The Science Authority

Software engineering

Article by:

Frakes, William B. Virginia Polytechnic Institute and State University, Northern Virginia Center, Falls Church, Virginia.

Publication year: 2014

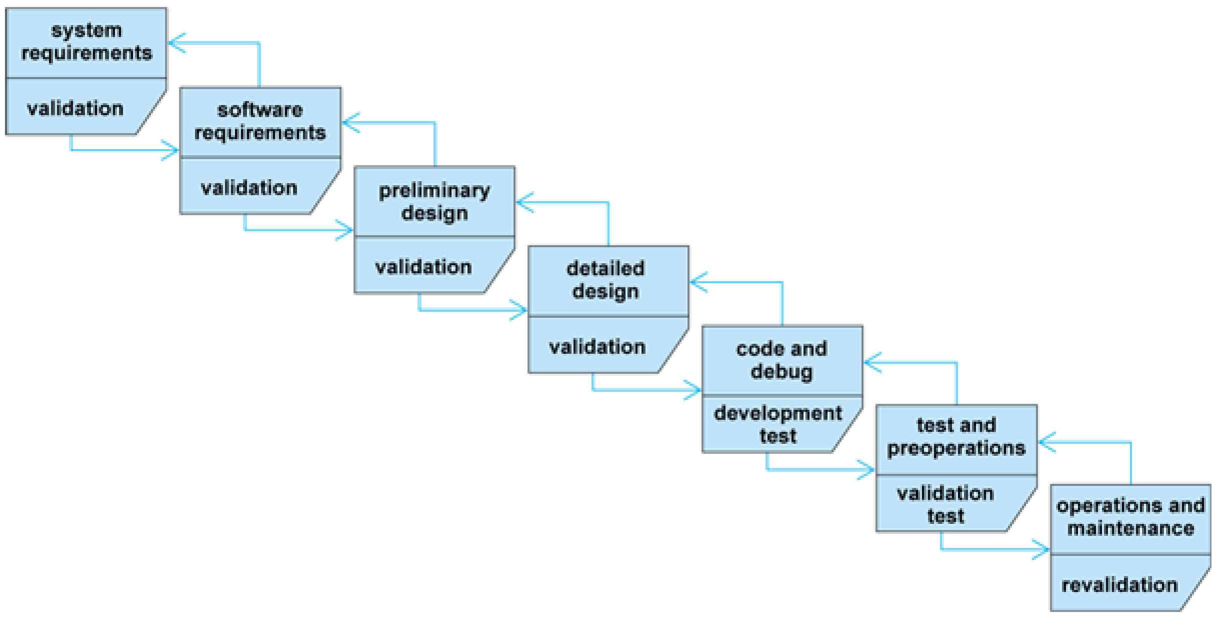
DOI: http://dx.doi.org/10.1036/1097-8542.631400 (http://dx.doi.org/10.1036/1097-8542.631400)

Content

The process of manufacturing software systems. A software system consists of executable computer code and the supporting documents needed to manufacture, use, and maintain the code. For example, a word processing system consists of an executable program (the word processor), user manuals, and the documents, such as requirements and designs, needed to produce the executable program and manuals.

Software engineering is ever more important as larger, more complex, and life-critical software systems proliferate. The rapid decline in the costs of computer hardware means that the software in a typical system often costs more than the hardware it runs on. Large software systems may be the most complex things ever built. This places great demands on the software engineering process, which must be disciplined and controlled.

To meet this challenge, software engineers have adapted many techniques from older engineering fields, as well as developing new ones. For example, divide and conquer, a well-known technique for handling complex problems, is used in many ways in software engineering. The software engineering process itself, for example, is usually divided into phases. The definition of these phases, their ordering, and the interactions between the phases specify a software life-cycle model. The best-known life-cycle model is the waterfall model (see **illustration**) consisting of a requirements definition phase, a design phase, a coding phase, a testing phase, and a maintenance phase. The output of each phase serves as the input to the next. *See also:* **Systems engineering (/content/systems-engineering/676000)**



Waterfall software life-cycle model. (*After B. W. Boehm, Software engineering, IEEE Trans. Comp., C-25:1226–1241, 1975*)

Requirements

The purpose of the requirements phase is to define what a system should do and the constraints under which it must operate. This information is recorded in a requirements document. A typical requirements document might include a product overview; a specification of the development, operating, and maintenance environment for the product; a high-level conceptual model of the system; a specification of the user interface; specification of functional requirements; specification of nonfunctional requirements; specification of interfaces to systems outside the system under development; specification of how errors will be handled; a listing of possible changes and enhancements to the system; and other auxiliary information such as a glossary and index. Each requirement, usually numbered for reference, must be testable. The requirements definition process is difficult for several reasons.

