Software Engineering Design

Theory and Practice

Software Engineering Design

Theory and Practice

Carlos E. Otero

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20120511

International Standard Book Number-13: 978-1-4665-6586-9 (eBook - PDF)

This book is dedicated to my wife, Kelly,

children Allison, Amanda, Michael, and Ashley,

and parents Angel L. Otero and Lydia E. Rivera.

Contents

Preface

This book is the result of an effort that I began in 2010 at the University of Virginia's College at Wise to create a course in software engineering design consistent with the 2004 IEEE/ACM curriculum guidelines for undergraduate programs in software engineering (SE). In a broad context, the recommended topics for undergraduate SE programs include design concepts, design strategies, architectural design, detailed design, human–computer interface design, and design evaluation. As a former industry practitioner, I learned first-hand the difference between hearing or "learning" about these topics and developing the necessary skills to apply them in a way that adds value to some development team, program, project, or business. With that in mind, I set out to compile material that I could use (from previous industry experience) to help students become proficient in designing software-intensive systems. Throughout the process, many of the original examples considered dry or hard to follow by students were replaced with new problem domains (e.g., gaming systems) that helped students assimilate the concepts better. Because of the "hands-on" approach required to master these concepts, the teaching style evolved to emphasize both theory and practice. The theory portion was used to present acceptable general design principles or a body of design principles to explain successful software systems' designs. The practice portion provided the avenue for transforming design theory into skills that can be employed directly to real-life industrial settings. The knowledge and experience gained from these efforts have been captured in this textbook, which can be useful for both industry practitioners and students in software engineering, computer science, and information technology programs.

INTRODUCTION

This book provides an introduction to the essential concepts employed by software engineers who design large-scale, software-intensive systems in a professional environment. It bridges the gap between industry and academia by providing students with a comprehensive view of software design using industry-proven concepts for designing complex software systems. Its unique blend of theory and practice provides both students and industry practitioners with key concepts that are immediately relevant to today's software designers. The book contains examples, review questions, and chapter exercises carefully selected to bring real-world problems into a classroom environment. More importantly, it incorporates an effective learn-by-doing approach that allows students to transform design theory into the skills required to design complex software systems. The book starts by providing a general overview of software design, including the fundamentals of software design, the importance of studying software design, and different practical concepts used for designing software.

As part of the introductory material, the software engineering process is covered briefly to provide the context in which software design takes place and a formal top-down design process is presented. The top-down approach consists of several design phases and activities that occur at varied levels of detail/abstraction, including the software architecture, detailed design, and construction design. As part of the top-down approach, detailed coverage of applied architectural, creational, structural, and behavioral design patterns is provided and a collection of standards and guidelines for structuring high-quality code is presented. The book also provides techniques for evaluating software design quality at different stages and much needed coverage of management and engineering leadership for software designers. This provides software engineers with the necessary management, ethical, and leadership knowledge required to build products for the public domain. The book also provides coverage of the software design document and other forms of documentation important during the design of software systems. Collectively, the book comprehensively introduces students and practitioners to software engineering design and provides the knowledge required to emerge and succeed as tomorrow's professional design leaders.

USE AS A TEXTBOOK

The textbook provides a comprehensive (sophomore-level) introduction to required concepts in software design. When used as textbook, instructors are encouraged to visit the textbook website to download slides, the solutions manual, and other exercises developed as part of the ongoing effort to improve education in software engineering design. The material presented in the book's 10 chapters can be easily extended to 16 weeks, especially when covering the topics of design patterns. A recommended approach includes conducting microdesign reviews, where students (or student groups) design, implement, and present their work regularly, while other students evaluate, critique, and provide peer-review comments. Because of the nature of the topics covered, students are expected to meet the following prerequisites:

- Introduction to Programming (with object-oriented language)
- Data Structures and Algorithms
- Introduction to Software Engineering

Ultimately, the most important feature of software designs is their applicability to build software; therefore, the course should require students to implement a large portion of the designs created as part of the course. A recommended approach (when possible) is to adopt a unified modeling language (UML) modeling tool capable of forward and reverse engineering and use the textbook as a guide for creating and assigning design problems centered around the topics discussed throughout the book, since they are essential to all software engineering students from ABET-accredited programs.

Acknowledgments

Many people have contributed to this effort in one way or another. First, I would like to thank Dr. Jacob Somervell from the University of Virginia's College at Wise, and Dr. Luis Daniel Otero from Florida Institute of Technology for providing the material for Chapters 9 and 10, respectively. Your work made the overall product complete, and many students and practitioners will benefit from your contributions. I also thank Dr. Ira Weissberger from Harris Corporation for his contribution on the section on software design documentation in Chapter 5 and Stefan Joe-Yen from Northrop Grumman Corporation for his contribution to the architectural patterns chapter. I would also like to thank the reviewers and staff at Taylor & Francis—in particular John Wyzalek—for their help throughout this long process. Finally, and most importantly, I thank my wife, Kelly, and children Allison, Amanda, Michael, and Ashley, for their sacrifices, patience, and support.

About the Author

Carlos E. Otero, PhD, is assistant professor in the College of Technology and Innovation at the University of South Florida Polytechnic (USFP). Prior to joining USFP, Dr. Otero worked as assistant professor of software engineering in the Department of Mathematics and Computer Science at the University of Virginia's College at Wise, where he created the software engineering design course for Virginia's first and (at the time of writing) only ABET-accredited BS in software engineering.

Prior to his academic career, Dr. Otero spent 11 years in the private industry, where he worked as design and development engineer in a wide variety of military computer systems, including satellite communications systems, command and control systems, wireless security systems, and unmanned aerial vehicle systems. Currently, he continues to consult with industry in the areas of requirements engineering, software systems design and development, quality assurance, and mobile systems engineering.

Dr. Otero received his BS in computer science, MS in software engineering, MS in systems engineering, and PhD in computer engineering from Florida Institute of Technology in Melbourne. He has published over 25 technical publications in scientific peer-reviewed journals and conferences proceedings. He is a senior member of the IEEE, a science advisor for the National Aeronautics and Space Administration (NASA) DEVELOP program, an active professional member of the Association for Computing Machinery (ACM), and a member of several journal editorial boards in technology and engineering.

1

Introduction to Software Engineering Design

CHAPTER OBJECTIVES

- Understand software design from the engineering perspective
- Understand the importance of software design in developing complex products
- Understand the issues that make software design challenging
- Understand the software design process and differentiate between its activities
- Become familiar with software design principles, considerations, and strategies

CONCEPTUAL OVERVIEW

Software design is an indispensable phase of the software engineering process for creating and evaluating software models that guide the construction effort for developing high-quality software systems on time and within budget. Conceptually, design is the process of transforming functional and nonfunctional requirements into models that describe the technical solution before construction begins. To achieve this, the concept of software design, its activities, and tasks must be well understood so that a problem-solving framework for designing quality into software products can be established. In today's modern software systems, there are numerous design principles, processes, strategies, and other factors affecting how designers execute the software design phase. When equipped with the proper design foundation knowledge, an understanding of the designer's roles and responsibilities can be acquired, allowing designers to become effective in designing large-scale software systems under a wide variety of challenging conditions. This chapter presents the fundamental concepts of software engineering design, within context, and provides the motivation for the rest of the book.

ENGINEERING DESIGN

Design is an integral part of every engineering discipline. Airplanes, bridges, buildings, electronic devices, cars, and many other products of similar complexity are all designed. In civil engineering, designs are used to specify detailed plans for developing physical and naturally built environments, such as bridges, roads, canals, dams, and buildings. In electrical engineering, designs are used to capture, evaluate, and specify the detailed qualitative and quantitative description of solutions for telecommunication systems, electrical systems, and electronic devices. In mechanical engineering, designs are used for analyzing, evaluating, and specifying technical features required to construct machines and tools, such as industrial equipment, heating and cooling systems, aircrafts, robots, and medical devices. In all other engineering disciplines, design provides a systematic approach for creating products that meet their intended functions and users' expectations. Formally, Dym and Little (2008, p. 6) define engineering design as

> A systematic, intelligent process in which designers generate, evaluate and specify designs for devices, systems or processes whose form(s) and function(s) achieve clients' objectives and users' needs while satisfying a specified set of constraints.

Design is a lengthy and complex process requiring significant investments in time and effort. So why conduct design in engineering disciplines? There are many possible answers to this question, stemming from simple common sense to more complicated ones involving professional, ethical, social, and legal implications. From the commonsense perspective, products of such complexity are hard to create, are costly to change, and, when built carelessly or incorrectly, can significantly impact human life. When working toward the creation of complex products, teams must organize in a disciplined manner, and a systematic approach needs to be employed to carefully ensure that products are built to meet their specifications. Consider the construction of a bridge that spans over a body of water and is required to support a particular weight, to maintain access to watercrafts navigating underneath, to withstand expected wind speeds, and to provide other features such as sidewalks—all while being bound by a schedule and budget. The successful construction of such a bridge is a nontrivial task and requires years of experience, formal education, and large teams collaborating together to achieve the construction goals. If constructed incorrectly, reconstructing the bridge can skyrocket from its original construction cost; worse yet, if defects are undetected, the bridge could collapse, resulting in the catastrophic loss of human life. Similar to the construction of the bridge, teams engineering other products, such as airplanes, watercrafts, medical devices, and safety-critical software systems, share comparable challenges, and failure of these products can also result in catastrophic events. In an engineering environment, before product construction begins, the design of products needs to be carefully and extensively planned, evaluated, verified, and validated to ensure the product's success. This is mainly achieved through design.

ENGINEERING PROBLEM SOLVING

Throughout the design process, designers are constantly engaging in problem-solving activities that are fundamental to all modern engineering projects. In a broad sense, engineers can be characterized as specialized problem solvers. Their work requires them to identify, evaluate, and propose solutions to complex problems (in particular domains) under tight project constraints. In some situations, engineers tackle problems that have never been solved before, creating challenges to meet not only functional aspects of products but also their established schedule and budget. Before engaging in more concrete design topics, a formal discussion on problem solving is necessary to identify fundamental concepts that are well understood by successful designers; these serve as basis for establishing a holistic problem solving framework that can be employed any time during design.

To become a good designer, engineers must be good problem solvers. This may require years of experience solving problems in a particular domain. In many cases, experience allows engineers to reuse already proven solutions across separate but similar problems. In other cases, where unsolved problems are encountered, designers are required to "think out of the box" and carefully craft a systematic approach for solving the problem in an acceptable manner, which may require problem classification, identification of the solution approach and type of adequate solution, and identifying the overall strategy for reaching its solution. In a general sense, problem solving during design occurs in three different states (Plotnik and Kouyoumdjian 2010):

- Initial state
- Operation state
- Goal state

Through these states, designers employ several techniques and strategies to create a landscape suitable for problem solving. The initial state is where problems are formulated and interpreted. In some cases, achieving full understanding of the problem is a problem itself. Once problems are well understood, designers move to the operational state, where thinking about the problem occurs and viable solutions come to light. Once an appropriate solution is identified, evaluated, and validated, designers move to the goal state, where a final solution to the problem is found, marking the end of the problem-solving process.

TABLE 1.1

Problem Classification

Problem	Description
Well-defined	Problem with clear goals and known constraints
Ill-defined	Problem with undefined or ambiguous goals and unknown constraints
Wicked	Problem with no definite solution; not understood until after the formulation of its solution

Initial State

Design problems are not all the same; they vary in size, complexity, and, based on these characteristics, the amount of time and effort required for their solution. In some cases, it quickly becomes evident that certain problems are harder to solve than others. When this determination is made, the strategy for the solution approach is adjusted to account for the additional complexity. Being able to differentiate between types of problem is crucial in helping designers account for the amount of effort, time, and risk associated with the solution approach. Therefore, an important problem-solving skill involves identifying and classifying the type of problem encountered, which includes *well-defined*, *ill-defined*, and *wicked problems*, as presented in Table 1.1 (Giachetti 2010).

