Module 1: Introduction to Software Engineering

Topics

---

1. Emergence of Software Engineering

In the software industry, we have seen the complexity of computer-based systems increase dramatically over the past decades along with advances in technology. This new technology has increased the demand for computer-based systems to control many infrastructures with software. As a result, designing and building cost-effective, reliable, and high-quality software has become the focus of software engineering in the computer industry.

In the past the processes used for designing and developing software were very informal, which contributed to the rise in development and maintenance costs. The results of ad hoc development processes contributed to a higher percentage of unreliable and lesser quality products entering the marketplace. Many accidents resulted from failures in computer-based systems with hardware devices that were controlled with software. At the time, the industry was considered to be in a crisis state, which then led to the emergence of new practices and methods in software engineering.

Technological advances have had a big impact on the complexity level required in software systems. The emergence of new communication protocols, hardware devices, and graphical user interface components have placed a greater demand on software engineers to design quality, reliable, and safe software.

A Brief History of Software Engineering

In the 1950s and the early 1960s, the various engineering disciplines were beginning to analyze how aspects of the engineering field could be applied to methods used in developing software products. As computing power evolved over the decades, the demand increased along with the complexity of the problems that needed to be addressed in the design of software. The term **software engineering** was introduced in 1968 at the first international software engineering conference, held by the North Atlantic Treaty Organization (NATO) Science Committee (Mahoney 2004). Many practitioners believe this is the milestone that marked the emergence of the software engineering discipline.

Software was developed to control critical hardware devices in the mid- to late-1960s and early 1970s. During this time, cases emerged that involved operational errors and accidents resulting in the loss of human lives and damage to property. Defects in software were uncovered, which heightened public awareness to the need for better quality and reliability of software. The escalating cost of building quality and reliable software was on the rise in the computer industry and the demand for skilled programmers could not be met. The state of software development was viewed by practitioners as being in a "crisis" state and was commonly referred to as the software crisis.

In response to the software crisis, researchers and practitioners have been trying to develop a set of methodologies, processes, and tools as the "silver bullet" for building software. The combination of these methodologies, processes, and tools is what comprises the field of software engineering, which continues to emerge today.

The software crisis can be summed up in economic terminology: there is a greater demand for software than there is a supply. The software in question is custom software written to solve large, unique data-processing problems. Usually, the small ubiquitous problems can be addressed by some combination of commercially available software products, such as word processing programs, spreadsheets, and so on.

But these types of problems are not the focus of this course. Our focus instead is on large, unique problems that require custom software solutions.

Software Engineering Defined

To fully understand the definition of software engineering, we need to define what is meant by the term **software**. Software is much more than just the code of a computer program. Software is accompanied by a set of documentation that is necessary to describe the details of the requirements, design, and any external aspects of the software that are necessary for it to execute successfully. These documents include any configuration files and any aspects of the external environment that surround the building and use of the software. We can refer to these items as a collective whole, the **software system** or the **software configuration**.

Software has unique characteristics that differ from hardware, which is manufactured and tangible. Software is abstract and intangible and is not manufactured or governed by physical laws. Creative programming solutions are required by humans because software is developed from a conceptual idea. There are four characteristics that best describe software: its maintainability, dependability, efficiency, and usability. These characteristics are inherent of the design process that was used to build the software. The uniqueness of programming solutions makes these characteristics difficult to measure.

In order to build software, we use processes to address a unique problem to be solved or a particular need to be satisfied. Software engineering focuses on improving these processes to build a better quality and more reliable product. In early times these processes were often ad hoc and chaotic and resulted in poor quality and unreliable software that was expensive to maintain after deployment. The rising cost of maintaining software was very expensive and unacceptable by software companies. The ultimate goal of good software engineering methodologies is to reduce these maintenance costs by building a better product earlier in the development phases of the life cycle.

To define software engineering, we can look at the definition provided by the Institute of Electrical and Electronics Engineers (IEEE, 1993)

1. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is the application of engineering to software.

2. The study of approaches as in 1.

The IEEE refers to a "systematic, disciplined, and quantifiable" approach to developing software that can be measured for effectiveness to allow for continuous improvement. As we mentioned earlier, software is intangible and developed from a conceptual idea. However, not all software teams may be receptive and suitable to follow a strict disciplined process. Pressman (2005) stresses that a process needs to be designed to foster creativity and to be adaptable and agile to change.

It should be noted that the IEEE definition does not provide the technical "how tos" to build software. In order to build quality software, companies need to be committed to developing sound processes, which requires the establishment of a set of methods and tools. When we develop these processes we also need to consider support for the business needs of the organization or company.

If you search the Internet you can find a variety of automated tools that have been developed to provide support for software processes and methods. These tools are referred to as computer-aided software engineering (CASE) tools. CASE tools have been designed to ease software development and to allow for integration of information from one process to another. It is important to understand that complex processes rely on individuals' judgment and creativity, so not every process can be automated by using a CASE tool. We will elaborate more on CASE tools in section 3.

Social and Ethical Responsibility

Software engineering is a respected profession and engineers are expected to use good judgment when developing software products. Software engineering is no different than any other engineering discipline. Software engineers are bound by the constraints of the legal and social framework when developing software (Sommerville, 2004).

Software engineers possess technical skills and they need to use them in an honest and moral way when developing software products. Applications to solve a wide range of problems are created and used in society, and society expects the software to be correct and of high quality. Companies entrust engineers with their assets and private data. It is the responsibility of the software engineer to maintain professionalism in the areas of confidentiality, competence, intellectual property rights, and computer misuse (Sommerville, 2004).

Professional software engineers are encouraged to join organizations in which ethical standards are established. The Association for Computing Machinery (ACM) and IEEE are known for establishing rules of professional conduct and ethics.

2. Introduction to Systems Engineering

In this section we will define system engineering and list some of its distinct characteristics. To help you understand the development of the software component within the system framework, we will introduce the basic system life cycle model. Because software requirements are derived from system requirements, we will introduce the System Specifications document.

Definition of System Engineering

The basic definition of system engineering (Blanchard, 1998, p. 12) is:

> … the effective application of scientific and engineering efforts to transform an operational need into a defined system configuration through a top-down iterative process of requirements analysis, functional analysis and allocation, synthesis, design optimization, test and evaluation, and validation.

This definition highlights the various phases in the life cycle of a system, which we will discuss further in the next section.

The Department of Defense (1996) defines system engineering as the … process that shall:

- Transform operational needs and requirements into an integrated system design solution through concurrent considerations of all life-cycle needs (i.e., development, manufacturing, test, and evaluation, verification, deployment, operations, support, training, and disposal).
- Ensure the compatibility, interoperability, and integration of all functional and physical interfaces and ensure that system definitions and design reflect the requirements for all systems elements (i.e., hardware, software, facilities, people, data).
- Characterize and manage technical risks.

The Department of Defense's definition is similar to the one provided by Blanchard in mentioning the life cycle, but it also stresses risk-management activities as a separate process.

System Engineering Characteristics

System engineering has distinct characteristics (Stevens, n.d.) that make its life cycle unique from software engineering:

- handles communication among disparate disciplines, each with its own technical language
- is primarily a project management activity
- deals with multiple semi-independent subsystems
- deals with engineering requirements, constraints for hardware
- is difficult to change
- has mechanics and architecture that make progress easier to measure

- is unlikely to require taking as many risks
- has a concrete working environment dependent on hardware
- has components that are manufactured or replicated and that can be damaged
- has upgrades requiring physical contact to implement because systems contain tangible components
- has development costs that are easier to determine
- has the primary maintenance goal of returning to its original state by repairing units

If we think of system engineering from the perspective of hardware, these characteristics can be easily understood. After all, hardware is tangible, and it can be manufactured.

