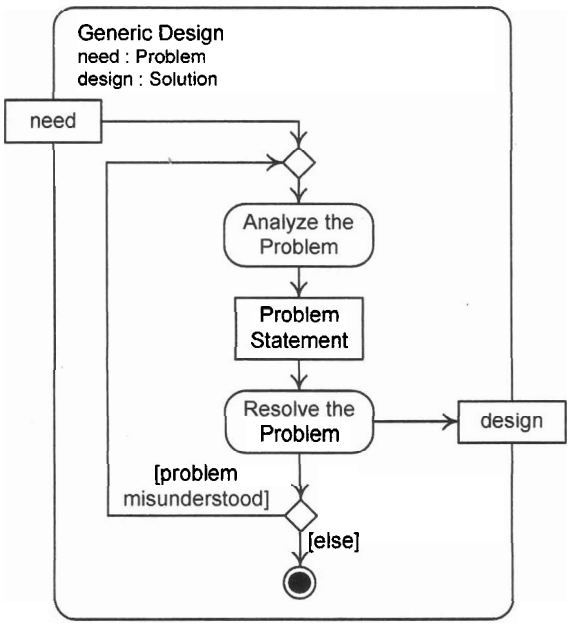

Figure 2-2-Generic Design Process

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.

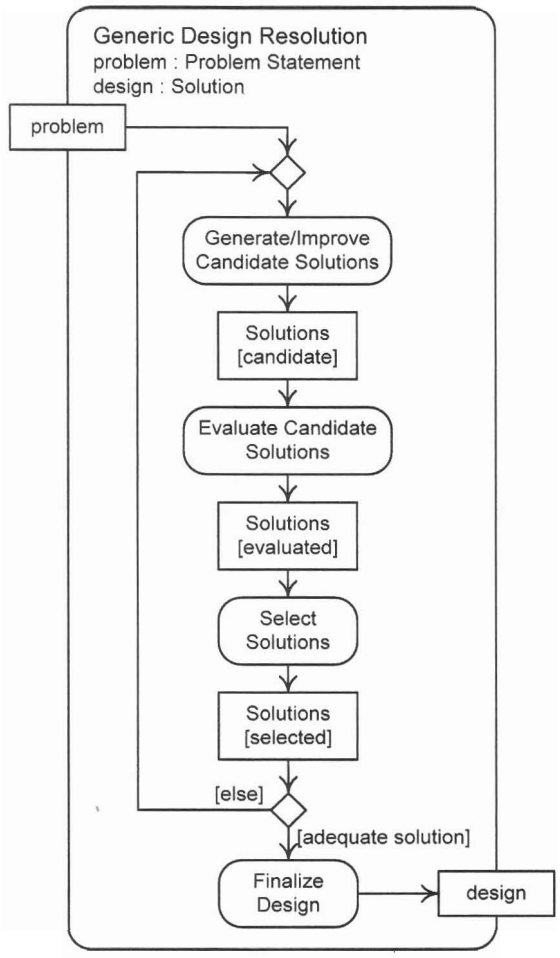

Figure 2-2-4 Generic Design Resolution Process

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.

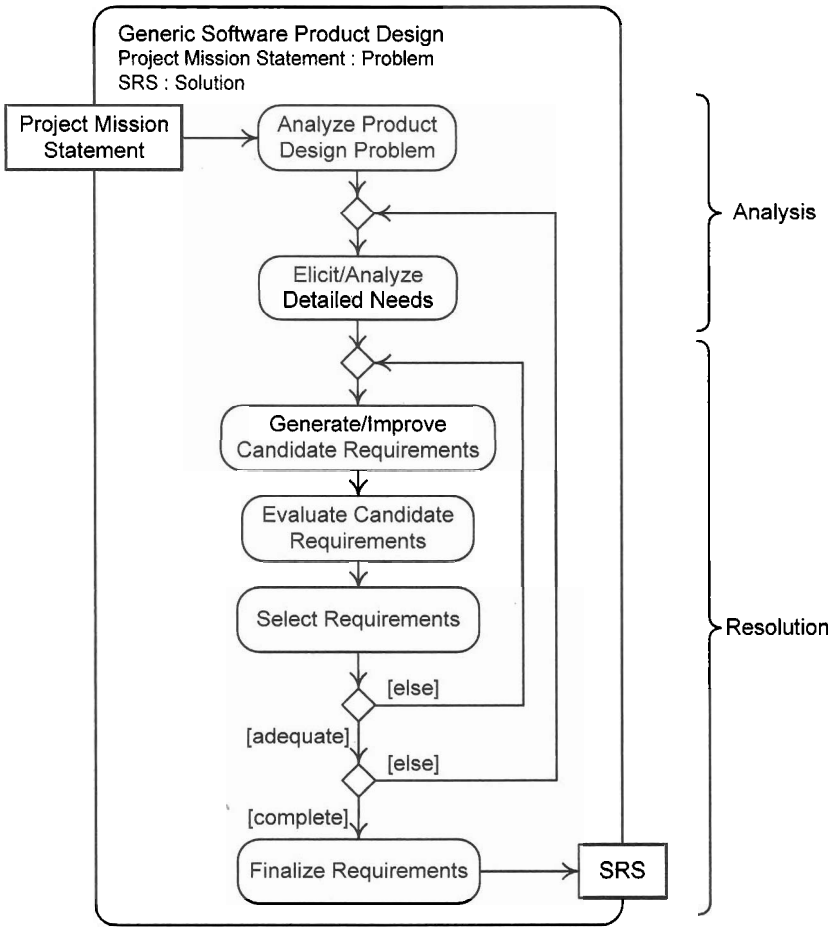

Figure 2-2-5 Generic Software Product Design Process

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.