Developing requirements requires that the customers of a system be identified and that their needs be understood. There may well be communication problems between customers who know their business lines but may be unfamiliar with computer technology, and software engineers who know computer technology but do not understand customers' business lines. Since there is a time lag between the requirements definition and the delivery of the system, the requirements of the customer may change while the system is being built. This problem is known as requirements volatility.

Another problem stems from the language that is used to express the requirements. Requirements stated in a natural language, such as English, are expressive and understandable by users of the system but may be imprecise. Formal languages such as predicate calculus, set-theoretic notations, and finite-state machines are precise but may lack expressiveness and may be difficult or impossible for end users of the system to understand. In practice, most requirements documents are written in natural language, but formal languages are sometimes used for parts of the system. Finite-state machines, for example, are often used to specify telecommunication system activities such as call handling.

The traceability problem refers to the difficulty of cleanly mapping a given requirement to its implementation in design and code. This makes it difficult to determine which parts of a system implement a given requirement and whether a given requirement has been completely and accurately implemented.

To deal with these problems, many approaches to requirements definition have been tried. One of the more successful is the

JAD (joint application design) method, which uses a focused multiday meeting of customers and developers to produce a requirements document. JAD uses rapid prototyping to help specify user interfaces. Data collected from JAD sessions indicate that significant improvements in requirements productivity and quality can be achieved using JAD.

A prototype is a partial implementation of a software system. Throwaway prototypes are built to clarify or validate requirements or to explore feasibility and then are discarded. A breadboard prototype is built as the initial part of a system and is not discarded. A key to prototyping is to use a language tool that allows rapid prototype creation. For example, interface generators allow user interfaces to be drawn, and then the system generates the code for the interface.

Design

In the design phase, a plan is developed for how the system will implement the requirements. The plan is expressed using a design method and notation. Many methods and notations for software design have been developed. Each method focuses on certain aspects of a system and ignores or minimizes others. This is similar to viewing a building with an architectural drawing, a plumbing diagram, an electrical wiring diagram, and so forth.

Principles

Principles of software design have also emerged. Abstraction, the suppression of details, is used throughout the design process to identify key design objects and operations. Separation of concerns guides the decomposition of a system into smaller parts that can be solved separately. Reusable patterns and components are used to improve system quality and reliability. Design tools support design tasks, though they primarily support the drawing of diagrams using a particular design notation and database functions, rather than guiding the design process per se.

Researchers have identified several common architectural styles for software systems. These include the pipeline architecture where the system is viewed in terms of data flowing through a pipe and transformed by data filters, data abstraction where each subsystem hides a type representation, a repository architecture where computational processes write to and read from a common data store, and a layered architecture where each architectural layer provides certain services and hides lower levels of abstraction.

A key method for handling the complexity of large software systems is modularity. A system is modular if it is composed of simple independent parts that communicate through well-defined interfaces. The conceptual simplicity of the modular system parts allows them to be more easily documented and tested and to be better maintained.

Another key method for handling design complexity is information hiding, a decomposition method where each module hides internal processing details from other modules. Each module thus has two parts; a public part with information needed to use the component, and private information such as implementation details that a user does not need to know.

Two other important design principles are coupling and cohesion. Cohesion refers to the degree to which parts of a program unit, for example functions in a module, are related. A highly cohesive module performs a single task and can usually be described by a specific verb and object like “print a file” or “sort a list.” The best type of cohesion is among elements that jointly implement an abstract data type. More cohesive modules tend to be more modular and to be easier to understand and document. Coupling refers to the strength of the linkage between two modules, based on the amount and kinds of information they exchange. Minimal coupling is a design goal. The worst type of coupling is among modules that share global data (data that are defined outside the scope of individual modules). The best is coupling where modules manipulate the same data type.

Design methods

The most common design methods are structured design, data structure design, dataflow design, and object-oriented design. Each of these methods has a defined process to guide design and a set of notations for recording the design. One of the earliest methods of design focused on the flow of control in a software system based on program branching and functional activities. This information was recorded in flowcharts which provided graphical symbols for procedures, decisions, starting and stopping, and connecting pages of the diagram. This method has fallen into disfavor because flow of control is thought to be a low-level implementation detail that should be specified at the end, rather than the beginning, of the design process.