Well-defined problems have clear defined goals and their constraints are well understood. This makes scoping the problem, proposing a solution approach, and arriving at the solution easier than with other types of problems, such as ill-defined and wicked problems. Ill-defined problems are problems where the mere interpretation of the problem is a problem itself; they are ambiguous with undefined goals and require more time and effort to clarify and interpret the problem to arrive at a solution. In some cases, with additional effort, ill-defined problems can be transformed into well-defined problems. Finally, wicked problems are problems where no single problem formulation exists. There may be many acceptable formulations of the problem and no definite solutions, and solutions are not deemed correct or incorrect but good or bad (Giachetti 2010). In many cases, wicked problems can lead to contradictive goals that need additional resolution before the problem solving can occur. When contradictive goals are present, providing a solution to one part of the problem results in the inability of solving other parts of the problem. In these types of problem, optimal solutions are hard to find, requiring additional struggle and collaborative brainstorming. Also, evaluation of alternative designs may require advanced techniques to determine the best course of action, which tends to require more time. In many cases, the solution to wicked problems is not known until after the problem is solved.

Operational State

The operational state of problem solving is where thinking about the problem solution takes place. It requires employing multiple techniques for problem solving such as using metaphors, decomposing problems into smaller, less complex problems (i.e., divide and conquer), reusing solutions (e.g., patterns), and so forth. In all of the techniques, designers

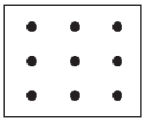

FIGURE 1.1
The nine-dot puzzle.

are expected to exhibit a "think outside the box" mentality to be able to solve complex problems. This requires shifting the mental model from a conventional approach to unconventional methodology where solutions to complex problems may arise from thinking in ways that deviate from conventional wisdom. For example, consider the popular nine-dot puzzle illustrated in Figure 1.1 (Kershaw and Ohlsson 2004).

The requirements for solving the nine-dot puzzle problem are as follows:

1. Draw four straight lines to connect all dots.
2. The pencil cannot be lifted from the paper once the line-drawing process begins.
3. No lines can be retraced.

Before moving on, think about this problem and attempt to provide a solution. At first, this may seem difficult because of the tendency of fixing the mental process to operate on the assumption that lines should begin and end on a dot. This *functional fixedness* limits the ability to find solutions based on objects having a different function from their usual ones (Plotnik and Kouyoumdjian 2010). In the case of the nine-dot puzzle, for some, functional fixedness makes it awkward or even impossible to propose solutions that involve lines going past the dots, which is what is required to solve this problem. To increase the chance of overcoming functional fixedness, problems need to be attempted several times and considered from many different viewpoints and unusual angles (Plotnik and Kouyoumdjian). Overcoming functional fixedness is critical for designers attempting to provide solutions at the operational state of problem solving.

Thinking about the Problem

Different types of thinking take place when finding solutions to problems. For example, when learning about a problem for the first time, problem solvers may begin by asking questions, which allows them to think about many different alternative solutions; as the problem-solving process moves forward, problem solvers can begin narrowing down the possibilities and think about the single best solution to the problem. These types of thinking are known as *convergent thinking* and *divergent thinking* (Table 1.2).

Both convergent and divergent thinking have significant roles in solving engineering problems. In many cases, problem solvers begin using divergent thinking with different levels of abstraction, and each level provides finer-grained solutions to the problem until convergent thinking can be employed to solve it.

TABLE 1.2

Types of Thinking

Type	Description
Convergent thinking	Type of thinking that seeks to find one single solution to a problem
Divergent thinking	Type of thinking that seeks to find multiple solutions to a problem

TABLE 1.3

Types of Problem Solution

Problem	Description
Algorithm	Fixed set of rules that lead to the solution of a problem
Heuristic	Rules of thumb (or procedure) that may or may not lead to the solution of a problem

Problem Solution

In many cases, determining the type of solution required for a given problem can reduce wasted time and effort spent in attempting to find a single, optimal solution. In such cases, designers can elect to seek approximate solutions—as opposed to optimal solutions—that are appropriate and acceptable for meeting project constraints. Determining the type of solution for a given problem can reduce time and budget required for building the system. Two types of solutions are *algorithms* and *heuristics*, as presented in Table 1.3.

Algorithms are step-by-step procedures for finding the correct solution to given problems. Algorithms do not normally involve subjective decisions or rely on intuition or creativity to find solutions (Brassard and Bratley 1995). For some types of problems, using algorithms to find solutions can be unrealistic, especially in time-driven, practical engineering problems. In these cases, heuristics provide a realistic approach for finding good approximations of the solution. In some cases, heuristics can lead to optimal solutions; in others, they can lead to solutions that are far from optimal or no solution at all (Brassard and Bratley 1995). Algorithms and heuristics are both used heavily in the design of engineering systems and determining their appropriateness for solving particular problems is essential to meeting other project demands.

Goal State

The goal state represents the final state of problem solving. It is where adequate solutions to given problems are determined. For many engineering problems, reaching the goal state is a nontrivial task that requires careful attention to all important aspects of the problem. The concepts of initial, operational, and goal state can be fused together to create a holistic problem-solving framework adequate to solving engineering problems at all stages of the development effort. The approach consists of the following tasks:

- Interpret problem
- Evaluate constraints
- Collaborative brainstorming

- Synthesize possibilities
- Evaluate solution
- Implement solution

The first task of the problem-solving approach involves *interpreting the problem*. This is where problem information is received and processed; problem classification is identified (e.g., well-defined, ill-defined) and activities are performed to formulate the problem. Interpreting the problem is a task performed during the initial state of problem solving. During the initial state, identification of stakeholders—persons, groups, or organizations that have direct or indirect stake in the problem and its solution—is essential. Once the problem is formulated, the *evaluate constraints* task is used to identify external problem constraints, which are negotiated, integrated, and used to set the bounds on the solution landscape. Once the problem and constraints are well understood, *collaborative brainstorming* can begin among problem solvers and stakeholders. During collaborative brainstorming, problem solvers use mostly divergent thinking to come up with alternative solutions that may bring to light new knowledge, which can trigger a transition back to the problem interpretation task. Once a set of acceptable solutions is identified, problem solvers *synthesize possibilities* to form the acceptable proposed solution to the problem. During this task, problem solvers shift from divergent thinking to convergent thinking to propose the best-known solution to the problem. The solution is shared and *evaluated* by everyone involved in the problem-solving process. Flaws in the solution may trigger a transition back to the collaborative brainstorming task; otherwise, implementation begins. *Collaborative brainstorming*, *synthesize possibilities*, and *evaluate solutions* are all tasks performed as part of the operational state of problem solving. During implementation, the proposed solution is executed until the problem is solved, which is a task performed during the goal state of problem solving. Together, these tasks are combined with other problem variables to provide a holistic approach to problem solving (Harrell, Ghosh, and Bowden 2004), as presented in Figure 1.2.

As seen in the figure, inputs are items that require processing during problem solving. Inputs come from many different sources and are interpreted and formulated for particular problems. They drive all activities by specifying the overarching need that promotes the execution of the problem-solving tasks. Constraints are external properties that come inherent with any problem and limit the solution approach. Outputs are the expected outcome in problem solving. In many engineering projects, merely coming up with the solution to a given problem is not enough, since the solution needs to be documented, formatted, decorated, specified in graphical model format, or placed under configuration management. Outputs coming out of the problem-solving process need to meet the appropriate standards as defined by the developing organization. The development organization may also set standards for activities, controls, and resources, which all impact problem solving. These variables are presented in Table 1.4.

Activities are internal tasks determined by the development organization that must be followed when solving problems. These are intended to help manage the problem-solving approach and may include review activities at different stages of problem solving, including preliminary and detailed stages, status reports, and documentation, which all impact the

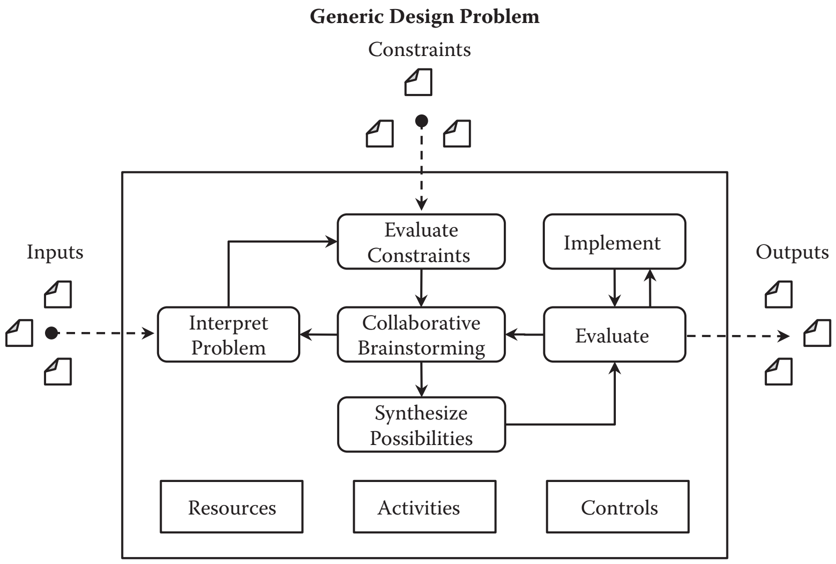

FIGURE 1.2
Holistic approach to problem solving.

TABLE 1.4
Problem-Solving Process Variables

Phase	Description
Activities	One or more tasks identified and required to solve the problem
Resources	Means by which activities are performed
Controls	Internal properties of the organization that place bounds on the solution, or the solution process, for the problem

time required to solve the problem. Controls, on the other hand, are internal constraints set by the development organization that limit the possible solutions so that they align well with the organizational goals and current practices. These controls can dictate when and where problem solving takes place, selection of strategies, permitted tools, personnel allowed to engage in problem solving, and measures for quality control. Finally, resources are the means by which activities are performed, which include people, software, and hardware, and their availability, which all impact problem solving. Together, all of these variables mix together to define the problem-solving landscape, which must be considered when tackling engineering problems.

Skill Development 1.1: Using the Holistic Approach in Problem Solving

Use and document all the steps of the holistic problem-solving approach presented in Figure 1.2 to solve the following problem. If possible, do this exercise as a team. The problem specification is as follows: there are six equal matches; connect each match to form four equilateral triangles. When done, explain how functional fixedness played a role in preventing you from arriving at the solution to this problem.

SOFTWARE ENGINEERING DESIGN

In the previous sections, design was introduced as a systematic and intelligent process for generating, evaluating, and specifying designs for devices, systems, or processes. To support this process, the problem-solving skill was identified as an essential ingredient for designing complex products. These discussions provided a general perspective on the importance of these concepts in the engineering profession. As in other engineering disciplines, design and problem-solving are crucial to the development of professional, large-scale, software systems. Software systems are highly complex, difficult to create, costly to change, and—depending on the software product—critical to human safety. Similarly to other engineering disciplines, designs in software engineering are used to identify, evaluate, and specify the structural and behavioral characteristics of software systems that adhere to some specification. Software designs provide blueprints that capture how software systems meet their required functions and how they are shaped to meet their intended quality. Formally, software engineering design is defined as

> (1) The process of identifying, evaluating, validating, and specifying the architectural, detailed, and construction models required to build software that meets its intended functional and non-functional requirements; and (2) the result of such process.

