Software Engineering
Formal and Practical

Gene Fisher

California Polytechnic State University
San Luis Obispo

September 2009

Brief Table of Contents

Preface

This book is a comprehensive study of software engineering, with emphasis on the practical application of formal methods. While formal methods are an important part of software engineering, they are by no means the whole picture. Indeed, many aspects of software development involve principles of aesthetics and human communication that cannot be readily formalized. Since this text provides a comprehensive treatment of the field, both formal and non-formal aspects are fully covered. Throughout the coverage, the book integrates formal methods into the major phases of software development, to provide a foundation for the development process.

On Formality

From the author's perspective, the use of formal methods has been significantly neglected in most software engineering texts as well as in industrial practice. In general, far too many software engineers view formal methods as impractical and largely irrelevant to their regular activities. This is a rather unusual view when one compares software engineering to other science and engineering disciplines. In almost all such disciplines, formal mathematical analysis is a common practice.

One cause for lack of rigor in software engineering is that formal reasoning can be hard. When given the chance, human nature will steer us away from hard tasks. The civil engineer, for example, might like to draw some simple pictures and perform some informal analysis when designing a bridge, without devoting endless hours to formal modeling. Fortunately, the competent civil engineer knows that informal analysis is not sufficient, and that a bridge may well collapse if a careful mathematical specification is not developed. The civil engineer learns this as part of basic training and the practice of civil engineering demands that formal analysis is an integral part of the job.

Like the civil engineer, the software engineer should learn that careful formal reasoning can be an integral part of building complex software systems. In current practice, informal analysis and design are frequently considered adequate for software engineering projects. Furthermore, most software engineers are not trained in formal methods, nor does the practice of software engineering require the same degree of mathematical rigor as is required in other branches of engineering. This author firmly believes that as software engineering matures into a genuine engineering discipline, acceptance and use of formal methods will be an important part of its maturation.

A key distinction between this text and most others is that here formal methods are considered to be an integral part of software development, rather than an alternate form of development. Again, that formal methods are an integral part of development by no means implies that all aspects of software development are fully formalized or formalizable.

On Practicality

Another aspect of software engineering texts that the author has found disappointing is the depth and thoroughness of the examples. As an instructor, I am convinced that presentation of coherent and detailed examples is critically important to teaching software engineering effectively.

The technical chapters of the book present material in the context of a single large example. The example is an office calendaring system. When software engineering concepts are first introduced, small excerpts of the example are presented to focus on basic principles. As the presentation of concepts evolves, larger example components are presented and relevant details are focused upon. The online supplemental materials have the example in full, including a complete requirements document, a formal specification, design, Java implementation, testing artifacts, and other supporting material.

The motivation for using a single large example is to avoid a problem the author has found in other texts, where concepts are presented with relatively small, independent, and generic examples. Such example presentation fails to show the big picture of a complete, non-trivial piece of software. In this book, the initial use of small excerpts avoids overwhelming the reader with too much detail at first. As the reader gains a better understanding of concepts, the complete details of the example can be consulted to understand how the pieces fit together and to appreciate the scope of the completed product.

The chosen example is large enough in size and scope to embody a number of general technical problems that are encountered in a wide range of software applications, not just in the domain of office products. Major technical aspects of the example include the following:

- a substantial end-user interface, with a reasonably wide range of interface elements
- a sufficiently large size to require non-trivial design and implementation techniques, including use of multiple design patterns
- a sufficiently large size to require non-trivial testing techniques
- basic and advanced data design, including interface to external data stores
- basic and advanced functional design, including exception handling and event-based processing
- distributed processing and remote data access
- a sufficiently large size to require non-trivial project management, configuration control, and documentation

This book does not provide in-depth coverage of advanced Computer Science topics such as databases and distributed computing. The focus of the book in such advanced areas is on the specification and design of software that must address technical problems in these areas, and the use of software libraries that provide implementation solutions to the problems.

Another motivation for using a single large example is to provide continuity as the book progresses. The reader sees how initial product storyboards evolve into requirements, a model specification, a design, and an implementation. The activities of testing, management, configuration control, and documentation happen along the way. In effect, this evolution tells the story of software engineering.

The Approach Used in the Book

The approach taken in software engineering textbooks is based predominantly on the methodologies that the books cover. In general, the treatment of methodologies varies widely. Some books present a survey of different methodologies, giving equal treatment to each of them. Others focus exclusively on a single methodology.

This book takes an approach somewhere between survey and single-methodology. Overall, the book follows a specific methodology, defined precisely in terms of the software process and artifacts presented in Chapters 2 and 3. In terms of survey, most chapters include discussion of alternative methodologies, comparing and contrasting them to the specific approach taken in that chapter.

The methodology used in the book is general in scope. It is applicable to a wide range of end-user software that performs non-trivial computational tasks and requires a non-trivial human-computer interface. Not all aspects of the methodology are entirely applicable to all types of software, as is the case with most methodologies. For example, the part of the methodology that focuses on the human-computer interface is not applicable to systems software or embedded software that has no significant human interface. Where appropriate, the book discusses methodological differences in building different types of software and describes how the specific methods used in the book can be adapted as necessary.

Nothing in the book's methodology is fundamentally new. Each phase of development is based on concepts that have appeared extensively in the software engineering literature and have been applied in practice. One linguistic element of the methodology that is the Formal Modeling and Specification Language (FMSL). Prior to its appearance in this book, FMSL has not been used extensively except at the university where the author teaches. The concepts of FMSL are far from new; they are rooted in well-established principles of formal software specification that have been in existence, if not wide-spread use, for at least two decades.

Following the process definition in Chapter 2, Chapter 3 overviews the software artifacts produced by the process. Subsequent chapters are organized around the steps of this process and its artifacts.