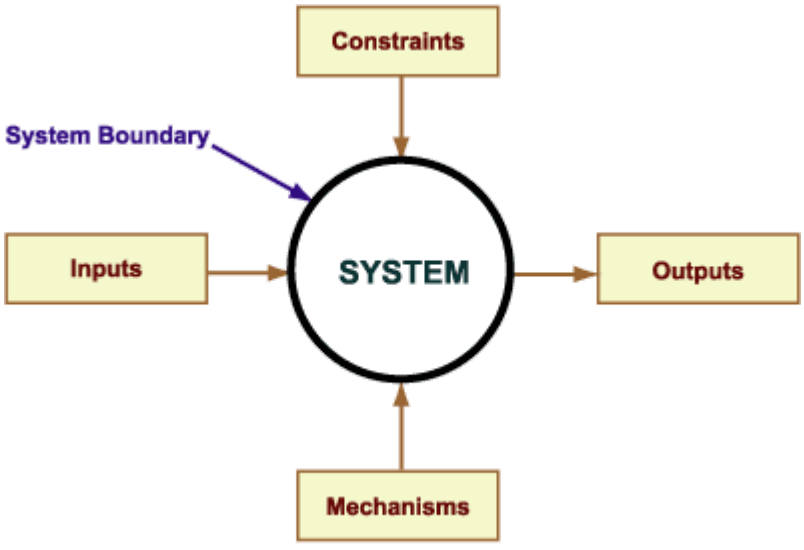
System Boundaries and Environment

Before we discuss the life cycle of a system, we need to define what is meant by a system that involves one or more computers and software. The term *system* is used loosely in society to define systems that emerge from organizations and institutions, systems that consist of hardware components, and systems that consist of hardware and software components. Local and federal governments are often referred to as a system. We also refer to various institutions, the educational system, and the financial or banking system. Hardware systems involve mechanical parts that work together to perform a specific task; power tools are good examples. Systems that contain both hardware and software components are best described as integrated subsystems that work together to accomplish an objective. Sommerville (2004, p. 21) defines a system as "a purposeful collection of interrelated components that work together to achieve some objective." The focus of our systems engineering discussion will be on systems that contain hardware and software components, or subsystems, which are interrelated. An example is a self-service checkout system that is used in stores by customers to purchase items.

A system operates within a set of **boundaries** and **limitations** unique to its overall **environment**. These are not considered part of the overall system itself, but the mechanisms and constraints can affect the system state at any given time. Anything that crosses the system boundary is defined as the system's distinct set of **inputs** and **outputs**. The following diagram demonstrates these relationships.

Click on each item in figure 1.1 to read its description and see some examples.

Figure 1.1
The System Environment



As you review the contents of this diagram, remember that a system has a designated purpose and it responds to an identified need. Further analysis of any system will result in a unique set of inputs, outputs, and external constraints that are imposed on the system, as well as the required mechanisms

to achieve the desired results. This **system framework** contains all the necessary products and processes to fulfill the intended need.

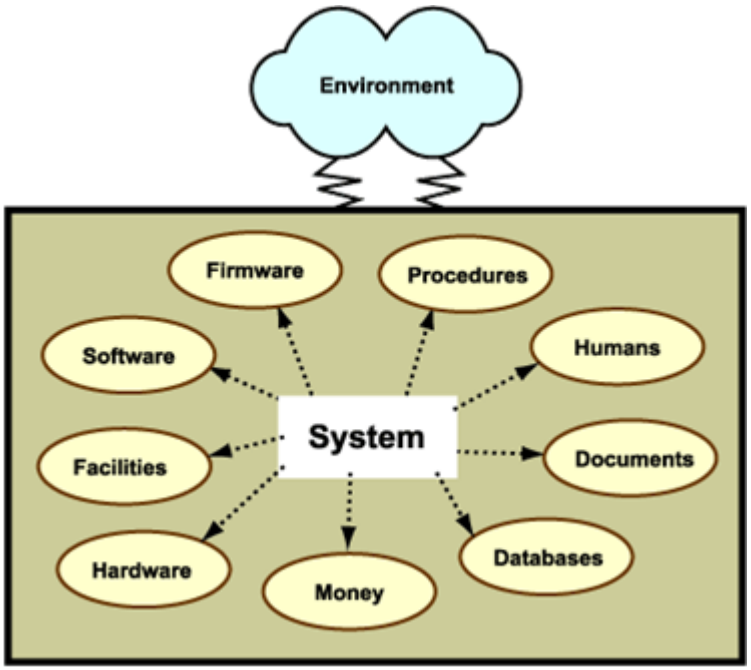
Systems are functionally dependent on their physical and organizational environment. It may also be the intent of the system to change its environment.

System Components or Subsystems

Functionality within a system is performed at the **component** level. A component can also be referred to as a **subsystem**. Components require the involvement of engineers from different disciplines working together to perform all the necessary functions required within the entire system. Interdisciplinary involvement consists of engineers from the software, electronics, mechanical, structural, civil, human factors, environmental, and architectural disciplines. Domain specialists and representatives from the user community should work with the engineers. The complexity level of the system, required to satisfy the functionality, determines both the number of subsystems and the interrelationships of the disciplines necessary to satisfy the system requirements.

The diagram below shows the various components of a system.

Figure 1.2
Components of a System

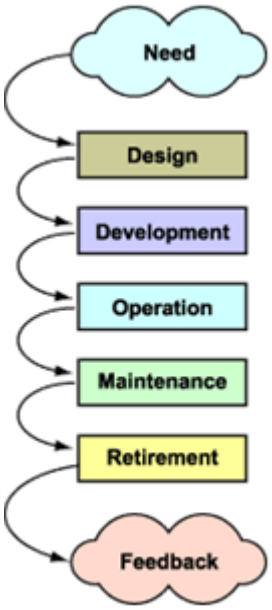


Relationships between subsystems are defined in their **interfaces**. These interfaces need to be defined at the system level.

System Life Cycle

Now that we have defined a system as integrated hardware and software components working together in its environment to fulfill an objective, we will discuss its life cycle. The basic life cycle of a system begins with identification of a need to be fulfilled through the system's development, implementation, and retirement phases. The various stages of the system life cycle are identified in figure 1.3.

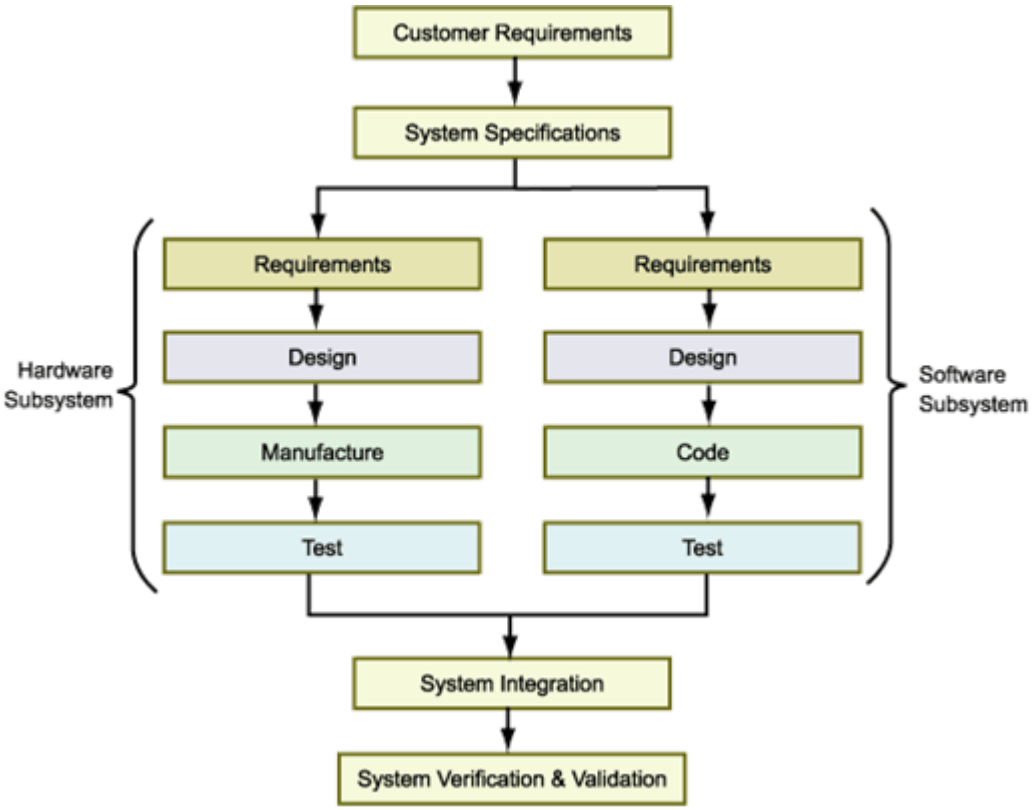
Figure 1.3
System Life Cycle



In order to develop software, it is necessary to understand how the software interacts with the hardware to fulfill the need or objective of the system. The role and boundaries of the software subsystem need to be well-defined within the system framework.