The term software design is used interchangeably in practice as a means to describe both the process and product of software design. From a process perspective, software design is used to identify the phase, activities, tasks, and interrelationship between them required to model software's structure and behavior before construction begins. From a product development perspective, software design is used to identify the design artifacts that result from the identified phase, activities, and tasks; therefore, these products by themselves, or collectively, are referred to as software design. Design products vary according to several factors, including design perspective, language, purpose, and their capabilities for evaluation and analysis. For example, designs can be in architectural form, using architectural notations targeted for specific stakeholders. These types of design can be presented using block diagrams, Unified Modeling Language (UML) diagrams, or other descriptive form of black-box design documentation. In other cases, design can be in detailed form, where a more white-box representation of the system is used to model structural and behavioral aspects. These can include software models that contain class diagrams, object diagrams, sequence diagrams, or activity diagrams. Other design products include models that represent interfaces, data, or user interface designs. Due to the many ways software design is used in practice, a common pitfall in software engineering projects is to associate design with a particular type of design artifact, therefore neglecting other forms of design or the activities required to create complete and correct software designs. Collectively, both process and products, including all variety of design products, are considered software design and are essential in most professional software projects.

WHY STUDY SOFTWARE ENGINEERING DESIGN?

On February 25, 1991, a software error on the Patriot missile defense system operating during operation Desert Storm caused it to fail to track and intercept an incoming Scud, which resulted in the death of 28 Americans (GAO, 1992). In 1996, a software error caused the Ariane 501 satellite launch to fail catastrophically, resulting in a direct cost of approximately $370 million (Dowson 1997). The software error that caused Ariane 501 to fail could be attributed to its software design. Similarly, the literature is swamped with many examples of disastrous results of software-based products. The reason for many of these disasters is that developing high-quality software on time and within budget is a daunting task. From the outset, the landscape for software development projects is plagued with a variety of challenges that increase complexity in software projects. Software design plays an integral part in managing the complexity and the challenges encountered in any software development effort.

During the software design phase, the system is decomposed to allow optimum development of the software; requirements are mapped to conceptual models of the operational software; roles are assigned to software teams on the same or remote sites; well-known interfaces for software components are created; quality attributes are addressed and incorporated into the design of the system; the user interface is created; the software's capability is analyzed; function and variable names are identified; design documentation goals are established; and the foundation for the rest of the software engineering life cycle is established. Given its impact on the creation and management of software products, mastering software design becomes essential to successfully engineer software products. The reasons for studying software engineering design can be described using a product development perspective and a project management perspective.

Reasons for Studying Software Design in Product Development

From the product development view, studying software design is important because designs form the foundation for all other software construction activities. Software designs allow software engineers to create models that represent the structure and behavior of the software system. Through these models, the main components and their interconnection for the solution are identified. Characteristics of quality code, such as modularization, cohesiveness, and coupling, are all born in the design phase. For complex tasks, abstractions and encapsulation are used in software design as means to provide a systematic approach for problem solving. In addition, software designs are reusable; therefore, they can be applied to different projects to provide ready-made solutions to common problems. Software design also provides the means to evaluate and incorporate the quality attributes necessary for software systems. Therefore, issues such as performance, usability, portability, and security can all be addressed early on in the development project. These benefits are carried over to all other subsequent phases of the software development life cycle and have direct impact on the implementation, testing, and maintenance phase.

Reasons for Studying Software Design in Project Management

Managing software projects characterized by changing requirements, tight schedules, cost constraints, and high expectations for software quality is tough. Among these, requirement changes are common drivers for all other project characteristics. This means that, as requirements change, projects should expect some impact in their cost, schedule, and quality. In some cases, requirement changes can easily translate to extended schedules and increased cost; in others, where schedules are not extended, requirement changes translate to decreased software quality. Good software design can minimize (or counter) the effects of requirements volatility in managing software projects. From the management's point of view, software design is important because it helps accommodate changes to the requirements or system updates, therefore minimizing impact on schedule, cost, and quality. In addition, good software design increases efficiency in human resource allocation tasks. By decomposing the software into independent units, resources can be assigned to software components so that they can be built in parallel in the same or different construction sites, therefore having significant impact on software schedules and cost. By compartmentalizing the design, the effects of unwanted employment attrition (i.e., employees leaving the company) can also be minimized, since new employees need only to take on the individual design component assigned to that employee. Good software designs provide an efficient mapping of customer requirements to software solutions, therefore facilitating requirements tracing throughout the design. Having a strong grasp on software design helps management abstract project tasks and acquire better appreciation of the work to be done. Overall, having a strong grasp in software design helps management improve the project planning, organization, staffing, and tracking and provide overall guidance for the project.

SOFTWARE DESIGN CHALLENGES

Today, the software design phase has evolved from an ad hoc and sometimes overlooked phase to an essential phase of the development life cycle. Furthermore, the increasing complexity of today's systems has created a set of particular challenges that makes it hard for software engineers to meet the continuous customer demand for higher software quality. These challenges have prompted software engineers to pay closer attention to the design process to better understand, apply, and promulgate well known design principles, processes, and professional practices to overcome these challenges. Some of the major challenges include requirements volatility, design process, quality issues (e.g., performance, usability, security), distributed software development, efficient allocation of human resources to development tasks, limited budgets, unreasonable expectations and schedules, fast-changing technology, and accurate transformation from software requirement to a software product. A brief discussion of these challenges is presented next.

Design Challenge 1: Requirements Volatility

A major reason for the complexity of software projects is the constant change of requirements. When designed properly, software can be modified or extended easily; however, when designed poorly, modifying software can become overwhelming and lead to all sorts of complex problems. Unlike the development of computer hardware, bridges, houses, or mechanical parts, software's very own nature allows itself to change to provide different or new functionality to systems. This same trait that makes software so desirable is what makes it also so complex. Although much effort is put into the requirements phase to ensure that requirements are complete and consistent, that is rarely the case; leaving the software design phase as the most influential one when it comes to minimizing the effects of new or changing requirements. Requirements volatility is challenging because they impact future or current development efforts. This forces designers to create designs that provide solutions to problems at a given state while also anticipating changes and accommodating them with minimal effort. This requires designers to have a strong understanding of the principles of software design and develop skills to manage complexity and change in software development.

Design Challenge 2: Process

Software engineering is a process-oriented field. Software processes allow engineers to organize the steps required to develop software solutions with schedule and cost constraints. Therefore, at the core of every software development company, there should be a sound, well-understood, and consistent process for software development. Processes can also be developed and customized for particular phases of the software engineering life cycle. In the design phase, software processes involve a broad set of activities and tasks that bridge the gap between requirements and construction while adhering to a set of project-specific (or company-specific) constraints. These activities include common ones, such as architectural and detailed design, as well as other supporting activities. These supporting activities include establishing a design review process, defining design quality evaluation criteria, evaluating design reuse, establishing design change management and version control procedures, adopting design tools, and allocating resources. In many cases, a company's design process is not well established, is poorly understood, or is approached with minimalistic expectations that ignore aspects that are essential to executing a successful design phase. Focusing design efforts on creating independent software products, such as a simple class diagram or user interface, while ignoring other design activities may create complexities later on during system's test and maintenance. The design process is challenging because essential design process activities are often overlooked, done in an ad hoc manner, or simply not done at all. In many cases, a well-established and well carried out design process serves an indication of future project's success.

Design Challenge 3: Technology

Software is meant to be everywhere. From health-care systems and education to defense and everyday ubiquitous devices, software is required to operate on a massive and always evolving technology landscape. Besides the operating environment, the technology for designing and implementing today's software systems continues to evolve to provide improved capabilities. Examples of these include modeling languages and tools, programming languages, development environments, design patterns, and design strategies. As new technologies emerge, software engineers are required to assimilate and employ them all at the same time. In some cases, emerging technologies do not completely replace old ones. Some software systems are required to interoperate with old legacy systems designed with older design methodologies. This results in software designers employing different design methodologies and technologies, all on the same software system. In other cases, design models need to be derived from existing code, modified, and made interoperable with newer technologies. This technology-driven aspect of the design phase creates a demand for capable software designers that can assimilate new technology quickly and effectively to succeed at designing software. The technology aspect of software design is challenging because it is fast and ever-changing; therefore, designers must keep abreast of the latest advances and become proficient in the application of these advancements while maintaining rooted in legacy technology.

Design Challenge 4: Ethical and Professional Practices

Designers create blueprints that drive the construction of the software. During this creation process, designers are required to determine how design decisions affect the environment and the people that use the software. In many cases, the software development process is traditionally carried out under tight schedule constraints. Inherently, all phases of the development life cycle suffer from this, including the design phase. This creates external pressures that can lead designers to deviate from the normal design approach to meet these demands, which can have catastrophic consequences. No matter how tight deadlines are, how much animosity exists within the design team, or how much other external/personal factors are brought into the design phase, software designers must exhibit strong ethical and professional practices to ensure that the systems they build are of highest quality and that all design considerations are properly evaluated. In many cases, this requires designers to exert strong leadership skills to influence and negotiate with stakeholders, motivate the design team, and lead the design process to accomplish the project's goals. Designers are also responsible for enforcing ethical guidelines during the design process; evaluating the social impacts of their designs in the public domain or in safety-critical systems; and to follow the appropriate professional practices to ensure success in the overall system. The ethical and professional practices aspect of software design are challenging because designers are constantly faced with numerous pressures from stakeholders that influence designers' decisions, most of which have consequences of social, ethical, or professional nature.

Design Challenge 5: Managing Design Influences

Designs are shaped by many different influences from stakeholders, the development organization, and other factors. These influences can have cyclical effects between the system and its external influences, such that external factors affect the development of the system and the system affects its external factors (Bass, Clements, and Kazman 2003). Managing these influences is essential for maximizing the quality of systems and their related influence on future business opportunities. Of specific importance are design influences that come from the system stakeholders and its developing organization.

Stakeholders

Designing software is a nondeterministic activity. If given the same task to different designers, different solutions will be proposed, each of them being perfectly acceptable (McConnell 2004). Now add to the mix the multitude of influences that come from different stakeholders, and you can easily get a variety of design alternatives for meeting a variety of stakeholders' concerns, all conflicting with each other. This creates a challenge when trading off design alternatives that meet all stakeholders concerns. Making such design trade-offs is difficult, especially on large-scale design efforts. Consider a project with multiple customers, each with conflicting goals affecting design decisions. In such projects, creating a design that sacrifices some desired customer capability but provides other desired properties, such as quick time-to-market, reliability, or lower cost, can lead to the development of a high-quality system that maintains acceptable levels of satisfaction among stakeholders. This is an example of how stakeholders affect design decision, and the design, in turn, influences the stakeholder goals (Bass et al. 2003). Managing stakeholders' influences is challenging because it requires designers to exert a high-level of communication, negotiation, and technical skills to ensure that design decisions are made to accommodate all concerns without negatively affecting the project.

Development Organization's Structure

The development organization's structure influences the development of software products, in particular, the design of those products. As example, consider the case of distributed software engineering. In today's global market, more and more cases of distributed software development are taking place. A wide variety of reasons exist for developing software at different sites. Consider companies that have sites in multiple states, where various levels of domain expertise are found at different sites. Or consider the case of software engineers resigning, creating a gap in the development team that is hard to fill with local resources. Finally, consider companies that simply want to reduce cost by hiring software engineers from different countries. These and many other reasons exist for having development across site boundaries. In each of these cases, the structure of the development's organization makes it complicated to, for example, coordinate design efforts, evaluate and discuss design alternatives, conduct peer reviews, and manage version control. In these cases, designers need to consider not only technical aspects of the design but also the distribution of employees, organizational goals, resource availability, and so forth. Designs that support integration of distributed expertise across sites can introduce capabilities for building new software products that could not be engineered otherwise. This in turn can influence the developing organization to target new areas of businesses, therefore allowing the software design to influence its business goals. Managing the influences of the development organization is challenging because it requires designers to span out of the technical domain to have a keen interest on the organization as a whole.