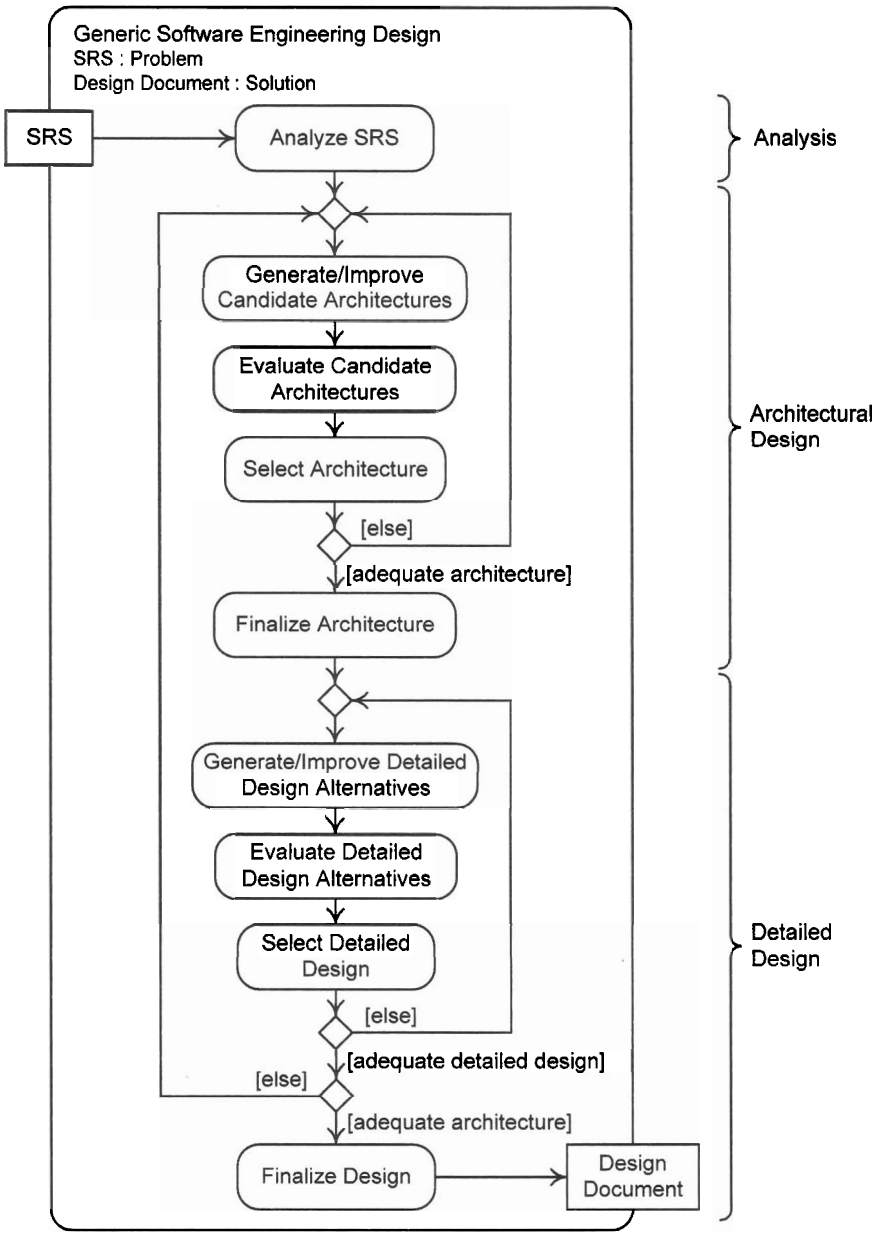

Figure 2-2-6 Generic Software Engineering Design Process

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 though 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.

