Software engineering

Article by:

Gupta, Pranshu Department of Mathematics and Computer Science, DeSales University, Center Valley, Pennsylvania.

Publication year: 2015

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

Content

The production of software through a progression of disciplined and controlled steps. As the need for complex, real-time, and life-critical software proliferates, there is a demand for an effective software development process that guarantees the correctness and quality of the software produced. To meet this challenge, the computer science community has developed a process called software engineering that aims to build software that meets all the specifications for its intended use. The term engineering requires that software be produced with the objective of delivering a successful final product with as much certainty as that of civil engineers constructing a building or a road. For this reason, it is necessary that software engineers design software with a full understanding of the intended design and the certainty of its correctness and reliability of operation in a specific operating environment. In other words, software engineers should aim to develop the counterpart of the civil engineer's “blueprints” to guarantee the functionality, correctness, and behavior of the required software. However, whereas civil engineers have been constructing roads, buildings, and bridges for centuries, software has been built only since the early 1940s. It is for this reason that software engineering, as currently viewed, is an evolving discipline for manufacturing software systems that, in a typical modern configuration, have many interacting software and hardware components, with the purpose of accomplishing specific tasks, and that include, as an essential component of the final system, documentation to substantiate the needs of the system and how it operates. *See also:* **Software (/content/software/757374)**

Software engineering process

A number of techniques have been defined for developing software systems, some of which are considered to be among the most complex and difficult products ever built. As mentioned earlier, software engineering, following the lead of its most mature siblings such as civil and hydraulic engineering, has adopted and adapted the strategy of “divide and conquer” to make the development of software and its ever-increasing complexity more manageable. Therefore, to better accomplish this task, the software engineering process is divided into phases or developmental stages, such as requirements elicitation, design, coding, testing, debugging, deployment, and maintenance. The definition of these phases, their ordering,

and their interactions are known as the software development life-cycle model. Currently, there are different models for, or approaches to, the software development process. Among the most popular ones are the waterfall, prototype, incremental, iterative, spiral, scrum, rapid application development, and agile models. In general, during the software development process, the output of each phase serves as the input to the next one. Although there may be some reiteration among some consecutive phases, the aim is that, as a result of this progressive developmental process, the final product is a correctly working system that satisfies all the requirements and the users' needs and, at the same time, is easy to understand and maintain. Each phase of the software development life cycle will be discussed.

Requirements elicitation

The purpose of the requirements elicitation phase is to record and document the customer or system requirements of the perceived system and its operating constraints. In other words, in this phase, we are trying to answer the question, “What is the system supposed to do?” A typical requirements document includes a product overview, functional and nonfunctional characteristics, system performance, user interface specifications, development specifications, the operating and maintenance environment for the system, a high-level conceptual model of the system, error-handling specifications, potential enhancements to the system, and other auxiliary information such as a user's manual, glossary, and index. This requirements elicitation phase is generally accomplished by using a combination of techniques and tools, such as interviews, questionnaires, checklists, data-flow diagrams, entity-relationship diagrams, and the like. This process can become difficult as a result of miscommunication among the customer, the business and requirements analyst (BA; acts as a go-between for the customer and the developer), and the developer. Factors that may play a major role in miscommunications among the parties during information gathering are imprecision as a result of a lack of expressiveness or understanding by the customer as to what the system needs to accomplish; the customer's assumption that the BA or the developer already understands the requirements in detail without much clarification; the customer's unfamiliarity with the technology; or the developer's unfamiliarity with the customer's business operations, data flow, and volatility of requirements. The outcome or deliverable product of this phase is a document called the software requirements specification (SRS). The requirements elicitation process is critical, as the outcome and acceptance of the final system depends on the accuracy of the SRS. This is because each function in the system must be mapped to a system requirement for traceability and vice versa. A potential weakness of this process is the incompleteness of the SRS, therefore resulting in a partial picture of the needed system.

Design

Just as the system requirements phase allowed us to answer the “What” question, the design phase will help us answer the question, “How is the system to be implemented?” This phase uses the SRS document as its input and delivers as its output the architecture document, the implementation plan, performance analysis, and a test plan.

The design phase maps the system requirements to an architecture that defines the components of the system, their interfaces, and their behaviors using modeling tools, some of which are briefly described later on. The design documents provide detailed information about programming languages, environments, system architecture, algorithms, and data structures used to implement the system. During this phase, the designers also refer to cost estimation models such as the constructive cost model (COCOMO). COCOMO uses a basic regression formula that includes historical, current, and future project data as input parameters. The effort estimation is equally important. Methods such as the analysis effort method are used to estimate the duration of the project.