CONTEXT OF SOFTWARE DESIGN

In today's modern software systems, software design plays a key role in the development of software products; however, it is only one phase of the complete software engineering life cycle. To understand how design fits within the whole software engineering process, it is necessary to provide the appropriate context so that clear distinctions can be made between the different life cycle phases and an appreciation of the importance of software design activities and tasks can be acquired. For this reason, an overview of software engineering and its life cycle is required. Software engineering is defined by the IEEE (1990, p. 67) as

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
> (2) The study of approaches as in (1).

The fundamental software engineering life cycle phases include requirements, design, construction, test, and maintenance, as presented in Table 1.5.

The requirements phase is where stakeholders are identified and customer needs, wants, and the (often overlooked) nonfunctional requirements are determined (Laplante 2009). During this phase, requirements are analyzed in their raw form to address issues such as requirements that don't make sense, contradict each other, or are incomplete, vague, or just wrong (Laplante 2009); requirements are classified and prioritized; and the specification of the software system, which typically results in the production of a document, or its electronic equivalent is reviewed and validated (Abran, Moore, Bourque, and Dupuis

TABLE 1.5

Fundamental Software Engineering Phases

Phase	Description
Requirements	Initial stage in the software development life cycle where requirements are elicited, analyzed, specified, and validated
Design	The requirement's specification is used to create the software design, which includes its architecture and detailed design
Construction	Relies on the requirements' specification, the software architecture, and detailed design to implement the solution using a programming language; a great deal of design can also occur at this phase
Test	Ensures that the software behaves correctly and that it meets the specified requirements
Maintenance	Modifies software after delivery to correct faults, improve performance, or adapt it for a different environment

2005). Once the requirements for the system are specified, designing the system takes place, which is the main topic of this book.

The construction phase begins once the design phase has been executed and all requirements can be traced to a section of the software design models. The construction phase is where designs are implemented using the programming language of choice. In this phase, code is generated according to a style guide. In addition, the code is unit tested, debugged, and peer-reviewed; programming errors are detected, tracked, and resolved; code is managed by using change management and version control software; and, finally, code is prepared for delivery using a predefined set of conventions for formatting. The construction phase is tightly related to the design phase and in some cases (typically on smaller projects) the line dividing both phases can be hard to identify. There are several reasons for this, the main one being that detailed designs can be directly translated to code; therefore, software engineers tend to design and code at the same time. In other cases, where design and construction are clearly delineated by the process, it is common for some construction tasks, such as identifying appropriate class, function, and variable names, to be performed during detailed design. Finally, because many discoveries made well into the construction phase give rise to functionality that requires design work, engineers must iterate back and forth between construction and design activities. Once all the design artifacts are implemented with programming and all assigned requirements can be validated through execution of code during unit testing, the construction phase is complete.

The testing phase is typically the final step before the software goes out the door. The main purpose of the testing phase is to verify and validate the software to ensure that it meets the predefined functions and level of quality defined in the software requirement's phase. Formally, the IEEE (1990, p. 76) defines testing as

> (1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.
> (2) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item.

The software testing phase serves as a gateway between product development and product release. Therefore, verification and validation efforts need to be made to ensure that the software meets the specification and the integrity of the software can be assured under normal and harsh conditions. It is important to note that no desired quality attribute can be verified during testing if it hasn't been designed into the product first. Therefore, even though testing is typically credited for ensuring product quality, design is fundamental in supporting a successful testing phase. Once software is delivered, the maintenance phase begins to implement corrections, adaptations, or improvements to the software. Corrections are typically made on a smaller scale to rectify faulty behavior or output of the software. These typically do not require design work. However, for adaptations or improvements, design work may be required to accommodate the changes. Together, all phases of the software engineering life cycle work together to define the functions that the software must provide, to transform these functions into technical solutions, to implement those solutions, and to validate their implementation and ensure the quality of the system throughout future versions.

SOFTWARE DESIGN PROCESS

In the previous section, the design phase was briefly mentioned as a means for determining its place within the software engineering process. However, as it will be seen, the design phase incorporates many activities and tasks conducted by different teams and typically managed by personnel other than designers. This requires a formal process to ensure that the design phase is conducted properly and that it addresses all the concerns identified for the software system being built. Many processes exist to carry out phases, activities, and tasks throughout the software engineering life cycle, including the unified process (UP), Scrum, and the dynamic systems development method (DSDM) (Pressman 2010). What follows is a discussion on the software design process in terms of the fundamental activities and tasks required to build software products. These activities and tasks are essential and typically built into other formal processes such as the ones already mentioned. The hope is that by placing more emphasis on the fundamental activities and tasks and less on particular process approaches readers can obtain a more concise and understandable coverage of the topic.

In today's professional software engineering landscape, software engineers are being asked to build larger and more complex software systems in the same or different sites. Therefore, both design processes and artifacts are increasing in complexity. This means that it is not enough to know how to model structural and behavioral aspects of the system in the design phase, but it is also essential that software designers know about the particular process (e.g., UP, Scrum) required to manage, create, and control software design activities. Sommerville (2010) defines a software process as a set of activities that lead to the production of a software product. Similarly, a software design process is a set of activities and controls that specify how resources work together for the production of software design artifacts. The software engineering body of knowledge identifies two major activities for software design: software architecture and detailed design (Abran et al. 2005). These are the essential activities for managing the complexity involved in developing large-scale software systems. However, numerous other important activities are required for supporting the creation of architectural and detailed designs. Therefore, when planning and identifying an appropriate software design process, the effort required for these activities needs to be considered. In addition, because of the emphasis that some forms of design place on construction, the detailed design activity process can be modified to explicitly present the construction design activity that addresses design issues encountered during the construction phase. With this in mind, a holistic approach to software design, which includes architecture, detailed and construction design, management, and documentation, is presented in Figure 1.3.

As seen in the figure, software architecture is the first activity conducted in the design process. Architectural designs are elaborated through detailed designs, which are further elaborated through construction designs. All of these design activities need to be documented, and the process for design and documentation needs to be managed. Figure 1.3 also presents a necessary differentiation between the software design phase and the distribution of its activities throughout the software engineering life cycle. In some cases, the architectural design activities can begin during the analysis activity of the requirements

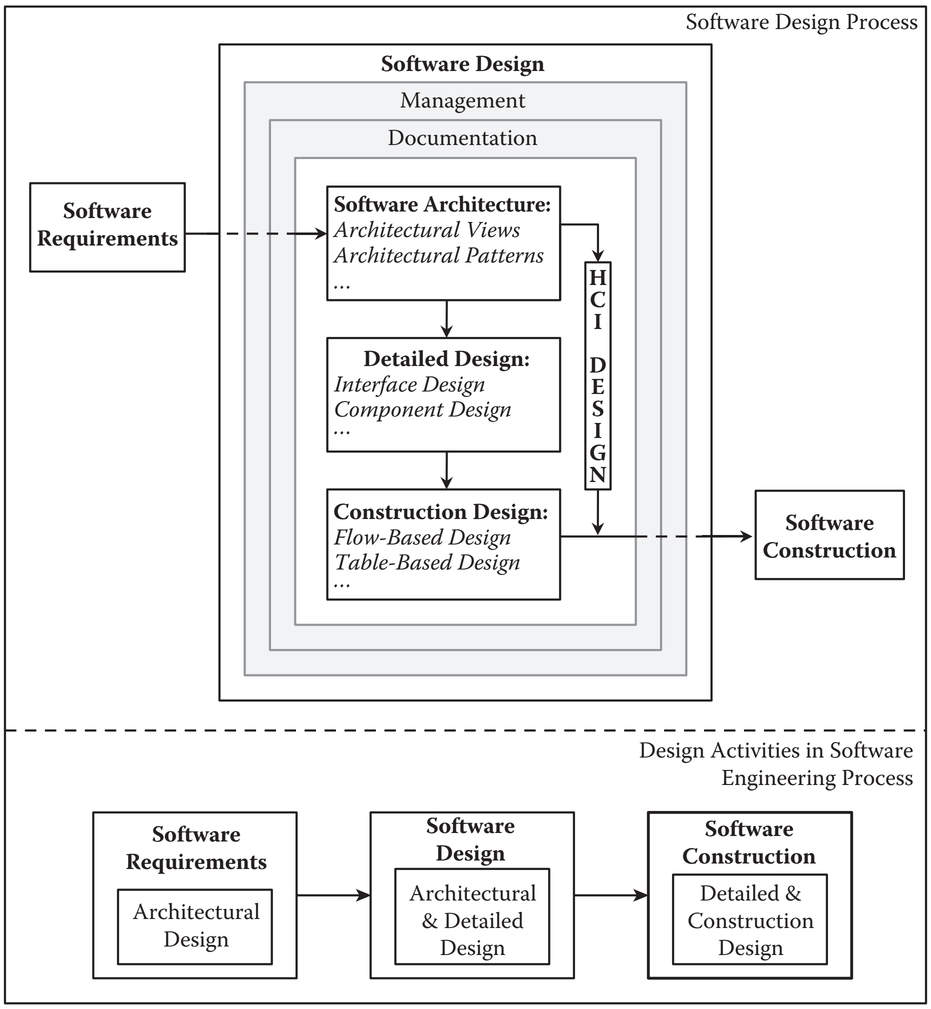
Software Design Process
Software Design
Management
Documentation
Software Requirements
Software Architecture: Architectural Views Architectural Patterns …
Detailed Design: Interface Design Component Design …
Construction Design: Flow-Based Design Table-Based Design …
HCI DESIGN
Software Construction
Design Activities in Software Engineering Process
Software Requirements
Architectural Design
Software Design
Architectural & Detailed Design
Software Construction
Detailed & Construction Design

FIGURE 1.3
The software design process and design activities during the SWE process.

phase and span through the design phase; in others, it begins after the requirements are specified and validated. In a similar fashion, the detailed design activity can start at the project's design phase and span through the software construction phase. These scenarios are highly project dependent; therefore, following to a strict waterfall-like process for software development is impractical for all but the simplest software applications.

Software Architecture

The software architecture activity corresponds to a macrodesign approach for creating models that depict the quality and function of the software system. It provides black-box models used to evaluate the system's projected capabilities as well as its expected quality, all from multiple perspectives. Therefore, architectural designs allow different stakeholders, with different backgrounds and expertise, to evaluate the design and ensure that the software architecture is addressing their concerns. For example, from the systems engineering perspective, architectural designs can provide information about the physical deployment of the system, including subsystems located at different locations, the artifacts executing in the subsystems, and how the system as a whole communicates. From the configuration management perspective, architectural designs can provide information about the hierarchy of files in the file system and how these files are interconnected to build and deploy the software system. From the software engineering perspective, different architectural designs can help decompose the software and define the major structural components of the system, identify interfaces between the components, map the requirements to them, evaluate concurrency issues, and provide overall insight into the design solution. A major benefit of architectural designs is their capacity to evaluate high-level concerns from stakeholders that deal mostly with nonfunctional requirements (e.g., performance, usability, security). For these purposes, architectural designs serve as important communication, reasoning, and analysis tools that support the development and growth of the systems (Bass et al. 2003). Software architecture lays the foundation for all subsequent work in the software engineering life cycle.

Detailed Design

The detailed design step begins after the software architecture is specified, reviewed, and deemed sufficiently complete for detailed design to begin. The detailed design activity builds on the software architecture to provide white-box design elements of the structure and behavior of the software system and in many cases is the last major effort before software construction begins. Detailed design is the activity that deals with refining the software architecture to reach a point where the software design, including architecture and detailed design, is deemed sufficiently complete for construction to begin. Whereas the software architecture places a major emphasis on quality (nonfunctional requirements), the detailed design activity places a major focus on addressing functional requirements of the system. In object-oriented systems, the detailed design activity is where components are refined into one or more classes, interfaces are realized, relationships between classes are specified, class functions and variable names are created, design patterns are identified and applied, and, if applicable, design tools are configured for code generation. Two major tasks of the detailed design activity are *interface design* and *component design*.