- Chapters 4 through 6 cover requirements analysis.
- Chapter 7 is devoted to requirements testing, using formalized inspection techniques.
- Chapters 8 through 10 cover the requirements modeling process, resulting in a formal software specification.
- Chapter 11 presents techniques to test the specification, prior to its refinement into a program design.
- Chapter 12 discusses rapid software prototyping, as a means to convey requirements to the end user in terms of a partially operational program.
- Chapters 13 through 17 cover software design, from general design principles through formal design specification.
- Chapter 18 covers design testing, by inspection and partial execution techniques.
- Chapter 19 covers program implementation; since low-level programming is the topic of prerequisite courses to software engineering, only a single chapter is devoted to implementation.
- Chapters 20 through 22 are devoted to program debugging, testing, and verification.
- Chapter 23 covers post-development software deployment, during which the software is installed, used, maintained, and upgraded.
- Chapters 24 through 26 cover the pervasive phases of development devoted to project management, configuration control, and documentation.
- Chapter 27 is an overview of software engineering tools.
- Chapter 28 is an introduction to the topics of software ethics and law.
- The chapters conclude with number 29, which covers the future of software engineering.

Organizationally, the book follows what can be considered a "traditional" software process. This makes sense for a comprehensive treatment of the subject matter, since a traditional process generally has more steps and more surviving artifacts than many non-traditional processes. Much of the book's material can be used in a process-independent manner, by changing the order in which material is presented, and omitting coverage of material that is not germane to a particular style of process.

Reader Background

The reader of this book is assumed to have training and/or experience at the introductory level of Computer Science. Specifically, the reader should understand basic concepts of programming, including the design and implementation of data structures. The reader should also understand basic concepts of discrete mathematics, including predicate logic with quantifiers.

Using the Book in Courses

The primary audience for the book is undergraduate and graduate students in software engineering. The book may be of interest to other readers who desire an introduction to the practical use of formal methods in software engineering.

The author uses this book in a two-quarter undergraduate sequence, covering twenty weeks in total. The first quarter covers Chapters 1 through 10 in full, and partially covers Chapters 11 and 12. The second quarter covers chapters 11 through 22 in full, and partially covers Chapters 23 and 24. During both quarters, selected topics from Chapters 25 through 30 are introduced.

The book is also used in a two-quarter graduate sequence where it is supplemented with readings from current software engineering literature. In addition, the graduate courses involve projects that treat formal specification, testing, and verification in greater depth than in the undergraduate sequence.

Since most colleges and universities use the semester rather than quarter system, coverage for a sixteen-week semester can be adjusted in a number of ways. One suggestion is to cover Chapters 1 through 10 and 13 through 22 in depth, plus selected topics from other chapters

Online Supplemental Material

The book is supplemented by a collection of online materials, available at http://www.csc.calpoly.edu/~gfisher/se-book. All of the examples presented in the text are available online. Also available are supplemental lecture and course project materials. Executable tools and documentation are available for the formal specification language introduced in Chapters 8 and 9. Other experimental tools are also available for suitably brave and tolerant users. See the site for further information on currently available tools.

Acknowledgments

The author thanks first and foremost the many students who have endured ten years of draft versions of this text, in the form of lecture notes and other course materials. Their patience and constructive criticism have improved the material immeasurably. Most recently, a number of graduate students at Cal Poly University have been particularly helpful in solidifying the concepts presented in the book. They include Michael Porcelli, Rick Doty, Ira Weiny, Larry Bolef, and Mal Sikand. Faculty colleagues who have taught me much about software engineering include Peter Freeman, Carl Levitt, Dan Stearns, and Clark Turner. Finally, I thank my faithful spouse who has endured many fits and frustrations during the writing of this book, asking only occasionally when it would be done. Thanks Lori.

Chapter 1

Introduction

Software engineering is the disciplined creation of software. The discipline is based on general principles of scientific and engineering problem solving, applied to the specific task of software development. Problem solving principles employed by the software engineer include defining a problem clearly before starting its solution, and using strategies to manage the complexity of large problems. General engineering principles include the use of formal modeling to specify a product precisely, and the use of rigorous testing to verify that a product meets its specification.

The principle of defining a problem clearly is fundamental to any problem solving activity. For software engineering, the "problem" to be solved is based on the needs that people have for software. Hence, the task of problem definition for the software engineer entails analyzing the requirements that people have for a software product. Once the requirements are analyzed and understood, they constitute the definition of the problem to be solved, that is, the definition of the software to be built. Several early chapters of this book are devoted to the analysis of software requirements and their precise specification.

Like other engineered artifacts, software systems can be large and complex. Scientists and engineers have learned to employ various strategies to cope with problem size and complexity. An important general strategy is that of "divide-and-conquer", which entails breaking down a large and complicated problem into smaller pieces, so that each piece can be more clearly understood and solved. Software engineers employ such strategies in the design of a software product, to define and organize the overall product architecture. Principles of software design are discussed in chapters of the book following those on requirements analysis.

Engineers of all kinds build models to help them understand a problem to be solved. A model is a simplified version of a complex engineered artifact. The model allows the engineer to focus on basic properties of the artifact before all of the details of the finished product are completed. Modeling techniques in software engineering have evolved over several decades, and the use of modeling is becoming increasingly widespread in the development of complex software. The book discusses software modeling in the requirements specification phase of software development, as well as in the architectural design phase.

Thorough testing is essential for any well-engineered product. The general goal of testing is to ensure that a product meets its specification. For a software product, testing can be conducted by thorough inspection of the product components, in much the same sense that any product is inspected. Software engineers also utilize testing techniques designed specifically for software. These techniques involve the development of thorough testing plans that systematically exercise each and every software function to ensure there are no product defects. Software testing is discussed thoroughly, in individual chapters devoted to the different aspects of testing.

While software engineers employ many of the same general techniques used by other engineers, there are some important differences between software engineering compared to other branches of engineering. One of the most significant differences is the nature of the product itself. Software is "soft". It is not built with the "hard" physical materials used for most other engineered artifacts. Software does not "wear out" in the same physical way that other products do. These characteristics of software can lead to a number of potential benefits, including inexpensive mass production and high product reliability. The book discusses how these benefits can be realized when software is well engineered and thoroughly tested.

Another difference between software engineering and its fraternal disciplines is its relatively young age. The origin of software engineering can be traced to the late 1960s, when the first international conference was held on the subject [Naur 68]. Since that time, there have been many advances in software engineering practices, and there are likely to be many new advances before the discipline fully matures.

