UNIT-I: INTRODUCTION TO SOFTWARE ENGINEERING

Let us first understand what software engineering stands for. The term is made of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

IEEE defines software engineering as:

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

EVOLVING ROLE OF SOFTWARE:

software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product.

As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.

As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time—information.

Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems.

The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.And yet, the same questions asked of the lone programmer are being asked when modern computer-based systems are built:

1)Why does it take so long to get software finished?

2)Why are development costs so high?

3)Why can't we find all the errors before we give the software to customers?

4)Why do we continue to have difficulty in measuring progress as software is being developed?

CHANGING NATURE OF SOFTWARE/APPLICATIONS OF SOFTWARE

It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows, neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

System software. System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.

Real-time software. Software that monitors/analyzes/controls real-world events as they occur is called real time. Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

Business software. Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., pointof-sale transaction processing).

Engineering and scientific software. Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

Embedded software. Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

Personal computer software. The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

Web-based software. The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.

Artificial intelligence software. Artificial intelligence (AI) software makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledgebased systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category

SOFTWARE CHARACTERISTICS:

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. **Software is developed or engineered, it is not manufactured in the classical sense.**

   Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different (see Chapter 7). Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. **Software doesn't "wear out." Figure 1.1 depicts failure rate as a function of time for hardware.**

The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

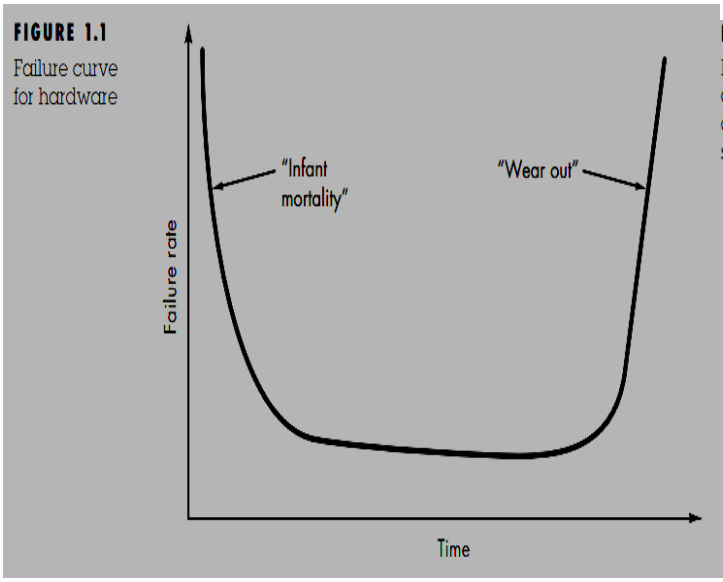


FIGURE 1.1 Failure curve for hardware

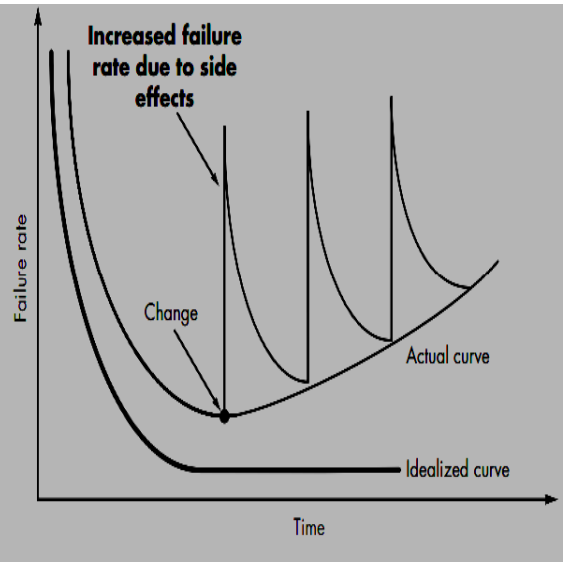


FIGURE 1.2 Idealized and actual failure curves for software

3. **Although the industry is moving toward component-based assembly, most software continues to be custom built.**

   Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an IC or a chip) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

SOFTWARE MYTHS

Today, most knowledgeable professionals recognize myths for what they are— misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed.

Management myths. Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: We already have a book that's full of standards and procedures for building software, won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

Myth: My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them effectively.

Myth: If we get behind schedule, we can add more programmers and catch up (sometimes called the Mongolian horde concept).

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks [BRO75]: "adding people to a late software project makes itlater." At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource3 the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software+e projects.

Customer myths. A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs— we can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. Figure 1.3 illustrates the impact of change. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost. When changes are requested during software design, the cost impact grows rapidly. Resources have been committed and a design framework has been established. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have

a severe impact on cost. Change, when requested after software is in production, can be over an order of magnitude more expensive than the same change requested earlier.

Practitioner's myths. Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it'll take you to get done." Industry data ([LIE80], [JON91], [PUT97]) indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review. Software reviews (described in Chapter 8) are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes many elements. Documentation provides a foundation for successful engineering and, more important, guidance for software support.

Myth: Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software professionals recognize the fallacy of the myths just described. Regrettably, habitual attitudes and methods foster poor management and technical practices, even when reality dictates a better approach. Recognition of software realities is the first step toward formulation of practical solutions for software engineering

THE SOFTWARE PROBLEM:

The software system needs to be of high quality with respect to properties like reliability, usability, portability, etc. This need for **high quality** and to satisfy the the end users has a major impact on the way software is developed and its cost. The rule of thumb Brooks gives suggests that the **industrial-strength software** may cost about 10 times the **student software**.The software industry is largely interested in developing industrial-strength software, and the area of software engineering focuses on how to build such systems. That is, the problem domain for software engineering is industrialstrength software. In the rest of the book, when we use the term software, we mean industrial-strength software. In the remainder of this chapter, we will learn – That quality, cost, and schedule are the main forces that drive a (industrialstrength) software project. – How cost and productivity are defined and measured for such a project, and how quality of software is characterized and measured. – That large scale and change are important attributes of the problem domain and solution approaches have to handle them.

Cost, Schedule, and Quality: Though the need for high quality distinguishes industrial strength software from others, cost and schedule are other major driving forces for such software. In the industrial-strength software domain, there are three basic forces at play—cost, schedule, and quality.

The software should be produced at reasonable **cost**, in a reasonable time, and should be of good quality. These three parameters often drive and define a software project. Industrial-strength software is very expensive primarily due to the fact that software development is extremely labor-intensive. To get an idea of the costs involved, let us consider the current state of practice in the industry. Lines of code (LOC) or thousands of lines of code (KLOC) delivered is by far the most commonly used measure of software size in the industry. As the

main cost of producing software is the manpower employed, the cost of developing software is generally measured in terms of person-months of effort spent in development. And productivity is frequently measured in the industry in terms of LOC (or KLOC) per person-month.

Schedule is another important factor in many projects. Business trends are dictating that the time to market of a product should be reduced; that is, the cycle time from concept to delivery should be small. For software this means that it needs to be developed faster, and within the specified time. Unfortunately, the history of software is full of cases where projects have been substantially late.