To satisfy the need for the system, a solution is established and presented to the customer in a document called the *system specifications* (SS). The SS describes what is needed by the system, its subsystems, and the interfaces that need to exist between the subsystems and other systems. When the SS is complete it is submitted to the customer for approval. Once the SS is approved by the customer, the life cycle for each subsystem can independently commence. In figure 1.4, we illustrate the parallel development cycle of hardware and software subsystems.

Figure 1.4
Parallel Development of Hardware and Software Subsystems



System Specifications

In figure 1.4, we can see that the SS outlines the "top-level" requirements for the overall system. Defining the requirements for the software subsystem can commence after the system requirements are agreed upon by the stakeholders and the engineers. Subsequent development phases of software and hardware can continue in parallel and then be integrated for testing on a scheduled date as a whole working system. Integration of software and hardware can take months, even years, depending on the size and complexity of the system.

Development of a project begins at the system level by stating its desired function. The functionality of the system is relayed to designers as a set of requirements that address the question *"What is this system supposed to do?"* At this point in the development life cycle, the system is viewed as a whole working unit encompassing the software and the hardware.

System engineers are responsible for defining the requirements and the interfaces for the whole system and designing the components of the system operations. The System Specifications document is prepared by the engineers, and it defines the architecture, the functional and non-functional requirements, and defines the necessary interfaces between all the components. We will concentrate on the required interfaces between the software and hardware components in the architecture for a self-service checkout system.

We view a system as a conceptual model. Consistency and communication are the goals of the specifications phase in the system life cycle. General guidelines for the SS include:

- stating the goals of the system
- defining the system boundary, which includes any constraints imposed on the system
- specifying the inputs and outputs
- identifying the various components
- defining the structure
- specifying any interactions between the components
- identifying any safety concerns

Upon its completion, the SS document is submitted, reviewed, revised, and approved by all of the stakeholders and serves as the basis for the top-level design that defines the system architecture.

Architectural Diagram

The basic process for gathering system requirements consists of effective communications between the customer and the technical community and a thorough understanding of the environment in which the system is to operate. To illustrate the system and its environment, a top-level architectural diagram is used. Pressman (2005) describes the architectural context diagram (ACD) as a block diagram that represents the flow of information into and out of the system. Refer to the following diagram to view a graphical representation of a self-service checkout system as seen in Wal-Mart and major grocery store chains.

Figure 1.5
Architectural Context Diagram of Self-Service Checkout System

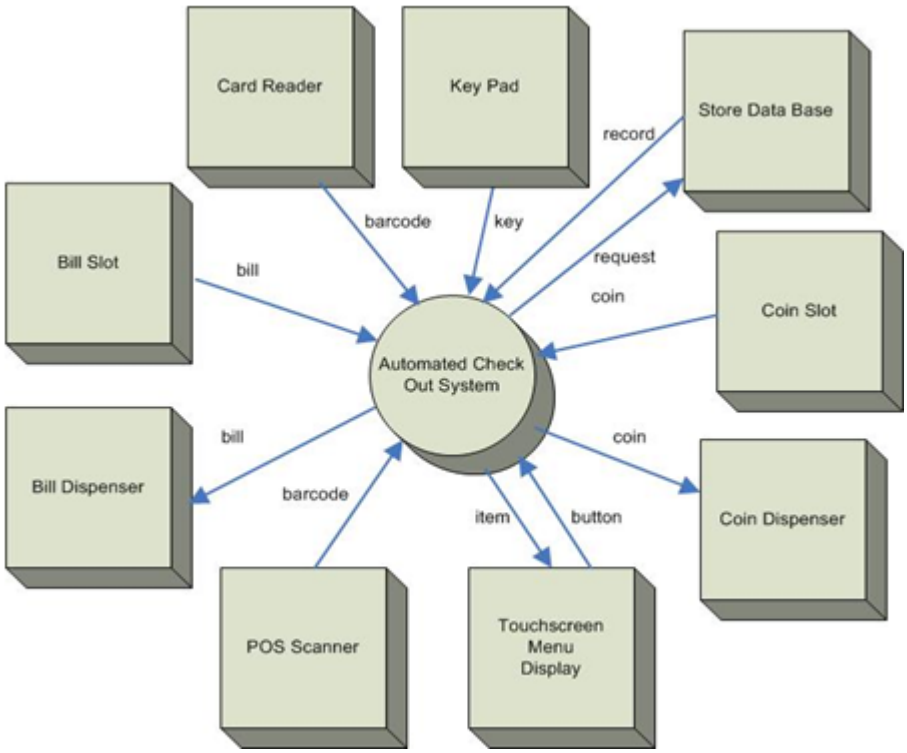


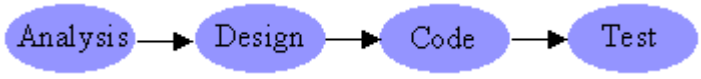
Figure 1.5 represents all the external entities (shown as rectangles) or terminators and the necessary relationships to produce a working checkout system. The proposed software subsystem is placed in the middle of the diagram (shown as a circle) surrounded by the external entities. The software subsystem is treated as a black box, without regard for the internal workings. The connecting lines with directional arrows reflect the expected flow of data inputs and outputs between the external entities and the software component. You should note that each directional line has a label describing the data flow or object. Proper labeling of all inputs and outputs at the system specifications stage will form the foundation for subsequent definition of the operational scenarios and data objects during software requirements development. We will discuss operational scenarios later in this module and data objects in modules 2 and 3.

IEEE Guide for Developing System Requirements Specifications

The IEEE has provided some guidelines for writing a System Specifications document in the IEEE Guide for Developing System Requirements Specifications, IEEE Standard 1223a-1998. Later in this module we will discuss diagrams that can be used to graphically represent information that needs to be included in the SS. The use of diagrams is recommended in technical documents and all diagrams should be supplemented with a textual description.

3. Software Process and Product

In this section we will study the discipline of software engineering by reviewing the development process and the software product. There are many software products, with code being just one of them. Process and product are really two sides of the same coin. The goal of the process is the product, and there's no product without the process. To accommodate the differing roles of software products, we will study several process models that vary the sequence and frequency of the basic tasks of software development. The basic tasks, however, are invariant and are illustrated below in one of the predominant software development processes, linear-sequential:

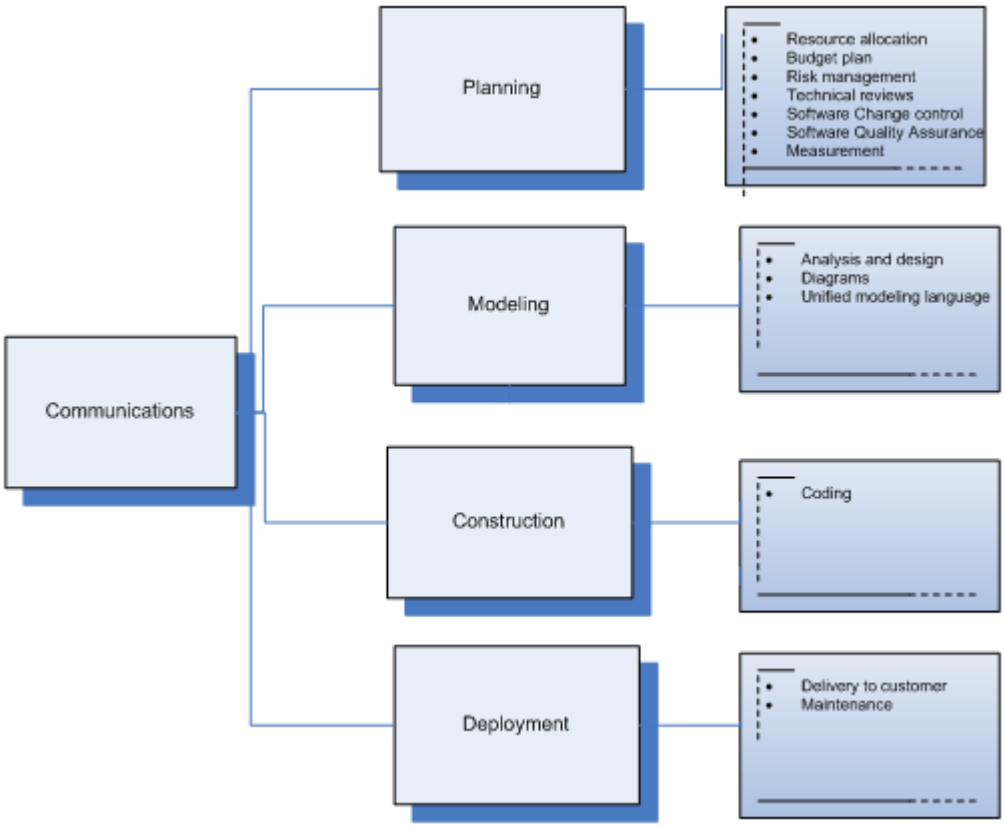


Roger Pressman defines a software process as "the approach that is taken as software is engineered" (Pressman, 2005, p. 21). This approach includes all the technical methods and automated tools that

create the process used to make a complex software product. You should keep in mind that a well-designed process is customized to be applicable to the developer's work culture and to stimulate creativity and support business decision making. One process does not fit all projects.

As you can imagine, many activities are involved in engineering software products. These activities include more than just the development cycle of software, analysis, design, code, and test. To convey these activities, Pressman outlines a framework to form a foundation for a complete software process for any project (Pressman, 2005, pp. 22-23). This common process framework provides support for umbrella activities and is outlined below in figure 1.6; it is applicable to all types and sizes of projects.

Figure 1.6
Process Framework with Umbrella Activities



You should note that the most important element of this framework is communications. It is essential for effective communication to be present in all activities used in the phases of the life cycle of software.

Process Improvement

Organizations that develop software products need to improve their development process for several reasons. The demand for software today requires high-quality systems to be delivered to the customer at a low cost and within a specified time frame. The organization is faced with challenges to meet this demand and to produce software products within tight constraints. A greater challenge exists for the development of the safety-related system because software and project risk factors can impede the completion of the final product. There is no guarantee that a software system can be 100 percent safe and reliable, but studies have shown that implementing a good technical and management process in the development life cycle can help reduce the chance of software and project failures.

At this point in our discussion, let's define a process and its maturity in the context of the development of a software product. A process is best described as the means used to bring together the necessary knowledge that is integrated into the final product (Pressman, 2005). This definition includes the means of communication and the software tools used to aid in this knowledge integration. Maturity of a process describes the effectiveness of the means that are being used.

Process Improvement Models

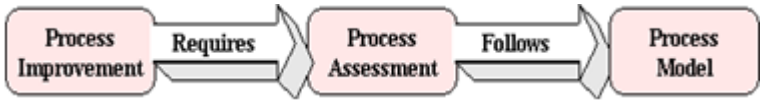
Organizations can improve their technical and management process and lower their risk factors by following the guidelines of the Capability Maturity Model Integrated (CMMI) process improvement model. We will discuss how this model can help organizations achieve a higher level of process maturity.

A process improvement model offers a framework for assessing the current state of an organization's way of developing software. This blueprint of effective activities is used to evaluate the current state of the process and to offer a baseline for improvement to reach a higher level of maturity.

The underlying premise of process improvement is based on Total Quality Management principles that were taught by Shewhart, Juran, Deming, and Humphrey (1989). These principles stress the importance of the process that is used to develop and maintain a product. The process used to develop the product will determine its overall quality; if a high-quality process is used in software development, the product will also be of high quality.

Figure 1.7 illustrates the relationship between process improvement and a process model.

Figure 1.7
Achieving Process Improvement



Phillips (2003) provides several reasons why process models should be used by an organization or company that develops software:

- to set process improvement objectives and priorities
- to improve the current process
- to provide guidance to achieve a mature process

A process model can only offer guidance on what to do to improve maturity of the process. No specific direction is given on how to actually do it (Phillips, 2003). Each organization must customize its activities according to its business objectives and the type of product it produces. It should be noted that process models vary and not all guidelines may be applicable to a particular project or application.

To help select a process model, Pressman (2005) recommends making the process model decision based on several factors:

- the nature of the project and application
- the methods and tools to be used
- the required controls and deliverables

Humphrey (1989) further defines a software process as all the tools, methods, and practices that are used in the creation of the software product. Some examples of activities that are used in a process during the development life cycle are:

- software quality assurance
- configuration management
- project management
- subcontract management
- requirements management

You will notice Humphrey has highlighted activities in the software process that are relative to managing the **software system or software configuration**.

A summary of benefits (Phillips, 2003) derived from studies on organizations that have followed process models are categorized as improvements in the following areas:

- predictability of schedules and budgets
- development cycle time
- productivity levels

- quality (measured by the number of defects)
- customer satisfaction
- employee morale
- return on investment
- cost factor of quality

Process Assessment

To assess an organization's software process, all activities used in building the product must be evaluated against a set of standards. These standards provide guidance because the process used in defining the standards has been proven effective over time. Successful evaluation of process maturity is achieved by following a framework that can offer direction to the organization and its internal operations. Standards provide a gauge that can be used to determine whether the process is in compliance with the standard.

Process assessment is used by organizations and companies in different ways:

- to determine the capability of a development organization in the selection process
- to determine the organization's capability to keep in line with its business aims

To summarize our discussion on process improvement, companies can use process assessment as an evaluation tool or incorporate it into their internal operation. The selection of a development organization to build software is a very important task, especially for systems that contain safety-critical components. Companies can also adopt process assessment to determine their capability to stay in line with their business aims, to stay marketable.

Capability Maturity Model

In 1984, Carnegie Mellon University, in response to software needs in the Department of Defense (DOD), formed the Software Engineering Institute (SEI). The SEI developed the Software Process Maturity Model as part of its early work at the DOD. In 1986, the SEI initiated the Software Capability Evaluation (SCE) project as requested by the United States Air Force (Humphrey, 1992). After several more years of refinement, in 1991, the SEI developed the Capability Maturity Model (CMM) at Carnegie Mellon to denote a class of maturity models.

Over the years, the SEI has revised the CMM based on new knowledge acquired from studies conducted on private industry and government. This new knowledge was compiled based on process assessments, capability evaluations, and feedback submitted by the participants in the studies. The conceptual framework for the CMM was developed by Watts Humphrey, who acted as the advisor during the refinement process of the model (Paulk et al., 1993).

In December 2001, the Software Engineering Institute released the Capability Maturity Model Integrated (CMMI), which was an upgrade from the CMM. The CMMI is currently being supported by the SEI and followed by organizations and companies in the computer industry.

In module 5 we will discuss project management issues, continue our discussion on process improvement, and take a closer look at the key activities involved in CMMI.

Computer-Aided Software Engineering (CASE)

In this section we will introduce computer-aided software engineering tools (CASE) that were developed to support process activities. We will discuss the different types of CASE tools that are available to automate process activities by classifying them according to functionality. It should be noted that many tools exist in the marketplace and some of these packages contain a full set of tools to support multiple functions.