Over the years, software engineers have developed different methodologies and notations to express system requirements that try to mimic, to some degree, the blueprints used by architectures, civil engineers, or electricians. As such, their notation is aimed at highlighting what is important to them, while minimizing the other aspects of the design. However, during the software design process, regardless of the notation methodology used, there are certain common principles. The overall objective of these principles is to have a clear understanding of how the pieces fit together before worrying about how they will be implemented. The main principles used are those of abstraction, modularity, information hiding, and coupling and cohesion.

Abstraction, as its name indicates, is concerned with the identification of key objects and functionality that, if possible, can be reused throughout the system. Closely associated with abstraction is the principle of modularity. A module is a simple unit of the system that accomplishes a particular subtask, has a well-defined interface, and can be independently tested. It is the totality and interplay of these modules that allows a particular functionality of the overall system to be achieved. Modules can be thought of as “black boxes” in which the internal details are hidden from other modules. However, modules need to interact, and therefore each one of them has a “public” component that allows communication with the others. In the composition of each module, or even larger units, it is necessary to take into account the relationships of its internal components, or cohesiveness. We can think of cohesiveness as a measure of how well all the components are working together for a common goal. A pragmatic approach for determining whether a module is performing a single task is to try to describe its functionality through a single sentence that contains a single subject, verb, and object. If this is not possible, then the module is performing more than one task and needs to be rewritten. A tighter relationship, or high cohesiveness, in a module is desirable because it makes its functionality easier to understand, test, and document.

In addition to taking into account how the internal components of a module interact to accomplish a task, it is crucial to consider how the different modules interact with one another (coupling). Coupling can be thought of as a measure of the strength of the linkages between modules, based on the amount and type of information that they exchange; that is, how closely connected the modules are. In this regard, minimal coupling is a very desirable goal in software design. The notions of coupling and cohesion are related. “Low coupling” generally correlates well with high cohesion and vice versa. Low coupling is regarded as an indicator of good design. If high cohesion among modules also exists, the desirable goals of high readability and maintainability of any individual system software component as well as of the overall system will be achieved. *See also:* **Algorithm (/content/algorithm/022150)**; **Computer programming (/content/computer-programming/195000)**; **Data structure (/content/data-structure/757547)**; **Modeling languages (/content/modeling-languages/801930)**; **Programming languages (/content/programming-languages/547550)**

Coding

The coding phase of the software development life cycle is concerned with implementing the software components that will satisfy the system requirements as stated in the SRS document using one or more suitable programming languages. The coding of the system may involve the use of either a high- or a low-level language. High-level languages that may be used for writing appropriate code can be of different types, such as functional, declarative, imperative, or object-oriented. Low-level languages that can be used are the machine or assembly languages. For pragmatic reasons, most systems are generally developed using a combination of high-level languages; however, whenever fast performance or minimum size of executable code (footprint) is required, the use of a low-level language or a high-level language that allows direct interaction with the hardware is desirable. As a result of this programming process, the deliverable for this phase is a working version of the envisioned system. The coding phase considers the essential issues of quality, performance, and debugging. *See also:* **Language theory (/content/language-theory/801220)**; **Object-oriented programming (/content/object-oriented-programming/757337)**

Testing

Although the testing phase can be viewed as an independent phase of the software development life cycle, it is highly integrated with the coding phase. This depends, in part, on the testing approaches used, as described later in this section, and the fact that programmers continually test their modules. The main objective of the testing phase is to examine the working version of the system to determine whether it meets the system requirements as specified in the SRS document. In other words, we need to find where the system fails to meet these specifications as the result of errors, bugs, or overlooking a requirement implementation. As indicated earlier, all functionalities of the system must be mapped to a particular set of requirements and vice versa. This is necessary not only for coding, but also for all life-cycle phases and their deliverables. The software testing process is often divided into subphases. The first subphase is unit testing of the software developed by a single programmer. The second part is integration testing, where all units are combined and tested as a single group and where the test cases are developed directly from the SRS document. System testing can be done in either a top-down or a bottom-up fashion. In top-down testing, high-level routines are implemented and tested first and then used as a testing environment for the lower-level routines. This is called the test harness. Testing the system in a bottom-up fashion proceeds by first developing low-level routines and testing them, then progressively combining these routines and testing them as parts of larger and larger program units. In practice, when both methods are used concurrently, the process is called sandwich testing. Acceptance of the system is finally done by its intended users. When the new system is intended to replace an existing one, both systems are run in parallel until the user is satisfied with the new system's performance. The final acceptance of a system is generally preceded by a walk-through and inspection testing, in which users and developers drill the system rigorously to examine how it functions. *See also:* **Software testing (/content/software-testing/757613)**

