Chapter 1

Introduction

Ask any student who has had some programming experience the following question: You are given a problem for which you have to build a software system that most students feel will be approximately 10,000 lines of (say C or Java) code. If you are working full time on it, how long will it take you to build this system?

The answer of students is generally 1 to 3 months. And, given the programming expertise of the students, there is a good chance that they will be able to build a system and demo it to the Professor within 2 months. With 2 months as the completion time, the productivity of the student will be 5,000 lines of code (LOC) per person-month.

Now let us take an alternative scenario—we act as clients and pose the same problem to a company that is in the business of developing software for clients. Though there is no "standard" productivity figure and it varies a lot, it is fair to say a productivity figure of 1,000 LOC per person-month is quite respectable (though it can be as low as 100 LOC per person-month for embedded systems). With this productivity, a team of professionals in a software organization will take 10 person-months to build this software system.

Why this difference in productivity in the two scenarios? Why is it that the same students who can produce software at a productivity of a few thousand LOC per month while in college end up producing only about a thousand LOC per month when working in a company? Why is it that students seem to be more productive in their student days than when they become professionals?

The answer, of course, is that two different things are being built in the two scenarios. In the first, a *student system* is being built whose main purpose is to demo that it works. In the second scenario, a team of professionals in an organization is building the system for a client who is paying for it, and whose business may depend on proper working of the system. As should be evident, building the latter type of software is a different problem altogether. It is this problem in which software engineering is interested. The difference between the two types of software was recognized early and the term *software*

engineering was coined at NATO sponsored conferences held in Europe in the 1960s to discuss the growing software crisis and the need to focus on software development.

In the rest of the chapter we further define our problem domain. Then we discuss some of the key factors that drive software engineering. This is followed by the basic approach followed by software engineering. In the rest of the book we discuss in more detail the various aspects of the software engineering approach.

1.1 The Problem Domain

In software engineering we are not dealing with programs that people build to illustrate something or for hobby (which we are referring to as student systems). Instead the problem domain is the software that solves some problem of some users where larger systems or businesses may depend on the software, and where problems in the software can lead to significant direct or indirect loss. We refer to this software as *industrial strength software*. Let us first discuss the key difference between the student software and the industrial strength software.

1.1.1 Industrial Strength Software

A student system is primarily meant for demonstration purposes; it is generally not used for solving any real problem of any organization. Consequently, nothing of significance or importance depends on proper functioning of the software. Because nothing of significance depends on the software, the presence of "bugs" (or defects or faults) is not a major concern. Hence the software is generally not designed with quality issues like portability, robustness, reliability, and usability in mind. Also, the student software system is generally used by the developer him- or herself, therefore the need for documentation is nonexistent, and again bugs are not critical issues as the user can fix them as and when they are found.

An *industrial strength software system*, on the other hand, is built to solve some problem of a client and is used by the clients organization for operating some part of business (we use the term "business" in a very broad sense—it may be to manage inventories, finances, monitor patients, air traffic control, etc.) In other words, important activities depend on the correct functioning of the system. And a malfunction of such a system can have huge impact in terms of financial or business loss, inconvenience to users, or loss of property and life. Consequently, the software system needs to be of high quality with respect to properties like dependability, reliability, user-friendliness, etc.

This requirement of high quality has many ramifications. First, it requires that the software be thoroughly tested before being used. The need for rigorous testing increases the cost considerably. In an industrial strength software project, 30% to 50% of the total effort may be spent in testing (while in a student software even 5% may be too high!)

Second, building high quality software requires that the development be broken into phases such that output of each phase is evaluated and reviewed so bugs can be removed. This desire to partition the overall problem into phases and identify defects early requires more documentation, standards, processes, etc. All these increase the effort required to build the software—hence the productivity of producing industrial strength software is generally much lower than for producing student software.

Industrial strength software also has other properties which do not exist in student software systems. Typically, for the same problem, the detailed requirements of what the software should do increase considerably. Besides quality requirements, there are requirements of backup and recovery, fault tolerance, following of standards, portability, etc. These generally have the effect of making the software system more complex and larger. The size of the industrial strength software system may be two times or more than the student system for the same problem.

Overall, if we assume one-fifth productivity, and an increase in size by a factor of two for the same problem, an industrial strength software system will take about 10 times as much effort to build as a student software system for the same problem. The rule of thumb Brooks gives also says that industrial strength software may cost about 10 times the student software[25]. The software industry is largely interested in developing industrial strength software, and the area of software engineering focuses on how to build such systems. In the rest of the book, when we use the term software, we mean industrial strength software.

IEEE defines *software* as the collection of computer programs, procedures, rules, and associated documentation and data [91]. This definition clearly states that software is not just programs, but includes all the associated documentation and data. This implies that the discipline dealing with the development of software should not deal only with developing programs, but with developing all the things that constitute software.

1.1.2 Software is Expensive

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.

The productivity in the software industry for writing fresh code generally ranges from 300 to 1,000 LOC per person-month. That is, for developing software, the average productivity per person, per month, over the entire development cycle is about 300 to 1,000 LOC. And software companies charge the client for whom they are developing the software upwards of $100,000 per person-year or more than $8,000 per person-month

(which comes to about $50 per hour). With the current productivity figures of the industry, this translates into a cost per line of code of approximately $8 to $25. In other words, each line of delivered code costs between $8 and $25 at current costs and productivity levels! And even small projects can easily end up with software of 50,000 LOC. With this productivity, such a software project will cost between $ 0.5 million and $1.25 million!

Given the current compute power of machines, such software can easily be hosted on a workstation or a small server. This implies that software that can cost more than a million dollars can run on hardware that costs at most tens of thousands of dollars, clearly showing that the cost of hardware on which such an application can run is a fraction of the cost of the application software! This example clearly shows that not only is software very expensive, it indeed forms the major component of the total automated system, with the hardware forming a very small component. This is shown in the classic hardware-software cost reversal chart in Figure 1 [17].

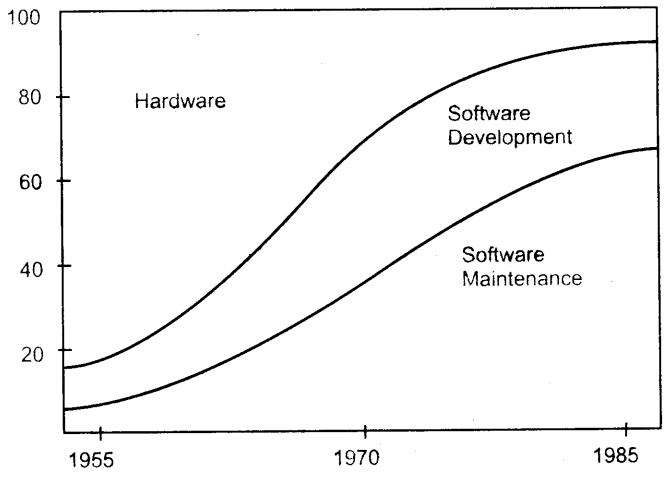

Figure 1.1: Hardware-software cost trend.

As Figure 1 shows, in the early days, the cost of hardware used to dominate the system cost. As the cost of hardware has lessened over the years and continues to decline, and as the power of hardware doubles every 2 years or so (the Moore's law) enabling larger software systems to be run on it, cost of software has now become the dominant factor in systems.

1.1.3 Late and Unreliable

Despite considerable progress in techniques for developing software, software development remains a weak area. In a survey of over 600 firms, more than 35% reported having some computer-related development project that they categorized as a *runaway*[131]. A runaway is not a project that is somewhat late or somewhat over budget—it is one where the budget and schedule are out of control. The problem has become so severe that it has spawned an industry of its own; there are consultancy companies that advise how to rein such projects, and one such company had more than $30 million in revenues from more than 20 clients [131].

Similarly, a large number of instances have been quoted regarding the unreliability of software; the software does not do what it is supposed to do or does something it is not supposed to do. In one defense survey, it was reported that more than 70% of all the equipment failures were due to software! And this is in systems that are loaded with electrical, hydraulic, and mechanical systems. This just indicates that all other engineering disciplines have advanced far more than software engineering, and a system comprising the products of various engineering disciplines finds that software is the weakest component. Failure of an early Apollo flight was also attributed to software. Similarly, failure of a test firing of a missile in India was attributed to software problems. Many banks have lost millions of dollars due to inaccuracies and other problems in their software [122].

A note about the cause of unreliability in software: software failures are different from failures of, say, mechanical or electrical systems. Products of these other engineering disciplines fail because of the change in physical or electrical properties of the system caused by aging. A software product, on the other hand, never wears out due to age. In software, failures occur due to bugs or errors that get introduced during the design and development process. Hence, even though a software system may fail after operating correctly for some time, the bug that causes that failure was there from the start! It only got executed at the time of the failure. This is quite different from other systems, where if a system fails, it generally means that sometime before the failure the system developed some problem (due to aging) that did not exist earlier.

1.1.4 Maintenance and Rework

Once the software is delivered and deployed, it enters the *maintenance* phase. Why is maintenance needed for software, when software does not age? Software needs to be maintained not because some of its components wear out and need to be replaced, but because there are often some residual errors remaining in the system that must be removed as they are discovered. It is commonly believed that the state of the art today is such that almost all software that is developed has residual errors, or bugs, in it. Many of these surface only after the system has been in operation, sometimes for a long time. These errors, once discovered, need to be removed, leading to the software being changed. This is sometimes called *corrective maintenance.*

Even without bugs, software frequently undergoes change. The main reason is that software often must be upgraded and enhanced to include more features and provide more services. This also requires modification of the software. It has been argued that once a software system is deployed, the environment in which it operates changes. Hence, the needs that initiated the software development also change to reflect the needs of the new environment. Hence, the software must adapt to the needs of the changed environment. The changed software then changes the environment, which in turn requires further change. This phenomenon is sometimes called the *law of software evolution.* Maintenance due to this phenomenon is sometimes called *adaptive maintenance.*

Though maintenance is not considered a part of software development, it is an extremely important activity in the life of a software product. If we consider the total life of software, the cost of maintenance generally exceeds the cost of developing the software! The maintenance-to-development-cost ratio has been variously suggested as 80:20, 70:30, or 60:40. Figure 1 also shows how the maintenance costs are increasing.