Interface Design

Interface design refers to the design activity that deals with specification of interfaces between components in the design (Sommerville 2010). Interface design can be focused on specifying the interfaces used internally within software components or externally across software components. In both cases, interfaces provide a standardized way for specifying how services are accessed and provided by software components. Interface design allows subsystems to be designed independently and in parallel; therefore, it is typically one of the first tasks performed as part of the detailed design. Other forms of interface design specify communication between systems, for example, custom binary or Extensible Markup Language (XML) messaging specifications used for communication between two or more subsystems through the network.

Component Design

During architecture, the software system is decomposed into logical components that abstract required system functions. During detailed design, these logical components are refined and their interactions are modeled to verify the validity of their structural composition. The execution of the detailed design activity requires a shift from the macrodesign approach to the microdesign approach to further decompose and refine system components into one or more fine-grained elements, functions, and data variables required for supporting the internal structure and behavior of components that meet assigned roles during the software architecture activity. Component design refers to modeling the *internal structure and behavior* of components—which includes the internal structure of both logical and physical components—identified during the software architecture phase. During this activity, fine-grained components are derived from the architecture, and their internal structure and behavior are designed. Components are not limited to object-oriented systems; therefore, component designs can be realized in many ways. In object-oriented systems, the internal structure of components is typically modeled using UML through one or more diagrams, including class and sequence diagrams. When modeling the internal structure of components, several design principles, heuristics, and patterns are used to create and evaluate component designs.

Construction Design

The idea of the detailed design activity is to get as close to the solution as possible without beginning the construction phase. In many cases, in object-oriented systems, this amounts to identifying classes, their attributes and functions, and interrelationships with other classes. These tasks are done while abstracting and deferring details of implementation to the construction phase. In some cases, however, implementing complex software functions identified during the detailed design activity requires additional design work to ensure they work properly and maintain the quality standards sought during the software architecture activities. In these cases, construction design is necessary. Construction design is not a new concept. Many other authors have proposed it as an important design activity. For example, McConnell (2004) specifies five levels of software design; one of them, being at the lowest level, deals with internal routine design. Similarly, Fox (2006) identifies a form of low-level design that fills the gap between detailed design and programming and deals with issues such as operation specification, including operation name, parameter types, and return types among others. Other authors, such as Meyers (2005), have highlighted the importance of designing code at low levels, during construction. Construction design is the last design activity—typically conducted during the construction phase—required to support the system's quality attributes, such as performance, maintainability, and testability.

Human–Computer Interface Design

The human–computer Interface (HCI) design activity is where general principles are applied to optimize the interface between humans and computers. Visual designs have a major role on the success or failure of software systems. Systems that meet functional requirements but that are not usable cannot succeed. The HCI design activity can be executed in parallel to the software architecture or detailed design activities. In some cases, HCI design is considered an architectural task, while in others it is considered a detailed design task. Regardless of where HCI design fits within design processes adopted by specific organizations, it is a major design activitiy that requires careful attention. The major concerns of the HCI designs may include the evaluation and use of modes, navigation, visual designs, response time and feedback, and design modalities, such as forms and menu-driven. HCI designs directly influence the quality of any system and are essential to understanding and addressing the factors that affect the overall usability of the system. Many design principles and evaluation techniques exist to succesfully design user interfaces.

Software Design Documentation

Similar to the specification activity of the requirements phase, software design documentation, also known as software design description (SDD), plays a big role in professional, large-scale, or software-intensive systems. Its importance is specified by the IEEE (1998, p. iii) as follows:

> SDDs play a pivotal role in the development and maintenance of software systems. During its lifetime, a given design description is used by project managers, quality assurance staff, configuration managers, software designers, programmers, testers, and maintainers. Each of these users has unique needs, both in terms of required design information and optimal organization of that information. Hence, a design description must contain all the design information needed by those users.

SDD should include the necessary information that properly captures the design of the system. As part of this activity, other issues such as tools for generating design documents, validation, and configuration management must be addressed. The software design documentation activity typically begins at the design phase and continues throughout the lifetime of the software system.

Software Design Management

Management plays a big role in software engineering projects. Griffin (2010, p. 5) defines management as

> A set of activities (including planning and decision making, organizing, leading, and controlling) directed at an organization's resources (human, financial, physical, and information), with the aim of achieving organizational goals in an efficient and effective manner.

In the design phase, management refers to the set of activities required to efficiently create and implement quality design artifacts, within schedule and budget constraints. This definition encompasses a broad set of activities that are particular to specific organizations. However, at the core of every organization's management activities, quality is a focal point. The quality of software designs can be assessed in various ways. From the management's perspective, quality of software designs can be evaluated in terms of cost and scheduling. From the engineering point of view, quality in designs can be evaluated using a set of well-known design principles as well as modeling and evaluating the quality attributes that the software must exhibit, which are specified via nonfunctional, quality requirements. From the configuration management's perspective, design quality can be achieved through change management processes that control how designs are created, modified, and improved. In large-scale software projects, software design management is essential to plan, organize, staff, track, and lead the activities required to carry out successfully the software architecture and detailed design steps.

ROLES OF THE SOFTWARE DESIGNER

From the discussions provided so far, it should be evident that designers are not all equal. In many design efforts, designers have different roles, with different titles and responsibilities that focus on specific design problems of the software system. There are many factors in place that determine the designer's role, including an engineer's work preference, experience, and capabilities. When studying software design, it is important to understand how these roles differ, the type of work performed, and capabilities required to perform the activities required of each role. In some cases, software designers are heavily involved in the requirements and construction phases; therefore, they must have expertise not only in design but also in requirements engineering and software construction. In other cases, a clear organizational delineation exists, allowing designers to focus on their area of expertise. A list of typical designer roles is presented in Table 1.6 (Giachetti 2010).

TABLE 1.6

Typical Roles in Software Design

Designer	Description
Enterprise architect	Designs the enterprise's strategy, processes, information, and organizational structure
Software architect	Designs software systems using a black-box modeling approach; concern is placed on the external properties of software components that determine the system's quality and support the further design of functional requirements
Component designer	Focuses on designing the internal structure of software components identified during the software architecture phase; has strong programming skills
User Interface designer	Designs the software's user interface; skilled in determining ways that increase usability of the system
System engineer	Designs systems using a holistic approach, which include designing how software and hardware collaborate to achieve the system's goals

Systems Engineer

The systems engineer designs the overall development process of systems as a whole, including processes for development of both the software and hardware that are part of the system. As a specialization of system engineering, software systems engineers design software at the system level; in many cases, the work performed by software systems engineers is similar to that of a software architect. Systems engineers work closely with customers to provide a holistic view of systems, their interfaces, and the distribution of requirements to subsystems. Software systems engineers are typically experts in the problem domain, and, depending on the type of system (e.g., embedded, web), they also develop expertise on other nonsoftware-related parts, such as hardware, communications, and avionics. This is essential at all phases of the software development process, since they must be able to communicate with other engineering disciplines, such as electrical, mechanical, and civil. In this role, designers have typically accumulated experience in other design roles, such as software architecture, component design, and in some cases construction. In addition to technical skills, systems engineers are required to have strong leadership skills to ensure the successful system development.

Software Architect

The software architect is in charge of designing the software architecture. Software architects can be found under a wide variety of titles, such as software lead, senior software engineer, or principal software engineer. Regardless of the title, software architects have extensive experience architecting systems that meet their intended requirements. Experience is typically acquired while moving up through the ranks, from software programmer all the way up to software architect. Software architects have strong leadership skills and are required to be skilled in initiation, communication, and negotiation. They also need to have a keen understanding of the developing organization to determine ways software systems can influence the organizational business goals and increase new business ventures leveraged from existing architectures. Other skills beneficial to software architects include project management skills.

Component Designer

Component designers are highly noticeable during detailed and construction designs, since they are typically the ones constructing the software. Therefore, they have strong programming skills and a strong foundation in design principles. For object-oriented component designers, strong object-oriented skills including knowledge of design patterns are essential. Component designers create both static and dynamic models of the software system at levels appropriate to drive construction; these include (when applicable) UML class diagrams and sequence diagrams. They have deep knowledge and understanding of the software requirements assigned to them; they are knowledgeable about other tools that support the design and development effort, such as modeling tools, integrated development environments, forward and reverse engineering, and configuration management. When designing at the component level, component designers have a full understanding of style guides for the project, since they dictate naming, spacing, and commenting conventions and other aspects that shape the structure of code. Component designers devise construction designs as needed and are proficient at creating effective unit tests that verify the quality of their product developed. Finally, component designers need to be comfortable scheduling and conducting peer reviews and accepting feedback and evaluating it objectively to improve their designs.

SOFTWARE DESIGN FUNDAMENTALS

Within the design process, many principles, considerations, and strategies help designers execute the software design process in an effective and consistent manner. For the most part, these help designers manage and simplify problems, consider the impacts of their proposed solutions, and establish a foundation for decision making during design. In this context, design principles refer to knowledge matter that has been found effective throughout the years in multiple projects on different domains. Design principles are applicable on most design projects; therefore, their use is expected to help achieve high-quality designs. On the other hand, design considerations are recommendations that help designers in the design process; they may or may not be followed. Finally, design strategies consist of tactical approaches in which design principles and considerations can be employed to drive the design process. These concepts are further discussed in the next sections.

General Software Design Principles

Throughout the history of software engineering, many design principles have emerged to become fundamental drivers for decision making during the software design process. These design principles are used as a basis for reasoning and serve as justification for almost all design decisions. They also provide designers with a foundation from which other more sophisticated design methods can be applied (Pressman 2010). These principles are not specific to any particular design strategy (e.g., object oriented) or process, so they are fundamental to all software design efforts and can be applied during architectural, detailed, and construction designs. The principles include (Abran et al. 2005):

- Modularization
- Abstraction
- Encapsulation
- Coupling and cohesion
- Separation of interface and implementation
- Sufficiency and completeness

Modularization

Modularization is one of the most important (and perhaps oversimplified) design principles in software design. Modularity allows software systems to be manageable at all levels of the development life cycle. That is, the work products of the requirements, design, construction, and testing efforts can all be modularized to efficiently carry out the operations. In the design phase, modularization is the principle that drives the continuous decomposition of the software system until fine-grained components are created. Modularization plays a key role during all design activities, including software architecture and detailed and construction design; when applied effectively, it provides a roadmap for software development starting from coarse-grained components that are further modularized into fine-grained components directly related to code. If applied properly, modularization can lead to designs that are easier to understand, resulting in systems that are easier to develop and maintain. Efficient modularization can be achieved by following and applying the principles of *abstraction* and *encapsulation*. With proper modularization, software systems can be decomposed into modules that allow the system's complexity to be manageable and allow the system to be efficiently built, maintained, and reused.