Structured design, sometimes called top-down design, begins with a high-level functional view of a system and progressively decomposes it into lower and lower levels of detail. Structure charts are one notation for recording top-down designs. This notation provides symbols for system units at various levels of decomposition, and connectors that show how data are passed among units. Structured design is widely used and is fairly easy to understand, but fails to provide detailed guidance for system decomposition.

Dataflow design focuses on the data that flows through the system and on the processes needed to transform it. Flow of control issues are deferred until the end of the design process. Dataflow notation provides symbols for data flowing through the system, data transforms, logical alternation, and the beginning and end of data processes. Data dictionaries provide precise definitions of the data in the system, and pseudo-code is used in the final steps of the design process to record flow of control sequences. Dataflow design provides a complete method, and dataflow notation is useful in communicating system development issues to end users. The method is less appropriate for systems that are algorithmically intense, such as mathematical software. The method also provides little opportunity to incorporate abstract data types. *See also:* **Dataflow systems (/content/dataflow-systems/181000)**

A data type is a set of data values and allowed operations on those values. For example, an integer data type has integers as its data values and the operations addition, subtraction, multiplication, and division. An abstract data type is one with implementation details abstracted away, providing an implementation-neutral mathematical abstraction. Formally defining an abstract data type involves developing a name for it, defining the sets needed for its description, defining the syntax of its operations, and defining the semantics of the operations. *See also:* **Abstract data type (/content/abstract-data-type/001750)**; **Data structure (/content/data-structure/757547)**

Data abstraction is a key idea in object-oriented design, which views a software system as a collection of objects which communicate via message passing. An object is an encapsulated collection of data and operations, called methods. A message consists of the name of an object, the name of one of its methods, and the parameters needed to use the method. A class is a specification of the shared features of related objects. Objects belonging to a class are said to be instances of the class. Inheritance is the process by which one class acquires the variables and operations of another. *See also:* **Object-oriented programming (/content/object-oriented-programming/757337)**

Coding

The coding phase of the software life-cycle is concerned with the development of code that will implement the design. This code is written is a formal language called a programming language. Programming languages have evolved over time from sequences of ones and zeros directly interpretable by a computer, through symbolic machine code, assembly languages, and finally to higher-level languages that are more understandable to humans. *See also:* **Programming languages (/content/programming-languages/547550)**

Most coding today is done in one of the higher-level languages. When code is written in a higher-level language, it is translated into assembly code, and eventually machine code, by a compiler. Many higher-level languages have been developed, and they can be categorized as functional languages, declarative languages, and imperative languages.

Functional languages are modeled on mathematical function theory. Computations in these languages, such as Lisp and ML, are specified using the application of functions.

Declarative languages are modeled on formal logic. Computations in these languages, such as Prolog, which is based on a subset of predicate calculus, are specified by stating facts and using logical implication.

Imperative (algorithmic) languages, such as Fortran, Cobol, and C, use a precise series of instructions to specify computations. Object-oriented languages, such as C++ and Java, are a special kind of imperative language that adds constructs to support object-oriented programming. These constructs include class declarations and inheritance. Almost all engineering of large software systems is done using an imperative language.

Following the principle of modularity, code on large systems is separated into modules, and the modules are assigned to individual programmers. A programmer typically writes the code using a text editor. Sometimes a syntax-directed editor that “knows” about a given programming language and can provide programming templates and check code for syntax errors is used. Various other tools may be used by a programmer, including a debugger that helps find errors in the code, a profiler that shows which parts of a module spend most time executing, and optimizers that make the code run faster.

Testing

Testing is the process of examining a software product to find errors. This is necessary not just for code but for all life-cycle products and all documents in support of the software such as user manuals.

The software testing process is often divided into phases. The first phase is unit testing of software developed by a single programmer. The second phase is integration testing where units are combined and tested as a group. System testing is done on the entire system, usually with test cases developed from the system requirements. Acceptance testing of the system is done by its intended users.

Testing can be done either top-down of bottom-up. In top-down testing, high-level routines are implemented and tested and then used as a testing environment, called a test harness, for lower-level routines. Bottom-up testing proceeds by first developing low-level routines, testing them, and then progressively combining and testing them as parts of larger and larger program units. In practice, both methods are used together in a process called sandwich testing.

The basic unit of testing is the test case. A test case consists of a test case type, which is the aspect of the system that the test case is supposed to exercise; test conditions, which consist of the input values for the test; the environmental state of the system to be used in the test; and the expected behavior of the system given the inputs and environmental factors.