Maintenance work is based on existing software, as compared to development work that creates new software. Consequently, maintenance revolves around understanding existing software and maintainers spend most of their time trying to understand the software they have to modify. Understanding the software involves understanding not only the code but also the related documents. During the modification of the software, the effects of the change have to be clearly understood by the maintainer because introducing undesired side effects in the system during modification is easy. To test whether those aspects of the system that are not supposed to be modified are operating as they were before modification, *regression testing* is done. Regression testing involves executing old test cases to test that no new errors have been introduced.

Thus, maintenance involves understanding the existing software (code and related documents), understanding the effects of change, making the changes—to both the code and the documents—testing the new parts, and retesting the old parts that were not changed. Because often during development, the needs of the maintainers are not kept in mind, few support documents are produced during development to help the maintainer. The complexity of the maintenance task, coupled with the neglect of maintenance concerns during development, makes maintenance the most costly activity in the life of software product.

Maintenance is one form of change that typically is done after the software development is completed and the software has been deployed. However, there are other forms of changes that lead to rework during the software development itself.

One of the biggest problems in software development, particularly for large and complex systems, is that what is desired from the software (i.e., the requirements) is not understood. To completely specify the requirements, *all* the functionality, interfaces, and constraints have to be specified before software development has commenced! In other words, for specifying the requirements, the clients and the developers have to *visualize* what the software behavior should be once it is developed. This is very hard

to do, particularly for large and complex systems. So, what generally happens is that the development proceeds when it is believed that the requirements are generally in good shape. However, as time goes by and the understanding of the system improves, the clients frequently discover additional requirements they had not specified earlier. This leads to requirements getting changed. This change leads to *rework*; the requirements, the design, the code all have to be changed to accommodate the new or changed requirements.

Just uncovering requirements that were not understood earlier is not the only reason for this change and rework. Software development of large and complex systems can take a few years. And with the passage of time, the needs of the clients change. After all, the current needs, which initiate the software product, are a reflection of current times. As times change, so do the needs. And, obviously, the clients want the system deployed to satisfy their most current needs. This change of needs while the development is going on also leads to rework.

In fact, changing requirements and associated rework are a major problem of the software industry. It is estimated that rework costs are 30 to 40% of the development cost [22]. In other words, of the total development effort, rework due to various changes consume about 30 to 40% of the effort! No wonder change and rework is a major contributor to the software crisis. However, unlike the issues discussed earlier, the problem of rework and change is not just a reflection of the state of software development, as changes are frequently initiated by clients as their needs change.

1.2 The Software Engineering Challenges

Now we have a better understanding of the problem domain that software engineering deals with, let us orient our discussion to Software Engineering itself. *Software engineering* is defined as the systematic approach to the development, operation, maintenance, and retirement of software [91]. In this book we will primarily focus on development.

The use of the term *systematic approach* for the development of software implies that methodologies are used for developing software which are repeatable. That is, if the methodologies are applied by different groups of people, similar software will be produced. In essence, the goal of software engineering is to take software development closer to science and engineering and away from ad-hoc approaches for development whose outcomes are not predictable but which have been used heavily in the past and still continue to be used for developing software.

As mentioned, industrial strength software is meant to solve some problem of the client. (We use the term client in a very general sense meaning the people whose needs are to be satisfied by the software.) The problem therefore is to (systematically) develop software to satisfy the needs of some users or clients. This fundamental problem that software engineering deals with is shown in Figure 1.

Though the basic problem is to systematically develop software to satisfy the client, there are some factors which affect the approaches selected to solve the problem. These

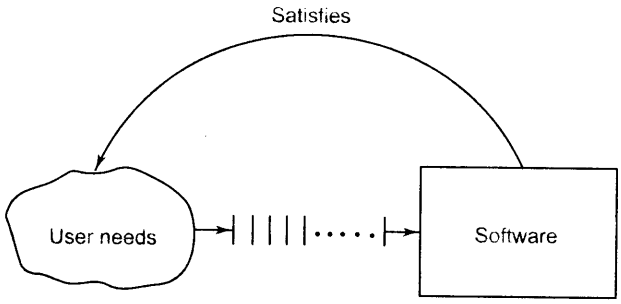

Figure 1.2: Basic problem.

factors are the primary forces that drive the progress and development in the field of software engineering. We consider these as the primary challenges for software engineering and discuss some of the key ones here.

1.2.1 Scale

A fundamental factor that software engineering must deal with is the issue of scale; development of a very large system requires a very different set of methods compared to developing a small system. In other words, the methods that are used for developing small systems generally *do not scale up* to large systems. An example will illustrate this point. Consider the problem of counting people in a room versus taking a census of a country. Both are essentially counting problems. But the methods used for counting people in a room (probably just go row-wise or column-wise) will just not work when taking a census. 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 lines needs to be developed. A different set of methods must be used for developing large software. Any large project involves the use of engineering and project management. For software projects, by engineering we mean the methods, procedures, and tools that are used. In small projects, informal methods for development and management can be used. However, for large projects, both have to be much more formal, as shown in Figure 1.

As shown in the figure, when dealing with a small software project, the engineering capability required is low (all you need to know is how to program and a bit of testing) and the project management requirement is also low. However, when the scale changes to large, to solve such problems properly, it is essential that we move in both directions—the engineering methods used for development need to be more formal, and the project

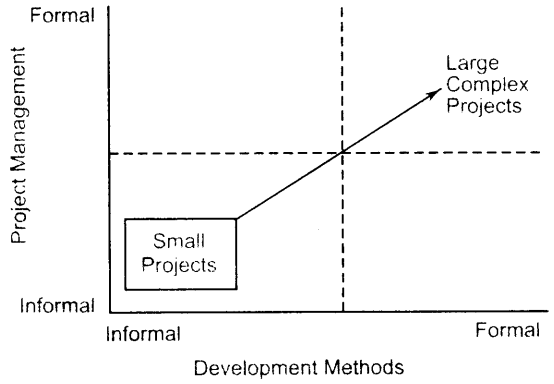

Figure 1.3: The problem of scale.

management for the development project also needs to be more formal. For example, if we leave 50 bright programmers together (who know how to develop small programs well) without formal management and development procedures and ask them to develop an on-line inventory control system for an automotive manufacturer, it is highly unlikely that they will produce anything of use. To successfully execute the project, a proper method for engineering the system has to be used and the project has to be tightly managed to make sure that methods are indeed being followed and that cost, schedule, and quality are under control.

There is no universally acceptable definition of what is a "small" project and what is a "large" project, and the scales are clearly changing with time. However, informally, we can use the order of magnitudes and say that a project is *small* if its size is less than 10 KLOC, *medium* if the size is less than 100 KLOC (and more than 10), *large* if the size is less than one million LOC, and *very large* if the size is many million LOC. To get an idea of the sizes of some real software products, the approximate sizes of some well known products is given in Table 1.1.

1.2.2 Quality and Productivity

An engineering discipline, almost by definition, is driven by practical parameters of cost, schedule, and quality. A solution that takes enormous resources and many years may not be acceptable. Similarly, a poor-quality solution, even at low cost, may not be of much use. Like all engineering disciplines, software engineering is driven by the three major factors: cost, schedule, and quality.

The cost of developing a system is the cost of the resources used for the system, which, in the case of software, is dominated by the manpower cost, as development

Size (KLOC)	**Software**	**Languages**
980	gcc	ansic, cpp, yacc
320	perl	perl, ansic, sh
305	teTeX	ansic, perl
200	openssl	ansic, cpp, perl
200	Python	python, ansic
100	apache	ansic, sh
90	cvs	ansic, sh
65	sendmail	ansic
60	xfig	ansic
45	gnuplot	ansic, lisp
38	openssh	ansic
30,000	Red Hat Linux	ansic, cpp
40,000	Windows XP	ansic, cpp

Table 1.1: Size in KLOC of some well known products.

is largely labor-intensive. Hence, the cost of a software project is often measured in terms of person-months, i.e., the cost is considered to be the total number of person-months spent in the project. (Person-months can be converted into a dollar amount by multiplying it with the average dollar cost, including the cost of overheads like hardware and tools, of one person-month.)

Schedule is an 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.

Productivity in terms of output (KLOC) per person-month can adequately capture both cost and schedule concerns. If productivity is higher, it should be clear that the cost in terms of person-months will be lower (the same work can now be done with fewer person-months.) Similarly, if productivity is higher, the potential of developing the software in shorter time improves—a team of higher productivity will finish a job in lesser time than a same-size team with lower productivity. (The actual time the project will take, of course, depends also on the number of people allocated to the project.) In other words, productivity is a key driving factor in all businesses and desire for high productivity dictates, to a large extent, how things are done.

The other major factor driving any production discipline is quality. Today, quality is a main mantra, and business strategies are designed around quality. 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 discussion. We use the international standard on software product quality as the basis of our discussion here [94].

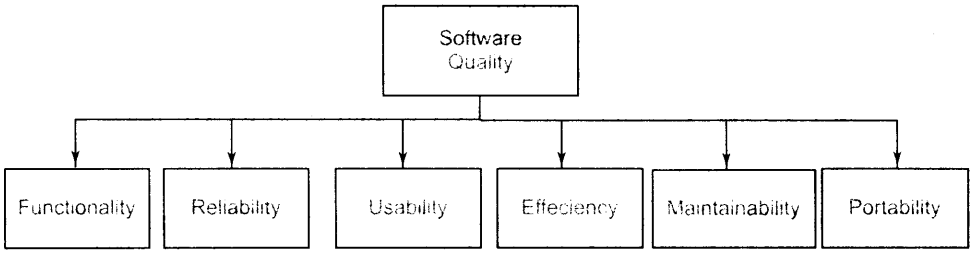

Figure 1.4: Software quality attributes.