Besides cost and schedule, the other major factor driving software engineering is **quality.** Today, quality is one of the main mantras, and business strategies are designed around it. Unfortunately, a large number of instances have occurred regarding the unreliability of software—the software often does not do what it is supposed to do or does something it is not supposed to do. Clearly, developing high-quality software is another fundamental goal of software engineering. However, while cost is generally well understood, the concept of quality in the context of software needs further elaboration.software quality comprises six main attributes, as shown in Figure 1.1. These attributes can be defined as follows:

– Functionality. The capability to provide functions which meet stated and implied needs when the software is used.

– Reliability. The capability to provide failure-free service. – Usability. The capability to be understood, learned, and used.

– Efficiency. The capability to provide appropriate performance relative to the amount of resources used.

– Maintainability. The capability to be modified for purposes of making corrections, improvements, or adaptation.

– Portability. The capability to be adapted for different specified environments without applying actions or means other than those provided for this purpose in the product.

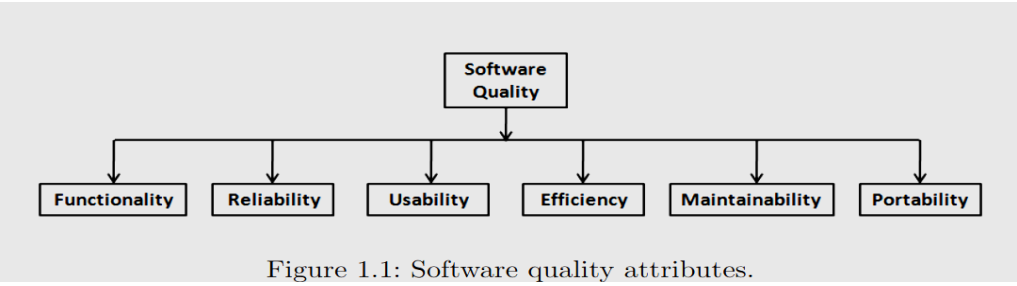


Figure 1.1: Software quality attributes.

Scale and Change: Though cost, schedule, and quality are the main driving forces for a project in our problem domain (of industry strength software), there are some other characteristics of the problem domain that also influence the solution approaches employed. We focus on two such characteristics—scale and change.

Scale :Most industrial-strength software systems tend to be large and complex, requiring tens of thousands of lines of code.A different set of methods will have to be used for conducting a census, and the census problem will require considerably more management, organization, and validation, in addition to counting. Similarly, methods that one can use to develop programs of a few hundred lines cannot be expected to work when software of a few hundred thousand l ines needs to be developed.

A different set of methods must be used for developing large software. Any software project involves the use of engineering and project management. In small projects, informal methods for development and management can be used. However, for large projects, both have to be much more rigorous, as illustrated in Figure 1.2. In other words, to successfully execute a project, a proper method for engineering the system has to be employed and the project has to be tightly managed to make sure that cost, schedule, and quality are under control. Large scale is

a key characteristic of the problem domain and the solution approaches should employ tools and techniques that have the ability to build large software systems.

Change: Change is another characteristic of the problem domain which the approaches for development must handle. As the complete set of requirements for the system is generally not known (often cannot be known at the start of the project) or stated, as development proceeds and time passes, additional requirements are identified, which need to be incorporated in the software being developed. This need for changes requires that methods for development embrace change and accommodate it efficiently. Change requests can be quite disruptive to a project, and if not handled properly, can consume up to 30 to 40% of the development cost

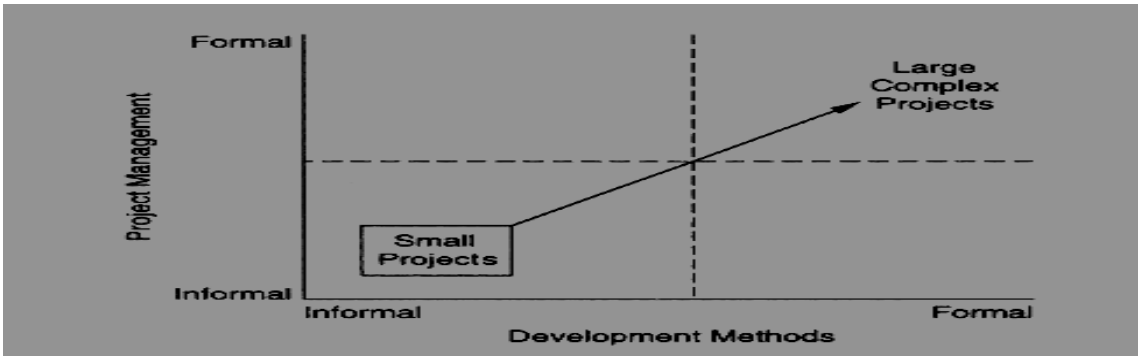


Figure 1.2: The problem of scale.

SOFTWARE PROCESS

Process and Project :A process is a sequence of steps performed for a given purpose .As mentioned earlier, while developing (industrial strength) software, the purpose is to develop software to satisfy the needs of some users or clients, as shown in Figure 2.1. A software project is one instance of this problem, and the development process is what is used to achieve this purpose.

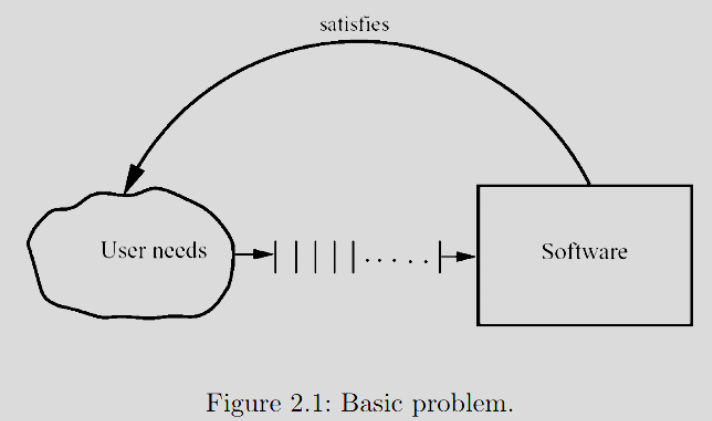


Figure 2.1: Basic problem.

So, for a project its development process plays a key role—it is by following the process the desired end goal of delivering the software is achieved. However, as discussed earlier, it is not sufficient to just reach the final goal of having the desired software, but we want that the project be done at low cost and in low cycle time, and deliver high-quality software.

A process model specifies a general process, which is “optimum” for a class of projects. That is, in the situations for which the model is applicable, using the process model as the project’s process will lead to the goal of developing software with high02 Q&P. A process model is essentially a compilation of best practices into a “recipe” for success in the project.