Certain characteristics stand out in young disciplines. For one thing, young disciplines are generally less formal than they will eventually become. The early years of a discipline can involve a lot of trial and error, until eventually a set of well-defined principles evolves. Software engineering is beyond its earliest trial-and-error stages, but software engineering practices are still far less formal than most other types of engineering. A fundamental tenet of this book is that formal methods of software engineering need to be more widely practiced. To date, there are few studies that validate or invalidate the use of formal methods in software engineering. In providing significant coverage of formal methods, this book seeks to help make the case for their utility.

Another characteristic of young disciplines is breadth of scope, such that a young discipline encompasses what may evolve into separate sub-disciplines in its future. What is broadly called Software Engineering today may in future branch into separate disciplines of software architecture, formal engineering, and software construction. Software architects will interact with human clients in the user-level design of software products, in a way similar to how building architects interact with their clients. Software engineers, in a more narrow sense than defined today, will focus on the formal mathematical models of the software system as designed by the architects. Software engineers will play a role analogous to that played by civil engineers in constructing a building. Software construction specialists will then build the final software product, as designed and specified by the architects and engineers. Software construction, like building construction, will be considered more a craft than an engineering science.

Whatever the future may hold, software engineering today covers the range of activities that include gathering user requirements, specifying a product to be built, designing the product, building it, testing it, and deploying it. This book covers all of these activities. It also covers supporting activities of project management, configuration control, documentation, and reuse.

1.1. The Different Types of Software

Software is ubiquitous in today's world. It is applied to tasks as varied as flying spacecraft to balancing a checkbook. The term *application domain* refers to the specific area in which software is applied. For example, spacecraft software is in the domain of aerospace applications; checkbook software is a financial application.

The fact that software applications are virtually limitless is a major challenge for the software engineer. It means that software engineers must often work in application domains where they have little or no experience. In so doing, they must rely on the expertise of others to ensure that a software product is properly specified and built. Further, the application domain can significantly affect the process used to develop a particular type of software. Software engineers must therefore be skilled in communicating with those who understand the application domain of a software product.

1.1.1. General Categories of Software

While there is no limit to the types of software application, there are some general application categories that affect the structure of a software product and how it is engineered. Three broad application categories are the following:

- end-user software
- system software
- embedded software

These categories are based on who the ultimate user of the software is. With end-user software, the users are human beings who apply the software to perform useful work. Word processors and web browsers are examples of end-user software. From an engineering standpoint, a key aspect of end-user software is its human-computer interface (HCI). The HCI provides the services that allow the human user to communicate with the software. HCI development is a very substantial part of the process of engineering end-user software. In particular, the requirements for end-user software are heavily influenced by the HCI. The software process followed in this book employs a *user-centered* technique for requirements analysis, whereby the central means of presenting the requirements is from the standpoint of the HCI. The design phase of the software process is also heavily influenced by HCI structure.

System software is not used directly by end-users, but rather by other software. In a web browser, for example, system software performs tasks such as underlying network communication and remote data access. The end user is aware of the work performed by the system software, but has little or no direct communication with it. Given this, system software generally has little or no HCI. Instead, system software has what is called an *application programmer interface* (API). As this name suggests, system software can be considered to have human users, but of a different sort than end-product users. The human users of system software are the application programmers who use the system software to write their applications. Even though most programmers are human, there is a fundamental difference between HCI versus API. The HCI is an *external* human interface, whereas the API is an *internal* program interface. From an engineering standpoint, the lack of a HCI affects the process used to develop system software. In particular, details of user-centered requirements analysis are not applicable to system software. There are, however, important commonalities in the design process for end-user and system software. These commonalities are discussed in later chapters of the book.

Embedded software is used within hardware devices that operate with no direct human control. Examples of embedded software are the controller for an automobile engine or the program that controls some fully automated manufacturing process. As with system software, there is little or no HCI to embedded software. The direct user of embedded software is not a human, but rather a hardware device. From an engineering standpoint, the process of embedded software development has some key differences from that for end-user or system software. A common requirement for embedded software is *real-time* operation. This means that the software must respond to external events in a fixed amount of time. End-user software generally does not have real-time requirements. For example, a web browser that does not respond in a timely manner may annoy its user, however it can still perform its task properly. When an embedded real-time program does not respond in time, it has fundamentally failed to do its job. Embedded software also has requirements based on the hardware devices with which in must communicate. Analysis of these requirements and implementation of the software that meets them often requires specialized technical knowledge. Despite some unique characteristics of embedded software, there are commonalities in its development process compared to that for end-user and system software, as will be discussed in later chapters.

1.1.2. Categories of Software Clientele

There is another major categorization of software that is independent of its application domain. This categorization is based on the clientele who purchase a software product. From a clientele perspective, the two general types of software are *off-the-shelf* versus *custom*.

Off-the-shelf software is also referred as *shrink-wrap*. It is built by software developers who sell their product on the open market. Off-the-shelf software can have a broad market, such as a word processor or web-authoring tool. It can also be built for more specialized applications, but for which there is still a large enough market to be profitably sold. Software for medical or legal applications are examples of more specialized off-the-shelf products.

In contrast to off-the-shelf, custom software is built to satisfy the needs of a specific customer, typically an organization of some kind. Custom software is also referred to as *bespoke*. When an organization has needs that cannot be met in whole or in part by an off-the-shelf product, it orders a custom software product.

An important distinction between off-the-shelf versus custom software is in the area of ownership and control. Off-the-shelf software is typically licensed to its users, under specific terms of use. The developers often retain control and ownership of program source code and other artifacts of software development. With "open-source" products, the source code is available to customers, but it is still typically licensed, and the developers retain some forms of control.

With custom software, control and ownership are most typically retained by the customer. The customer contracts with a developer to deliver a product. Once delivered, the customer owns the product wholly. A variant of custom software is that developed *in-house* by an organization. In this case, the organization does not contract for software to be built, but rather develops using its own internal staff. In this case, the customer and developer are one in the same organization.

From an engineering perspective, the major difference between off-the-shelf versus custom software is in the requirements process. With custom software, the requirements are developed to meet the needs of specific customers, and the requirements analysis process engages those specific customers. By its nature, off-the-shelf software is developed for a broader customer community. This means that requirements analysis for off-the-shelf software is based on the needs of *prospective* customers. Off-the-shelf software developers often have a marketing department, whose job it is to identify a prospective customer base, and develop requirements to meet the needs of those perspective customers. During the requirements analysis process, members of the marketing staff serve in effect as representatives of an actual customer base.