According to the quality model adopted by this standard, software quality comprises of six main attributes (called characteristics) as shown in Figure 1 [94]. These six attributes have detailed characteristics which are considered the basic ones and which can and should be measured using suitable metrics. At the top level, for a software product, these attributes can be defined as follows [94]:

- **Functionality.** The capability to provide functions which meet stated and implied needs when the software is used
- **Reliability.** The capability to maintain a specified level of performance
- **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

The characteristics for the different attributes provide further details. Usability, for example, has characteristics of understandability, learnability, operability; maintainability has changeability, testability, stability, etc.; while portability has adaptability, installability, etc. Functionality includes suitability (whether appropriate set of functions are provided,) accuracy (the results are accurate,) and security. Note that in this

classification, security is considered a characteristic of functionality, and is defined as "the capability to protect information and data so that unauthorized persons or systems cannot read or modify them, and authorized persons or systems are not denied access to them."

There are two important consequences of having multiple dimensions to quality. First, software quality cannot be reduced to a single number (or a single parameter). And second, the concept of quality is project-specific. For an ultra-sensitive project, reliability may be of utmost importance but not usability, while in a commercial package for playing games on a PC, usability may be of utmost importance and not reliability. Hence, for each software development project, a quality objective must be specified before the development starts, and the goal of the development process should be to satisfy that quality objective.

Despite the fact that there are many quality factors, reliability is generally accepted to be the main quality criterion. As unreliability of software comes due to presence of defects in the software, one measure of quality is the number of defects in the delivered software per unit size (generally taken to be thousands of lines of code, or KLOC). With this as the major quality criterion, the quality objective is to reduce the number of defects per KLOC as much as possible. Current best practices in software engineering have been able to reduce the defect density to less than 1 defect per KLOC.

It should be pointed out that to use this definition of quality, what a defect is must be clearly defined. A defect could be some problem in the software that causes the software to crash or a problem that causes an output to be not properly aligned or one that misspells some word, etc. The exact definition of what is considered a defect will clearly depend on the project or the standards the organization developing the project uses (typically it is the latter).

1.2.3 Consistency and Repeatability

There have been many instances of high quality software being developed with very high productivity. But, there have been many more instances of software with poor quality or productivity being developed. A key challenge that software engineering faces is how to ensure that successful results can be repeated, and there can be some degree of consistency in quality and productivity.

We can say that an organization that develops one system with high quality and reasonable productivity, but is not able to maintain the quality and productivity levels for other projects, does not know good software engineering. A goal of software engineering methods is that system after system can be produced with high quality and productivity. That is, the methods that are being used are repeatable across projects leading to consistency in the quality of software produced.

An organization involved in software development not only wants high quality and productivity, but it wants these consistently. In other words, a software development

organization would like to produce consistent quality software with consistent productivity. Consistency of performance is an important factor for any organization; it allows an organization to predict the outcome of a project with reasonable accuracy, and to improve its processes to produce higher-quality products and to improve its productivity. Without consistency, even estimating cost for a project will become difficult.

Achieving consistency is an important problem that software engineering has to tackle. As can be imagined, this requirement of consistency will force some standardized procedures to be followed for developing software. There are no globally accepted methodologies and different organizations use different ones. However, within an organization, consistency is achieved by using its chosen methodologies in a consistent manner. Frameworks like ISO9001 and the Capability Maturity Model (CMM) encourage organizations to standardize methodologies, use them consistently, and improve them based on experience. We will discuss this issue a bit more in the next chapter.

1.2.4 Change

We have discussed above how maintenance and rework are very expensive and how they are an integral part of the problem domain that software engineering deals with. In todays world change in business is very rapid. As businesses change, they require that the software supporting to change. Overall, as the world changes faster, software has to change faster.

Rapid change has a special impact on software. As software is easy to change due to its lack of physical properties that may make changing harder, the expectation is much more from software for change.

Therefore, one challenge for software engineering is to accommodate and embrace change. As we will see, different approaches are used to handle change. But change is a major driver today for software engineering. Approaches that can produce high quality software at high productivity but cannot accept and accommodate change are of little use today—they can solve only very few problems that are change resistant.

1.3 The Software Engineering Approach

We now understand the problem domain and the basic factors that drive software engineering. We can view high quality and productivity (Q&P) as the basic objective which is to be achieved consistently for large scale problems and under the dynamics of changes. The Q&P achieved during a project will clearly depend on many factors, but the three main forces that govern Q&P are the people, processes, and technology, often called the Iron Triangle, as shown in Figure 1.

So, for high Q&P good technology has to be used, good processes or methods have to be used, and the people doing the job have to be properly trained. In software engineering, the focus is primarily on processes, which were referred to as systematic approach in the definition given earlier. As processes form the heart of software engineering (with

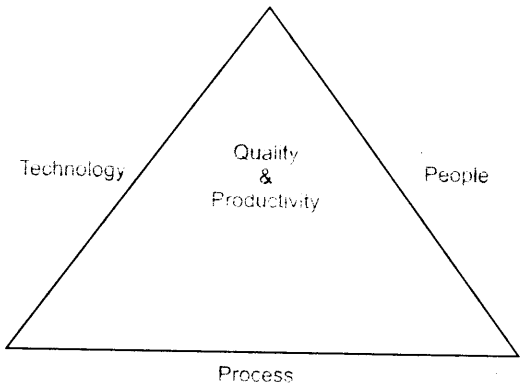

Figure 1.5: The iron triangle.

tools and technology providing support to efficiently execute the processes,) in this book we will focus primarily on processes. Process is what takes us from user needs to the software that satisfies the needs in Figure 1.

The basic approach of software engineering is to separate the process for developing software from the developed product (i.e., the software). The premise is that to a large degree the software process determines the quality of the product and productivity achieved. Hence to tackle the problem domain and successfully face the challenges that software engineering faces, one must focus on the software process. Design of proper software processes and their control then becomes a key goal of software engineering research. It is this focus on process that distinguishes Software Engineering from most other computing disciplines. Most other computing disciplines focus on some type of product—algorithms, operating systems, databases, etc.—while software engineering focuses on the process for producing the products. It is essentially the software equivalent of "manufacturing engineering." Though we will discuss more about processes in the next chapter, we briefly discuss two key aspects here—the development process and managing the development process.

1.3.1 Phased Development Process

A development process consists of various phases, each phase ending with a defined output. The phases are performed in an order specified by the process model being followed. The main reason for having a phased process is that it breaks the problem of developing software into successfully performing a set of phases, each handling a different concern of software development. This ensures that the cost of development is lower than what it would have been if the whole problem was tackled together. Furthermore, a phased process allows proper checking for quality and progress at some defined points

during the development (end of phases). Without this, one would have to wait until the end to see what software has been produced. Clearly, this will not work for large systems. Hence, for managing the complexity, project tracking, and quality, all the development processes consist of a set of phases. A phased development process is central to the software engineering approach for solving the software crisis.

Various process models have been proposed for developing software. In fact, most organizations that follow a process have their own version. We will discuss some of the common models in the next chapter. In general, however, we can say that any problem solving in software must consist of requirement specification for understanding and clearly stating the problem, design for deciding a plan for a solution, coding for implementing the planned solution, and testing for verifying the programs.

For small problems, these activities may not be done explicitly, the start and end boundaries of these activities may not be clearly defined, and no written record of the activities may be kept. However, systematic approaches require that each of these four problem solving activities be done formally. In fact, for large systems, each activity can itself be extremely complex, and methodologies and procedures are needed to perform them efficiently and correctly. Though different process models will perform these phases in different manner, they exist in all processes. We will discuss different process models in the next chapter. Here we briefly discuss these basic phases; each one of them will be discussed in more detail during the course of the book (there is at least one chapter for each of these phases).

Requirements Analysis

Requirements analysis is done in order to understand the problem the software system is to solve. The emphasis in requirements analysis is on identifying what is needed from the system, not how the system will achieve its goals. For complex systems, even determining what is needed is a difficult task. The goal of the requirements activity is to document the requirements in a *software requirements specification* document.

There are two major activities in this phase: problem understanding or analysis and requirement specification. In problem analysis, the aim is to understand the problem and its context, and the requirements of the new system that is to be developed. Understanding the requirements of a system that does not exist is difficult and requires creative thinking. The problem becomes more complex because an automated system offers possibilities that do not exist otherwise. Consequently, even the users may not really know the needs of the system.

Once the problem is analyzed and the essentials understood, the requirements must be specified in the requirement specification document. The requirements document must specify all functional and performance requirements; the formats of inputs and outputs; and all design constraints that exist due to political, economic, environmental, and security reasons. In other words, besides the functionality required from the system, all the factors that may effect the design and proper functioning of the system should

be specified in the requirements document. A preliminary user manual that describes all the major user interfaces frequently forms a part of the requirements document.

Software Design

The purpose of the design phase is to plan a solution of the problem specified by the requirements document. This phase is the first step in moving from the problem domain to the solution domain. In other words, starting with *what* is needed, design takes us toward *how* to satisfy the needs. The design of a system is perhaps the most critical factor affecting the quality of the software; it has a major impact on the later phases, particularly testing and maintenance.

The design activity often results in three separate outputs—*architecture design*, *high level design*, and *detailed design*. *Architecture* focuses on looking at a system as a combination of many different components, and how they interact with each other to produce the desired results. The *high level design* identifies the modules that should be built for developing the system and the specifications of these modules. At the end of system design all the major data structures, file formats, output formats, etc., are also fixed. In *detailed design*, the internal logic of each of the modules is specified.