Maintenance

Software systems are dynamic; that is, they frequently change during and after deployment because of factors such as adding new functionalities, transporting the system to new environments, or just fixing errors. Maintenance, by its very nature, is done after deployment, and, for some systems, may cost more than all the other software life-cycle phases combined. Therefore, maintenance is a primary concern of software development organizations. 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 testing. Studies have shown that about half of all maintenance involves 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 all maintenance effort. Corrective maintenance is the process of fixing errors in a system after its release and accounts for about 20% of all maintenance effort. Because software systems change frequently, it is necessary to manage and control these changes through a well-structured software configuration management process. This activity consists primarily of tracking versions of life-cycle objects and monitoring the changes and the relationships among them. Typical configuration management activities also include handling and processing change requests and keeping records of these activities. *See also:* **Operating system (/content/operating-system/470405)**

Software engineering tools

Because of the inherent complexity of the development process, software engineers have introduced various tools to facilitate and monitor this critical phase of the software life cycle. In addition to the programming languages, other tools such the Unified Modeling Language (UML) and the Model-Driven Architecture (MDA) are used. UML provides a standard

way of visualizing the design of the system independently of the programming language used for its implementation. The MDA framework enhances the capabilities of the UML by providing model-to-model transformations, and thus is able to maintain platform-independent models of the system.

Software development life cycles

The phases of the software engineering process are generic phases of any software development life cycle. But because of the differences in the types of systems created and their implementation, more focused life-cycle models have been created for various systems under development. A few of these models will be discussed.

Waterfall model

The waterfall model is shown in **Fig. 1**. In this model, the requirements are finalized early in the cycle, allowing for fewer miscommunications and completion of the project in a timely manner. On the other hand, this characteristic can also be seen as a disadvantage because it is difficult to introduce new requirements at later phases of the development process. By its nature, the model does not lend itself well to progressive enhancement and incremental planning.

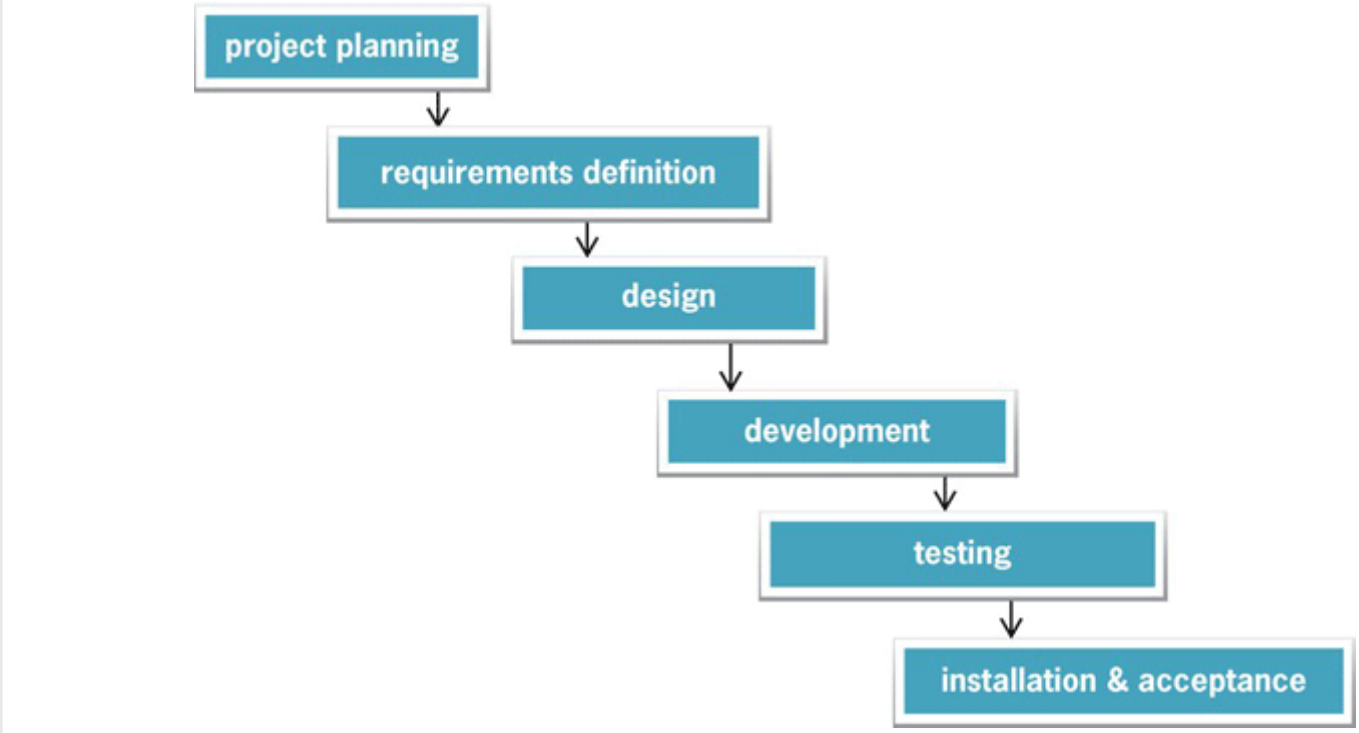

Fig. 1 Waterfall model.