The primary focus of this book is on end-user software, developed as either a custom or off-the-shelf product. End-user software is a broad and important category, and key aspects of its development process are applicable to system and embedded software as well. Where appropriate, distinctions will be drawn between software categories, but for the most part the emphasis is on end-user products. The book does not cover specific technical details of system and embedded software, such as communication with specialized system devices or real-time execution.

1.2. The People Involved with Software

A variety of people are involved in the development and use of a software product. These people are often referred to as *stakeholders*. As defined here, the term "stakeholder" means anyone who has some interest, large or small, in a software product. Stakeholders can be broadly categorized into the following groups:

- ***end users*** -- people who will use the software or people who represent those who will use it
- ***customers*** -- people who purchase the software, which they may or may not use themselves
- ***domain experts*** -- people who fully understand the application domain in which the software will run
- ***analysts*** -- members of the software development staff who specialize in requirements analysis and specification
- ***implementors*** -- members of the development staff who specialize in software design and implementation
- ***testers*** -- members of the development staff and user community who test the software to ensure that it meets the requirements specification
- ***managers*** -- those who manage the development process, as well as those who manage end users when the software is installed in an organization
- ***visionaries*** -- those who have the "big picture" for what the software is intended to do and how it will be developed
- ***maintainers and operators*** -- those who conduct post-development maintenance and operations, as necessary
- ***other interested parties*** -- anyone else interested in the software product, such as those with a financial investment, or those with societal or legal interest

The *end user* group is listed first as an indication of its overall importance. A key to a successful software product is acceptance by its end users. A key to user acceptance, in turn, is to consider the users to be active players in the development process. For custom software, this means involving as many of the actual users as possible in the requirements analysis activities. In some cases, end users can also be involved during the implementation phases of the project, to ensure that the product is being built to their satisfaction. For an off-the-shelf product, the team that develops the requirements must adequately represent the needs of potential end users as the requirements are defined, and as necessary during implementation.

The *customer* group are those who acquire the software on behalf of the end users, but may not themselves be end users. The customer group is more likely to exist for a custom software product, since customers and end users are typically the same for an off-the-shelf product. For custom software in an organization, the customers can include supervisory or administrative staff who have business-related software requirements for a software product. For example, a customer could require that a software product generally improves worker productivity, independent of the specific end-user requirements for the product.

The group of *domain experts* is always important to a successful software project, particularly so when the analysts and implementors are not familiar with the domain. Consider for example the application domain of medical software, where doctors are the domain experts. Since software analysts and implementors are not typically medical experts, the involvement of doctors in the project is critical in order for the requirements to be properly specified and the implementation to be properly tested.

The *analyst* and *implementor* groups do the actual software engineering. Their domain of expertise is the software. The analysts are skilled at communicating with end users and defining requirements. The implementors are skilled at software design and programming. In some projects, typically small ones, the analysts and implementors may be the same people. There are however distinct skills required to be a good analyst or implementor. In general, analysts need good "people skills" so they can interact

effectively with end users. Implementors need good technical programming skills, so they can design and build a correct and efficient piece of software. While there are people who are skilled in both of these areas, not everyone is able or desirous to excel in both analysis and implementation.

The group of software *testers* ensure that an implemented software product works. What it means for software to work is that it meets its requirements specification. In some cases software testing is conducted by the development staff, that is the analysts and implementors. In other cases, testing is conducted by a third-party team of *quality assurance* (QA) personnel. A final round of product testing must always be conducted by end users, or those acting as end users, to ensure that the product meets all user needs.

Any project of significant size needs good *managers*. While there are particular skills required for software management, much of what makes a good manager is product-independent. Managers need to be able to lead teams of developers and keep a project on track in terms of time and budget.

Successful software products are often based on the ideas of one or a few project *visionaries*. These are the people who understand the grand scheme of things. They can inspire and motivate all of the stakeholders to produce a good product. While it is certainly possible to develop good software without visionary guidance, it is often the case the software is the better for such guidance.

Once a software product is delivered to its users, it typically requires maintenance, and in some cases operations staff to keep it running properly. The *maintenance* staff is in charge of repairing post-delivery problems and upgrading the software to meet changing user needs. Such maintenance may be carried out by members of the original development team, or by staff hired specifically for the post-delivery lifetime of the software. For highly complex or configurable software, system *operators* may be needed to administer and configure a software installation.

A final group of stakeholders are *other interested parties* who do not participate directly the software development or use. People who have some financial investment in the software are included in this group. For custom software, financing is generally the province of a budget department that may have no involvement in the project other than paying the bills. Off-the-shelf software can be financed by venture capitalists who are neither end users nor developers, but who are clearly stakeholders due to their monetary investment.

Other interested parties can also be those with a societal or legal interest in the software. When a group of people view the use or misuse of a software product as beneficial or detrimental to society at large, they can become stakeholders. When the use or misuse of software has potential legal ramifications, those affected can become stakeholders. A high-profile example is the case of the music industry, who have viewed as theft the use of free music down-loading software, when it violates the music copyrights owned by the industry.

Depending on the scale of a software project, there may be overlap among the participating groups. For example, it is not uncommon for domain experts also to be end users. As mentioned above, the analyst, implementor, and tester groups may have overlapping membership. Given such overlaps, the groups can be considered *roles* that are assumed by the stakeholders. At different points, a single person can assume one or more roles. In the upcoming chapters of the book, these roles will be expanded upon further, in the context of the work that is performed during software development and use.

Different stakeholders have different and sometimes competing interests. Ideally, there should be cooperation and teamwork among all interested parties, to see that a software project is successful. Noteworthy areas of cooperation among stakeholders include the following:

- end users, domain experts, and analysts function as a team to develop the software requirements;
- analysts and implementors communicate as necessary to ensure that the implementation meets the requirements, and to deal with any requirements changes that arise during implementation;
- managers and those who they manage function as team, as in any well-run engineering project.

Given such cooperation among the stakeholders, a software project is much more likely to be successful than without it.