Abstraction

While the principle of modularization specifies what needs to be done, the principle of abstraction provides the guidance as to how it should be done. Modularizing systems in an ad hoc manner leads to designs that are incoherent, hard to understand, and hard to maintain. To modularize intelligently, a thorough understanding of abstraction is required (Liskov and Guttag 2010). Abstraction is the principle that deals with creating conceptual entities required to facilitate problem solving by focusing on essential characteristics of entities—in their active context—while deferring unnecessary details. When abstraction is applied, the level of detail required to think about a problem is adjusted to productively modularize a system; this allows for the creation of coherent entities that can be used to represent their possible variations in the problem's context and domain. The principle of abstraction can be applied iteratively at multiple levels during the design phase. At the software architecture level, abstraction helps during the identification of software components and their interfaces. At the detailed design phase, abstraction helps identify the entities, functions, and interfaces required to realize the component's provided services. At the construction level, abstraction helps in the further design of functions identified during detailed design. In all of these, abstraction is used to facilitate problem-solving by deferring details to later stages. The principle of abstraction can be classified as (Pressman 2010):

- Procedural abstraction
- Data abstraction

Procedural abstraction is a specific type of abstraction that simplifies *behavioral operations* containing a sequence of steps or other procedural abstractions. For example, consider a client–server application in which the client sends data to the server through the Internet. In this case, the *Send* procedural abstraction can be used to denote a series of operations, for example, retrieving the server's information (e.g., Internet Protocol [IP] address, port number), opening a connection, sending the message, and closing the connection. On the other hand, data abstraction is used to simplify the *structural composition* of data objects. Using the previous example, the *Message* data abstraction can be used to represent various messages with different attributes, such as the message's ID, content, and format. The definition of all of these properties can be deferred to later stages. Abstraction is fundamental for managing complexity in all activities of the software design phase.

Skill Development 1.2: The Abstraction Principle

The world is full of abstractions; without abstractions, communicating with our peers would be much more difficult. As an exercise, look for the nearest rectangular object that contains a knob and (maybe) a keyhole; if the object is blocking an entrance, change the state of the object so that it no longer blocks the entrance. Summarize this scenario by coming up with two abstractions: one data and the other procedural to increase communication with peers. When done, create a list of four other abstractions that surround you, and provide an abstraction as well as the detailed object description that would be required if the abstraction is not used. Ensure that there are two data abstractions and two procedural abstractions.

Encapsulation

In previous sections, modularization is presented as principle for decomposing monolithic systems into manageable units. While abstraction provides the principle for guiding the decomposition of the systems based on behavior and data, encapsulation provides the principle for enhancing the efficiency of the collaboration among modularized units. Encapsulation is the principle that deals with providing access to the services of conceptual entities (e.g., modules, components) by exposing only the information that is essential to carry out such services while hiding details of how the services are carried out. While abstraction is employed to find conceptual entities, encapsulation enforces that abstracted entities communicate between each other using a "need to know only" basis. When evaluated this way, the abstraction design principle helps create the modules and the encapsulation design principle enforces efficient communication between them. These principles are all essential in achieving efficient modularization. The relationship among modularization, abstraction, and encapsulation is presented in Figure 1.4. As seen, after the principle of abstraction is applied, the encapsulation principle is used to hide irrelevant details from the abstraction. In Figure 1.4, the shaded region corresponds to information that is irrelevant to other modules, while the white region corresponds to access points that modules can use to interoperate.

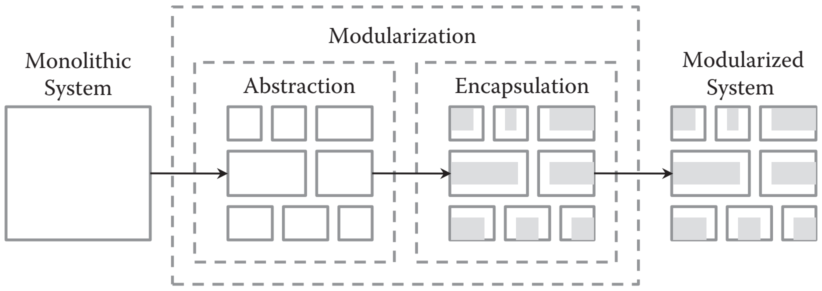

FIGURE 1.4
The modularization, abstraction, and encapsulation principles.

Coupling

Similar to abstraction and encapsulation, coupling and cohesion are design principles that lead to efficient module creation by emphasizing on the degree of dependency and belonging of modules, respectively. Formally, the IEEE (1990, p. 22) defines coupling as

> The manner and degree of interdependence between software modules.

Like all other design principles discussed so far, coupling can be applied during software architecture, detailed design, and construction design to measure the degree of dependency of design units, such as an architectural subsystem, a class in a detailed design's class diagram, or a function in code. In other words, the coupling principle can be used to determine how much an architectural subsystem depends on other architectural subsystems, how much a class depends on other classes, and how much a function depends on other functions. When measuring coupling, the number of dependencies between design units does not tell the whole story, since the nature of the dependencies plays an important role in decision making. For example, design units can depend on well-defined and stable interfaces, common data structures, and internal structure of other design units. It is not hard to support the idea that dependencies on well-defined and stable interfaces are less troublesome than dependencies on the internal structure of other design units. Three common types of coupling are

- Content coupling
- Common coupling
- Data coupling

Content coupling represents the most severe type of coupling, since it refers to modules that modify and rely on the internal details of other modules. *Common coupling* refers to dependencies based on a common access area, such as a global variable (IEEE 1990). When this occurs, changes to the global data area causes changes in all dependent modules. This type of coupling results in lesser severity than content coupling; however, it shares many of the undesired effects as content coupling. Finally, *data coupling* refers to the type of dependency in which design units communicate with each other only through a set of data parameters. Unlike content coupling, data coupling does not depend on the internals of other design units, and unlike common coupling it provides more control over the form of dependency. When dependency between modules relies on data parameters that are globally inaccessible, design units are shielded from undesired changes to the data by other design units. In all cases, a high degree of coupling gives rise to negative side effects. For example, as coupling increases, reusability and manageability of the design units decrease since errors or changes to the independent unit propagate to all dependent units. In other cases, when coupling increases, so does the complexity of managing and maintaining design units. Other types of coupling include *control coupling*, *hybrid coupling*, and *pathological coupling* (IEEE 1990).

Cohesion

While coupling gives insight to a design unit's degree of dependency, cohesion provides insight into its strengths. The IEEE (1990, p. 17) defines cohesion as

> The manner and degree to which the tasks performed by a single software module are related to one another.

Cohesion measures how well design units are put together for achieving a particular purpose and can be classified based on the measurement approach as

- Functional cohesion
- Procedural (or sequential) cohesion
- Temporal cohesion
- Communication cohesion

Functional cohesion measures a design unit's strength by the degree to which its tasks, operations, or subunits all contribute to perform a single function. When the function to be performed has a single logical meaning, functional cohesion can be seen as a form of logical cohesion. A highly functionally cohesive module is one whose internal details work toward achieving the same function. Functional cohesion is the most typical type of cohesion. *Procedural cohesion* measures the strength of a design unit by the degree to which its tasks work procedurally (in steps) to achieve the unit's purpose. Therefore, functional and procedural cohesion are not mutually exclusive; that is, modules can exhibit both high functional and procedural cohesion. *Temporal cohesion* measures strength by the degree to which all tasks in a design unit are performed at specific times. Consider a design unit responsible for carrying out the initialization of a system. This unit may be responsible for performing a power-on self-test that may include memory tests, file system checks, and communication checks. These are all different functions but need to be executed at the same time during initialization; therefore, the unit is temporally cohesive. Finally, *communication cohesion* measures a unit's strength by the degree to which its tasks produce or consume the same data.

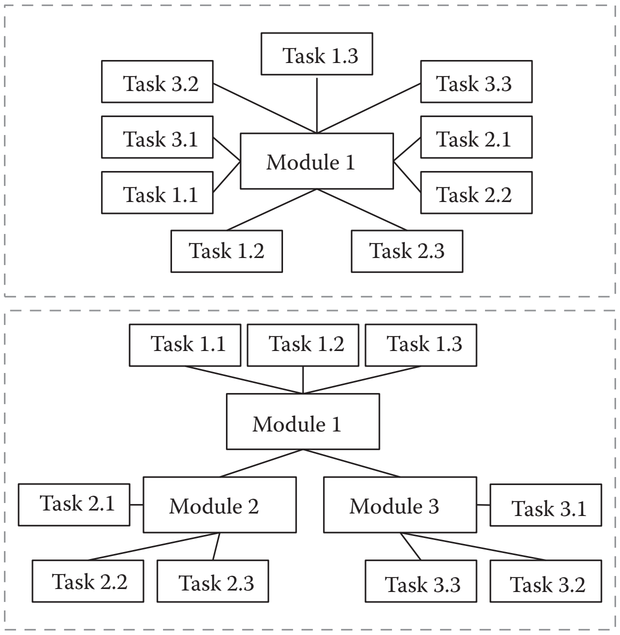

FIGURE 1.5
Example of principles of coupling and cohesion.

Cohesion provides an important principle that measures how much design units that are grouped together actually belong together based on different criteria. Cohesion can also be seen at different levels of the design process. During the software architecture activity, logical and communication cohesive modules are typical, whereas, during the detailed and construction design activities, functional, procedural, and temporal cohesiveness are more expected. In all cases, highly cohesive modules increase reusability. An example of the cohesion and coupling principles is presented in Figure 1.5. As seen in the top part of the figure, *Module 1* performs three unrelated different tasks (i.e., Task 1, Task 2, and Task 3), each requiring three independent subtasks. For example, *Task 1* requires three different subtasks, denoted by the labels *Task 1.1*, *Task 1.2*, and *Task 1.3*. As seen, *Module 1* has dependencies to nine different unrelated tasks, which can translate to a high degree of coupling and low degree of cohesion. The bottom part of Figure 1.5 shows how the system is decomposed into three more cohesive units, each with lower coupling than the original approach. In this case, the system is transformed to a modular system with higher cohesiveness and lower coupling. With this transformation, *Module 1* now has five dependencies and stronger functional cohesion. *Modules 2* and *3* have lower coupling than *Module 1* (both in its original and improve form) and are highly cohesive.

Separation of Interface and Implementation

The principle of separation of interface and implementation deals with creating modules in such way that a stable interface is identified and separated from its implementation. This design principle should not be confused with encapsulation. During encapsulation,

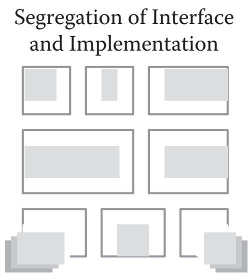

FIGURE 1.6
Principle of segregation of interface and implementation.

interfaces are created to provide public access to services provided by the design unit while hiding unnecessary details, which include implementation. While encapsulation dictates hiding the details of implementation, the principle of separation dictates their separation, so that different implementation of the same interface can be swapped to provide modified or new behavior. Figure 1.6 presents these concepts.

As seen, the bottom design units have separated interfaces; therefore, varied implementations can be employed without changes to a unit's interface and, subsequently, to dependent units. There are many benefits from this principle, including increased extensibility, reusability, and maintainability. Since implementation is compartmentalized, new capabilities can be added simply by including a new variation of the implementation without changes to old implementations. Also, in this way specific implementations can be reused.

Completeness and Sufficiency

The principles of completeness and sufficiency deal with efficient module creation. *Completeness* is a characteristic that measures how well design units provide the required services to achieve their intent. For example, during the detailed design activity, a communication class can be considered complete for a particular application if it provides services for establishing and terminating connections, sending and receiving messages. Missing any of these services would render the class incomplete. On the other hand, sufficiency measures how well design units are at providing only the services that are sufficient for achieving their intent. Consider the same communication class, which can include services for logging statistics, visualization of network activity, or any other capability applicable to the communication task. Although these capabilities enhance the class' service list, the class is considered sufficient by providing the required services of opening/terminating connections, sending, and receiving messages. That is, these sets of services are sufficient to achieve the unit's required functions, nothing more and nothing less.

Practical Software Design Considerations

Design principles are well-known throughout the software engineering community and are applied in one way or another in most projects. However, other considerations need to be made to provide the appropriate context in which these principles can be successfully applied for developing high-quality software systems. These considerations are discussed in the next sections.