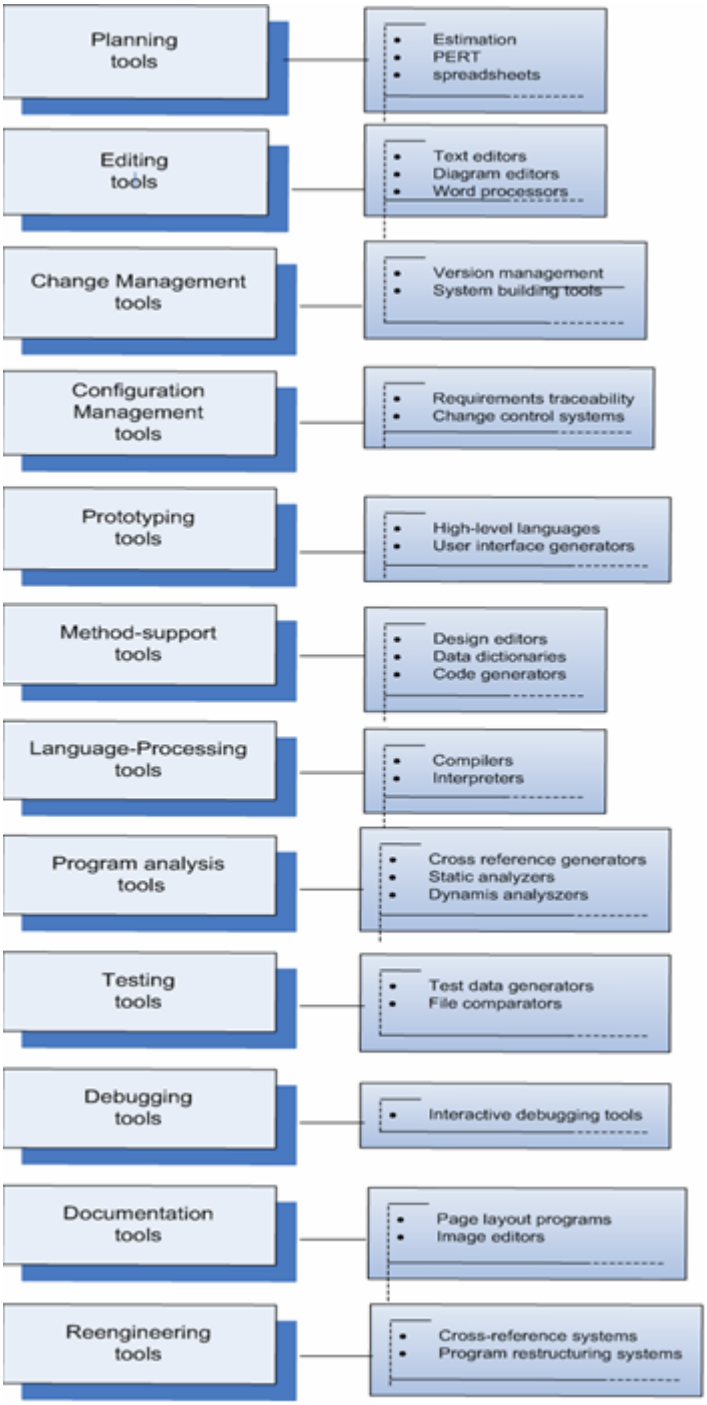
Types of CASE Tools

The software development process is information-intensive. The information generated takes different formats as software development progresses through the tasks of analysis, design, code, and test. Much of the information is textual, e.g., requirements, design descriptions, source code, test case descriptions, and so on. The remainder of the information is either graphical or binary, e.g., data flow diagrams (DFDs), entity-relationship (E-R) diagrams, state-transition diagrams (STDs), flowcharts, executable code, and so on.

For large systems, the amount of information that must be created, managed, controlled, and evaluated is far too great for developers to manage with paper and pencil. For this reason, the use of automated tools to assist in the development of software is crucial. Tools used for this purpose are called CASE (computer-aided software engineering) tools.

If you search the Internet you will discover that CASE tools are available for nearly every aspect of software development. To keep them straight, it is useful to view them in taxonomy. Sommerville (2004, p. 87) organizes tools by their specific function or utility. Read through these descriptions and identify their different roles in development. Note that these functions cover the range from documentation to support for testing. The sale of such tools is an active and important marketplace, essential for modern development projects. Care must be exercised to select those tools most suitable for the project at hand. An inappropriate tool can do more harm than good by diverting effort from development.

Figure 1.8
CASE Tools Classified by Functionality



Most CASE tools available today handle one or more different formats and support one or more development tasks, but fall short of supporting all tasks across the life cycle. For example, several commercial database front-end application programs assist in developing schema code. The software engineer defines the data schema (i.e., data model) either textually or graphically. The CASE tool will automatically generate the executable code to implement that schema. This CASE tool manages both textual and binary information formats, but only for the design and coding tasks, not for analysis or test.

One common perception is that the selection and use of a CASE tool will solve any problems associated with development. Unfortunately, nothing could be further from the truth. Although a tool can help to develop a product, and whereas not using tools can doom a project to failure (particularly if the system under development is large and complex), simply providing tools, by itself, is not sufficient to ensure success. The tools must be properly used in conjunction with a disciplined process.

In module 5, we will discuss CASE tools and how they can be used to support project management activities.

Integrated CASE Environments

The use of individual CASE tools can provide significant help to the software engineer, but the process of getting the tools to work together can sometimes be a difficult and frustrating task. If the tools work with different file formats, perform different functions, and execute on different machines, the engineer is left with having to expend extra effort just to coordinate the tools and their outputs. If, however, the tools can be integrated in some way so that they work together seamlessly, with compatible interfaces, then the engineer's job is made much easier and he or she can focus on the creative aspects of development. Specifically, the challenge is to integrate the various CASE tools together into a single capability, i.e., integrated case (I-CASE).

CASE tool interface protocols and input and output data formats must be standardized to permit individual CASE tools to communicate with each other. Integrating frameworks has evolved as an answer to the CASE tool communication problem, but the actual data formats and interface protocols have not yet been fully defined and standardized.

One key concept that is necessary to achieve I-CASE is that of the CASE repository. For individual CASE tools to share information during the software development process, each tool will require access to the outputs of the other CASE tools. One potentially effective approach to achieving this goal is to store the outputs of the CASE tools in a central location called a **repository**.

If, in addition to the collocation of CASE tool data, the CASE repository can be built with common protocols and data formats/conversions, then the CASE repository will serve as an integrating framework and be of even more assistance to the developers. Current industry publications that discuss "object repositories" and "object-bases" are really referring to the concept of CASE repository, in which information in different formats is stored centrally for access by any interested application program. For this discussion, the interested application program is the CASE tool.

4. Software Process Models

So far in our discussion we have introduced the concept of producing a high-quality software product by improving the activities used in the development process. In this section we will look closer at the various process models being used today in the computer industry. These process models have formed the basis for emerging process models, for example agile methodology, which we will discuss separately in section 5. As you read through this section, you should notice that all the process models have a variation of the basic tasks used in development: analysis, design, code, and test.

Life Cycle of Software

The full life cycle of a software system begins with its initial concept and ends with its retirement from service. Each step of this life cycle requires one or more activities to occur for the product to be completed successfully. To direct the processes within the development life cycle, the project manager selects a model that best describes the flow of the tasks needed within the project.

The IEEE (1996) generically describes a software life cycle model as a "framework that contains all the necessary processes, activities, and tasks to develop, to operate, and to maintain a software product" from the point of conception to termination of its use.

All life cycle models used in software development contain the following core components, as outlined by Christensen and Thayer (2001):

- a description of the major development phases
- a detailed description of the major processes and activities for each phase
- a specification of the products and inputs for each phase
- a mapping of the activities to a framework

Basic Life Cycle Models

There are different types of life cycle models that exist for the development of software. The basic ones are listed below. Move your mouse over the highlighted words to see a brief description.