Test cases can be either black-box or white-box. Black-box test cases are developed without reference to the internal structure of the code under test—that is, the code is treated as a black box. In white-box testing, the internals of the system are examined to help develop test cases.

Ideally, one would like to test all possible input-output-environment combinations, but this is impossible because such a set is typically very large, or even infinite. To address this problem, a method called equivalence partitioning is used. In equivalence partitioning, the test input and environment space are divided into classes based on common attributes. A few members of each class are then selected as representative test cases for the class.

Since it is impossible to test software exhaustively, various heuristics are used to allocate test cases to different portions of the system. One heuristic is to focus testing on the parts of the system that are most crucial. Another is to allocate testing resources on estimates of where the most errors are likely to be found, based on the size or complexity of system parts or on the parts that consume the most processing time.

Another practical problem is knowing when enough testing has been done and the product is ready to be released. This can be addressed using a cumulative fault plot, a bivariate graph of the cumulative faults found via testing versus time. Such a plotted curve typically displays an S shape since at first few faults are found as the testing process gears up, then the curve rises steeply as most of the errors are found, and finally the curve flattens again since the remaining errors are more difficult to detect. A system is considered ready to ship when the final part of the curve remains consistently flat.

In white-box testing, the aim is to ensure that all of the code has been tested. This measurement can be in terms of lines of code, branch points, or execution paths through the program. Tools to support white-box testing instrument (add tracking code to) assembly or source code in order to keep track of which parts of the code have been exercised by test cases and which have not.

The coverage analysis process uses this information to design test cases that cumulatively test the code to the desired percentage. The process is to first examine the code to see which parts have not been exercised, design test cases to exercise the untested code, run the test cases, and then run a coverage analysis tool. If the desired level of test coverage has been reached, the process ends; otherwise, the process is repeated.

When software is changed to fix a bug or add an enhancement, a serious error is often introduced. To ensure that this does not happen, all test cases must be rerun after each change. The process of rerunning test cases to ensure that no error has been introduced is called regression testing. Various tools, including Unix shell scripts, can be used to support regression testing. *See also:* **Software testing (/content/software-testing/757613)**

Walkthroughs and inspections

Walkthroughs and inspections are used to improve the quality of the software development process. Consequently, the software products created by the process are improved. A quality system is a collection of techniques whose application results in continuous improvement in the quality of the development process. Elements of the quality system include reviews, inspections, and process audits.

A quality system should address two different aspects of quality: process and product quality. Process quality is the fitness of the development process for producing a product, while product quality is the fitness for use of the product resulting from some process. Quality systems should answer two basic questions about process and product quality: Are we building the right product (assessing product quality)? Are we building the product right (assessing process quality)?

Many of the things that must be tested in the software engineering process are not amenable to the kinds of techniques discussed for software. For the many textual documents that are typically produced for a software product and for software engineering processes, the best available testing methods are walkthroughs and inspections. Since code is also a kind of document, it can be tested with walkthroughs and inspections. Several types of walkthroughs and inspections are used.

A software review is an objective evaluation of a software engineering product or process. Though informal reviews, lacking systematic procedures and structures, may be adequate for small simple projects, larger projects will benefit from formal reviews. According to data collected on many projects, inspections can detect errors earlier in the life-cycle. This is important because the cost to repair errors rises rapidly as the software life-cycle progresses. Earlier detection via inspections can give up to 25% improvements in productivity. Other data indicate that defects found by inspections can reduce costs tenfold.

Code inspections are group readings of code by its designer, programmer, and tester. A moderator manages the inspection.

During inspection, the programmer reads the code aloud and describes the intent of each statement. Errors are documented and categorized, but no attempt is made to fix the error during the inspection; this is done later. Average inspection rates for code are about 90 noncommentary source lines per hour. The inspection process can be guided by an error checklist of kinds of errors that might be encountered such as data declaration errors, computation errors, and control flow errors.

Another kind of inspection is a process audit whose purpose is to improve quality and productivity by improving the software production process. The best-known process evaluation for software is the SEI (Software Engineering Institute) process capability maturity model (CMM). This model is based on Philip Crosby's model for general process quality improvement. Crosby's model is based, in turn, on one of the basic principles of modern quality theory, which is that the best way to assure the quality of a product is to control the process which produces it. *See also:* **Quality control (/content/quality-control /561950)**

The CMM is an evaluation model that classifies a process as being at one of five levels: (5) Optimizing — constant improvement based on measurement and feedback. (4) Managed—measured process and quality and productivity. (3) Defined—process defined and institutionalized. (2) Repeatable—process dependent on individuals, but basic project controls established. (1) Initial—no formal procedures or management control. The highest level, 5, includes continual process improvement, another hallmark of modern quality theory. Experience has shown that most software organizations today are at level 1, and only a very few are at level 5.