Design for Minimizing Complexity

Design is about minimizing complexity. Every decision that is made during the design phase must take into account reducing complexity (McConnell 2004). In fact, the majority of design principles (e.g., modularization, abstraction, encapsulation) are meant to reduce complexity in one way or another. By doing this, details of the problem solution can be pushed further down the process, where they can be appropriately handled. As another example, consider HCI design: it is all about reducing complexity for the user. Finally, code design is about reducing complexity for other developers maintaining the software. As rule of thumb, when faced with competing design options, always choose the one that minimizes complexity.

Design for Change

As stated before, software will change; therefore, design with extension in mind. There are numerous reasons for this; for example, customers who like the software may want to extend its functionalities. On the other hand, customers who are discontent with the software may want to replace or remove functionality. In other cases, hardware changes may trigger a software change; advances in communications may cause software to change; or, simply, newer, better software technology becomes available triggering a change of software that introduces no new functionality but a more maintainable development technology that is supported by current practices. In any case, software will change; therefore, its very own nature requires software designers to plan for the future. A variety of techniques is available during the detailed design phase to achieve this.

Software Design Strategies

Throughout the years, a wide variety of strategies for designing software has been proposed. Some of these include structured design, object-oriented design, aspect-oriented design, data component-based design, and data structure-based design. Two popular strategies are discussed in the following sections.

Structured Design

In a broad context, structured design refers to any disciplined functional design approach where software systems are decomposed into independent, single-purpose modules, using an iterative top-down approach. The main focus of structured design is on the functions that systems need to provide, the decomposition of these functions, and the creation of modules that incorporate these functions. Structured design approaches are typically employed after structured analysis, where the main purpose is to derive a structure chart (i.e., software architecture) from data flow diagrams. Structured design introduced many benefits; for instance, by decomposing the system into independent, single-purpose modules, programs were simpler to understand, manage, code, debug, and reuse (Stevens 1981). However, structured design does not address the issues of data abstraction and information hiding and "is largely inappropriate for use with object-based and object-oriented programming languages" (Booch 1994, p. 22).

Object-Oriented Design

Unlike structured design, which focuses on functional decomposition of systems, object-oriented design focuses on object decomposition. Formally, the IEEE (1990, p. 51) defines object-oriented design as

> A design strategy in which a system or component is expressed in terms of objects and connections between those objects.

Objects provide numerous capabilities that make them desirable for efficiently designing software systems. For example, objects are capable of maintaining state information and provide services that can be used independently or relative to the object's state. Therefore, they are naturally good building blocks for creating good abstractions. Object-oriented designs also provide capabilities for inheritance and polymorphism, which provide various advantages when designing complex and large-scale software systems. Inheritance allows designers to create families of objects capable of reusing each other's interfaces or interfaces with implementations. While inheritance allows objects to inherit interfaces and implementations, polymorphism allows objects to change the behavior of inherited interfaces. Numerous design methods based on objects have been proposed. Today, the UP provides a popular framework for object-oriented software engineering using UML.

CHAPTER SUMMARY

Designs in software engineering are used to identify, evaluate, and specify the structural and behavioral characteristics of software systems that adhere to some specification. Software designs provide blueprints that capture how software systems meet their required functions and how they are shaped to meet their intended quality. Formally, software engineering design is defined as the process of identifying, evaluating, validating, and specifying the architectural, detailed, and construction models required to build software that meets its intended functional and nonfunctional requirements and the result of such a process. The term software design is used interchangeably in practice as means to describe both the process and product of software design. Throughout the design process, designers are constantly engaging in problem-solving activities that are fundamental to all modern engineering projects; therefore, they can be characterized as specialized problem solvers.

To ensure that all problem considerations are incorporated when solving design problems, a holistic problem-solving approach must be adopted, including all relevant concerns. Software design provides numerous advantages from both product development and process; however, many challenges must be considered and addressed before software designs can lead to complete and sufficient software models. In today's modern software systems, numerous design principles, processes, strategies, and other factors affect how designers execute the software design phase. When equipped with the proper design foundation knowledge, an understanding of the designer's roles and responsibilities can be acquired; allowing designers to become effective in designing large-scale software systems under a wide variety of challenging conditions.

REVIEW QUESTIONS

1. What is software engineering design, and why is it important?
2. What are the three states of problem solving? Describe each and explain how they apply to design problems?
3. What are two types of thinking employed during problem solving? Provide an example of how they are applied to design problems.
4. What is the difference between well-defined, ill-defined, and wicked problems and how these problems can affect software design?
5. What is the difference between an algorithm and a heuristic? Give examples of how both approaches can be applied during the design phase?
6. What is the holistic approach to problem solving? Explain.
7. How does design fits within the software engineering life cycle? Explain.
8. What are the major activities of the software design phase, and how do they differ from one another?
9. List and explain the challenges faced in software design.
10. Why is important to emphasize on documentation and management activities during design?
11. Compare and contrast the following: interface design, user interface design, and construction design.
12. What are the different roles of software designers? How do they differ?
13. Explain the difference between procedural and data abstraction.
14. What is content coupling, and how does it differ from other forms of coupling?
15. Explain in detail the concept of cohesion.
16. What do completeness and sufficiency mean?
17. What is the difference among the principles of modularization, abstraction, encapsulation, and separation of interface and implementation? Provide an example of each.
18. Compare and contrast the structured design strategy with the object-oriented design strategy.

REFERENCES

Abran, Alain, James W. Moore, Pierre Bourque, and Robert Dupuis. *Guide to the Software Engineering Body of Knowledge—2004 Version—SWEBOK.* Los Alamitos, CA: IEEE Computer Society Press, 2005.

Bass, Len, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, 2d ed. Boston: Addison-Wesley, 2003.

Booch, Grady. *Object-Oriented Analysis and Design with Applications*, 2d ed. Santa Clara, CA: Addison-Wesley, 1994.

Brassard, Gilles, and Paul Bratley. *Fundamentals of Algorithmics.* Upper Saddle River, NJ: Prentice Hall, 1995.

Dowson, Mark. "The Ariane 5 Software Failure." *ACM SIGSOFT Software Engineering Notes*, March 1997.

Dym, Clive L., and Patrick Little. *Engineering Design: A Project-Based Introduction.* Hoboken, NJ: Wiley, 2008.

Fox, Christopher. *Introduction to Software Engineering Design: Processes, Principles, and Patterns with UML2.* Boston: Addison Wesley, 2006.

Giachetti, Ronald E. *Design of Enterprise Systems: Theory, Architecture, and Methods.* Boca Raton, FL; CRC Press, 2010.

Griffin, Ricky W. *Management*, 10th ed. Mayfield Hts, Ohio: South-Western College Pub, 2010.

Harrell, C., Biman K. Ghosh, and Royce O. Bowden. *Simulation Using Promodel.* New York: McGraw-Hill, 2004.

IEEE. "IEEE Recommended Practice for Software Design Descriptions." 1998. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=741934.

IEEE. "IEEE Standard Glossary of Software Engineering Terminology." IEEE, 1990. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=159342.

IEEE/ACM. *Software Engineering 2004.* August 23, 2004. Available at: http://sites.computer.org/ccse/SE2004Volume.pdf (accessed September 22, 2010).

Kershaw, T. C., and S. Ohlsson. "Multiple Causes of Difficulty in Insight: The Case of the Nine-Dot Problem." *Journal of Experimental Psychology: Learning, Memory, and Cognition* 30:3–15, 2004.

Laplante, Phillip A. *Requirements Engineering for Software and Systems.* Boca Raton, FL: Auerbach Publications, 2009.

Liskov, Barbara, and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Boston: Addison-Wesley, 2000.

McConnell, Steve. *Code Complete*, 2d ed. Redmond, WA: Microsoft Press, 2004.

Meyers, Scott. *Effective C++: 55 Ways to Improve Your Programs and Designs*, 3d ed. Boston: Addison-Wesley, 2005.

U.S. General Accounting Office. (GAO). *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.* Washington, DC: U.S. Government Accountability Office, 1992.

Plotnik, Rod, and Haig Kouyoumdjian. *Introduction to Psychology*, 9th ed. Wadsworth Publishing, 2010.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach,* 7th ed. Belmont, CA: McGraw-Hill, 2010.

Sommerville, Ian. *Software Engineering*, 9th ed. Boston: Addison Wesley, 2010.

Stevens, Wayne P. *Using Structured Design: How to Make Programs Simple, Changeable, Flexible and Reusable.* Hoboken, NJ: John Wiley & Sons, 1981.

References

1 Chapter 1 - Introduction to Software Engineering Design

Abran, Alain, James W. Moore, Pierre Bourque, and Robert Dupuis. Guide to the So▮ware Engineering Body of Knowledge—2004 Version—SWEBOK. Los Alamitos, CA: IEEE Computer Society Press, 2005.

Bass, Len, Paul Clements, and Rick Kazman. So▮ware Architecture in Practice, 2d ed. Boston: Addison-Wesley, 2003.

Booch, Grady. Object-Oriented Analysis and Design with Applications, 2d ed. Santa Clara, CA: Addison-Wesley, 1994.

Brassard, Gilles, and Paul Bratley. Fundamentals of Algorithmics. Upper Saddle River, NJ: Prentice Hall, 1995.

Dowson, Mark. “ e Ariane 5 So▮ware Failure.” ACM SIGSOFT So▮ware Engineering Notes, March 1997.

Dym, Clive L., and Patrick Little. Engineering Design: A Project-Based Introduction. Hoboken, NJ: Wiley, 2008.

Fox, Christopher. Introduction to So▮ware Engineering Design: Processes, Principles, and Patterns with UML2. Boston: Addison Wesley, 2006.

Giachetti, Ronald E. Design of Enterprise Systems: ▮eory, Architecture, and Methods. Boca Raton, FL; CRC Press, 2010.

Gri▮n, Ricky W. Management, 10th ed. May¥eld Hts, Ohio: South-Western College Pub, 2010.

Harrell, C., Biman K. Ghosh, and Royce O. Bowden. Simulation Using Promodel. New York: McGraw-Hill, 2004.

IEEE. “IEEE Recommended Practice for So▮ware Design Descriptions.” 1998. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=741934.

IEEE. “IEEE Standard Glossary of So▮ware Engineering Terminology.” IEEE, 1990. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=159342.

IEEE/ACM. So▮ware Engineering 2004. August 23, 2004.

Available at: http://sites.computer.org/ccse/SE2004Volume.pdf (accessed September 22, 2010).

Kershaw, T. C., and S. Ohlsson. “Multiple Causes of Difficulty in Insight: e Case of the Nine-Dot Problem.” Journal of Experimental Psychology: Learning, Memory, and Cognition 30:3–15, 2004.

Laplante, Phillip A. Requirements Engineering for Software and Systems. Boca Raton, FL: Auerbach Publications, 2009.

Liskov, Barbara, and John Guttag. Program Development in Java: Abstraction, Specication, and Object-Oriented Design. Boston: Addison-Wesley, 2000.

McConnell, Steve. Code Complete, 2d ed. Redmond, WA: Microsoft Press, 2004.

Meyers, Scott. Effective C++: 55 Ways to Improve Your Programs and Designs, 3d ed. Boston: Addison-Wesley, 2005.

U.S. General Accounting Office. (GAO). Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia. Washington, DC: U.S. Government Accountability Office, 1992.

Plotnik, Rod, and Haig Kouyoumdjian. Introduction to Psychology, 9th ed. Wadsworth Publishing, 2010.

Pressman, Roger S. Software Engineering: A Practitioner’s Approach, 7th ed. Belmont, CA: McGraw-Hill, 2010.

Sommerville, Ian. Software Engineering, 9th ed. Boston: Addison Wesley, 2010.

Stevens, Wayne P. Using Structured Design: How to Make Programs Simple, Changeable, Flexible and Reusable. Hoboken, NJ: John Wiley & Sons, 1981.