Prototype model

In this model, the developers create a prototype of the application based on a limited version of the user requirements (**Fig. 2**). It is basically a hollow shell showing some of the basic features and functionality of the system. A critical drawback of this model is that, from the users' perspective, the prototype may be seen as the final product and some of the original requirements may now seem not to be needed. However, after using the prototype for a while, some of the requirements that were not considered necessary may become desirable or the user may have new requirements that were not initially considered. Because of this problem, the prototype may sometimes have to be redesigned to add new functionality, and this, in turn, increases the development cost.

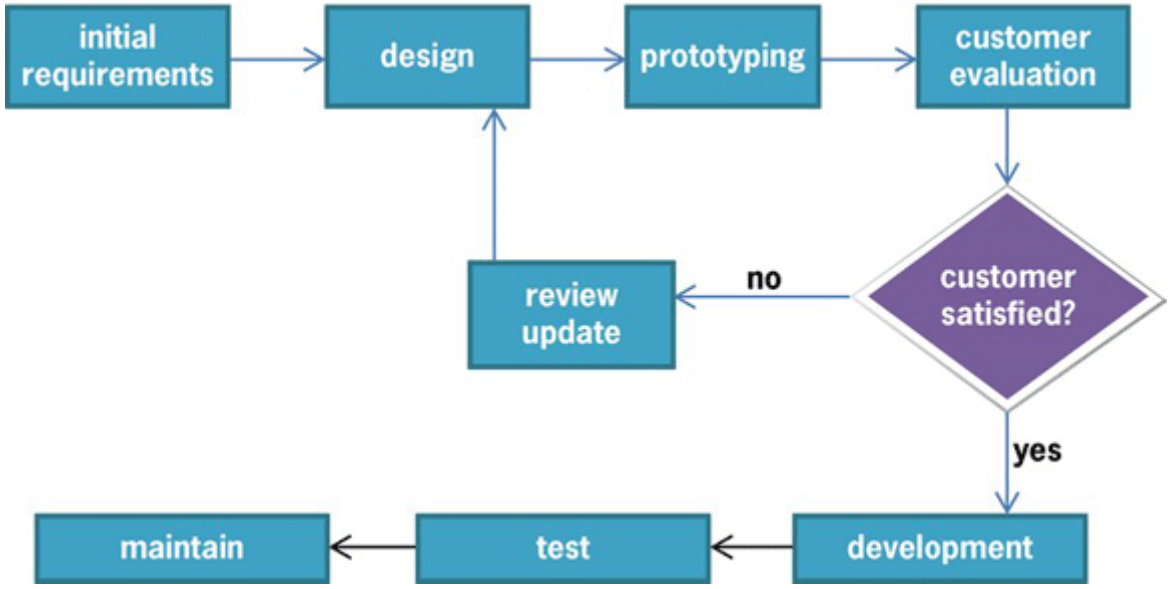

Fig. 2 Prototype model.

Agile

Agile software development is a group of software development methods in which requirements and solutions evolve through collaboration between the developers and the customers. This model provides the advantage of continuous improvement by implementing the changes in small increments over a short period of time. Thus, the agile method encourages rapid and flexible response to changes and a timely delivery. Some of the agile principles include customer satisfaction by rapid delivery of useful software, the flexibility to change requirements late in development, and close collaboration between users and developers; the working software in this model can be considered the principal measure of progress and adaptation to changing circumstances. *See also:* **Agile methods in software engineering (/content/agile-methods-in-software-engineering/YB150638)**

Scrum

Scrum is an iterative and incremental agile software development framework for managing product development. The new or changed requirements cannot be easily addressed in a traditional predictive or planned manner. This model takes into account that customers can change their minds and allows them to implement product changes using a developmental increment called a “sprint” that typically takes about three weeks to produce. The customer and the developer team agree on a set of features from the product backlog (sprint backlog) that still need to be implemented during a sprint.

Pranshu Gupta

Bibliography

S. L. Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*, 4th ed., Prentice Hall, Upper Saddle River, NJ, 2009

R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 8th ed., McGraw-Hill, New York, 2014

I. Sommerville, *Software Engineering*, 9th ed., Addison-Wesley, Boston, 2010

Additional Readings

Software Engineering Phases (http://infolab.stanford.edu/~burback/watersluice/node3.html)

Strategies for determining a design view: A preamble to DBMS Modeling [PDF] (http://ccscjournal.willmitchell.info/Vol16-00/east00/Ramon%20Mata-Toledo.pdf)