Maintenance

Large software systems are not static; rather, they change frequently both during development and after deployment. Maintenance is the phase of the software life-cycle after deployment. The maintenance phase may cost more than all of the others combined and is thus of primary concern to software organizations. The Y2K problem was, for example, a maintenance problem.

Activities

Maintenance consists of three activities: adaptation, correction, and enhancement. Enhancement is the process of adding new functionality to a system. This is usually done at the request of system users. This activity requires a full life-cycle of its own. That is, enhancements demand requirements, design, implementation, and test. Studies have shown that about half of maintenance effort is spent on enhancements.

Adaptive maintenance is the process of changing a system to adapt it to a new operating environment, for example, moving a system from the Windows operating system to the Linux operating system. Adaptive maintenance has been found to account for about a quarter of total maintenance effort. Corrective maintenance is the process of fixing errors in a system after release. Corrective maintenance takes about 20% of maintenance effort.

Software configuration management

Since software systems change frequently over time, an important activity is software configuration management. This consists of tracking versions of life-cycle objects, controlling changes to them, and monitoring relationships among them. Configuration management activities include version control, which involves keeping track of versions of life-cycle objects; change control, an orderly process of handling change requests to a system; and build control, the tracking of which versions of work products go together to form a given version of a software product.

Version control is supported by version control tools such as the Revision Control System (RCS) and Source Code Control System (SCCS). These tools support the storage and retrieval of versions of text files. To use them, a file is put into the change control system. After a version of a file is retrieved and modified, it is resubmitted to the system, which calculates changes between the old and new versions and stores only the changes, rather than both complete versions. This greatly reduces storage overhead. The system assigns unique numbers to versions and variations, allowing an orderly tracking process. A version replaces the original file, and a variation is an alternative to the original file.

Change control must be done in an orderly way on a large project. When a change is requested to some life-cycle work product, a form, sometimes called a modification request (MR), is created and logged into a change control database. A review board decides the severity of the problem described in the modification request and how the modification request should be handled, and then assigns personnel to address the handling of it.

Build control is concerned with how to put various versions of life-cycle work products together to build a system or part of a system. The make tool on UNIX, for example, allows the specification of which versions of software modules are needed to build an executable program, and the commands to actually do the build. In the C language, for example, an executable file depends on the object and library files needed to generate it. These, in turn, depend on source code files written in the C language, which are used by the compiler to create the object and library files.

Software reuse

The methods described so far, while effective in many cases, may not be adequate for building systems of the size and complexity sometimes required. One method that has been proposed to address this problem is software reuse, that is, the use of knowledge or artifacts from previous systems to build new ones. While software reuse has always been done in some way, recent research has attempted to get large productivity and quality gains by focusing on reuse that is systematic and domain-based.

A key idea in systematic domain-based reuse is that organizations seldom build completely new software systems. Rather, they build similar systems with variants to address different customer needs. These similar systems are said to be in a domain, which may be thought of informally as a business line. For example, a company's domain may be word processing software. The company will produce new versions with added functionality over time and may well provide variants for different operating systems.

Part of the impetus for more software reuse has come from considering other engineering disciplines. In electrical engineering, for example, systems are constructed from standardized components such as integrated circuits, resistors, and capacitors. A study of the history of hardware manufacturing has shown that that field made the transition from craft-based manufacturing to manufacturing based on interchangeable parts around 1840. Software production is in this transition process now.

Software reuse can be either horizontal or vertical. Horizontal reuse is reuse across domains. Much reuse is horizontal. For example, standard function libraries that support activities like searching and sorting are of this kind. Vertical reuse is reuse within a domain. For example, a component library might be developed specifically for word processing systems that includes modules to handle text buffers, font management, and so on.

Actions that can produce more reuse fall into four categories: economic, legal, technical, and managerial. Clearly reuse improvement is undertaken to produce some benefit. The bottom-line benefit for a company will be greater profits. More immediate benefits, presumed to improve profits, include higher productivity, better quality, faster time to market, and improved bid and late life-cycle estimation. There is evidence that reuse can indeed produce these results under the right circumstances. These circumstances will be different for different organizations, requiring the proper blend of managerial, technical, legal, and economic factors.