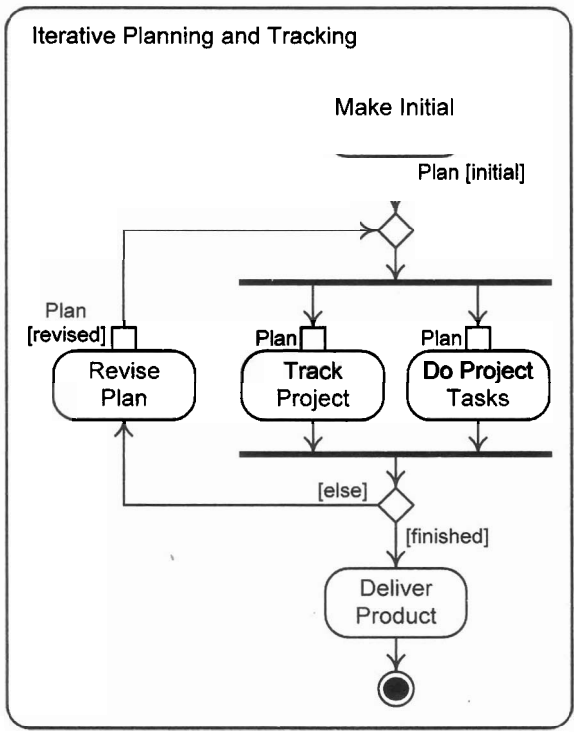

Figure 2-3-1 Iterative Planning and Tracking Process

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

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

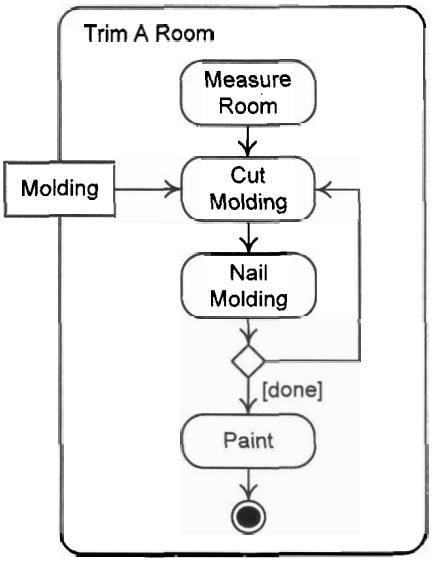

Figure 2-E-1 Activity Diagram for Exercise 1

2. Draw an activity diagram modeling the process of taking a shower between (but not including) undressing and getting dressed again.
3. Draw an activity diagram modeling the process of buying groceries, starting with making a shopping list.
4. Draw an activity diagram modeling the process of making a sandwich.
5. Draw an activity diagram modeling the process of buying a car.
6. Draw an activity diagram modeling the process of finding a book in the library and checking it out.
7. Draw an activity diagram modeling the process of making a schedule and registering for classes.
8. Draw an activity diagram modeling what happens when a Web browser starts up and loads its home page. Include object nodes in your diagram.
9. Draw an activity diagram modeling the process you went through to apply for college. Your diagram should have concurrent flows.
10. 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.

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.

Research Projects

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.

BLANK PAGE

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.

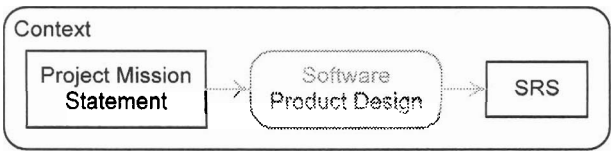

Figure 3-O-1 Software Product Design

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

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.

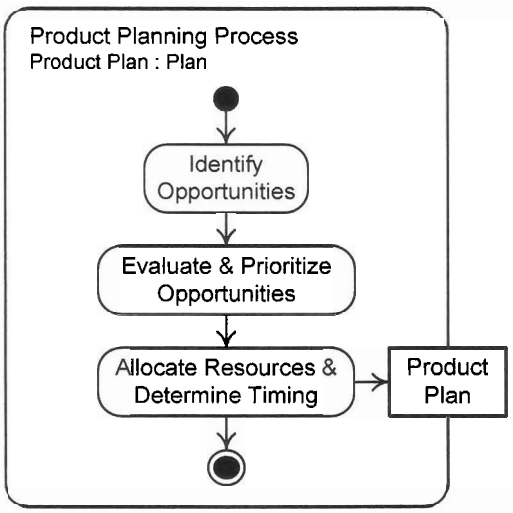

Figure 3-2-1 Product Planning Process

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.

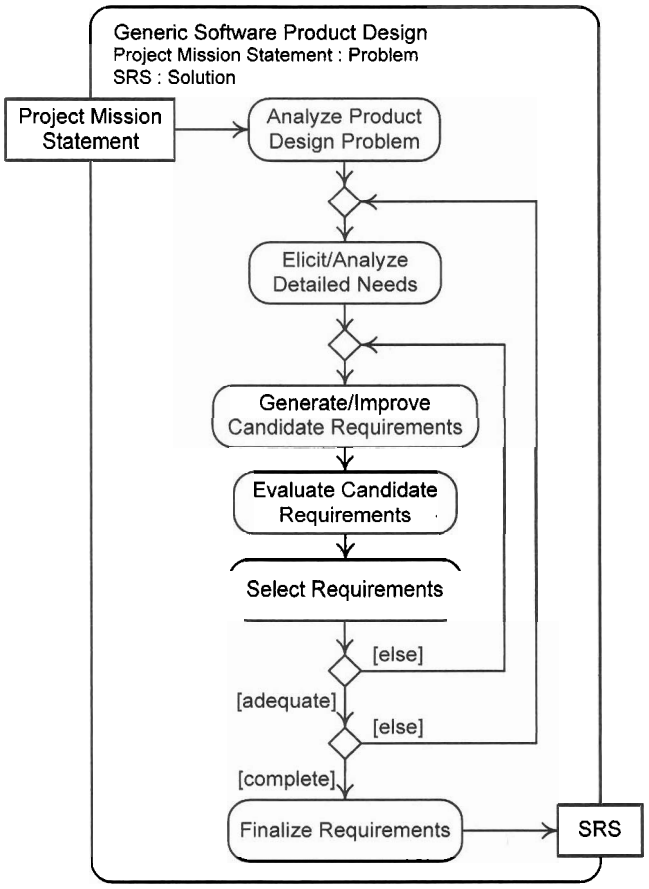

Figure 3-3-1 Generic Software Product Design Process

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.