1.3. The Software Life Cycle

Like any other engineered product, software evolves through a number of phases as it is developed and put to use. Theses phases are often called the software product *life cycle*. The major phases of the software life cycle have already been identified in the preceding discussion of software types and people. Here is a summary of the major phases:

- formulate an idea for a software product
- gather and specify software requirements
- design and implement the software
- test the software implementation against the requirements
- deliver, use, and maintain the software

These are only the most basic phases of software development, with many important details left out. In order for a software product to be successfully engineered, these generalized life cycle phases must be used as the basis for a more formal *software development process*. The process defines the specific activities of each development phase, and the order in which the activities are carried out.

Chapter 2 of the book provides thorough coverage of software processes, including a discussion of the different approaches to process definition. While the approaches can differ substantially in the details, at the heart of any successful software process are the basic steps of problem solving outlined at the beginning of this chapter. These steps, and the corresponding phase of the software life cycle are shown in Table 1.

Among the various approaches to the software development process, a key difference is the sequence in which these core problem-solving steps are a carried out. In a *sequential* software process, each step is carried out to completion before the next step is begun. That is, a complete set of requirements are first developed, then the design and implementation are completed, then the final product is tested.

Problem-Solving Phase	**Software Development Phase**
define the problem	gather and specify software requirements
solve the problem	design and implement the software
verify the solution	test the software implementation against the requirements

Table 1: Software development as problem solving.

In contrast to a fully sequential approach is an *iterative* style of development. In this process approach, a problem-solving step need only be partially completed before the next is begun. The process starts by defining a small part of the product requirements. Then that part alone is implemented and tested. The process then iterates back to the first phase, where the next part of the requirements are specified, and so on.

The sequential and iterative approaches have advantages and disadvantages that depend on a number of factors, including project size, application domain, and the skills of the development team. In practice, software is most often built with a combination of sequential and iterative development, not purely one form or another. These process-related issues will be discussed fully in Chapter 2. Also discussed in that chapter are additional activities of the software process not sequentially related to the core problem-solving phases. Important among these are the activities of project management, configuration control, and documentation.

1.4. Software Artifacts

A software *artifact* is something produced during the software development process. The ultimate goal of the process is to produce an operational program, that satisfies user needs. From an end user's perspective, this working program is the artifact of primary interest. Customers also need documentation artifacts that tell them how to use the software. This documentation can include users' manuals, tutorials, and online program help.

From the developers' perspective, there are many more software artifacts than just the working program and its user documentation. During the stages of development, other important artifacts are produced. Table 2 summarizes the software artifacts that can be produced during each stage of the software life cycle.

Stage of the Software Life Cycle	Associated Artifacts
formulate an idea	storyboards of software behavior, rough drafts of the requirements
gather and specify requirements	requirements document, abstract model specification
design and implement the software	architectural design model, program code
test the software	test plans, test results
deliver, use, and maintain the software	user documentation, technical documentation, defect tracking logs

Table 2: Artifacts produced at each stage of the software life cycle.

When formulating the initial ideas for a product, developers can use *storyboards* to work out the way a program will appear to its end users. Storyboarding is a practice borrowed from the movie industry, where a director sketches out the scenes of a movie before it is filmed. In a similar way, a software analyst can sketch out the user interface of a program before it is implemented. These storyboards are captured as artifacts so they can be reviewed by users and used to refine the requirements.

During the requirements specification stage of development, a detailed requirements document is produced. This is based on the storyboards, plus additional analysis of user needs. From these requirements, the more formal engineering starts, with the definition of an abstract program model. This model specifies the behavior of the program, but without low-level programming details.

Artifacts of the design and implementation stage include a concrete design of software architecture and the program code itself. The architectural design model is a refinement of the abstract requirements model. Software engineers may apply well-known design patterns to refine the architectural model. A design pattern is a pre-packaged piece of design based on software design experience that has been gained over the years by software engineers. The most concrete software artifact is the program code, produced during the implementation stage of development.

Testing artifacts are defined to plan the software testing to be conducted. Once planned, tests are performed by the testing staff, with the results recorded. If any test failures occur, the test record artifacts are reviewed by the development team to perform the necessary corrections.

Several important artifacts are produced before and during the post-development stage of the software life cycle. The user-level documentation is produced prior to delivery. Developer-level documentation is provided for the staff who will maintain and enhance the delivered software product. If defects are detected during software use, they are recorded in some form of defect tracking log. This artifact is used by the maintenance staff to correct the defect as necessary. When defects are corrected or new product features are added, the documentation artifacts must be updated accordingly.

Other important software artifacts are produced by project management activities that are conducted throughout the stages of the software life cycle. Management-related artifacts include project scheduling plans, resource allocation plans, and a records of project meetings.

The artifacts introduced here are samples of the kinds of work products that can be produced in a software project. The specific details of software artifacts depend on the details of the software process that produces them. For example, a highly iterative software process may produce a less formal requirements specification, relying instead on the program itself to embody the requirements. Chapters 2 and 3 of the book discuss details of software process and the artifacts produced during the process.

1.5. Software Languages, Notations, and Methodologies

Software is not built with the same hard materials that are used for other kinds of engineered products. One might consider the computer hardware on which software runs to be its material, but the engineering of computer hardware is a separate task from engineering software. Computer engineering and software engineering are in fact separate disciplines.

The "materials" used to build software are the languages and notations in which software artifacts are expressed. For example, the English language is a "material" used to express software requirements. A graphical program diagram can be a material for software design. The distinction between the terms "language" and "notation" is not critically important, but for clarity they are distinguished in this book. The term "language" is used to refer to a primarily textual form of communication; the term "notation" is used

to refer to a primarily graphical form.

Table 3 summarizes the kinds of languages and notations that can be used for different software artifacts. Some of the language names appearing in the table may not be familiar to the reader. The languages are defined briefly here in the introduction, and more thoroughly as they are used in the book.

Since software requirements must be understood by end users, they are most typically expressed in English and pictures. Requirements documents can be produced in paper form. When requirements are extensive, viewing them in online format can be convenient. A widely used form for electronic documents is HTML (Hyper-text Mark-up Language) [World Wide Web Consortium 99]. This is the language displayed by a normal web browser. The requirements examples in the online supplement for this book are in HTML.