- linear sequential
- prototyping
- evolutionary
- rapid application development
- component-based

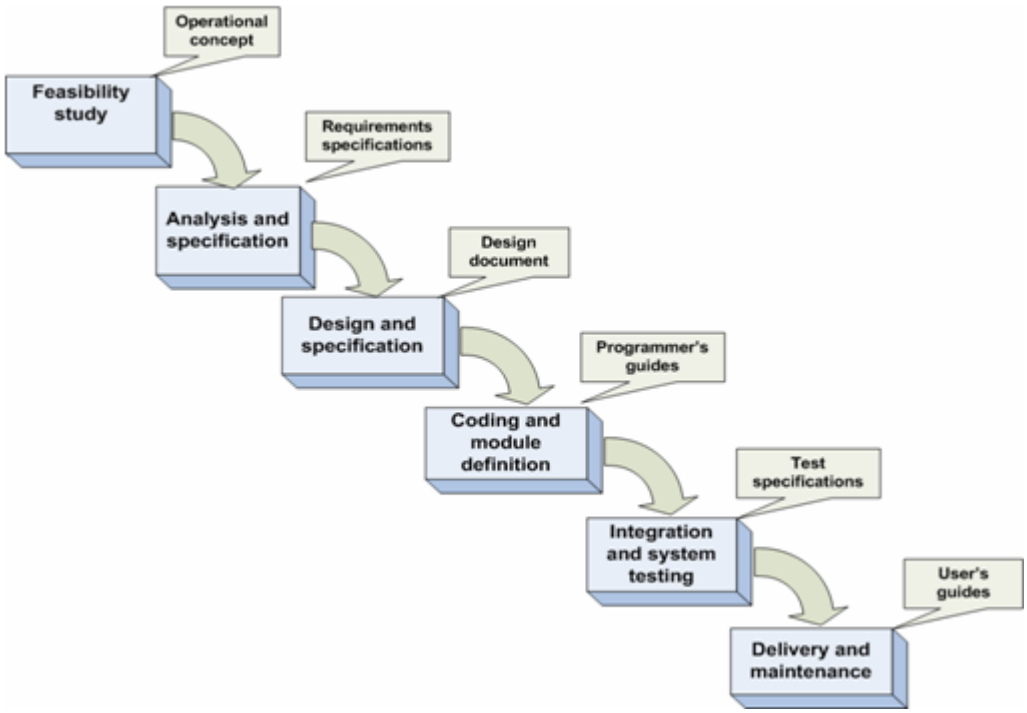
To complete this section we will expand our process model discussion and look at some specific models. You are encouraged to do research on the Internet to learn more about the various development models in software engineering.

Waterfall Model

The classic linear-sequential life-cycle model for software engineering is sometimes referred to as the *waterfall model*. The basis of the *waterfall model* originated in the late 1960s to address needs for a complex military software development (Baird, 2002). The waterfall model consists of a systematic sequential approach to software development that begins at the system level and progresses through several phases or steps. At each phase or step it is recommended that one or more technical documents be produced as products that are completed and "signed off" before continuing on to the next phase (Sommerville, 2004). In 1970, Royce introduced the model following an iterative approach to software development in which he stressed the importance of documentation at each step. In Royce's paper, he also argued that the waterfall model was flawed and needed strengthening by providing additional supporting development steps to make it less risky.

In figure 1.9, we can trace through the phases of the life cycle and view the major supporting technical documentation proposed by Royce. You should note there can be several documents produced at each phase. We have only mentioned the major documents to help guide you through the phases in the life cycle and to guide you through your course assignment. We will discuss the software requirements specification, design document, and test specifications throughout the subsequent modules in this course.

Figure 1.9
Waterfall Model and Major Technical Documents



The lack of feedback between each phase and in getting the approval for each document of the life cycle has proved to be costly and has required a significant amount of rework when changes were introduced during later phases. For these reasons, the waterfall model is best suited for projects in which requirements are stable and well-defined (Sommerville, 2004, p. 67). Introducing changes late in the life cycle is difficult due to the sequential nature and dependency of phases followed in the waterfall model.

The waterfall model is the oldest and most widely used paradigm in the industry and is known to have inherent deficiencies. To address the deficiencies in the waterfall model, hybrids have been emerging over the decades that address:

- iteration of phases
- difficulty in getting all the requirements up front
- lateness of working product

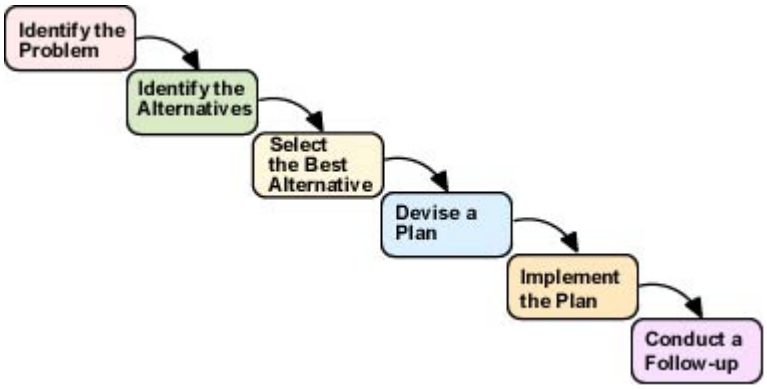
Boehm's Spiral Model

In this section we will discuss the evolutionary process model called *Boehm's Spiral Model.* The spiral model is highly recommended to guide the development processes necessary to produce high-quality, safety-critical software.

Dr. Barry Boehm was the founder of the spiral model that emerged in the software industry in 1988. A major feature of the spiral model is its iterative development and linear sequential nature. Each cycle of the development process is fully evolved and results in a new software release or build before the next cycle commences.

The basic philosophy of each cycle in the development process is based upon the principles of problem solving as outlined in Polya's (1957) approach and as summarized in figure 1.10.

Figure 1.10
Polya's Approach to Problem Solving



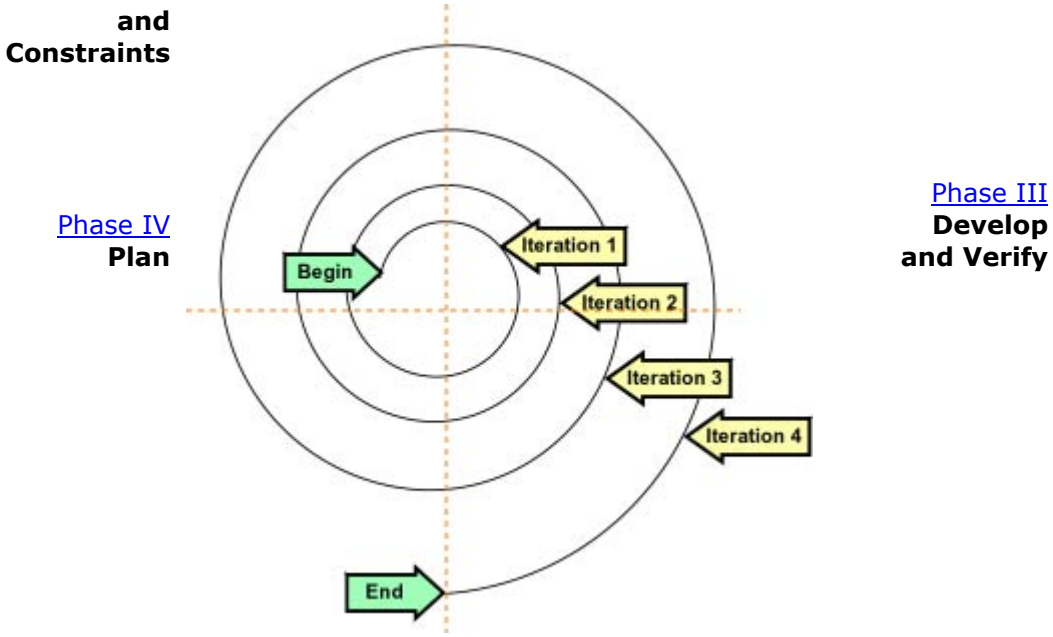
Boehm identifies four phases in the spiral model to encompass the basic principles of Polya's problem-solving approach. Each phase of the model is entered multiple times in a sequential fashion as the spiral continues to revolve. The activities within each iteration or cycle are completed and used as the foundation for the next iteration. Refer to figure 1.11 to see the basic layout of Boehm's spiral model (1988).