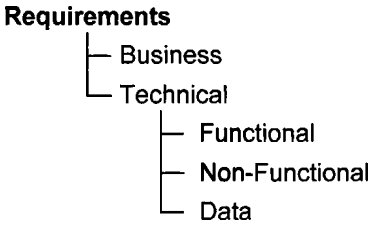

Figure 3-4-1 A Requirements Taxonomy

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?

Section 3.2

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.

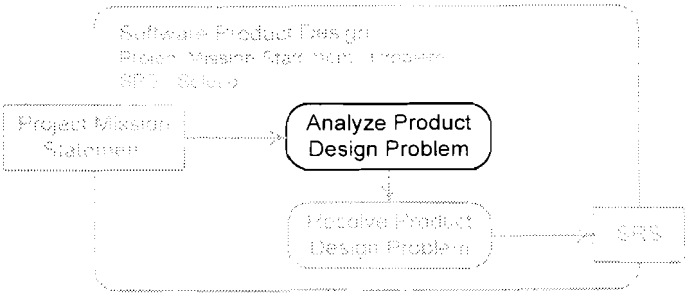

Figure 4-O-1 Software Product Design

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

Process Overview

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

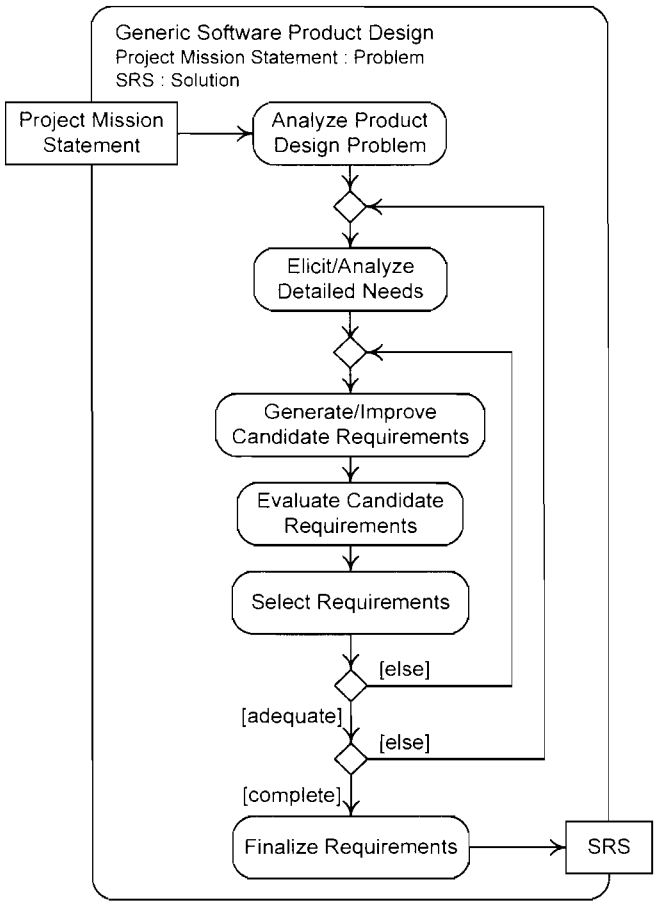

Figure 4-1-1 Generic Software Product Design Process

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 Vesus 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 ☐ Every stakeholder category in the stakeholders-goals list represents a group of legitimate stakeholders. ☐ Every need statement in the needs list accurately reflects the purported stakeholder's need. ☐ All need statement priorities are correct.
Scope ☐ All stakeholder goals are within the project scope. ☐ Every stated need is within the project scope.
Terminology ☐ Every specialized term is defined in the problem domain glossary. ☐ Specialized terms are used as defined in the problem domain glossary. ☐ Terms are used with the same meaning throughout. ☐ No synonyms are used.
Uniformity ☐ All need statements are at similar levels of abstraction. ☐ Similar items are treated in similar ways.
Completeness ☐ 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 goals. ☐ Every entity mentioned in the glossary and the needs list is included in the models. ☐ 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 understand 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.

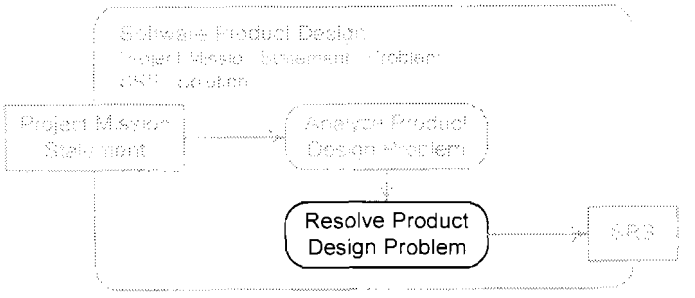

Figure 5-O-1 Software Product Design

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

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.