A formal software specification can be expressed in a language designed specifically for this purpose, such as Z (pronounced "Zed") [Spivey 92] or SpecL [Fisher 06]. SpecL is the language used in this book to express specifications. Software specifications can also be expressed in graphical form, using a notation such as UML (Unified Modeling Language) [Rumbaugh 98]. UML is also used in this book.

Software design can be expressed in a language or notation that focuses on the major structural elements of a program, leaving out the low-level details of implementation. For the Java programming language, Javadoc is such a notation. For expressing designs graphically, UML can be used. Since UML is a general-purpose notation, it can be used for both specification and design. The design examples in this book and the online supplement are expressed in Javadoc and UML.

Software implementations are expressed in programming languages, from which there are very many to choose. Implementation examples in this book use Java.

The languages and notations listed in Table 3 are those used predominately in this book. These particular languages and notations are by no means the only form of expression for software artifacts. While it is necessary to choose specific concrete languages in which to communicate, most of the underlying software engineering concepts are *language independent*. For example, the concepts of underlying software requirements are almost entirely independent of the particular natural language in which the requirements are expressed. Whether requirements are written in English, French, or some other natural language, the

Software Artifact	**Language or Notation**
requirements	English and pictures, in electronic or paper form
specification	a formal specification language, such as Z or SpecL; a modeling notation, such as UML
design	a structured software documentation format, such as Javadoc; a modeling notation, such as UML
implementation	a programming language, such as Java or C++; a graphical program diagramming notation

Table 3: Languages and notations used for software artifacts.

requirements for a particular software product must express the same product functionality. Whether a program is written in Java, C++, or some other language, it must behave in fundamentally the same way.

There are of course highly practical reasons for choosing one language over another. It would be silly for example to write software requirements in French for a user community who understands only English. At the implementation level, there are often practical reasons for choosing one programming language over another. The important point to emphasize here is that the fundamental concepts presented in this book are independent of the particular language or notation in which they are expressed. It is highly likely that most, if not all of the non-natural languages used in this book will become outdated within a decade. The reader, and alas the author, are therefore well advised to focus on concepts, not the particular details of language. In a field as rapidly changing as software engineering, its practitioners must be prepared to learn and use different languages during the course of their careers.

Software languages and notations are used in the context of a software engineering *methodology*. A methodology is a particular approach to developing software. Generally, a methodology defines a process of development and the structure of artifacts that the process produces. There are a wide variety of software methodologies that have been devised over the years. Some are very broad and general, encompassing the entire software life cycle. The Unified Software Process is a well-know example of a broad software methodology [Jacobson 99]. Other methodologies focus on a specific stage of the life cycle or a particular software artifact. For example, the methodology of design patterns focuses in particular on architectural software design [Gamma 95].

Overall, the book follows a specific methodology, defined precisely in terms of the software process presented in Chapter 2. The chapters that follow cover the parts of the methodology that apply to the different phases of the software life cycle. For requirements specification, a user-centered approach is employed, where the requirements are organized around scenarios that depict the way the user interacts with the software. A model-based specification methodology is used to formalize the requirements. The specification model defines precisely the software data and functions presented in the user-centered scenarios. The design and implementation stage of development employ object-oriented, function-oriented, and pattern-based methodologies. The object-oriented methodology organizes the design around software data objects; the function-oriented methodology focuses on the functions that manipulate the data. Design patterns are employed to take advantage of well-established forms of software design that have been recognized and refined by the community of experienced software designers. The testing phase of development is presented incrementally, with a separate chapter devoted to the testing of each of the major software artifacts.

1.6. Pervasive Principles

As noted at the beginning of the chapter, the practice of software engineering involves some general principles of problem solving. These are principles employed by scientists and engineers in a variety of disciplines, adapted in discipline-specific ways. Such principles should be considered general guidelines, not iron-clad rules. Being aware of these principles helps one understand problem solving in high-level terms, and helps organize one's thinking about solutions to complex problems.

1.6.1. Divide and Conquer

Divide and conquer is a fundamental problem solving strategy. Simply put, it means breaking up a large problem into smaller, more manageable pieces. In software engineering, the divide and conquer principle is applied at all levels of development.

The software development process is fundamentally based on divide and conquer. The overall process is subdivided into a series of process steps, with each step focusing on a particular aspect of development. The artifacts produced by each step form pieces of the overall software product.

Within each development step, divide and conquer is applied in specific ways. Software requirements are divided into individual cases of use, where each case focuses on a particular aspect of software functionality. The design and implementation of the software is divided into modular units, each one of which has a functionally cohesive purpose.

From a people perspective, divide and conquer applies to how development teams are composed and organized. Individual team members are assigned appropriately-sized pieces of work. Work assignments are based on team members' areas of expertise, so members' varying skills are applied most effectively to conquer individualized development tasks.

1.6.2. Hierarchy

Hierarchical organization is another basic principle of science and engineering. Herbert Simon, the famous economist and computer scientist, argues that hierarchy is a natural and fundamental property of complex systems [Simon 69].

For problem solving, hierarchical organization can be used in conjunction with divide and conquer. A large problem is not simply divided into a flat collection of parts. Rather, the problem is divided into parts, sub-parts, sub-sub-parts, and so on. Within each part/sub-part decomposition, the sub-parts are a logically related collection of pieces, organized to solve a particular aspect of the problem.

Part/subpart decomposition is pervasive in software engineering. For example, software requirements are organized into major categories of functionality. These are in turn divided into functional sub-categories. At the bottom of the hierarchy are specific use cases, each one of which focuses on a particular aspect of user-level functionality.

Hierarchy is used extensively in program design and implementation. In the Java language, for example, programs are hierarchically decomposed into packages, classes, methods, and data. Methods are organized into invocation hierarchies, i.e., how methods call other methods. Data are organized in predominantly two forms of hierarchy -- aggregation and inheritance. Aggregation is defined by the hierarchy of class data membership. Inheritance is defined by the class/sub-classes relationships, well-known in object-oriented program development.

Hierarchy is widely used in human organization. The "org chart" is a familiar way to depict the hierarchical structure of people within an organization, including a software development team. High-level project managers are near the top of the hierarchy, with individual developers nearer the bottom.

The use of hierarchy occurs in many other aspects of software engineering including user interfaces, modeling, testing, and documentation. These will be expanded upon in upcoming chapters.