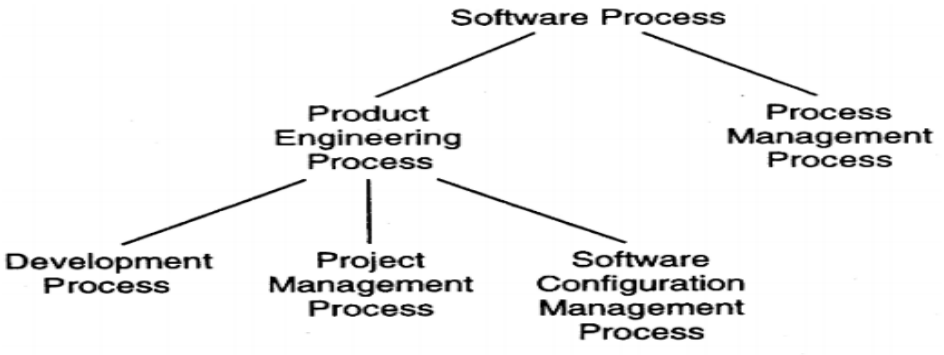
In other words, a process is a means to reach the goals of high quality, low cost, and low cycle time, and a process model provides a process structure that is well suited for a class of projects.

Component Software Processes:

The processes that deal with the technical and management issues of software development are collectively called the software process. As a software project will have to engineer a solution and properly manage the project, there are clearly two major components in a software process—a development process and a project management process.

The **development process** specifies all the engineering activities that need to be performed, whereas the **management process** specifies how to plan and control these activities so that cost, schedule, quality, and other objectives are met.

Effective development and project management processes are the key to achieving the objectives of delivering the desired software satisfying the user needs, while ensuring high productivity and quality. During the project many products are produced which are typically composed of many items (for example, the final source code may be composed of many source files). These items keep evolving as the project proceeds, creating many versions on the way.



As development processes generally do not focus on evolution and changes, to handle them another process called **software configuration control process** is often used. The objective of this component process is to primarily deal with managing change, so that the integrity of the products is not violated despite changes.

These three constituent processes focus on the projects and the products and can be considered as comprising the product engineering processes, as their main objective is to produce the desired product. If the software process can be viewed as a static entity, then these three component processes will suffice. However, a software process itself is a dynamic entity, as it must change to adapt to our increased understanding about software development and availability of newer technologies and tools. Due to this, a process to manage the software process is needed.

The basic objective of the process management process is to improve the software process. By improvement, we mean that the capability of the process to produce quality goods at low cost is improved. For this, the current software process is studied, frequently by studying the projects that have been done using the process. The whole process of understanding the current process, analyzing its properties, determining how to improve, and then affecting the improvement is dealt with by the process management process.

The relationship between these major component processes is shown in Figure 2.2. These component processes are distinct not only in the type of activities performed in them, but typically also in the people who perform the activities specified by the process. In a typical project, development activities are performed by programmers, designers, testers, etc.; the project management process activities are performed by the project management;

configuration control process activities are performed by a group generally called the configuration controller; and the process management process activities are performed by the software engineering process group (SEPG). In this book, we will focus primarily on processes relating to product engineering, particularly the development and project management processes. Much of the book discusses the different phases of a development process and the sub processes or methodologies used for executing these phases. For the rest of the book, we will use the term software process to mean product engineering processes, unless specified otherwise.

SOFTWARE DEVELOPMENT PROCESS MODELS

WATERFALL MODEL:

The simplest process model is the waterfall model, which states that the phases are organized in a linear order. The model was originally proposed **by Royce**, though variations of the model have evolved depending on the nature of activities and the flow of control between them.

In this model, a project begins with feasibility analysis. Upon successfully demonstrating the feasibility of a project, the requirements analysis and project planning begins. The design starts after the requirements analysis is complete, and coding begins after the design is complete. Once the programming is completed, the code is integrated and testing is done. Upon successful completion of testing, the system is installed. After this, the regular operation and maintenance of the system takes place.

The basic idea behind the phases is separation of concerns—each phase deals with a distinct and separate set of concerns. The requirements analysis phase is mentioned as "analysis and planning." Planning is a critical activity in software development. A good plan is based on the requirements of the system and should be done before later phases begin. However, in practice, detailed requirements are not necessary for planning. Consequently, planning usually overlaps with the requirements analysis, and a plan is ready before the later phases begin. This plan is an additional input to all the later phases.

Linear ordering of activities has some important consequences. First, to clearly identify the end of a phase and the beginning of the next, some certification mechanism has to be employed at the end of each phase. This is usually done by some verification and validation means that will ensure that the output of a phase is consistent with its input (which is the output of the previous phase), and that the output of the phase is consistent with the overall requirements of the system. The consequence of the need for certification is that each phase must have some defined output that can be evaluated and certified. That is, when the activities of a phase are completed, there should be some product that is produced by that phase. The outputs of the earlier phases are often called work products and are usually in the form of documents like the requirements document or design document. For the coding phase, the output is the code. Though the set of documents that should be produced in a project is dependent on how the process is implemented, the following documents generally form a reasonable set that should be produced in each project:

– Requirements document

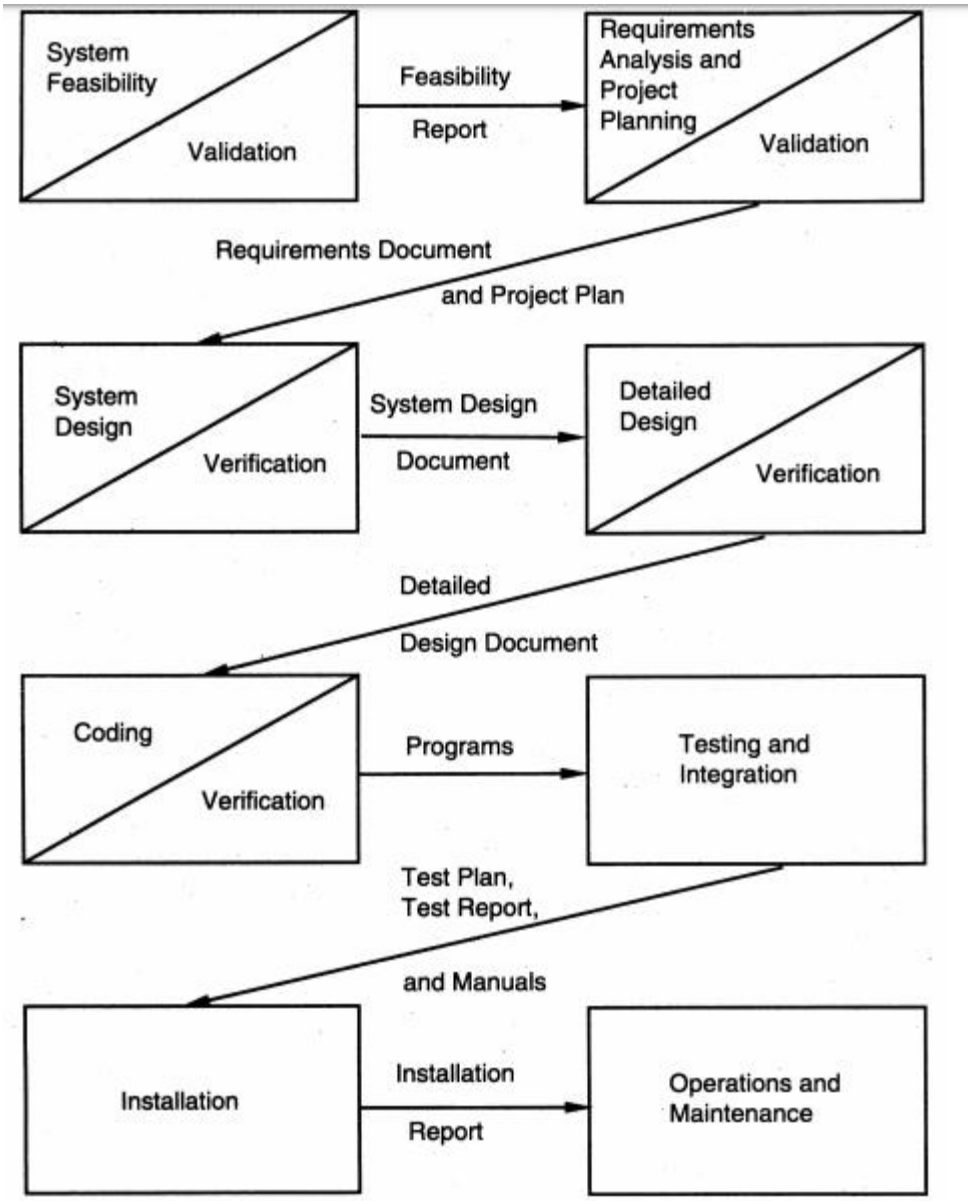
– Project plan

– Design documents (architecture, system, detailed)

– Test plan and test reports

– Final code

– Software manuals (e.g., user, installation, etc.)



One of the main **advantages** of the waterfall model is its simplicity. It is conceptually straightforward and divides the large task of building a software system into a series of cleanly divided phases, each phase dealing with a separate logical concern. It is also easy to administer in a contractual setup—as each phase is completed and its work product produced, some amount of money is given by the customer to the developing organization.

The waterfall model, although widely used, has some strong **limitations**. key limitations are:

1. It assumes that the requirements of a system can be frozen (i.e., baselined) before the design begins. This is possible for systems designed to automate an existing manual system. But for new systems, determining the requirements is difficult as the user does not even know the requirements. Hence, having unchanging requirements is unrealistic for such projects.

2. Freezing the requirements usually requires choosing the hardware (because it forms a part of the requirements specification). A large project might take a few years to complete. If the hardware is selected early, then due to the speed at which hardware technology is changing, it is likely that the final software will use a hardware technology on the verge of becoming obsolete. This is clearly not desirable for such expensive software systems.

3. It follows the "big bang" approach—the entire software is delivered in one shot at the end. This entails heavy risks, as the user does not know until the very end what they are getting. Furthermore, if the project runs out of money in the middle, then there will be no software. That is, it has the "all or nothing" value proposition.

4. It encourages "requirements bloating". Since all requirements must be specified at the start and only what is specified will be delivered, it encourages the users and other stakeholders to add even those features which they think might be needed (which finally may not get used).

5. It is a document-driven process that requires formal documents at the end of each phase.

Despite these limitations, the waterfall model has been the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is, if the developing organization is quite familiar with the problem domain and the requirements for the software are quite clear, the waterfall model works well, and may be the most efficient process.

PROTOTYPING MODEL:

The goal of a prototyping-based development process is to **counter the first limitation of the waterfall** model. The basic idea here is that instead of freezing the requirements before any design or coding can proceed, a **throwaway prototype** is built to help understand the requirements. This prototype is developed based on the currently known requirements.

Development of the prototype obviously undergoes design, coding, and testing, but each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an actual feel of the system, which can enable the client to better understand the requirements of the desired system. This results in more stable requirements that change less frequently.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determine the requirements. In such situations, letting the client "play" with the prototype provides invaluable and intangible inputs that help determine the requirements for the system. It is also an effective method of demonstrating the feasibility of a certain approach. This might be needed for novel systems, where it is not clear that constraints can be met or that algorithms can be developed to implement the requirements. In both situations, the risks associated with the projects are being reduced through the use of prototyping. The process model of the prototyping approach is shown in Figure .

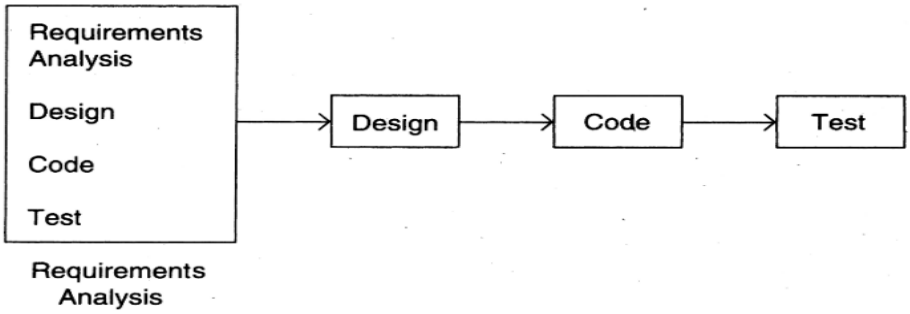


Figure 2.4: The prototyping model.

The development of the prototype typically starts when the preliminary version of the requirements specification document has been developed. At this stage, there is a reasonable understanding of the system and its needs and which needs are unclear or likely to change. After the prototype has been developed, the end users and clients are given an opportunity to use and explore the prototype. Based on their experience, they provide feedback to the developers regarding the prototype:

- what is correct,
- what needs to be modified,
- what is missing, what is not needed, etc.

Based on the feedback, the prototype is modified to incorporate some of the suggested changes that can be done easily, and then the users and the clients are again allowed to use the system. This cycle repeats until, in the judgment of the prototype developers and analysts, the benefit from further changing the system and

obtaining feedback is outweighed by the cost and time involved in making the changes and obtaining the feedback. Based on the feedback, the initial requirements are modified to produce the final requirements specification, which is then used to develop the production quality system.

Prototype is to be thrown away, only minimal documentation needs to be produced during prototyping. For example, design documents, a test plan, and a test case specification are not needed during the development of the prototype.

Another important cost-cutting measure is to reduce testing. Because testing consumes a major part of development expenditure during regular software development, this has a considerable impact in reducing costs. By using these types of cost-cutting methods, it is possible to keep the cost of the prototype to less than a few percent of the total development cost. And the returns from this extra cost can be substantial.

First, the experience of developing the prototype will reduce the cost of the actual software development.

Second, as requirements will be more stable now due to the feedback from the prototype, there will be fewer changes in the requirements. Consequently the costs incurred due to changes in the requirements will be substantially reduced. Third, the quality of final software is likely to be far superior, as the experience engineers have obtained while developing the prototype will enable them to create a better design, write better code, and do better testing. Developing a prototype mitigates many risks that exist in a project where requirements are not well known.

Overall, prototyping is well suited for projects where requirements are hard to determine and the confidence in the stated requirements is low. In such projects where requirements are not properly understood in the beginning, using the prototyping process model can be the most effective method for developing the software. It is also an excellent technique for reducing some types of risks associated with a project.

ITERATIVE DEVELOPMENT MODEL:

The iterative development process model **counters the third and fourth limitations of the waterfall** model and tries to combine the benefits of both prototyping and the waterfall model. The basic idea is that the software should be developed in increments, each increment adding some functional capability to the system until the full system is implemented. The iterative enhancement model is an example of this approach. In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system.