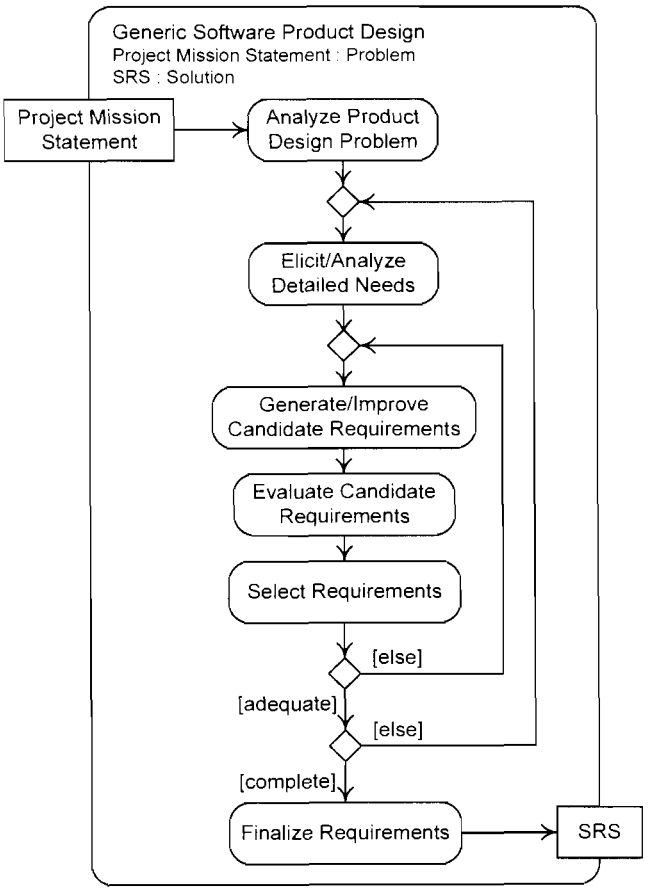

Figure 5-1-1 Generic Software Product Design Process

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.

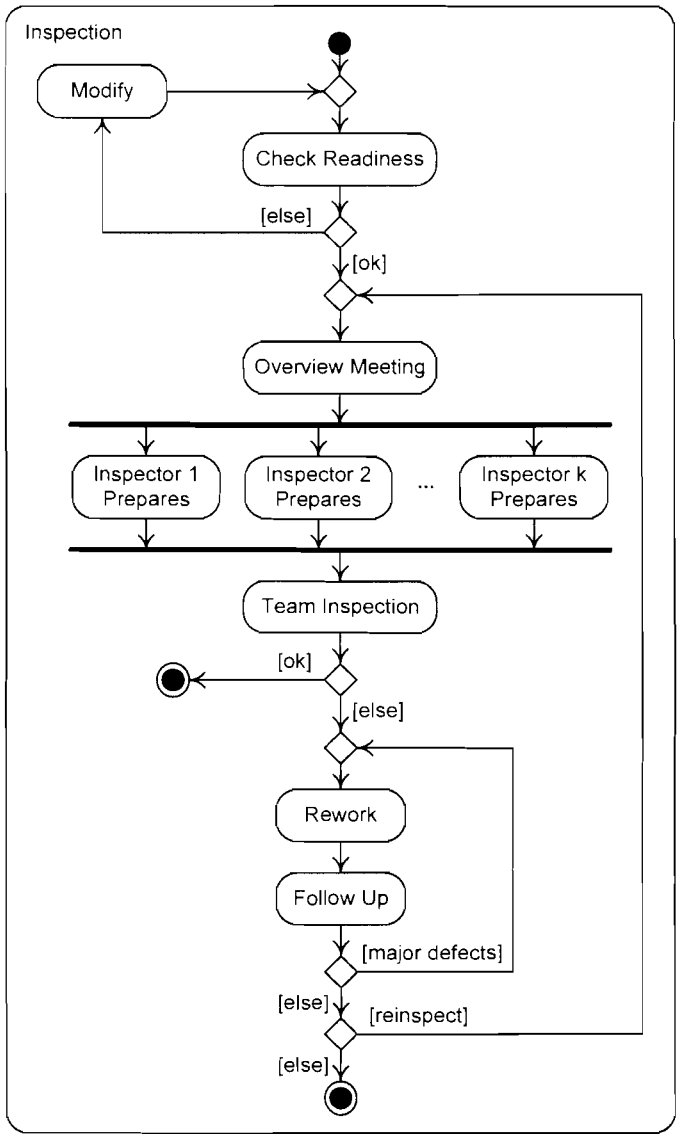

Figure 5-4-1 The Inspection Process

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.

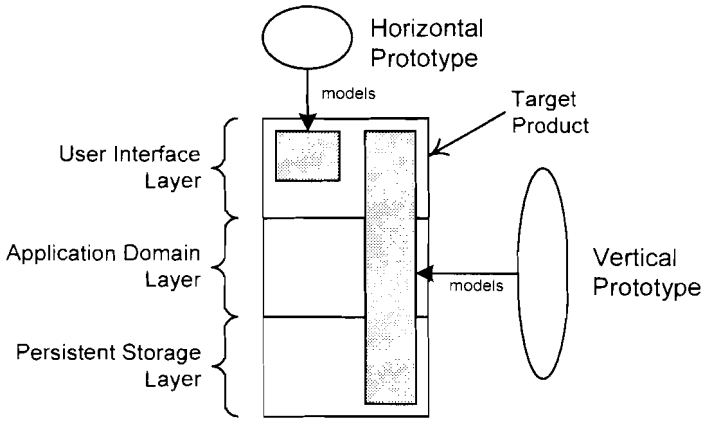

Figure 5-5-1 Horizontal versus Vertical Prototypes

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.