1.6.3. Multiple Views

When a problem has been divided and hierarchically organized, the pieces can be viewed from different perspectives. Assuming multiple perspectives helps the viewer see things that may go unnoticed when only a single perspective is taken.

The project stakeholders described in Section 1.2 can have widely varying viewpoints on a developing software product. Considering their varying perspectives is critically important to a project's success. To

do so, the software process must involve regular communication among the stakeholders, so their different viewpoints can be shared and reconciled.

Multiple views can also be based on different representations of software artifacts. *Data-oriented* and *function-oriented* representations provide two alternative views of software components. In the data-oriented representation, the software is viewed primarily as collection of data objects, with functions belonging to those objects. In a function-oriented view, the software is viewed primarily as a collection of functions, with data passed among the functions. Neither view is *the correct* way to look at software. Both views can be useful for understanding a complex software system, throughout its development.

Another aspect of multiple viewing is the use of two or more people on the same task. For highly technical tasks, paired work teams may help reduce errors. A common example of this is the overlapping duties performed by the pilot and co-pilot of a large aircraft. While flight procedures are very well known to the pilot, having the co-pilot explicitly check the pilot's work is a well-accepted means to avoid oversights. In software development, the concept of *pair programming* has been reported to be effective in design, implementation, and testing [Williams 00]. In this style of development, two programmers work side-by-side, on the same computer, to develop a single component of software. Under the proper circumstances, their paired work on the same task can produce better results than a single programmer working alone.

There are many other aspects software development where the use of multiple views can be beneficial. These will be discussed in upcoming chapters.

1.7. The Role of Formal Methods in Software Engineering

A "formal method" is one based on formal mathematical principles. At the implementation stage of development, software engineering is an inherently formal activity. This is because programming languages are based on formal principles. Not all programming languages are defined in terms of fully formal mathematics, however there is no question that programming is based on principles of formal mathematical logic. The colloquial term "program logic" is often used to describe the behavioral portions of software. Program implementors work with Boolean logic and discrete mathematics all the time. Hence, formal methods at the level of programming are nothing new to the software engineer.

As commonly used in the field of software engineering, the term "formal methods" means more than formalization at just the implementation level. In particular, it means applying formal mathematical reasoning to the requirements, specification, design, and testing phases of software development. It is at these levels that formal methods are far less widely used in the practice of software engineering.

For too many software engineers, there is an unfortunate mind-set that "formal" is synonymous with "not practical". A major goal of this book to demonstrate that this is not the case. Formal mathematical notations can be of significant practical value in a number of areas, including the following:

- thorough understanding of software requirements
- precise definition of a software design
- automated generation of program test plans

When analyzing requirements, formal methods can be described as a means to "keep the analyst honest". It is often surprising to discover just how vaguely understood some software requirement is until one tries to formalize the requirement. Formalization can reveal requirements flaws in the form of oversights or inconsistencies that could be very difficult to recognize otherwise. While such flaws can be be discovered when the requirements are implemented, they can be discovered too late, when their correction is more

costly in terms of program redesign than had a flaw been recognized earlier in the requirements. Worse yet, critical flaws in a requirements specification can go entirely undiscovered by a program design and implementation team. In some cases, such flaws can contribute to catastrophic software failures, as in the case of the Ariane 5 rocket [Lions 96]. With requirements for large and complex software, the use of formal methods has significant practical benefits in *requirements interaction management* [Robinson 03]. This is the task of discovering and managing critical relationships among the components of a large requirements specification.

In the design stage of software engineering, a practical technique based on formal methods is *design by contract*^{TM} [Mitchell 01]. With design by contract, the input/output behavior of program functions is defined using formal mathematical logic. The formal definition of a function is a precise *contract*, by which users of the function can be sure of its input requirements and output results. Having a precise definition of function behavior can be of significant practical value to the software designer, who needs to understand clearly what functions do in order to design a program that uses those functions.

In the area of testing, the use of formal methods can be quite practical when it comes to the generation of program testing plans. When program functions are defined using formal logic, *unit tests* for the functions can be automatically generated. Unit tests are used to exercise the functions and to report the test results, so that any errors detected during testing can be repaired before the program is delivered. There are commercially available tools that automatically generate unit tests from formal function definitions, for example JTest [Parasoft 01]. The use of such tools can have significant practical benefit, since the activity of manually generating unit tests can be quite tedious and time-consuming.

Formal methods are not applicable to all software artifacts. For example, the user-readable form of software requirements must be written in language that end users understand, which is most typically a human natural language like English. Even for artifacts that are not fully formalizable, having a formal mind-set can still be useful, in the sense that care is taken to be as precise as possible when writing documents in English.

It is important for advocates of formal methods not to over-sell the benefits. Formal methods are no panacea. The use of formal methods cannot guarantee that software will never fail. Guarantees about failure cannot be made for any engineering discipline, however formal its methods. For example, modern civil engineers use highly formal analysis in their work, but the structures they design do still fail sometimes. What using formal methods can do is give the engineer, software and otherwise, greater confidence that an engineered artifact will function correctly.

There is no question that using formal methods can be time consuming in the early stages of software development. It is the contention of formal methods advocates that this is time well spent, and that when the final product is delivered, it will work better, with less post-release time being spent on defect repair.

As noted earlier, there have been few studies that validate or invalidate the utility of formal methods in software engineering. Two notable studies with somewhat conflicting results are presented in [Pfleeger 97] and [Sobel 02]. There are many articles in the literature debating the pros and cons of formal methods. The positive or negative support in these articles is based largely on well-reasoned argumentation, rather than solid empirical data. More empirical studies need to be conducted.

A recurring point raised in the literature on formal methods is that they are not widely used in the practice of software engineering. Authors have cited varying reasons for this under-utilization, prominent among them being a lack of adequate training in formal methods. It is in the area of training where this book is intended to make a contribution, by explaining the techniques of formal methods thoroughly, and by providing concrete practical examples of their use. The reader may judge whether the practical benefits of

formal methods have in fact been demonstrated.

1.8. The Issue of Software Quality, or Lack Thereof

At the first NATO Software Engineering conference in 1968, the participants discussed a "crisis" in software development [Naur 68]. As perceived by a number of conference contributors, the crisis was due to software practices being inadequate to cope with the increasing size and complexity of software systems. As a result, software quality was low, particularly in terms of software failing to operate correctly.