In architecture the focus is on identifying components or subsystems and how they connect; in high level design the focus is on identifying the modules; and during detailed design the focus is on designing the logic for each of the modules. In other words, in architecture the focus is on what major components are needed, in high level design the attention is on *what* modules are needed, while in detailed design *how* the modules can be implemented in software is the issue. A *design methodology* is a systematic approach to creating a design by application of a set of techniques and guidelines. Most methodologies focus on high level design.

Coding

Once the design is complete, most of the major decisions about the system have been made. However, many of the details about coding the designs, which often depend on the programming language chosen, are not specified during design. The goal of the coding phase is to translate the design of the system into code in a given programming language. For a given design, the aim in this phase is to implement the design in the best possible manner.

The coding phase affects both testing and maintenance profoundly. Well-written code can reduce the testing and maintenance effort. Because the testing and maintenance costs of software are much higher than the coding cost, the goal of coding should be to reduce the testing and maintenance effort. Hence, during coding the focus should be on developing programs that are easy to read and understand, and not simply on developing programs that are easy to write. Simplicity and clarity should be strived for during the coding phase.

Testing

Testing is the major quality control measure used during software development. Its basic function is to detect defects in the software. During requirements analysis and design, the output is a document that is usually textual and nonexecutable. After coding, computer programs are available that can be executed for testing purposes. This implies that testing not only has to uncover errors introduced during coding, but also errors introduced during the previous phases. Thus, the goal of testing is to uncover requirement, design, and coding errors in the programs.

The starting point of testing is *unit testing*, where the different modules or components are tested individually. As modules are integrated into the system, *integration testing* is performed, which focuses on testing the interconnection between modules. After the system is put together, *system testing* is performed. Here the system is tested against the system requirements to see if all the requirements are met and if the system performs as specified by the requirements. Finally, *acceptance testing* is performed to demonstrate to the client, on the real-life data of the client, the operation of the system.

Testing is an extremely critical and time-consuming activity. It requires proper planning of the overall testing process. Frequently the testing process starts with a *test plan* that identifies all the testing-related activities that must be performed and specifies the schedule, allocates the resources, and specifies guidelines for testing. The test plan specifies conditions that should be tested, different units to be tested, and the manner in which the modules will be integrated. Then for different test units, a *test case specification document* is produced, which lists all the different test cases, together with the expected outputs. During the testing of the unit, the specified test cases are executed and the actual result compared with the expected output. The final output of the testing phase is the *test report* and the *error report*, or a set of such reports. Each test report contains the set of test cases and the result of executing the code with these test cases. The error report describes the errors encountered and the action taken to remove the errors.

1.3.2 Managing the Process

As stated earlier, a phased development process is central to the software engineering approach. However, a development process does not specify how to allocate resources to the different activities in the process. Nor does it specify things like schedule for the activities, how to divide work within a phase, how to ensure that each phase is being done properly, or what the risks for the project are and how to mitigate them. Without properly managing these issues relating to the process, it is unlikely that the cost and quality objectives can be met. These issues relating to managing the development process of a project are handled through project management.

The management activities typically revolve around a *plan*. A software plan forms the baseline that is heavily used for monitoring and controlling the development process

of the project. This makes planning the most important project management activity in a project. It can be safely said that without proper project planning a software project is very unlikely to meet its objectives. We will devote a complete chapter to project planning.

Managing a process requires information upon which the management decisions are based. Otherwise, even the essential questions—is the schedule in a project is being met, what is the extent of cost overrun, are quality objectives being met,—cannot be answered. And information that is subjective is only marginally better than no information (e.g., Q: how close are you to finishing? A: We are almost there.) Hence, for effectively managing a process, objective data is needed. For this, software metrics are used.

Software metrics are quantifiable measures that could be used to measure different characteristics of a software system or the software development process. There are two types of metrics used for software development: *product metrics* and *process metrics.*

Product metrics are used to quantify characteristics of the product being developed, i.e., the software. *Process metrics* are used to quantify characteristics of the process being used to develop the software. Process metrics aim to measure such considerations as productivity, cost and resource requirements, effectiveness of quality assurance measures, and the effect of development techniques and tools

Metrics and measurement are necessary aspects of managing a software development project. For effective monitoring, the management needs to get information about the project: how far it has progressed, how much development has taken place, how far behind schedule it is, and the quality of the development so far. Based on this information, decisions can be made about the project. Without proper metrics to quantify the required information, subjective opinion would have to be used, which is often unreliable and goes against the fundamental goals of engineering. Hence, we can say that metrics-based management is also a key component in the software engineering strategy to achieve its objectives.

Though we have focused on managing the development process of a project, there are other aspects of managing a software process. Some of these will be discussed in the next chapter.

1.4 Summary

Software cost now forms the major component of a computer system's cost. Software is currently extremely expensive to develop and is often unreliable. In this chapter, we have discussed a few themes regarding software and software engineering:

1. The problem domain for software engineering is industrial strength software.

2. This software is not just a set of computer programs but comprises programs and associated data and documentation. Industrial strength software is expensive and difficult to build, expensive to maintain due to changes and rework, and has high quality requirements.

3. Software engineering is the discipline that aims to provide methods and procedures for systematically developing industrial strength software. The main driving forces for software engineering are the problem of scale, quality and productivity (Q&P), consistency, and change. Achieving high Q&P consistently for problems whose scale may be large and where changes may happen continuously is the main challenge of software engineering.

4. The fundamental approach of software engineering to achieve the objectives is to separate the development process from the products. Software engineering focuses on process since the quality of products developed and the productivity achieved are heavily influenced by the process used. To meet the software engineering challenges, this development process is a phased process. Another key approach used in Software Engineering for achieving high Q&P is to manage the process effectively and proactively using metrics.

Exercises

1. Suppose a program for solving a problem costs C, and an industrial strength software for solving that problem costs 10C. Where do you think this extra 9C cost is spent? Suggest a possible breakdown of this extra cost.

2. If the primary goal is to make software maintainable, list some of the things you *will* do and some of the things you *will not* do during coding and testing.

3. List some problems that will come up if the methods you currently use for developing small software are used for developing large software systems.

4. Next time you do a programming project (in some course perhaps), determine the productivity you achieve. For this, you will have to record the effort you spent in the work. How does it compare with the illustrative productivity figures given in the Chapter.

5. Next time you do a programming project, try to predict the time you will take to do it in terms of hours as well as days. Then in the end, check how well your actual schedule matched the predicted one.

6. We have said that a commonly used measure for quality is defects per KLOC in delivered software. For a software product, how can its quality be measured? How can it be estimated before delivering the software?

7. **If you are given extra time to improve the reliability of the final product developing a software product, where would you spend this extra time?**

8. Suggest some ways to detect software errors in the early phases of the project when code is not yet available.

9. **How does a phased process help in achieving high Q&P, when it seems that we are doing more tasks in a phased process as compared to an ad-hoc approach?**

10. If absolutely no metrics are used, can you manage, or even define, a project? What is the bare minimum set of metrics that you must use for a development project?

Chapter 2

Software Processes

As we saw in the previous chapter, the concept of process is at the heart of the software engineering approach. According to Webster, the term *process* means "a particular method of doing something, generally involving a number of steps or operations." In software engineering, the phrase *software process* refers to the methods of developing software.

A software process is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced. The basic desired result is, as stated earlier, high quality and productivity. In this chapter, we will discuss the concept of software processes further, the component processes of a software process, and some models that have been proposed.

2.1 Software Process

In an organization whose major business is software development, there are typically many processes executing simultaneously. Many of these do not concern software engineering, though they do impact software development. These could be considered nonsoftware engineering process. Business processes, social processes, and training processes, are all examples of processes that come under this. These processes also affect the software development activity but are beyond the purview of software engineering.

The process that deals with the technical and management issues of software development is called a *software process.* Clearly, many different types of activities need to be performed to develop software. All these activities together comprise the software process. As different type of activities are being performed, which are frequently done by different people, it is better to view the software process as consisting of many component processes, each consisting of a certain type of activity. Each of these component

processes typically has a different objective, though they obviously cooperate with each other to satisfy the overall software engineering objective. In this section, we will define the major component processes of a software process and what their objectives are.

2.1.1 Processes and Process Models

A successful project is the one that satisfies the expectations on all the three goals of cost, schedule, and quality (we are including functionality or features as part of quality.) Consequently, when planning and executing a software project, the decisions are mostly taken with a view to ultimately reduce the cost or the cycle time, or for improving the quality. Software projects utilize a process to organize the execution of tasks to achieve the goals on the cost, schedule, and quality fronts.

A project's process specification defines the tasks the project should perform, and the order in which they should be done. The actual process exists when the project is actually executed. Although process specification is distinct from the actual process, we will consider the process specification for a project and the actual process of the project as one and the same, and will use the term process to refer to both of them. It should, however, be mentioned that although we are assuming that there is no difficulty in a project following a specified process, in reality it is not as simple. Often the actual process being followed in the project may be very different from the project's process specification. Reasons for this divergence vary from laziness to lack of appreciation of importance of process to "old habits die hard." Ensuring that the project is following the process it planned for itself is an important issue for organizations in the business of executing projects, and there are different ways to deal with it—we will not discuss this issue in this book.

A process model specifies a general process, usually as a set of stages in which a project should be divided, the order in which the stages should be executed, and any other constraints and conditions on the execution of stages. The basic premise behind a process model is that, in the situations for which the model is applicable, using the process model as the projects process will lead to low cost, high quality, or reduced cycle time. 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 generic guidelines for developing a suitable process for a project.

A project's process may utilize some process model. That is, the project's process has a general resemblance to the process model with the actual tasks being specific to the project. However, using a process model is not simply translating the tasks in the process model to tasks in the project. Typically, to achieve the project's objectives, a project will require a process that is somewhat different from the process model. That is, the project's process is generally a tailored version of a general process model. How the process model has to be tailored for a particular project, of course, depends on the project characteristics. What we need to understand is that a project's process may be obtained from a process model, by tailoring the process model to suit the project needs.