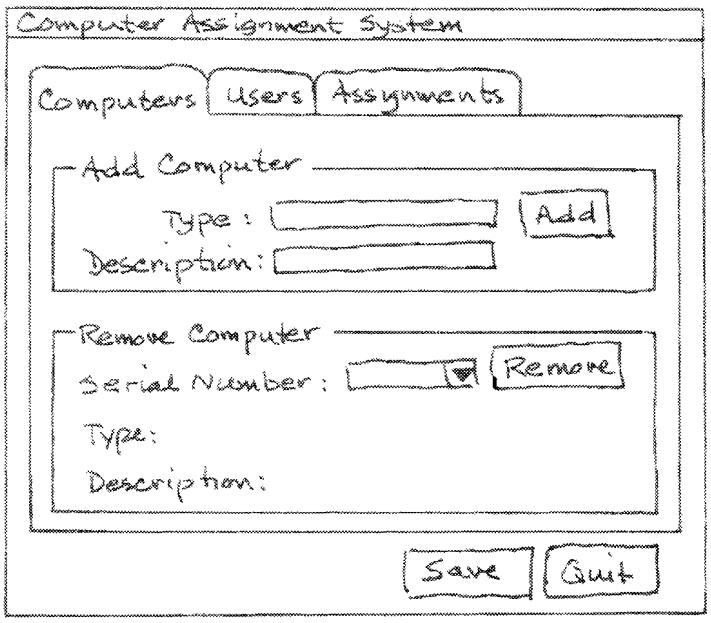

Figure 5-5-2 A Low Fidelity Prototype

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.

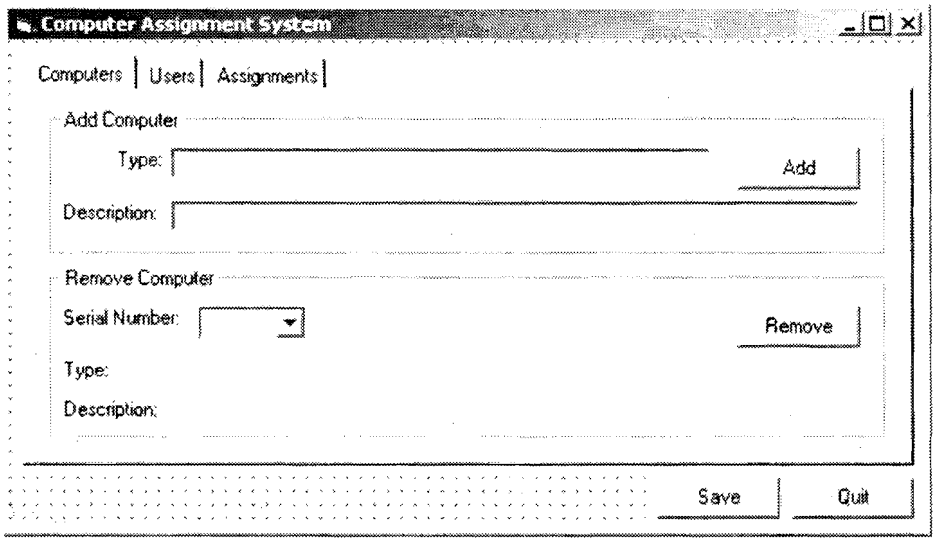

Figure 5-5-3 A High-Fidelity Prototype

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

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.

Prototyping for Requirements Generation and Refinement

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.

Prototyping for Requirements Evaluation and Selection

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.

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 [Weigers 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 Weigers [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.

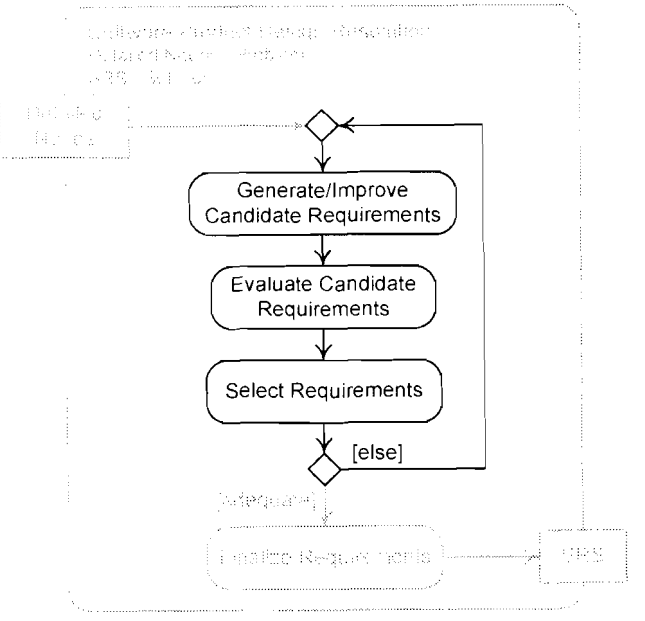

Figure 6-O-1 Software Product Design Resolution

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

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.

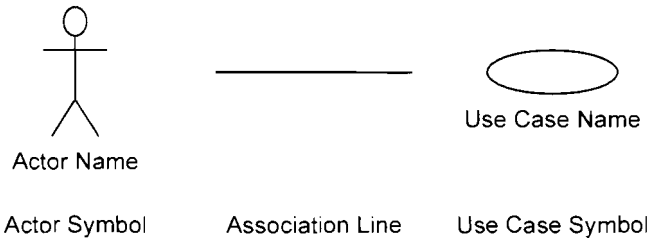

Figure 6-1-1 Use Case Diagram Elements

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.