2 Chapter 2 - Software Design with Unified Modeling Language

Booch, Grady, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. Object-Oriented Analysis and Design with Applications. Upper Saddle River, NJ: Addison-Wesley Professional, 2007.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. �e Unied Modeling Language User Guide. Santa Clara, CA: Addison-Wesley, 2005.

Qian, Kai, Xiang Fu, Lixin Tao, Chong-Wei Xu, and Jorge L. Diaz-Herrera. So�ware Architecture and Design Illuminated. Sudbury, MA: Jones & Barlett, 2009.

“UML 2.3 Superstructure.” Vers. 2.3. Object Management Group. May 2010. Available from: http://www.omg.org.

3 Chapter 3 - Principles of Software Architecture

Abran, Alain, James W. Moore, Pierre Bourque, and Robert Dupuis. Guide to the So�ware Engineering Body of Knowledge—2004 Version—SWEBOK. Los Alamitos, CA: IEEE Computer Society Press, 2005.

Bass, Len, Paul Clements, and Rick Kazman. So�ware Architecture in Practice, 2d ed. Boston: Addison-Wesley, 2003.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented So�ware Architecture: A System of Patterns. West Sussex, UK: Wiley, 1996.

Clements, Paul, Rick Kazman, and Mark Klein. Evaluating So�ware Architectures. Santa Clara, CA: Addison Wesley, 2001.

Gorton, Ian. Essential So�ware Architecture. Heidelberg, Germany: Springer, 2011.

Hofmeister, C., R. Nord, and D. Soni. Applied So�ware Architecture. Boston: Addison-Wesley, 2000.

Kruchten, Philippe. “Architectural Blueprints— e “4+1” View Model of So�ware Architecture.” IEEE So�ware 12, no. 6 (1995): 42–50.

Laplante, Phillip A. Requirements Engineering for So�ware and Systems. Boca Raton, FL: Auerbach Publications, 2009.

Pressman, Roger S. So�ware Engineering: A Practitioner’s Approach, 7th ed. Chicago: McGraw-Hill, 2010.

Taylor, Richard N., Nenad Medvidovic, and Eric M. Dashofy. So�ware Architecture: Foundations, �eory, and Practice. Hoboken, NJ: Wiley, 2009.

4 Chapter 4 - Patterns and Styles in Software Architecture

Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language: Towns, Buildings, Construction. New York: Oxford University Press, 1977.

Bass, Len, Paul Clements, and Rick Kazman. Software Architecture in Practice, 2d ed. Boston: Addison-Wesley, 2003.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns. West Sussex, UK: Wiley, 1996.

Clements, Paul, Rick Kazman, and Mark Klein. Evaluating Software Architectures. Santa Clara, CA: Addison Wesley, 2001.

Pressman, Roger S. Software Engineering: A Practitioner's Approach, 7th ed. Belmont, CA: McGraw-Hill, 2010.

Qian, Kai, Xiang Fu, Lixin Tao, Chong-Wei Xu, and Jorge L. Diaz-Herrera. Software Architecture and Design Illuminated. Sudbury, MA: Jones & Barlett, 2009.

Taylor, Richard N., Nenad Medvidovic, and Eric M. Dashofy. Software Architecture: Foundations, Theory, and Practice. Hoboken, NJ: Wiley, 2009.

5 Chapter 5 - Principles of Detailed Design

Booch, Grady, James Rumbaugh, and Ivar Jacobson. ■e Unied Modeling Langauge User Guide, 2d ed. AddisonWesley Professional, 2005.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented So■ware Architecture: A System of Patterns. West Sussex, UK: Wiley, 1996.

Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Sta¨ord. Documenting So■ware Architectures. Boston, MA: Addison Wesley, 2002.

Clements, Paul, Rick Kazman, and Mark Klein. Evaluating So■ware Architectures. Addison Wesley, 2001.

Douglas, Bruce P. Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley Professional, 1999.

Douglass, Bruce P. Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley Professional, 2002.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented So■ware. Boston: Addison-Wesley, 1995.

IEEE. “IEEE Standard for Information Technology-Systems Design-So■ware Design Descriptions.” 2009.

IEEE. “IEEE Standard Glossary of So■ware Engineering Terminology.” IEEE, 1990, p. 34.

Liskov, Barbara, and John Guttag. Program Development in Java: Abstraction, Specication, and Object-Oriented Design. Boston: Addison-Wesley, 2000.

Marin, Robert C. Agile So■ware Development: Principles, Patterns, and Practices. Upper Saddle River, NJ: Prentice Hall, 2003.

Meyer, Bertrand. Object-Oriented So■ware Construction, 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.

Ortega-Arjona, Jorge L. Patterns for Parallel So▯ware Design. West Sussex, UK: Wiley, 2010.

Pressman, Roger S. So▯ware Engineering: A Practitioner’s Approach, 7th ed. Chicago: McGraw-Hill, 2010.

Sommerville, Ian. So▯ware Engineering, 9th ed. Boston: Addison Wesley, 2010.

Tichy, Walter. Making So▯ware: What Really Works, and Why We Believe It. Sebastopol, CA: O’Reilly Media, 2010.

“UML 2.3 Superstructure.” Vers. 2.3. Object Management Group. May 2010. Available at: http://www.omg.org

Vermeulen, Allan, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Sta¨ord. e Elements of Java Styles. Cambridge, UK: Cambridge University Press, 2000.

Vora, Pawan. Web Application Design Patterns. Burlington, MA: Morgan Kaufmann, 2009.

8 Chapter 8 - Principles of Construction Design

Abran, Alain, James W. Moore, Pierre Bourque, and Robert Dupuis. Guide to the So�ware Engineering Body of Knowledge—2004 Version—SWEBOK. Los Angeles, CA: IEEE Computer Society Press, 2005.

Baldwin, Kenneth, Andrew Gray, and Trevor Misfeldt. �e Elements of C# Style. Cambridge, UK: Cambridge University Press, 2006.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. �e Unied Modeling Language User Guide, 2d ed.
Boston: Addison-Wesley, 2005.

Checkstyle 5.3. October 19, 2010. Available from: http://checkstyle.sourceforge.net/index.html (accessed March 11, 2011).

Collar, Emilio Jr. "An Investigation of Programming Code Textbase Readability Based on a Cognitive Readability Model." PhD thesis, University of Colorado at Boulder, 2005.

Fox, Christopher. Introduction to So�ware Engineering Design: Processes, Principles, and Patterns with UML2. Boston: Addison Wesley, 2006.

Galin, Daniel. So�ware Quality Assurance: From �eory to Implementation. Harlow, UK: Pearson Addison Wesley, 2003.

Hurley, Richard B. Decision Tables in So�ware Engineering. New York: Van Nostrand Reinhold, 1982.

IEEE. "IEEE Standard for Information Technology-Systems Design-So�ware Design Descriptions." 2009, p. 175.

Jones, Capers. Applied So�ware Measurement: Global Analysis of Productivity and Quality, 3d ed. New York: McGraw-Hill Osborne Media, 2008.

McCabe, omas J. "A Complexity Measure." IEEE Transactions on So�ware Engineering SE-2, no. 4 (1976): 308–320.

McConnell, Steve. Code Complete, 2d ed. Redmond, WA: Microso� Press, 2004.

Meyer, Bertrand. Object-Oriented So�ware Construction, 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.

Mills, Harlan D. Mathematical Foundations for Structured Programming. Gaithersburg, MD: IBM Federal Systems Division, IBM Corporation, 1972.

Misfeldt, Trevor, Gregory Bumgardner, and Andrew Gray. The Elements of C++ Style. Cambridge, UK: Cambridge University Press, 2004.

Pressman, Roger S. Software Engineering: A Practitioner's Approach, 7th ed. Chicago: McGraw-Hill, 2010.

Vermeulen, Allan, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick ompson. The Elements of Java Style. Cambridge, UK: Cambridge University Press, 2000.

9 Chapter 9 - Human-Computer Interface Design

Carroll, J. M. �e Nurnberg Funnell: Designing Minimalist Instruction for Practical Computer Skill. Cambridge, MA: MIT Press, 1990.

Carroll, J. M., and P. Aaronson. “Learning by Doing with Simulated Intelligent Help.” Communications of the Association for Computing Machinery, 1998: 1064–1079.

Hix, D., and H. R. Hartson. Developing User Interfaces: Ensuring Usability Through Product & Process. New York: John Wiley & Sons, 1993.

IEEE. “IEEE Standard Glossary of So�ware Engineering Terminology.” 1990. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=159342.

Lewis, C., P. Polson, C. Wharton, and J. Rieman. “Testing a Walkthrough Methodology for eory-Based Design of Walk-Up-and-Use Interfaces.” Chi ’90 Proceedings, 1990, 235–242.

McCrickard, D. S., C. M. Chewar, and J. Somervell. “Design, Science, and Engineering Topics? Teaching HCI with a Uni¥ed Method.” Technical Symposium on Computer Science Education (SigCSE’04). Norfolk, VA, 2004, 31–35.

Nielsen, J., and R. L. Mack. Usability Inspection Methods. New York: John Wiley & Sons, 1994.

Nielsen, J., and R. Molich. “Heuristic Evaluation of User Interfaces.” Proc. ACM CHI’90 Conference, Seattle, 1990, 249–256.

Rosson, M. B., and J. M. Carroll. Usability Engineering: Scenario-Based Development of Human-Computer Interaction. San Franciso: Morgan Kaufmann, 2002.

Scriven, M. �e Methodology of Evaluation in Perspectives of Curriculum Evaluation. Chicago, IL: Rand McNally, 1967.

Somervell, J., and D. S. McCrickard. “Better Discount Evaluation: Illustrating How Critical Parameters Support Heuristic Creation.” Interacting with Computers: Special Issue on Social Impact of Emerging Technologies 17, no. 5 (September 2005): 592–612.

Virzi, R. A., J. L. Sokolov, and D. Karis. “Usability

Problem Identi¥cation Using both Low- and High-Fidelity
Prototypes.” Proceedings of ACM CHI ’96, British Columbia,
Canada, 1996, 236–243.

10 Chapter 10 - Software Design Management, Leadership, and Ethics

Fisher, R., and W. Ury. Getting to Yes, 2d ed. New York: Penguin Books, 1991.

Gri�n, Ricky W. Management, 10th ed. Mason, OH: South-Western Publications, 2010.

IEEE Computer Society. “So�ware Engineering Code of Ethics and Professional Practice.” 2010. Available at

Judge, T. A., R. Ilies, J. E. Bono, and M. W. Gerhardt. “Personality and Leadership: A Qualitative and Quantitative Review.” Journal of Applied Psychology 87, no. 4 (2002): 765–768.

Li, Mei Yan, and Ying Zong Liu. “Study on Line Managers’ Competence-Based Abilities of Performance Management.” Applied Mechanics and Materials 40–41 (2010): 820–824.

Lussier, Robert, and Christopher Achua. Leadership: �eory, Application, & Skill Development, 4th ed. Florence, KY: Cengage Learning, 2010.

Meredith, Jack, and Samuel Mantel. Project Management: A Managerial Approach, 7th ed. Hoboken, NJ: John Wiley & Sons, 2009.

Nebus, J. “Building Collegial Information Networks: A eory of Advice Network Generation.” Academy of Management Review 31, no. 3 (2006): 615–637.

Pelosi, Marilyn K., and eresa M. Sandifer. Elementary Statistics: From Discovery to Decision. Hoboken, NJ: John Wiley & Sons, 2003.

Simkin, Mark. “ e Importance of Good Communication Skills on ‘IS’ Career Paths.” Journal of Technical Writing and Communication 26, no. 1 (1996): 69–78.

Veiga, J. F. “Special Topic Ethical Behavior in Management, Bringing Ethics into the Mainstream: An Introduction to the Special Topic.” Academy of Management Executive 18, no. 2 (2004): 37–38.