A project control list is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far along the project is at any given step from the final system. Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called the design phase, implementation phase, and analysis phase. The process is iterated until the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement model is shown in Figure 2.5.

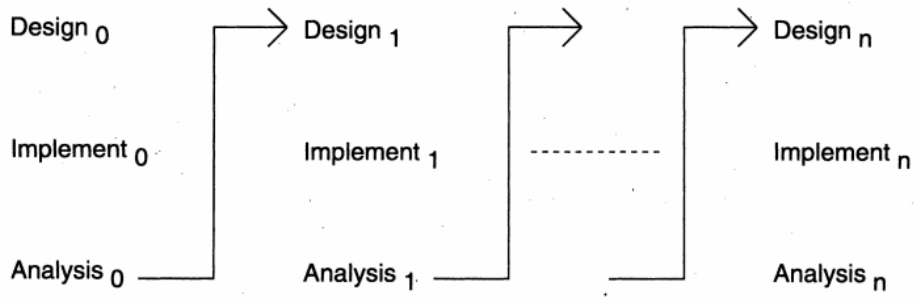


Figure 2.5: The iterative enhancement model

The project control list guides the iteration steps and keeps track of all tasks that must be done. Based on the analysis, one of the tasks in the list can include redesign of defective components or redesign of the entire system. However, redesign of the system will generally occur only in the initial steps. In the later steps, the design would have stabilized and there is less chance of redesign. Each entry in the list is a task that should be performed in one step of the iterative enhancement process and should be simple enough to be completely understood. Selecting tasks in this manner will minimize the chances of error and reduce the redesign work.

Overall, it may not offer the best technical solution, but the benefits may outweigh the costs in many projects. Another common approach for iterative development is to do the requirements and the architecture design in a standard waterfall or prototyping approach, but deliver the software iteratively. That is, the building of the system, which is the most time and effort-consuming task, is done iteratively, though most of the requirements are specified upfront. We can view this approach as having one iteration delivering the requirements and the architecture plan, and then further iterations delivering the software in increments. At the start of each delivery iteration, which requirements will be implemented in this release are decided, and then the design is enhanced and code developed to implement the requirements. The iteration ends with delivery of a working software system providing some value to the end user. Selecting of requirements for an iteration is done primarily based on the value the requirement provides to the end users and how critical they are for supporting other requirements. This approach is shown in Figure 2.6.

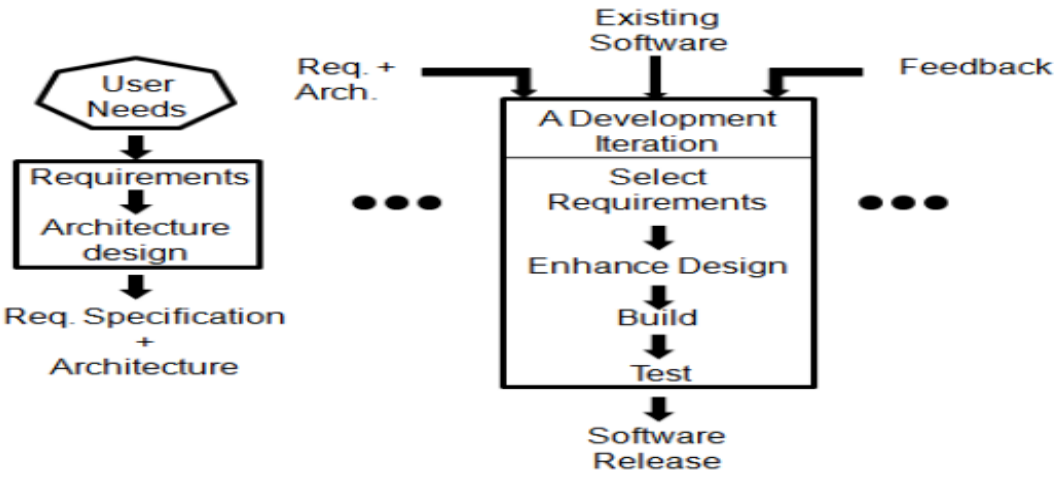


Figure 2.6: Iterative delivery approach.

The **advantage** of this approach is that as the requirements are mostly known upfront, an overall view of the system is available and a proper architecture can be designed which can remain relatively stable. With this, hopefully rework in development iterations will diminish. At the same time, the value to the end customer is delivered iteratively so it does not have the all-or-nothing risk. Also, since the delivery is being done incrementally, and planning and execution of each iteration is done separately, feedback from an iteration can be incorporated in the next iteration. Even new requirements that may get uncovered can also be incorporated. Hence, this model of iterative development also provides some of the benefits of the model discussed above. The iterative approach is becoming extremely popular, despite some difficulties in using it in this context.

There are a few key reasons for its increasing popularity.

First and foremost, in today's world clients do not want to invest too much without seeing returns. In the current business scenario, it is preferable to see returns continuously of the investment made. The iterative model permits this—after each iteration some working software is delivered, and the risk to the client is therefore limited.

Second, as businesses are changing rapidly today, they never really know the "complete" requirements for the software, and there is a need to constantly add new capabilities to the software to adapt the business to changing situations. Iterative process allows this.

Third, each iteration provides a working system for feedback, which helps in developing stable requirements for the next iteration

RATIONAL UNIFIED PROCESS MODEL:

Rational Unified Process (RUP) [51, 63] is another iterative process model that was designed by Rational, now part of IBM. Though it is a general pro- cess model, it was designed for object-oriented development using the Unified Modeling Language (UML). (We will discuss these topics in a later chapter).

RUP proposes that development of software be divided into cycles, each cycle delivering a fully working system. Generally, each cycle is executed as a separate project whose goal is to deliver some additional capability to an exist- ing system (built by the previous cycle). Hence, for a project, the process for a cycle forms the overall process. Each cycle itself is broken into four consecutive phases:

– Inception phase

– Elaboration phase

– Construction phase

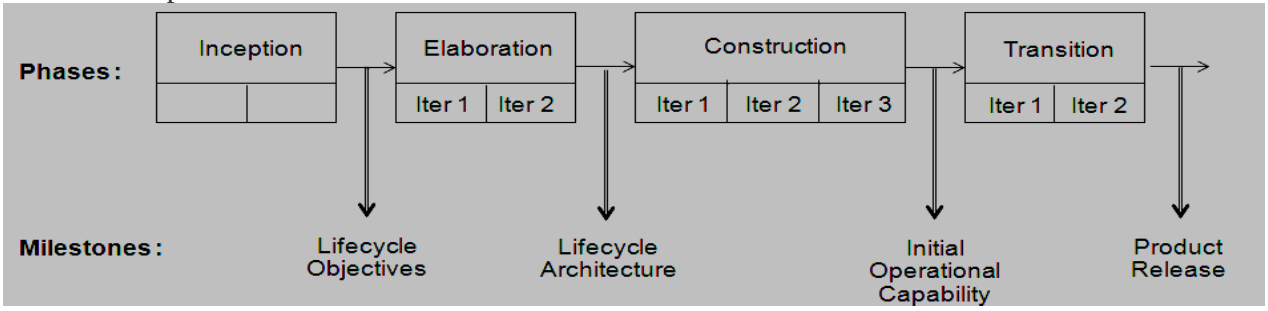
– Transition phase

the **inception phase** is to establish the goals and scope of the project, and completion of this phase is the lifecycle objectives milestone. This milestone should specify the vision and high-level capability of the eventual system, what business benefits it is expected to provide, some key illustrative use cases of the system, key risks of the project, and a basic plan of the project regarding the cost and schedule. Based on the output of this phase, a go/no-go decision may be taken. And if the project is to proceed, then this milestone represents that there is a shared vision among the stakeholders and they agree to the project, its vision, benefits, cost, usage, etc.