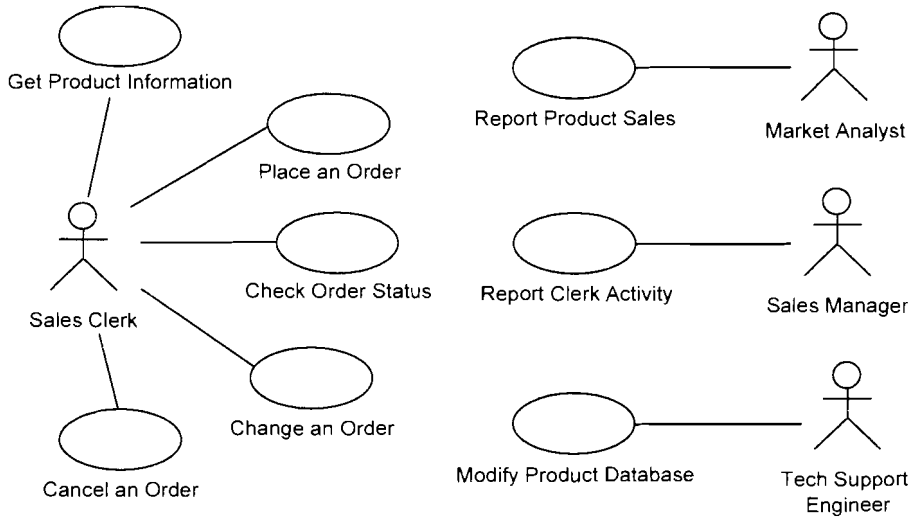

Figure 6-1-2 Sales Support System Use Case Diagram

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.

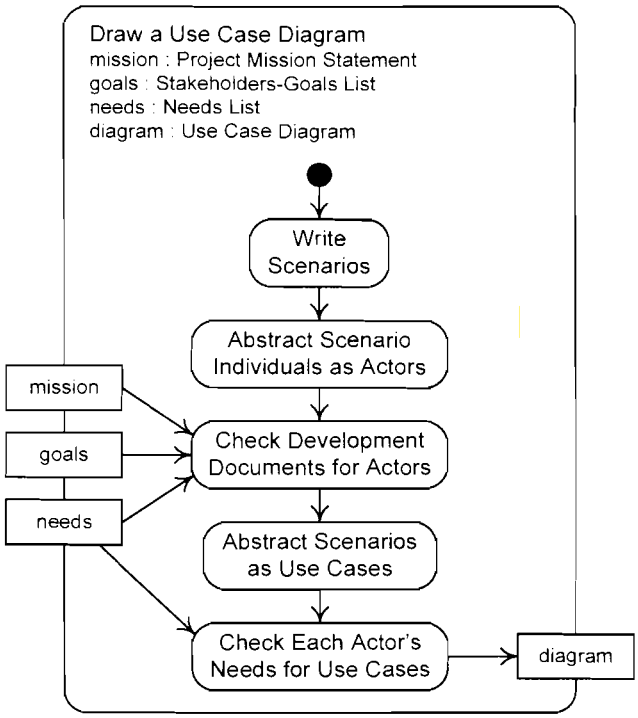

Figure 6-1-5 A Use Case Diagramming Process

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.

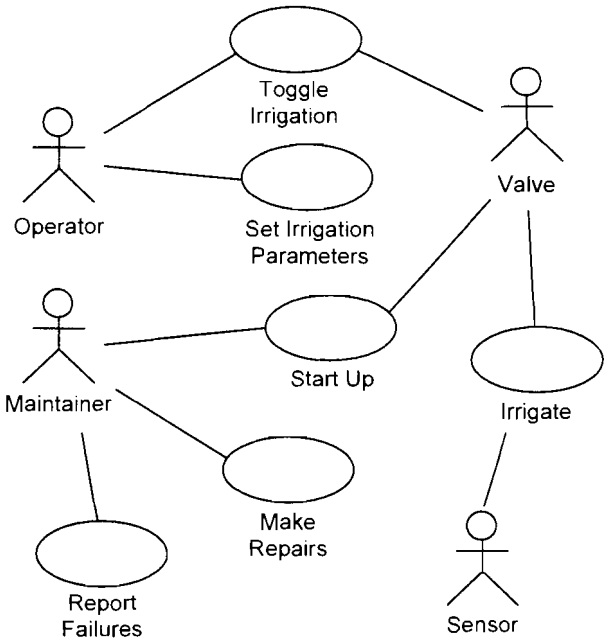

Figure 6-1-6 An AquaLush Use Case Diagram

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: name*
Actors: *actorList*
Stakeholders and Needs:
stakeholder—needsList.
...
stakeholder—needsList.
Preconditions: *what is assumed at the start.*
Postconditions: *what is guaranteed at the end.*
Trigger: *the event that starts the use case.*
Basic Flow:
 # stepDescription
...
Extensions:
extensionIdentifier condition:
extensionIdentifier # stepDescription
...
...