For organizations that use standard processes, tailoring is an important issue. We will not discuss it further—the reader can find more about tailoring in [96].

When a process is executed on a project, software products are produced, one of them being the final software. That is, a process specifies the steps, the project executes these steps, and during the course of execution products are produced. A process limits the degrees of freedom for a project by specifying what types of activities must be undertaken and in what order, such that the "shortest" (or the most efficient) path is obtained from the user needs to the software satisfying these needs. It should be clear that it is the process that drives a project and heavily influences the expected outcomes of a project. Due to this, the focus of software engineering lies heavily on the process.

2.1.2 Component Software Processes

We have mentioned that the development process is the central process which specifies the tasks to be done in a project. Planning and scheduling the tasks and monitoring their execution fall in the domain of project management process. Hence, there are clearly two major components in a software process—a development process and a project management process—corresponding to the two axes in Figure 1. The development process specifies the development and quality assurance 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.

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. To ensure that the software being produced uses the correct versions of these items requires suitable processes to control the evolution of these items. 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. Sometimes, changes in requirements may be handled separately by a *requirements change management process.*

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.*

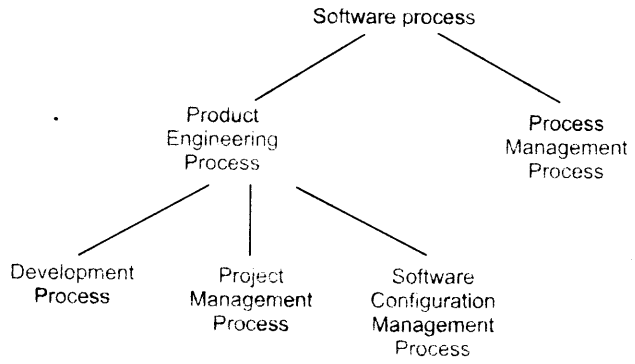

Figure 2.1: Software processes.

The relationship between these major component processes is shown in Figure 2.1. 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).*

Later in the chapter we will briefly discuss each of these processes, as well as the inspection process which is used for quality control of various work products. In the rest of the book, however, we will focus primarily on processes relating to product engineering, as process management is an advanced topic beyond the scope of this book. Much of the book discusses the different phases of a development process and the 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.

2.1.3 ETVX Approach for Process Specification

A process has a set of phases (or steps), each phase performing a well-defined task which leads a project towards satisfaction of its goals. To reduce the cost, a process should aim to detect defects in the phase in which they are introduced. This requires that

there be some verification at the end of each step, which in turn requires that there is a clearly defined output of a phase, which can be verified by some means. In other words, it is not acceptable to say that the output of a phase is an idea or a thought in the mind of someone; the output must be a formal and tangible entity. Such outputs of a development process, which are not the final output, are frequently called the *work products*. In software, a work product can be the requirements document, design document, code, prototype, and the like.

This restriction that the output of each step be some work product that can be verified suggests that the process should have a small number of steps. Having too many steps results in too many work products or documents. Due to this, at the top level, a process typically consists of a few steps, each satisfying a clear objective and producing a document which can be verified. How to perform the activity of the particular step or phase is generally addressed by *methodologies* for that activity. We will discuss various methodologies for different activities throughout the book.

As a process typically contains a sequence of steps, the next issue to address is when a phase should be initiated and terminated. This is frequently done by specifying the entry criteria and exit criteria for a phase. The *entry criteria* of a phase specifies the conditions that the input to the phase should satisfy to initiate the activities of that phase. The *exit criteria* specifies the conditions that the work product of this phase should satisfy to terminate the activities of the phase. The entry and exit criteria specify constraints of when to start and stop an activity. It should be clear that the entry criteria of a phase should be consistent with the exit criteria of the previous phase. In addition to the entry and exit criteria, the inputs and outputs of a step also need to be clearly specified. As errors can be introduced in every stage, a stage should end with some verification of its activities, and these should also be clearly stated. The specification of a step with its input, output, and entry and exit criteria is shown in Figure 2.2. This approach for process specification is called the ETVX (Entry criteria, Task, Verification, and eXit criteria) approach [128].

Besides the entry and exit criteria for the input and output, a step needs to produce some information to aid proper management of the process. This requires that a step produce some information that provides visibility into the state of the process. This information can then be used to take suitable actions, where necessary, to keep the process under control. The flow of information from a step and exercise of control is also shown in Figure 2.2.

2.2 Desired Characteristics of Software Process

We have not yet specified any process. Is any process suitable to use? Here we discuss some of the desirable characteristics of the software process. As a process may be used by many projects, it needs characteristics beyond satisfying the project goals. We will discuss some of the important ones in this section.

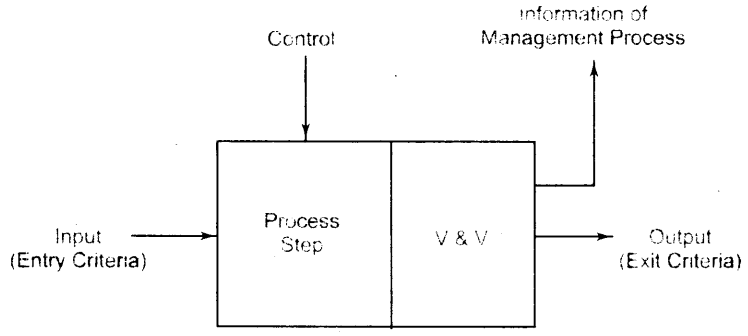

Figure 2.2: A step in a development process.

2.2.1 Predictability

Predictability of a process determines how accurately the outcome of following that process in a project can be predicted before the project is completed. Predictability can be considered a fundamental property of any process. In fact, if a process is not predictable, it is of limited use. Let us see why.

One way of estimating cost could be to say, "this project A is very similar to the project B that we did 2 years ago, hence A's cost will be very close to B's cost." However, even this simple method implies that the process that will be used to develop project A will be same as the process used for project B, *and* that following the process the second time will produce similar results as the first time. That is, this assumes that the process is *predictable.* If it was not predictable, then there is no guarantee that doing a similar project using the process will incur a similar cost.

The situation with quality is similar. The fundamental basis for quality prediction is that quality of the product is determined largely by the process used to develop it. Using this basis, quality of the product of a project can be estimated or predicted by seeing the quality of the products that have been produced in the past by the process being used in the current project. In fact, effective management of quality control activities largely depends on the predictability of the process. For example, for effective quality control, one method is to estimate what types and quantity of errors will be detected at what stage of the development, and then use them to determine if the quality assurance activities are being performed properly. This can only be done if the process is predictable; based on the past experience of such a process one can estimate the distribution of errors for the current project. Otherwise, how can anyone say whether detecting 10 errors per 100 lines of code (LOC) during testing in the current project is "acceptable"? With a predictable process, if the process is such that one expects around 10 errors per 100 LOC during testing, this means that the testing of this project

was probably done properly. But, if past experience with the process shows that about 2 errors per 100 LOC are detected during testing, then a careful look at the testing of the current project is necessary.

It should be clear that if we want to use the past experience to control costs and ensure quality, we must use a process that is predictable. With low predictability, the experience gained through projects is of little value. A predictable process is also said to be *under statistical control* [89, 101]. A process is under statistical control if following the same process produces similar results—results will have some variation, but the variation is mostly due to random causes and not due to process issues. This is shown in Figure 2.3; the y-axis represents some property of interest (quality, productivity, etc.), and x-axis represents the projects. The dark line is the expected value of the property for this process. Statistical control implies that most of the times the property of interest will be within a bound around the expected value. (Control charts provide a formal approach for defining these bounds. For a discussion on control charts and how to define optimal bounds, the reader is referred to [101].)

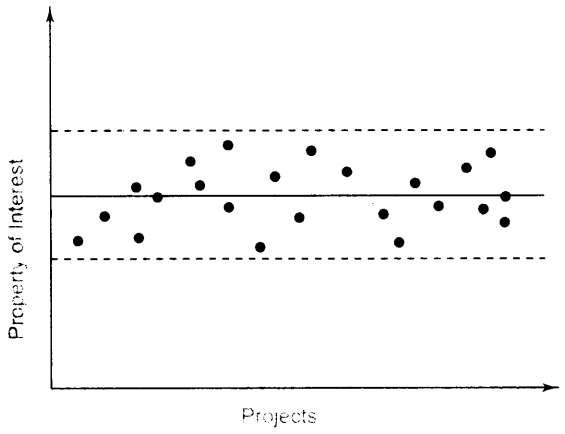

Figure 2.3: Process under statistical control

It should be clear that if one hopes to consistently develop software of high quality at low cost, it is necessary to have a process that is under statistical control. A predictable process is an essential requirement for ensuring good quality and low cost. Note that this does not mean that one can never produce high-quality software at low cost without following such a process. It is always possible that a set of bright people can do it. However, what this means is that without such a process, such things cannot be repeated. Hence, if one wants quality consistently across many projects, having a predictable process is essential. Because software engineering is interested in general

methods that can be used to develop different software, a predictable process forms the backbone of the software engineering methods.

2.2.2 Support Testability and Maintainability

We have already seen that in the life of software the maintenance costs generally exceed the development costs. Clearly, if we want to reduce the overall cost of software or achieve "global" optimality in terms of cost rather than "local" optimality in terms of development cost only, the goal of development should be to reduce the maintenance effort. That is, one of the important objectives of the development project should be to produce software that is easy to maintain. And the process used should ensure this maintainability.

Even in development, coding is frequently given a great degree of importance. We have seen that a process consists of phases, and a process generally includes requirements, design, coding, and testing phases. Of the development cost, an example distribution of effort with the different phases could be:

Requirements	10%
Design	10%
Coding	30%
Testing	50%

The exact numbers will differ with organization and the nature of the process. However, there are some observations we can make. First is that coding consumes only about a third of the development effort. This is against the common naive notion that developing software is largely concerned with writing programs and that programming is the major activity.