Managerial factors include a rewards program for producers and consumers of reusable assets. Making an asset reusable will require more effort than making an equivalent one-use asset. This additional cost must be amortized across the multiple uses of the component. Incentives for producers of components must be established. Some companies, for example, provide cash bonuses for reusable assets accepted by the company reuse library. Since most software engineers are not currently trained to do systematic reuse and domain engineering, management must also provide education and motivation.

Legal factors may become the greatest barrier to systematic reuse. Current laws regarding software copyrights, patents, and liabilities are unclear. For example, if a company sells an asset and it is integrated into an application that fails because of the asset, causing loss of life or property, does the purchasing organization have the right to sue for damages? Another problem is that many software algorithms are now being patented each year. This may, in the future, provide a disincentive to reuse.

Technical factors include the issues of how to create reusable assets, how to design new applications to take advantage of them, and what kind of software engineering environment is needed to support reuse. Other technical factors concern what kind of software development process is needed and how to create and maintain a reuse library.

Economic factors include how to do a cost benefit analysis of reuse, performance measurement of a reuse program, methods for system costing, product pricing criteria and strategies to take reuse into account, and how to finance support costs for a reuse program.

A reuse library consists of a database for storing reusable components and a search interface that allows users to find and retrieve items in the database. Components for the database can be purchased, built, or reengineered from existing software. Before being added to the library, the components must be certified to assure that they meet quality standards, and must be classified.

The most common method in classifying reusable components is enumerated classification, where the subject area is broken into mutually exclusive, usually hierarchical classes. Other classification methods are faceted classification where the subject area is analyzed into basic terms which are organized as attribute-value pairs, and free-text keyword where index terms are extracted directly for the reusable component description. Reuse library tools should ideally be integrated with tools for configuration management, since the reusable assets in the database will change as error corrections and enhancements are made.

Domain engineering

Domain engineering is the process of analyzing a domain, recording important information about the domain in domain models, creating reusable assets, and using the domain models and assets to create new systems in the domain. It has two phases: domain analysis and domain implementation.

Domain analysis is the process of finding common and variable parts of systems in a domain and constructing domain models. Code, documents, and expert information are used as inputs to the domain analysis process. A domain vocabulary is developed which is used to describe systems in the domain. A key model that must be developed is a generic architecture that describes all of the systems in the domain. Reusable components will also often be identified in the domain analysis process.

These models are used to develop tools to speed the process of developing systems in the domain. One tool is domain-specific components that can be integrated into systems in the domain. Domain-specific programming languages can also be developed. These languages can be either large complete languages that include domain-specific operators and operands, or little languages that are small domain-specific languages developed for a specific purpose. Another approach, if sufficient formal information about the domain is known, is to develop application generators.

Application generators are programs that produce all or most of a software system when given a high-level specification. Application generators can be particularly powerful tools for reuse, giving great quality and productivity improvements. To develop an application generator, a language must be developed that allows the desired system to be specified. A specification in this language is then passed to the generator that produces the code for the application in the desired target language. Lex, for example, is an application generator for lexical analyzers. The specifications for a desired lexical analyzer are written in the lex language. The lex generator then produces the C code to implement the lexical analyzer. *See also:* **Software (/content/software/757374)**

William B. Frakes

Bibliography

B. I. Blum, *Software Engineering: A Holistic Approach*, Oxford University Press, 1992

E. J. Braude, *Software Engineering: An Object-Oriented Perspective*, Wiley, 2000

M. Dorfman and R. H. Thayer (eds.), *Software Engineering*, IEEE Computer Society, 1996

R. S. Pressman, *Software Engineering: A Beginner's Guide*, McGraw-Hill, 1988

R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th ed., McGraw-Hill, 2000

I. Sommerville, *Software Engineering*, 6th ed., Addison-Wesley, 2000

Additional Readings

C. E. Otero, *Software Engineering Design: Theory and Practice*, CRC Press, 2012

V. Rajlich, *Software Engineering: The Current Practices*, CRC Press, 2012

F. Tsui, O. Karam, and B. Bernal, *Essentials of Software Engineering*, 3d ed., Jones & Bartlett Learning, 2014

J. Xiong, *New Software Engineering Paradigm Based on Complexity Science: An Introduction to NSE*, Springer Science+Business Media, 2011

Carnegie Mellon University: Software Engineering Institute (http://www.sei.cmu.edu/)

IEEE Transactions on Software Engineering (journal) (http://www.computer.org/portal/web/tse)