In the **elaboration phase**, the architecture of the system is designed, based on the detailed requirements analysis. The completion of this phase is the life- cycle architecture milestone. At the end of this phase, it is expected that most of the requirements have been identified and specified, and the architecture of the system has been designed (and specified) in a manner that it addresses the technical risks identified in the earlier phase. In addition, a high-level project plan for the project has been prepared showing the remaining phases and iter- ations in those, and the current perception of risks.

In the **construction phase**, the software is built and tested. This phase results in the software product to be delivered, along with associated user and other manuals, and successfully completing this phase results in the initial operational capability milestone being achieved.

the **transition phase** is to move the software from the devel- opment environment to the client's environment, where it is to be hosted. This is a complex task which can require additional testing, conversion of old data for this software to work, training of personnel, etc. The successful execution of this phase results in achieving the milestone product release.



RUP has carefully chosen the phase names so as not to confuse them with the engineering tasks that are to be done in the project, as in RUP the en- gineering tasks and phases are separate. Different engineering activities may be performed in a phase to achieve its milestones. RUP groups the activities into different subprocesses which it calls core process workflows. These sub- processes correspond to the tasks of performing requirements analysis, doing design, implementing the design, testing, project management, etc. Some of the subprocesses are shown in Table 2.1.

One key difference of RUP from other models is that it has separated the phases from the tasks and allows multiple of these subprocesses to function within a phase. In waterfall (or waterfall-based iterative model), a phase within a process was linked to a particular task performed by some process like re- quirements, design, etc. In RUP these tasks are separated from the stages, and it allows, for example, during construction, execution of the requirements process. That is, it allows some part of the requirement activity be done even in construction, something the waterfall did not allow. So, a project, if it so wishes, may do detailed requirements only for some features during the elaboration phase, and may do detailing of other requirements while the construction is going on (maybe the first iteration of it). This not only allows a project a greater degree of flexibility in planning when the different tasks should be done, it also captures the reality of the situation—it is often not possible to specify all requirements at the start and it is best to start the project with some requirements and work out the details later.

Though a subprocess may be active in many phases, as can be expected, the volume of work or the effort being spent on the subprocess will vary with phases. For example, it is expected that a lot more effort will be spent in the requirement subprocess during elaboration, and less will be spent in construc- tion, and still less, if any, will be spent in transition. The effort spent in a subprocess in different phases will, of course, de- pend on the project. However, a general pattern is indicated in Table 2.1 by specifying if the level of effort for the phase is high, medium, low, etc.

Table 2.1: Activity level of subprocesses in different phases of RUP.

	Inception	Elaboration	Construction	Transition
Requirements	High	High	Low	Nil
Anal. and Design	Low	High	Medium	Nil
Implementation	Nil	Low	High	Low
Test	Nil	Low	High	Medium
Deployment	Nil	Nil	Medium	High
Pro j. Mgmt.	Medium	Medium	Medium	Medium
Config. Mgmt	Low	Low	High	High

Overall, RUP provides a flexible process model, which follows an iterative approach not only at a top level (through cycles), but also encourages iterative approach during each of the phases in a cycle. And in phases, it allows the different tasks to be done as per the needs of the project.

Timeboxing Model

To speed up development, parallelism between the different iterations can be employed. That is, a new iteration commences before the system produced by the current iteration is released, and hence development of a new release happens in parallel with the development of the current release. By starting an iteration before the previous iteration has completed, it is possible to reduce the average delivery time for iterations. the basic unit of development is a time box, which is of fixed duration. Since the duration is fixed, a key factor in selecting the requirements or features to be built in a time box is what can be fit into the time box. This is in contrast to regular iterative approaches where the functionality is selected and then the time to deliver is determined. Timeboxing changes the perspective of development and makes the schedule a nonnegotiable and a high-priority commitment.

Each time box is divided into a sequence of stages, like in the waterfall model. Each stage performs some clearly defined task for the iteration and produces a clearly defined output. The model also requires that the duration of each stage, that is, the time it takes to complete the task of that stage, is approximately the same. Furthermore, the model requires that there be a dedicated team for each stage. That is, the team for a stage performs only tasks of that stage—tasks for other stages are performed by their respective teams. This is quite different from other iterative models where the implicit assumption is that the same team performs all the different tasks of the project or the iteration.

Having time-boxed iterations with stages of equal duration and having dedicated teams renders itself to pipelining of different iterations. (Pipelining is a concept from hardware in which different instructions are executed in parallel, with the execution of a new instruction starting once the first stage of the previous instruction is finished.)

To illustrate the use of this model, consider a time box consisting of three stages: requirement specification, build, and deployment. The requirement stage is executed by its team of analysts and ends with a prioritized list of requirements to be built in this iteration along with a high-level design. The build team develops the code for implementing the requirements, and performs the testing. The tested code is then handed over to the deployment team, which performs predeployment tests, and then installs the system for production use. These three stages are such that they can be done in approximately equal time in an iteration.

With a time box of three stages, the project proceeds as follows. When the requirements team has finished requirements for timebox-1, the requirements are given to the build team for building the software. The requirements team then goes on and starts preparing the requirements for timebox-2. When the build for timebox-1 is completed, the code is handed over to the deployment team, and the build team moves on to build code for requirements for timebox-
2, and the requirements team moves on to doing requirements for timebox-3. This pipelined execution of the timeboxing process is shown in Figure 2.8 [59].

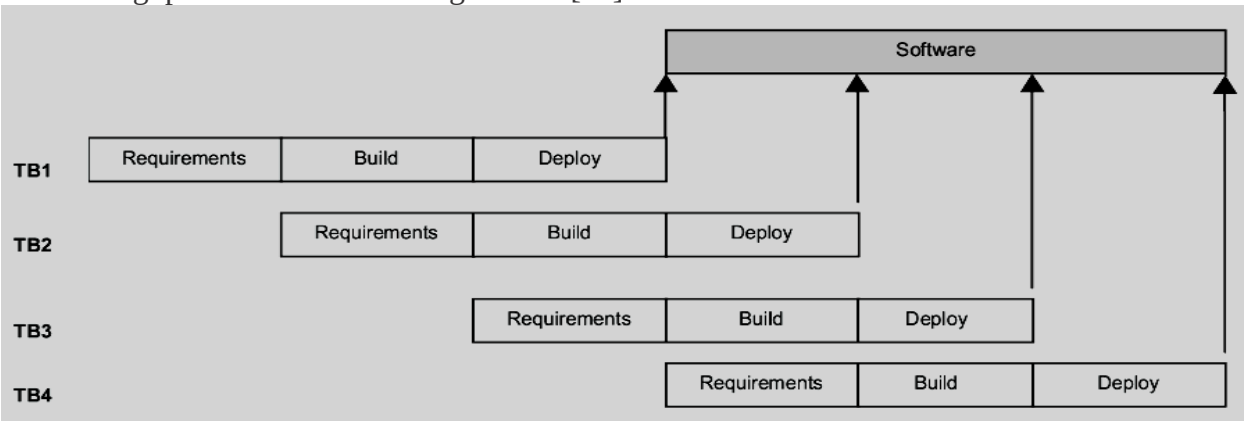


Figure 2.8: Executing the timeboxing process model.