Another way of determining the effort spent in programming is to study how programmers spend their time in a software organization. A study conducted in Bell Labs to determine how programmers spend their time, as reported in [60], found the distribution shown below:

Writing programs	13%
Reading programs and manuals	16%
Job communication	32%
Other (including personal)	39%

This data clearly shows that programming is not the major activity on which programmers spend their time. Even if we take away the time spent in "other" activities, the time spent by a programmer writing programs is still less than 25% of the remaining time. In the study reported by Boehm [20], it was found that programmers spend less than 20% of their time programming.

The second important observation from the data about effort distribution with phases is that testing consumes the most resources during development. This is, again,

contrary to the common practice, which considers testing a side activity that is often not properly planned. Underestimating the testing effort often causes the planners to allocate insufficient resources for testing, which, in turn, results in unreliable software or schedule slippage.

Overall, we can say that the goal of the process should not be to reduce the effort of design and coding, but to reduce the cost of testing and maintenance. Both testing and maintenance depend heavily on the quality of design and code, and these costs can be considerably reduced if the software is designed and coded to make testing and maintenance easier. Hence, during the early phases of the development process the prime issues should be "can it be easily tested" and "can it be easily modified".

2.2.3 Support Change

Software changes for a variety of reasons. In Chapter 1, we emphasized the pervasiveness of change as a basic property of the problem domain. Here we focus on changes due to requirement changes. Though changes were always a part of life, change in today's world is much more and much faster. As organizations and businesses change, the software supporting the business has to change. Hence, any model that builds software and makes change very hard will not be suitable in many situations.

Besides changing an existing and working software, which one can argue is beyond the development process, change also takes place while development is going on. After all, the needs of the customer may change during the course of the project. And if the project is of any significant duration, considerable changes can be expected.

Besides the change driven by business need, changes may occur simply because people may change their minds as they think more about possibilities and alternatives. So, some part of a software system may be developed and shown to the users, and the users or customer is likely to use the feedback to find that what he had requested was not correct, or that he needs more, or that he needs something different. In other words, change is prevalent, and a process that can handle change easily is desirable.

2.2.4 Early Defect Removal

The notion that programming is the central activity during software development is largely due to programming being considered a difficult task and sometimes an "art." Another consequence of this kind of thinking is the belief that errors largely occur during programming, as it is the hardest activity in software development and offers many opportunities for committing errors. It is now clear that errors can occur at any stage during development. An example distribution of error occurrences by phase is:

Requirements	20%
Design	30%
Coding	50%

As we can see, errors occur throughout the development process. However, the cost of correcting errors of different phases is not the same and depends on when the error is detected and corrected. The relative cost of correcting requirement errors as a function of where they are detected is shown in Figure 2.4 [20].

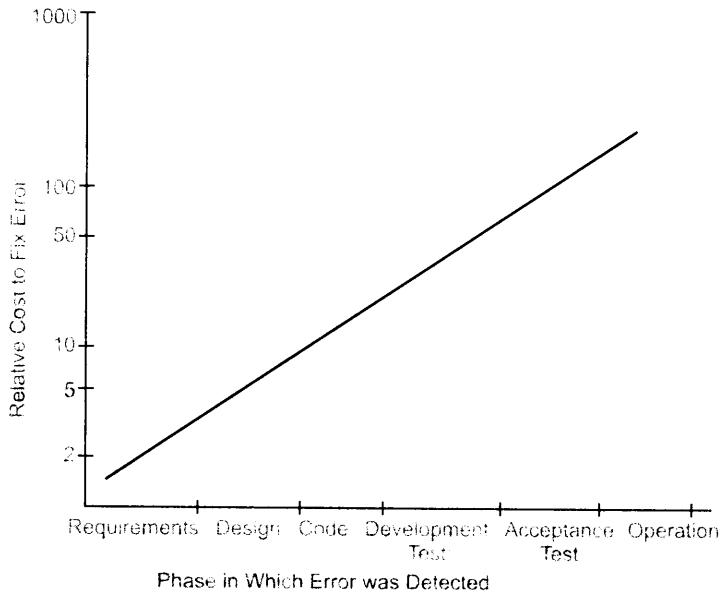

Figure 2.4: Cost of correcting errors.

As one would expect, the greater the delay in detecting an error after it occurs, the more expensive it is to correct it. As the figure shows, an error that occurs during the requirements phase, if corrected during acceptance testing, can cost up to 100 times more than correcting the error during the requirements phase itself.

The reason for this is fairly obvious. If there is an error in the requirements, then the design and the code will be affected by it. To correct the error after the coding is done would require both the design and the code to be changed, thereby increasing the cost of correction.

The main point of this discussion is that we should attempt to detect errors that occur in a phase during that phase itself and should not wait until testing to detect errors. Error detection and correction should be a continuous process that is done throughout software development. In terms of development phases, this means that we should try to verify the output of each phase before starting with the next (that is why the ETVX model has a V!) In other words, a process should have quality control activities spread through the process and in each phase. A quality control (QC) activity is one whose main purpose is to identify and remove defects.

Having QC tasks through the development is clearly an objective that should be supported by the process. However, it is even better to provide support for *defect prevention.* It is generally agreed that all the QC techniques that exist today are limited in their capability and cannot detect all the defects that are introduced. (Why else are there bugs in most software that is released that are then fixed in later versions?) Clearly, then, to reduce the total number of residual defects that exist in a system at the time of delivery and to reduce the cost of defect removal, an obvious approach is to prevent defects from being introduced. This requires that the process of performing the activities should be such that fewer defects are introduced. The method generally followed to support defect prevention is to use the development process to learn (from previous projects) so that the methods of performing activities can be improved. We will discuss this more in a later chapter.

2.2.5 Process Improvement and Feedback

As mentioned earlier, a process is not a static entity. Improving the quality and reducing the cost of products are fundamental goals of any engineering discipline. In the context of software, as the productivity (and hence the cost of a project) and quality are determined largely by the process, to satisfy the objectives of quality improvement and cost reduction, the software process must be improved.

Having process improvement as a fundamental objective requires that the software process be a closed-loop process. That is, the process must be improved based on previous experiences, and each project done using the existing process must feed information back to facilitate this improvement. As stated earlier, this activity of analyzing and improving the process is largely done in the process management component of the software process. However, to support this activity, information from various other processes will have to flow to the process management process. In other words, to support this activity, other processes will also have to take an active part.

Process improvement is also an objective in a large project where feedback from the early parts of the project can be used to improve the execution of the rest of the project. This type of feedback is eminently suited when the iterative development process model is used—feedback from one iteration is used to improve the execution of later iterations.

2.3 Software Development Process Models

In the software development process we focus on the activities directly related to production of the software, for example, design, coding, and testing. As the development process specifies the major development and quality control activities that need to be performed in the project, the development process really forms the core of the software process. The management process is decided based on the development process. Due to the importance of the development process, various models have been proposed. In this section we will discuss some of the major models.

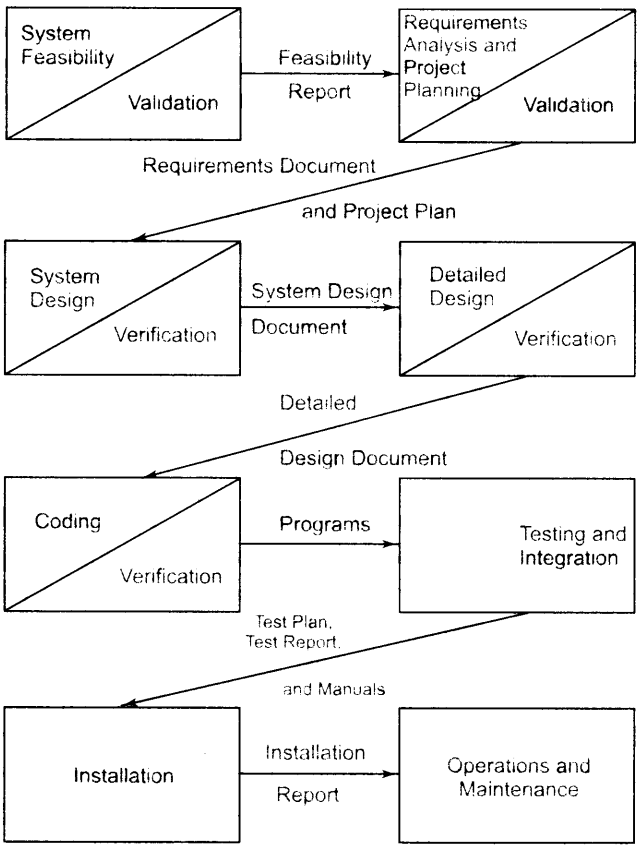

Figure 2.5: The waterfall model.

2.3.1 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 [132], 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 model is shown in Figure 2.5.

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.)

In addition to these work products, there are various other documents that are produced in a typical project. These include review reports, which are the outcome of reviews conducted for work products, as well as status reports that summarize the status of the project on a regular basis. Many other reports may be produced for improving the execution of the project or project reporting.

One of the main advantages of this 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. Some of the 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 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.

2.3.2 Prototyping

The goal of a prototyping-based development process is to counter the first two limitations 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, because the interactions with the prototype 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

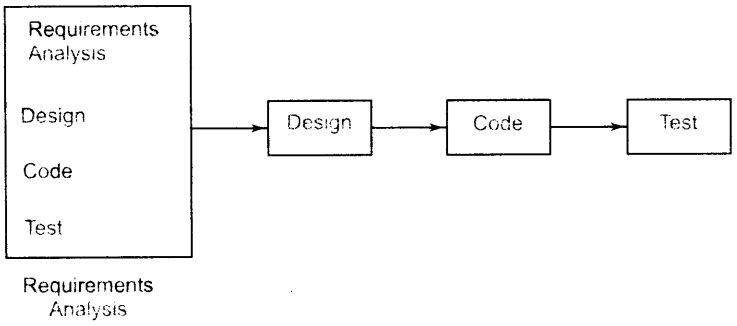

Figure 2.6: The prototyping model.