Click on each phase label to review its relevant activities.

Figure 1.11
Basic Layout of Boehm's Spiral Model

Phase I
Determine Objectives, Alternatives,

Phase II
Evaluate Alternatives



The spiral model is especially applicable for safety-critical software development because it provides a risk evaluation and analysis phase that is periodically visited throughout the development life cycle.

The spiral model can be used to perform periodic risk identification through the development and analysis of prototypes. The function-oriented nature of the spiral model supports parallel development of software modules by a team of programmers. The underlying concept of developing prototypes minimizes the overall risk factors by ensuring that the software requirements are stable.

The spiral model offers a workable framework to support the development of the system's hardware and the software components. Addressing risk periodically throughout their development cycles supports the elimination of unattractive alternatives early in the project, which will help keep the associated project costs down.

We have outlined the various phases of the spiral. The spiral model has been used extensively in the development of defense systems, but because of the high cost and time factors, it hasn't been used extensively in private industry. Therefore, the spiral model is being proposed only for extreme cases in which the customer is willing to pay the cost of developing a secure safety-critical system.

Rapid Application Development (RAD) Model

The Rapid Application Development approach to building software stresses incremental development and a short time to market. The concept of incremental software releases is used to deliver a product to the customer for evaluation and to plan for the next software release.

To gain a better understanding of a RAD approach, we can examine **evolutionary prototyping**, which delivers a system with functionality that can be used by the end-users. In order to build the initial prototype, parts of the system's requirements need to be clearly understood by the customer, developer, and the end-user. The prototype's functionality evolves with each release as the requirements for new features become fully defined. Prototypes can be used by the customer to discover new requirements for additional functionality or features. The main advantage of the evolutionary approach is to provide a means for the end-users to try out the system. The software engineers can also benefit because the prototype provides insight on how the end-user will use the system.

The evolutionary prototyping model stresses incremental software releases. A software release, sometimes referred to as a *build* in the industry, is identified with a version number. The version number is assigned sequentially to identify the difference in functionality from the previous release. Each version refines the requirements for additional functionality. It should be noted that the system following the evolutionary model is not complete when first released to the end-user until the requirements are finalized between the developer and customer.

Gradually building the system is suitable for systems in which specifications are difficult to fully define at the beginning of the life cycle. The evolutionary prototyping model stresses making changes quickly and demonstrating them to the customer and the end user in several revisions.

Rational Unified Process (RUP)

As you learn more about software engineering you will encounter many different processes and methodologies being used in the industry. In this section we briefly mention one methodology, the Rational Unified Process (RUP). RUP is a popular process that was developed to provide a disciplined approach to software development in large organizations. RUP is incorporated into IBM's Rational software (IBM Rational Unified Process), which uses the Unified Modeling Language extensively as a tool for supporting communications throughout the life cycle of software. RUP can be noted for its focus on risk reduction in projects by using an iterative development methodology (Baird, 2002). RUP portrays a flavor of the traditional waterfall model because support documentation plays an important role in its processes.

Code-Fix Approach

To complete our discussion of common approaches used in software development, we will mention the code-fix approach, sometimes referred to as the build-and-fix approach. The code-fix approach is not recognized as a software engineering methodology (Baird, 2002) but it does exist in the computer industry and you should be able to recognize it.

The code-fix approach commences from an ill-defined set of requirements received from the customer. The developer attempts to satisfy the requirements in a quick fashion by coding software that is presented to the customer for feedback. The developer proceeds in the development by providing fixes to the software and repeating this cycle until the customer is satisfied with the software product. Practitioners question the quality and scalability of the final product (Baird, 2002). Often "spaghetti" code results from the code-fix approach and the code is often not documented and is difficult to maintain or change.

5. Agile Approach to Software Development

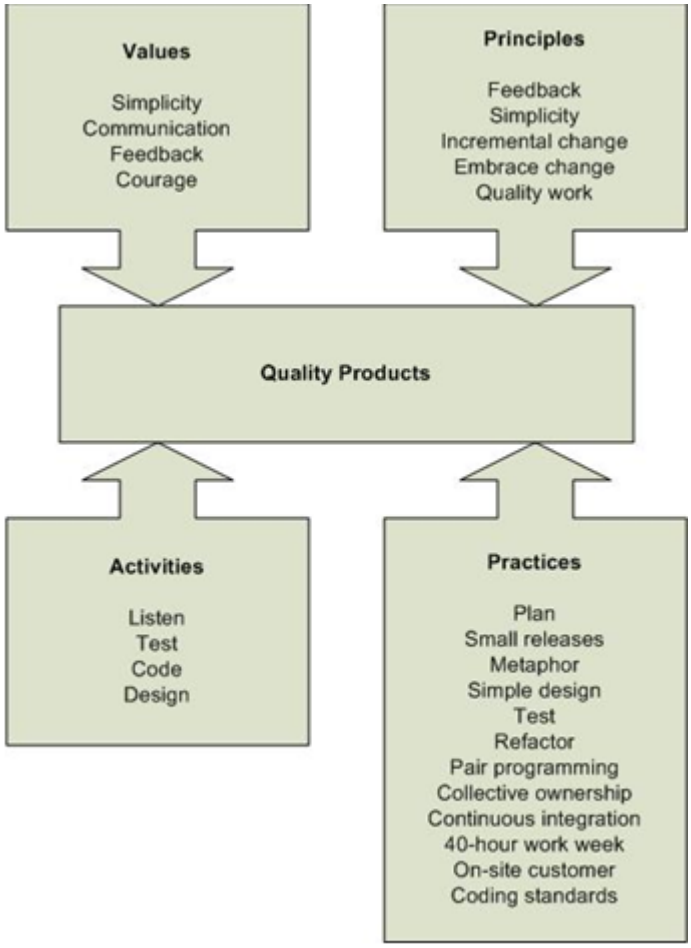
In this section we will discuss a new approach to software development. The agile approach was conceived in response to weaknesses discovered in earlier software development practices and was spurred by the Internet. The growing trend for web applications heavily influenced the need for agility in a development approach. The nature of web programming requires flexibility to adapt and respond to continuous and quick changes in requirements and design. In our discussion on the traditional waterfall model, we mentioned that stability in requirements was necessary and changes were hard to accommodate. Therefore, this inflexibility to allow change has led researchers to develop a more agile model to accommodate development of applications for the web.

The next section of this module will discuss a popular agile process model, Extreme Programming, which can be used for projects that require flexibility to adapt to a changing requirement environment.

Extreme Programming (XP)

Extreme Programming (XP) has been emerging since the late 1990s as a proposed solution for small- to medium-sized teams developing projects with continuously changing requirements. The basic philosophy of XP is built from a set of values, principles, practices, and activities (Baird, 2002) that are outlined in figure 1.12 to produce high-quality software products.

Figure 1.12
Extreme Programming Philosophy



In figure 1.12 we can see a set of values that is used to establish the overall tone throughout the XP development process. The focus of XP is to keep the design and structure as simple as possible to avoid adding unnecessary features to the software and to limit documentation. Because support documentation is kept at a minimum, it's necessary to establish two-way communication between all involved parties. It is preferred that the customer be located on-site so that the means of communication can be oral. Informal means of communications, such as daily stand-up meetings, are used to transfer information between parties. Written documents, reports, and plans are kept at a minimum to save time and effort. Quick and constructive feedback from the customer is encouraged, which builds confidence among the team members. The confidence level of the team fosters quick redevelopment when needed to keep activity moving forward.

The strength of the XP methodology is found in several principles that are used to direct developmental activities. We mentioned feedback and simplicity in XP values but now we will extend these to include short feedback loops and to limit the design to a current iteration. The way in which changes in requirements are handled is the most notable difference in XP as compared to traditional approaches to development. XP encourages small and frequent changes to address and resolve problems as they occur. It’s easy to understand how this philosophy plays an important role in developing applications for the web, because the Internet environment is constantly changing. The principles of XP contrast with the traditional philosophies but the focus is the same – to produce a quality product. XP focuses on quality of work by concentrating on the coding and testing activities.