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

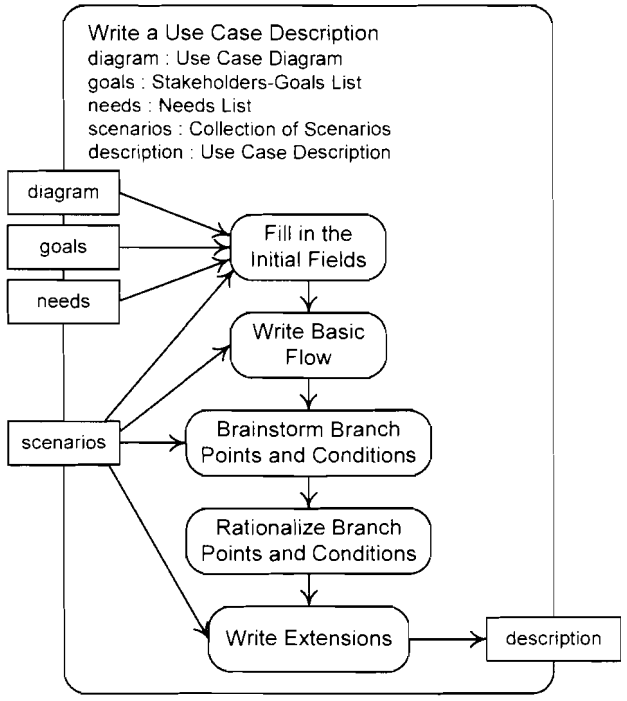

Figure 6-2-7 A Use Case Description Writing Process

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 is 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.

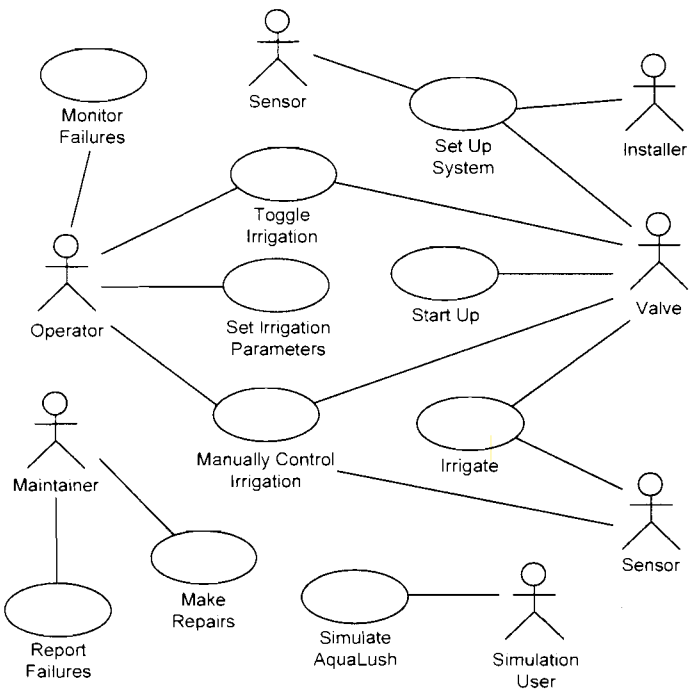

Figure 6-3-1 AquaLush Use Case Diagram A

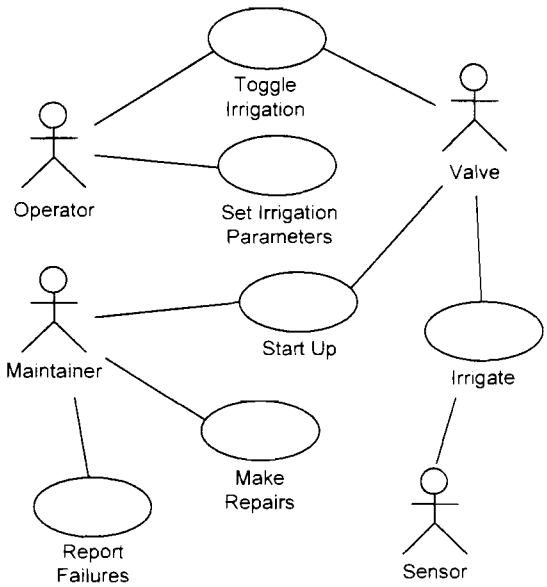

Figure 6-3-2 AquaLush Use Case Diagram B

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.

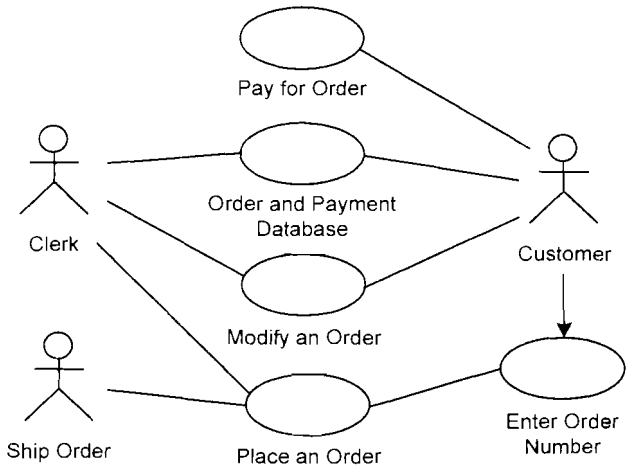

Figure 6-E-1 An Erroneous Use Case Diagram for Exercise 2

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 on purpose.
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.