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 2.6.

A development process using throwaway prototyping typically proceeds as follows [72]. 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 the prototype and play with it. 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 prototypers 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.

For prototyping for the purposes of requirement analysis to be feasible, its cost must be kept low. Consequently, only those features are included in the prototype that will have a valuable return from the user experience. Exception handling, recovery, and conformance to some standards and formats are typically not included in prototypes.

In prototyping, as the prototype is to be discarded, there is no point in implementing those parts of the requirements that are already well understood. Hence, the focus of the development is to include those features that are not properly understood. And the development approach is "quick and dirty" with the focus on quick development rather than quality. Because the 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 type of cost-cutting methods, it is possible to keep the cost of the prototype less than a few percent of the total development cost.

Prototyping is often not used, as it is feared that development costs may become large. However, in some situations, the cost of software development without prototyping may be more than with prototyping. There are two major reasons for this. First, the experience of developing the prototype might reduce the cost of the later phases when the actual software development is done. Secondly, in many projects the requirements are constantly changing, particularly when development takes a long time. We saw earlier that changes in requirements at a late stage of development substantially increase the cost of the project. By elongating the requirements analysis phase (prototype development does take time), the requirements are "frozen" at a later time, by which time they are likely to be more developed and, consequently, more stable. In addition, because the client and users get experience with the system, it is more likely that the requirements specified after the prototype will be closer to the actual requirements. This again will lead to fewer changes in the requirements at a later time. Hence, the costs incurred due to changes in the requirements may be substantially reduced by prototyping. Hence, the cost of the development after the prototype can be substantially less than the cost without prototyping; we have already seen how the cost of developing the prototype itself can be reduced.

Prototyping is well suited for projects where requirements are hard to determine and the confidence in the stated requirements is low. In such projects, a waterfall model will have to freeze the requirements in order for the development to continue, even when the requirements are not stable. This leads to requirement changes and associated rework while the development is going on. Requirements frozen after experience with the prototype are likely to be more stable. Overall, in 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 an excellent technique for reducing some types of risks associated with a project. We will further discuss prototyping when we discuss requirements specification and risk management.

2.3.3 Iterative Development

The iterative development process model counters the third limitation 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. At each step, extensions and design modifications can be made. An advantage of this approach is that it can result in better testing because testing each increment is likely to be easier than testing the entire system as in the waterfall model. Furthermore, as in prototyping, the increments provide feedback to the client that is useful for determining the final requirements of the system.

The iterative enhancement model [7] 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 2.7.

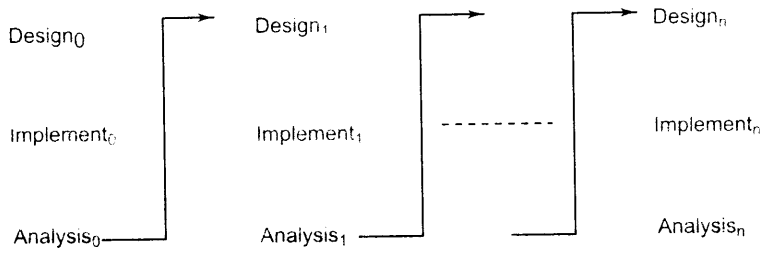

Figure 2.7: The iterative enhancement model.

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

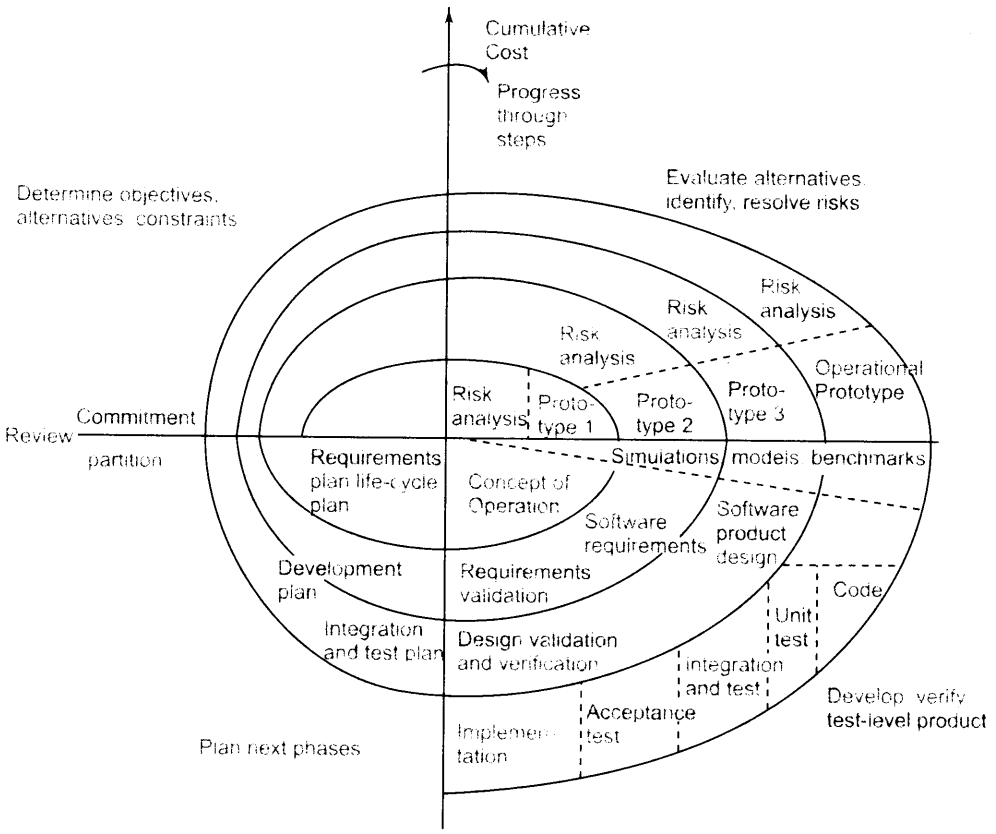

Figure 2.8: The spiral model.

chances of error and reduce the redesign work. The design and implementation phases of each step can be performed in a top-down manner or by using some other technique.

The spiral model is another iterative model that has been proposed [18]. As the name suggests, the activities in this model can be organized like a spiral that has many cycles as shown in Figure 2.8 [18].

Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exist. The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project. The next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping. Next, the software is developed, keeping in mind the risks. Finally the next stage is planned.

One effective use of the iterative model is often seen in product development, in which the developers themselves provide the specifications and therefore have a lot of control on which specifications go in the system and which stay out. Generally, a version of the product is released that contains some capability. Based on the feedback from users and experience with this version, technology changes, business changes, etc., a list of additional desirable features and capabilities is generated. These features form the basis of enhancement of the software, and are included in the next version. In other words, the first version contains some core capability. And then more features are added to later versions.

In a customized software development, where the client has to provide and approve the specifications, this process model is becoming extremely popular, despite some difficulties in using it in this context. The main reason is the same—as businesses are changing very 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. Furthermore, customers do not want to invest too much for a long time 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.

The iterative approach to software development is now widely used. Many contemporary development approaches like extreme programming [10] and Agile approaches [38] consider iterative development as a basic strategy for developing software for current times. Rational Unified Process (RUP) [108] also employs an iterative process.

2.3.4 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. However, to support parallel execution, each iteration has to be structured properly and teams have to be organized suitably. The timeboxing model proposes an approach for these [100, 99].

In the timeboxing model, 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. Time-boxing changes the perspective of development and makes the schedule a non-negotiable 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.) Let us consider a time box with duration T and consisting of n stages—S_1, S_2, ..., S_n, each stage S_i being executed by a dedicated team. The team of each stage has T/n time available to finish their task for a time box, that is, the duration of each stage is T/n. When the team of a stage i completes the tasks for that stage for a time box k, it then passes the output of the time box to the team executing the stage $i+1$, and then starts executing its stage for the next time box $k+1$. Using the output given by the team for S_i, the team for S_{i+1} starts its activity for this time box. By the time the first time box is nearing completion, there are $n-1$ different time boxes in different stages of execution. And though the first output comes after time T, each subsequent delivery happens after T/n time interval, delivering software that has been developed in time T.

As an example, 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 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 requirement team has finished requirements for timebox-1, the requirements are given to the build team for building the software. The requirement team then goes on and starts preparing the requirements for timebox-2. When the build for the 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 2.9 [99].

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

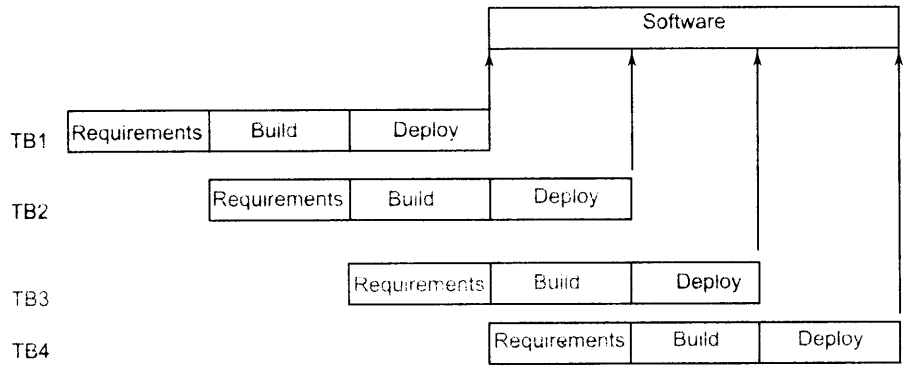

Figure 2.9: Executing the timeboxing process model.

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 will be made 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 team-wise activity for the 3-stage pipeline discussed above is shown in Figure 2.10 [99].