We will now turn the focus of our discussion to how these values and principles are implemented in the actual activities of the life cycle. The basis for all practices in XP is verbal communications; practitioners therefore must master good listening skills. Another interesting concept in XP is on testing activities. Test cases are continually written as the requirements emerge before any code is actually developed. This school of thought is based on building quality into the code instead of catching errors at a later stage in development. Another prominent feature of XP is the pair programming concept. Two programmers are selected for compatibility and assigned to work on a code module to produce better quality code quickly with fewer errors.

We have finished our discussion on the core of XP methodology. You should now be able to apply the values, principles, and activities as outlined in figure 1.12 to the XP practices. We encourage you to conduct research on the Internet to read more about XP practices and its success stories in the computer industry.

6. Unified Modeling Language (UML)

In this section we will take a brief look at the history of the Unified Modeling Language (UML) and how it can be applied to the life cycle phases of software development. We introduce UML at this point in our discussion to assist you in preparing diagrams to represent your ideas in the system specifications document.

A. History of Unified Modeling Language

UML has emerged in the software industry and has steadily matured over the decades to become a standard. As a modeling language, UML represents best practices in analysis and design of large-scale software systems.

We can trace the roots of UML to the beginnings of object-oriented languages. With the onset of early object-oriented languages, Stimula-67 and Smalltalk, Objective-C, techniques for object-oriented analysis and design began to appear in software engineering. In the early 1990s, a variety of diagramming techniques began to appear in publications written by common authors Coad, Yourdon, Booch, Rumbaugh, Premerlini, Eddy, Lorensen, Jacobsen, Christerson, Jonsson, and Overgaard (Bennet et al, 2001). You may have purchased or seen books on object-oriented analysis and design in your favorite bookstore or library that were written by one or more of these authors. If you scan through a few of these books you will notice that each approach outlines a slightly different diagramming notation and method to represent classes, objects, and relationships. The difference in these notations and methods was very confusing to developers when sharing designs. To effectively communicate an object-oriented design, a visual language with an established set of rules was needed in the industry. UML attempts to provide this commonality of methods in the software industry and to become the standard notation language used among designers and developers.

B. Unified Modeling Language

The Unified Modeling Language provides a definition of a set of elements and a set of rules on how to define these elements to make programs (Bennet et al, 2001). The elements in UML are defined by using a set of symbols, lines, rectangles, ovals, and other shapes, which are grouped together to create a graphical representation of a conceptual idea. These visual representations increase understanding among all interested parties involved in the design model.

In the current version of UML, there are different types of diagrams that are defined and divided into three major categories: structure, behavior, and interaction.

As we proceed through the phases of the life cycle you will learn and practice with a few of these diagrams to present your ideas for your project. You will be developing a set of technical documents for a proposed system as a required assignment for this course. The diagrams we will present in the subsequent modules are summarized in table 1.1 and mapped to the documents where you can use them throughout the life cycle. Before you start to work on your first assignment, the SS document, make sure you review the notes and examples on Use Case and State diagrams. Click on the highlighted text in the table below.

Table 1.1
Maps the UML Diagrams to Technical Documents

UML Diagram	Document(s)	Representation
Operational or Use Case Diagram	System Specifications	Scenarios describing how the system will be used by actors of the system.

	Software Requirements Specifications	
Statechart or State Transition Diagram	System Specifications	States of system operation and the events to trigger these different states.
Activity Diagram	System Specifications Software Requirements Specifications	Activity flow of the various processes needed to accomplish the task.
Class Diagram	Software Requirements Specifications Software Design	Define classes and the necessary relationships between classes.
Object Diagram	Software Design	Class instances and relationships among these instances.
Component Diagram	Software Design	Code module breakdown of the processes.

A variety of drawing tools exist on the market that provide the symbol set necessary to draw UML diagrams. You can download a trial version of SmartDraw software from the Internet to create the required diagrams for your project assignment. You are encouraged to perform a search on the Internet for other available drawing tools.

Case Tools and UML

We mentioned earlier in this module that there are many CASE tools available on the market to automate the tedious tasks involved in software development. UML provides a specification written for members of the Object Management Group who develop CASE tools used in the industry to provide standardization of notation. You are encouraged to search the Internet and become familiar with the availability of CASE tools that support UML features.

References

Baird, Stewart. *Teach Yourself Extreme Programming in 24 Hours.* Indianapolis, Indiana: SAMS Publishing, 2002.

Bennett, S., Skelton, J., and Lunn, K. Schaum's Outlines, UML. Italy: L.E.G.O. Spa, Vincenza, 2001.

Blanchard, Benjamin S. *System Engineering Management*, second edition. John Wiley & Sons, 1998.

Boehm, Barry W. "A spiral model of software development and enhancement." *Computer,* 21(5), 61–72. 1988.

Carnegie Mellon Software Engineering Institute. (2003). *Upgrading from SW-CMM to CMMI.* Retrieved November 12, 2003, from http://www.sei.cmu.edu/cmmi/adoption/pdf/upgrading.pdf

Christensen, Mark J.; and Thayer, Richard H. *The project manager's guide to software engineering's best practices.* Los Alamitos, CA: IEEE Computer Society Press, 2001.

Department of Defense Regulation 5000.2-R. (March 15, 1996). "Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information Systems (MAIS) Acquisition Program," Part 4, Paragraph 4.3. U.S. Department of Defense.

Dorfman, M.; Thayer, R.H.; and foreword by Boehm, B. Software Engineering. IEEE Computer Society. 1997.

Extreme Programming. Retrieved on July 7, 2006 from http://www.extremeprogramming.org/.

Humphrey, W.S. *Managing the software process.* Reading, MA: Addison-Wesley, 1989.

Humphrey, W.S. *Introduction to software process improvement* (Technical Report CMU/SEI-92-TR-7). Pittsburgh, PA: Carnegie Mellon University, 1992.

IEEE. IEEE Standards Collection: Software Engineering, IEEE Standard 610.12-1990. 1993.

Institute of Electrical Engineers (IEEE)/EIA. *Industry implementation of International Standard ISO/IEC 12207:1995 standard for information technology—software life-cycle processes.* New York: IEEE, 1996.

Institute of Electrical Engineers (IEEE). *IEEE Guide for Developing System Requirements Specifications:* Software Engineering, IEEE Standard 1223a-1998. 1998.

Mahoney, Michael S. *Finding a History for Software Engineering*. IEEE Annals of the History of Computing, pp 8-19. IEEE Computer Society. 2004.

Object Management Group (OMG). Retrieved July 7, 2006 from http://www.omg.org/.

Paulk, M.C.; Weber, C.V.; Garcia, S.M.; Chrissis, M.B.; and Bush, M. *Key practices of the capability maturity model, version 1.1* (Technical Report CMU/SEI-93-TR-25). Pittsburgh, PA: Carnegie Mellon University, 1993.

Phillips, Mike. (2003). *CMMI version 1.1 tutorial.* Carnegie Mellon University. Retrieved November 12, 2003, from http://www.sei.cmu.edu/cmmi/presentations/euro-sepg-tutorial/.

Polya, G. *How to solve it* (second edition). Princeton, NJ: Princeton University Press, 1957.

Pressman, Roger S. *Software engineering: A practitioner's approach* (sixth edition). New York: McGraw-Hill, 2005.

Rational Unified Process. Retrieved July 7, 2006 from http://www-306.ibm.com/software/rational/) International Business Machines.

Royce, Winston W. "Managing the Development of Large Software Systems." Published in the Institute of Electrical Engineers (IEEE) WESCON Proceedings, pp. 1-9. August 1970.

Software Engineering Laboratory. Retrieved July 5, 2006, from http://sel.gsfc.nasa.gov/.

Sommerville, Ian. *Software Engineering* (seventh edition). England: Pearson Education Addison Wesley, 2004.

Stevens. Software Engineering Related to Systems Requirements, Information Systems Division ESRIN, European Space Agency, ESRIN.

Unified Modeling Language. Retrieved July 7, 2006, from http://www.uml.org/.

Return to top of page