Nearly thirty-five years after the NATO conference, the cover article in a prominent technology magazine was entitled "Why Software Is So Bad" [Mann 02]. While not proclaiming a software crisis, articles such as this represent a view held by many about software products. The view is that significant improvements still need to be made in software engineering practices if the quality of software is to improve.

There are nearly as many definitions of "software quality" as there are authors who write about it. Here is a simple definition that captures what this book's author believes are the two key criteria of a quality software product:

- the product works correctly
- the users like it

What it means for a product to "work correctly" is that it functions according to its specification. This in turn means that everyone concerned with the product must understand the specification. For end users, the understandable version of a specification typically comes in two forms. During product development, user-level requirements present the specification in terms understandable to the end user. After delivery[1], user-level product documentation describes what the software does, as does actual product behavior.

For the software developers, the specification needs to be precise enough to ensure that the implementation works correctly. To meet this need, this textbook advocates the development of a more formal version of the product specification, with technical details not normally understandable to the end user. The point of this more formal specification is to ensure that the developers understand the product precisely. As the specification is refined, the analysts' formal understanding is (re)conveyed to the users in language they can understand, namely the requirements document, user documentation, and actual program behavior.

A formal specification is key to ensuring that the product works correctly from an engineering standpoint. What it means formally for a program to work correctly is that each and every component of the program implementation functions according to the specification of that component. Without a specification that is precise, complete, and unambiguous, it can be difficult to ensure fully correct operation from an engineering standpoint.

The lack of a sufficiently precise specification is a typical cause for poor software quality in terms of the "works correctly" quality criterion. Consider the following scenario. A user has a clear idea of what a custom software product is supposed to do. The user consults with a software analyst and they work out the details of the product requirements. The analyst then communicates these requirements to the implementors, who build a program. Upon delivery of the program, the customer says "Hey wait, this isn't what it's supposed to do. What's going on here?"

---

[1] For an iterative development process, delivery may come in a number of phases.

What is "going on" in the preceding scenario can be common in software development. Somewhere in the chain of communication from user to analyst to implementor, the user's "clear" idea was not properly communicated. There are a number of approaches that have been proposed to solve this problem. One solution is to ensure that the requirements specification is written in a clear and precise enough form that no one misunderstands it. Another solution is to shorten the communication chain between customer and implementor, so that they communicate more directly and more frequently. With either or both of these solutions, the fact remains that a clearly understood requirements specification is a key to correctly operating software, and hence to quality software. In the article cited above on bad software, the authors concluded that poor understanding of requirements is a major causative factor of poor-quality software.

The second quality criterion of a user "liking" the software is more difficult to measure than correct operation. Liking a product can be based on a wide range of subjective product qualities, including the way it looks and how easy it is to use. The best way to achieve these kind of qualities is by fully involving the user in the analysis and testing of product requirements. In this way, the user has the opportunity to evaluate the product before it is delivered, to ensure that it has the necessary likeable qualities. Developers should not underestimate the psychological benefit of end-user involvement. A user who participates fully in the requirements process has a sense of investment and ownership in a product. This sense can very positively impact the user's perception of product quality, that is, how much the user likes it.

For a custom software product, the software analyst should involve as many actual end users as possible throughout the requirements analysis and end-user testing phases of development. For off-the-shelf products, those who develop the requirements must be sure that they fully understand and represent the ideas of the ultimate product users. Some prominent software flops have been the result of a developer's "really cool idea" that resulted in a product that no real users wanted. The moral of such flops is that one can develop software that works brilliantly well, but that is worthless because no one wants to use it.

This discussion of software quality has been user-centered. There are other aspects of software quality that are invisible to the end user. These aspects have to do with the engineering process for building the software, and the internal software artifacts that the end user rarely or never sees. Internal artifacts include the formal specification, the program design, and the program implementation. Quality criteria for these and other internal artifacts are discussed in later chapters of the book, in the context of the specific techniques used to build those artifacts.

References

[Fisher 06]
Fisher, G. A Formal Modeling and Specification Language, Version 4, Department of Computer Science Technical Report, California Polytechnic State University, San Luis Obispo (December 2006).

[Gamma 95]
Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns,* Addison-Wesley (1995).

[Jacobson 99]
Jacobson, I., G. Booch, and J. Rumbaugh. *The Unified Development Process,* Addison-Wesley (1999).

[Lions 96]
Lions, J. L. *ARIANE 5 Flight 501 Failure, Report by the Inquiry Board*, Paris (19 July 1996).

[Mann 02]
Mann, C. C. Why Software Is So Bad, *MIT's Technology Review* **105**(4)(July/August 2002).

[Mitchell 01]
Mitchell, R. and J. McKim. *Design by Contract by Example,* Addison-Wesley (2001).

[Naur 68]
Naur, P., B. Randell, and (Eds.). *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee,* Scientific Affairs Division, NATO (October 1968).

[Parasoft 01]
Parasoft. *Using Design by Contract™ to Automate Java™ Software and Component Testing,* Parasoft Corporation (2001).

[Pfleeger 97]
Pfleeger, S. L. and L. Hatton. Investigating the Influence of Formal Methods, *IEEE Computer* **30**(2) p. 33-43 (February 1997).

[Robinson 03]
Robinson, W. N. and S. D. Pawlowski. Requirements Interaction Management, *ACM Computing Surveys* **35**(2) p. 375-408 (June 2003).

[Rumbaugh 98]
Rumbaugh, J., I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual,* Addison-Wesley (1998).

[Simon 69]
Simon, H. A. *The Sciences of the Artificial,* MIT-Press (1969).

[Sobel 02]
Sobel, A. E. K. and M. R. Clarkson. Formal Methods Application: An Empirical Tale of Software Development, *IEEE Transactions on Software Engineering* **28**(3) p. 308-320 (March 2002).

[Williams 00]
Williams, L., R. R Kessler, and W. Cunningham. "Strengthening the Case for Pair Programming", *IEEE Software* **17**(4) p. 19-25 (July/August 2000).

[World Wide Web Consortium 99]
World Wide Web Consortium. *"HTML 4.01 Specification"*. December 1999.