With a three-stage time box, at most three iterations can be concurrently in progress. If the time box is of size T days, then the first software delivery will occur after T days. The subsequent deliveries, however, will take place after every T/3 days. For example, if the time box duration T is 9 weeks (and each stage duration is 3 weeks), the first delivery is made 9 weeks after the start of the project. The

second delivery is made after 12 weeks, the third after 15 weeks, and so on. Contrast this with a linear execution of iterations, in which the first delivery will be made after 9 weeks, the second after 18 weeks, the third after 27 weeks, and so on.

There are three teams working on the project—the requirements team, the build team, and the deployment team. The teamwise activity for the 3-stage pipeline discussed above is shown in Figure 2.9 [59].

It should be clear that the duration of each iteration has not been reduced.

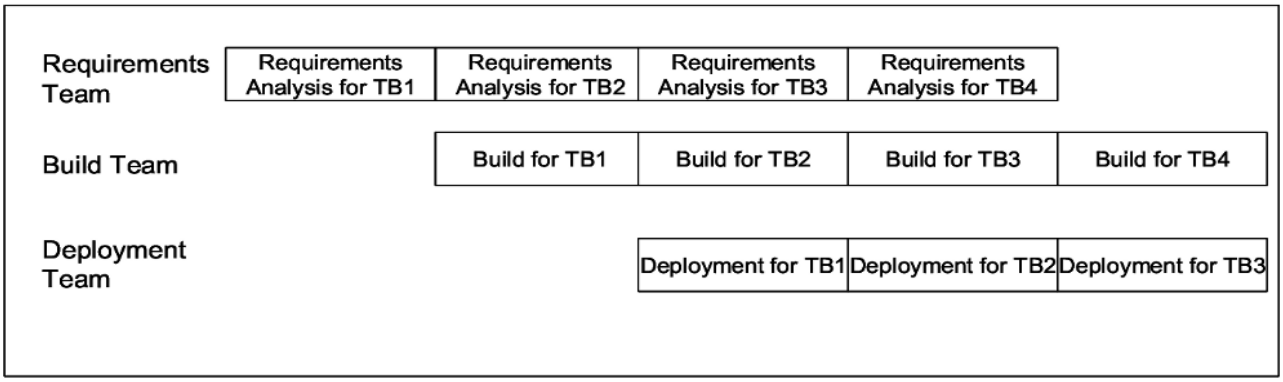


Figure 2.9: Tasks of different teams.

The total work done in a time box and the effort spent in it also remains the same—the same amount of software is delivered at the end of each iteration as the time box undergoes the same stages. If the effort and time spent in each iteration also remains the same, then what is the cost of reducing the delivery time? The real cost of this reduced time is in the resources used in this model. With timeboxing, there are dedicated teams for different stages and the total team size for the project is the sum of teams of different stages. This is the main difference from the situation where there is a single team which performs all the stages and the entire team works on the same iteration.

Hence, the timeboxing provides an approach for utilizing additional man- power to reduce the delivery time. It is well known that with standard methods of executing projects, we cannot compress the cycle time of a project substan- tially by adding more manpower. However, through the timeboxing model, we can use more manpower in a manner such that by parallel execution of different stages we are able to deliver software quicker. In other words, it provides a way of shortening delivery times through the use of additional manpower.

Timeboxing is well suited for projects that require a large number of features to be developed in a short time around a stable architecture using stable technologies.

Extreme Programming and Agile Processes

Agile development approaches evolved in the 1990s as a reaction to documen- tation and bureaucracy-based processes, particularly the waterfall approach. Agile approaches are based on some common principles, some of which are [www.extremeprogramming.org]:

– Working software is the key measure of progress in a project.
– For progress in a project, therefore, software should be developed and deliv- ered rapidly in small increments.
– Even late changes in the requirements should be entertained (small-increment model of development helps in accommodating them).
– Face-to-face communication is preferred over documentation.
– Continuous feedback and involvement of customer is necessary for developing good-quality software.
– Simple design which evolves and improves with time is a better approach than doing an elaborate design up front for handling all possible scenarios.
– The delivery dates are decided by empowered teams of talented individuals (and are not dictated).

Many detailed agile methodologies have been proposed, some of which are widely used now. Extreme programming (XP) is one of the most popular and well-known approaches in the family of agile methods. Like all agile approaches, it believes that changes are inevitable and rather than treating changes as undesirable, development should embrace change. And to accommodate change, the development process has to be lightweight and quick to respond. For this, it develops software iteratively, and avoids reliance on detailed and multiple documents which are hard to maintain. Instead it relies on face-to-face communication, simplicity, and feedback to ensure that the desired changes are quickly and correctly reflected in the programs. Here we briefly discuss the development process of XP, as a representative of an agile process.

An extreme programming project starts with user stories which are short (a few sentences) descriptions of what scenarios the customers and users would like the system to support. They are different from traditional requirements specification primarily in details—user stories do not contain detailed requirements which are to be uncovered only when the story is to be implemented, therefore allowing the details to be decided as late as possible. Each story is written on a separate card, so they can be flexibly grouped.

The empowered development team estimates how long it will take to imple- ment a user story. The estimates are rough, generally stated in weeks. Using these estimates and the stories, release planning is done which defines which stories are to be built in which system release, and the dates of these releases. Frequent and small releases are encouraged, and for a release, iterations are employed. Acceptance tests are also built from the stories, which are used to test the software before the release. Bugs found during the acceptance testing for an iteration can form work items for the next iteration. This overall process is shown in Figure 2.10.

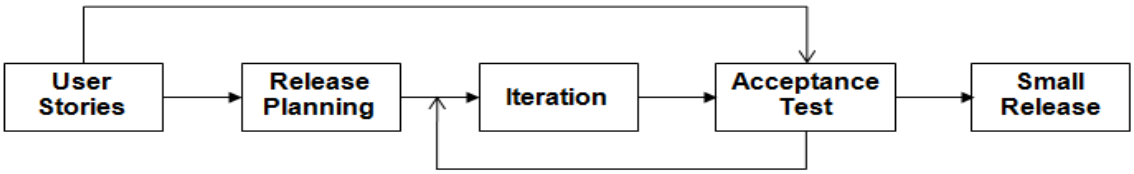


Figure 2.10: Overall process in XP.

Development is done in iterations, each iteration lasting no more than a few weeks. An iteration starts with iteration planning in which the stories to be implemented in this iteration are selected—high-value and high-risk stories are considered as higher priority and implemented in early iterations. Failed acceptance tests in previous iteration also have to be handled. Details of the stories are obtained in the iteration for doing the development.

The development approach used in an iteration has some unique practices. First, it envisages that development is done by pairs of programmers (called pair programming and which we will discuss further in Chapter 7), instead of individual programmers. Second, it suggests that for building a code unit, automated unit tests be written first before the actual code is written, and then the code should be written to pass the tests. This approach is referred to as test-driven development, in contrast to regular code-first development in which programmers first write code and then think of how to test it. (We will discuss test-driven development further in Chapter 7.) As functionality of the unit increases, the unit tests are enhanced first, and then the code is enhanced to pass the new set of unit tests. Third, as it encourages simple solutions as well as change, it is expected that the design of the solution devised earlier may at some point become unsuitable for further development. To handle this situation, it suggests that refactoring be done to improve the design, and then use the refactored code for further development. During refactoring, no new functionality is added, only the design of the existing programs is improved. (Refactoring will be discussed further in Chapter 7.) Fourth, it encourages frequent integration of different units. To avoid too many changes in the base code happening together, only one pair at a time can release their changes and integrate into the common code base. The process within an iteration is shown below.

This is a very simplified description of XP. There are many other rules in XP relating to issues like rights of programmers and customers, communi- cation between the team members and use of metaphors, trust and visibility to all stakeholders, collective ownership of code in which any pair can change any code, team management, building quick spike solutions to resolve diffi- cult technical and architectural issues or to explore some approach, how bugs are to be handled, how what can be done within an iteration is to be esti- mated from the progress made in the previous iteration, how meetings are to be conducted, how a day in the development should start, etc. The website www.extremeprogramming.org is a good source on these, as well as other as- pects of XP.

XP, and other agile methods, are suitable for situations where the volume and pace of requirements change is high, and where requirement risks are con- siderable. Because of its reliance on strong communication between all the team members, it is effective when teams are collocated and of modest size, of up to about 20 members. And as it envisages strong involvement of the customer in the development, as well as in planning the delivery dates, it works well when the customer is willing to be heavily involved during the entire development, working as a team member.

Pro ject Management Process

While the selection of the development process decides the phases and tasks to be done, it does not specify things like how long each phase should last, or how many resources should be assigned to a phase, or how a phase should be monitored. Quality and productivity in the project will also depend critically on these decisions. To meet the cost, quality, and schedule objectives, resources have to be properly allocated to each activity for the project, and progress of different activities has to be monitored and corrective actions taken when needed. All these activities are part of the project management process. Hence, a project management process is necessary to ensure that the engineering process ends up meeting the real-world objectives of cost, schedule, and quality.

The project management process specifies all activities that need to be done by the project management to ensure that cost and quality objectives are met. Its basic task is to ensure that, once a development process is chosen, it is implemented optimally. That is, the basic task is to plan the detailed implementation of the process for the particular project and then ensure that the plan is properly executed. For a large project, a proper management process is essential for success.

The activities in the management process for a project can be grouped broadly into three phases**: planning, monitoring and control, and termination analysis**.

Project management begins with **planning,** which is perhaps the most critical project management activity. The goal of this phase is to develop a plan for software development following which the objectives of the project can be met successfully and efficiently. A software plan is usually produced before the development activity begins and is updated as development proceeds and data about progress of the project becomes available. During planning, the major activities are cost estimation, schedule and milestone determination, project staffing, quality control plans, and controlling and monitoring plans. Project planning is undoubtedly the single most important management activity, and it forms the basis for monitoring and control. We will devote one full chapter.

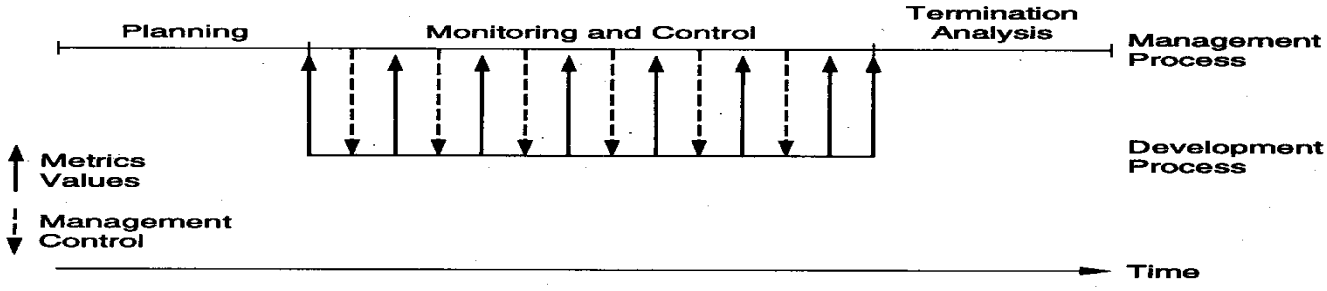
later in the book to project planning.

Project **monitoring and control** phase of the management process is the longest in terms of duration; it encompasses most of the development process. It includes all activities the project management has to perform while the development is going on to ensure that project objectives are met and the development proceeds according to the developed plan (and update the plan, if needed). As cost, schedule, and quality are the major driving forces, most of the activity of this phase revolves around monitoring factors that affect these. Monitoring potential risks for the project, which might prevent the

project from meeting its objectives, is another important activity during this phase. And if the information obtained by monitoring suggests that objectives may not be met, necessary actions are taken in this phase by exerting suitable control on the development activities.

Monitoring a development process requires proper information about the project. Such information is typically obtained by the management process from the development process. Consequently, the implementation of a development process model should ensure that each step in the development process produces information that the management process needs for that step. That is, the development process provides the information the management process needs. However, interpretation of the information is part of monitoring and control.

Whereas monitoring and control last the entire duration of the project, the last phase of the management process—**termination analysis**—is performed when the development process is over. The basic reason for performing termi- nation analysis is to provide information about the development process and learn from the project in order to improve the process. This phase is also of- ten called postmortem analysis. In iterative development, this analysis can be done after each iteration to provide feedback to improve the execution of further iterations. We will not discuss it further in the book.The temporal relationship between the management process and the development process is shown in Figure 2.12. This is an idealized relationship show- ing that planning is done before development begins, and termination analysis is done after development is over. As the figure shows, during the development, from the various phases of the development process, quantitative information flows to the monitoring and control phase of the management process, which uses the information to exert control on the development process.



UNIT-2:Software Requirements Analysis and Specification

Value of good SRS:

A basic purpose of the SRS is to bridge this communication gap so they have a shared vision of the software being built. Hence, one of the main advantages of a good SRS is:

– An SRS establishes the basis for agreement between the client and the sup- plier on what the software product will do.

This basis for agreement is frequently formalized into a legal contract between the client (or the customer) and the developer (the supplier). So, through SRS, the client clearly describes what it expects from the supplier, and the developer clearly understands what capabilities to build in the software. A related, but important, advantage is:

– An SRS provides a reference for validation of the final product.

That is, the SRS helps the client determine if the software meets the require- ments. Without a proper SRS, there is no way a client can determine if the software being delivered is what was ordered, and there is no way the developer can convince the client that all the requirements have been fulfilled.