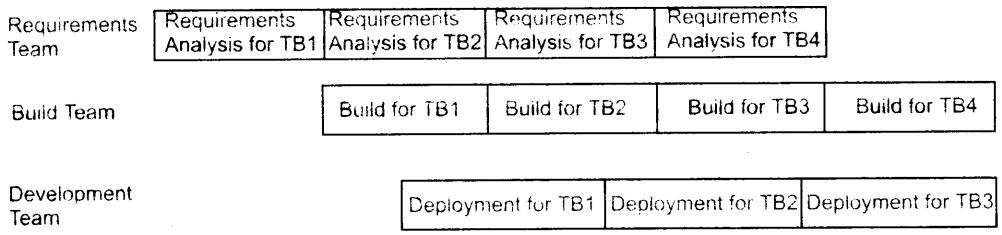

Figure 2.10: Tasks of different teams.

It should be clear that the duration of each iteration has not been reduced. 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 same effort and time is 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 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.

For example, consider an iterative development with three stages, as discussed above. Suppose that it takes 2 people 2 weeks to do the requirements for an iteration, it takes 4 people 2 weeks to do the build for the iteration, and it takes 3 people 2 weeks to test and deploy. If the iterations are serially executed, then the team for the project will be 4 people (the maximum size needed for a stage)—in the first 2 weeks two people will primarily do the requirements, then all the 4 people will do the task of build, and then 3 people will do the deployment.

If this project is executed using the timeboxing process model, there will be 3 separate teams—the requirements team of size 2, the build team of size 4, and the deployment team of size 3. So, the total team size for the project is (2+4+3) = 9 persons. This is more than twice the peak team size if iterations are executed serially. It is due to this increase in team size that the throughput increases and the average delivery time decreases.

Hence, the timeboxing provides an approach for utilizing additional manpower 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 substantially 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. These features should be such that there is some flexibility in grouping them for building a meaningful system in an iteration that provides value to the users.

The model is not suitable for projects where it is difficult to partition the overall development into multiple iterations of approximately equal duration. It is also not suitable for projects where different iterations may require different stages, and for projects whose features are such that there is no flexibility to combine them into meaningful deliveries. We have only discussed the basic process model and have not discussed the impact of unequal stages, exceptions on the execution of this model, project management issues, etc. For further details about the model, as well as a detailed example of applying the model on a real commercial project, the reader is referred to [100, 99].

2.3.5 Comparision of Models

As discussed earlier, each process model is suitable for some context, and the main reason for studying different models is to develop the ability to choose the proper model for a given project. Using a model as the basis, the actual process for the project can

be decided, which hopefully is the optimal process for the project. To help select a model, we summarize the strengths and weaknesses of the different models, along with the types of projects for which they are suitable, in Figure 2.11.

Strengths	Weaknesses	Types of projects
Waterfall		
Simple Easy to execute Intuitive and logical	All or nothing approach Requirements frozen early Disallows changes Cycle time too long May choose outdated hardware technology User feedback not allowed Encourages req. bloating	For well understood problems, short duration project, automation of existing manual systems
Prototyping		
Helps in requirements elicitation Reduces risk Leads to a better system	Front heavy process Possibly higher cost Disallows later changes	Systems with novice users When uncertainities in requirements When UI very important
Iterative		
Regular/quick deliveries Reduces risk Accommodates changes Allows user feedback Allows reasonable exit points Avoids req. bloating Prioritizes requirements	Each iteration can have planning overhead Cost may increase as work done in one iteration may have to be undone later System architecture and structure may suffer as frequent changes are made	For businesses where time is of essence Where risk of a long project cannot be taken Where requirements are not known and will be known only with time
Timeboxing		
All strengths of iterative Planning and negotiations somewhat easier Very short delivery cycle	Project management is complex Possibly increased cost Large team size	Where very short delivery times needed Flexibility in grouping features exists

Figure 2.11: Comparison of process models.

2.4 Other Software Processes

Though the development process is the central process in software processes, other processes are needed to properly execute the development process and to achieve the desired characteristics of software processes. There are processes for each of the activities in the development process, e.g., design process, testing process, etc. These processes are often called methodologies and we will discuss them in their respective chapters. Here we discuss those processes that span the entire project and are not particular to any task in the development process. We discuss some of the important processes that are involved when developing software.

2.4.1 Project Management Process

Proper management is an integral part of software development. A large software development project involves many people working for a long period of time. We have seen that a development process typically partitions the problem of developing software into a set of phases. 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, if needed. All these activities are part of the project management process.

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. The focus is on issues like planning a project, estimating resource and schedule, and monitoring and controlling the project. In other words, the basic task is to plan the detailed implementation of the process for the particular project and then ensure that the plan is followed. 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. As shown earlier in Figure 2.2, the implementation of a development process model should be such 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.

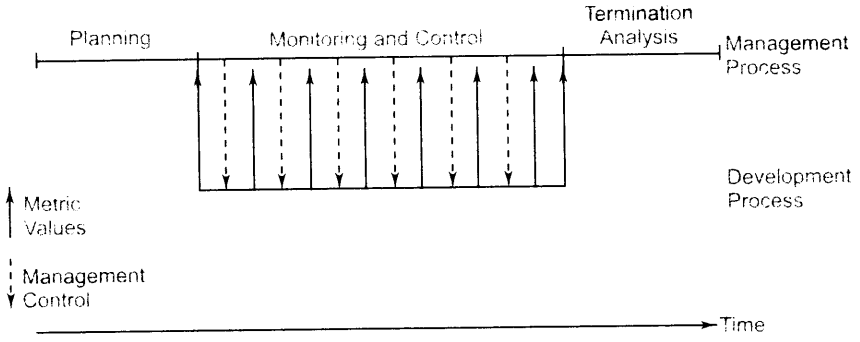

Figure 2.12: Temporal relationship between development and management process.

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 termination analysis is to provide information about the development process and learn from the project in order to improve the process. This phase is also often 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; for an example of a postmortem report the reader is referred to [96].

The temporal relationship between the management process and the development process is shown in Figure 2. This is an idealized relationship showing 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.

2.4.2 The Inspection Process

The main goal of the inspection process is to detect defects in work products. Software inspections were first proposed by Fagan [58, 59]. Earlier inspections were focused on code, but over the years its use has spread to other work products too. In other words, the inspection process is used throughout the development process. Software inspections are now a recognized industry best practice with considerable data to support that they help in improving quality and also improve productivity (e.g., see reports given in [70, 77, 144]). There are books on the topic which describe in great detail how inspections should be conducted [70, 68].

An inspection is a review of a software work product by a group of peers following a clearly defined process. The basic goal of inspections is to improve the quality of the work product by finding defects. However, inspections also improve productivity by finding defects early and in a cost effective manner. Some of the characteristics of inspections are:

- An inspection is conducted by technical people for technical people
- It is a structured process with defined roles for the participants
- The focus is on identifying problems, not resolving them
- The review data is recorded and used for monitoring the effectiveness of the inspection process

As inspections are performed by a group of people, they can be applied to any work product, something that cannot be done with testing. The main advantage of this is that defects introduced in work products of the early parts of the life cycle, or in the work products produced by other processes like the project management process or the CM process, can be detected in that work product itself, thereby not incurring the much higher cost of detecting defects in later stages.

Inspections are performed by a team of reviewers (or inspectors) including the author, with one of them being the *moderator*. The moderator has the overall responsibility to ensure that the review is done in a proper manner and all steps in the review process are followed. Most methods for inspections are similar with minor variations. Here we discuss the inspection process employed by a commercial organization [97] The different

stages in this process are: planning, preparation and overview, group review meeting, and rework and follow-up. These stages are generally executed in a linear order. We discuss each of these phases now.

Planning

The objective of the planning phase is to prepare for inspection. The author of the work product ensures that the work product is ready for inspection. The moderator checks that the entry criteria are satisfied by the work product. The entry criteria for different work products will be different. For example, for code an entry criteria is that the code compiles correctly and the available static analysis tools have been applied. The review (inspection) team is also formed in this phase.

The package that needs to be distributed to the review team is prepared. The package includes the work product to be reviewed, the specifications for that work product, relevant checklists and standards. The specifications for the work product are frequently the output of the previous phase and are needed to check the correctness of the current work product. For example, when a high level design has to be reviewed, then the package must include the requirement specification also, without which checking the correctness of design may not be possible.

Overview and Preparation

In this phase the package for review is given to the reviewers. The moderator may arrange an opening meeting, if needed, in which the author may provide a brief overview of the product and any special areas that need to be looked at carefully. The objective and overview of the inspection process might also be given in this meeting. The meeting is optional and can be omitted. In that case, the moderator provides a copy of the group review package to the reviewers.

The main task in this phase is for each reviewer to do a *self-review* of the work product. During the self-review, a reviewer goes through the entire work product and logs all the potential defects he or she finds in the self-preparation log. Often the reviewers will mark the defect on the work product itself. The reviewers also record the time they spent in the self-review. A standard form may be used for the self-preparation log; an example form is shown in Figure 2 [97].

Relevant checklists, guidelines, and standards may be used while reviewing. Checklists specifying the type of defects to look for are particularly useful. Ideally, the self review should be done in one continuous time span. The recommended time is less than two hours—that is, the work product is small enough that it can be fully examined in less than two hours. This phase of the review process ends when all reviewers have properly performed their self review and filled the self-review logs.

Project name and code :
Work product name and ID:
Reviewer name:
Effort spent for preparation (hrs):
Defect List:

Sl	Location	Description	Criticality / Seriousness

Figure 2.13: Self review log.

Group Review Meeting

The basic purpose of the group review meeting is to come up with the final defect list, based on the initial list of defects reported by the reviewers and the new ones found during the discussion in the meeting. The entry criterion for this step is that the moderator is satisfied that all the reviewers are ready for the meeting. The main outputs of this phase are the defect log and the defect summary report.

The moderator first checks to see if all the reviewers are prepared. This is done by a brief examination of the effort and defect data in the self-review logs to confirm that sufficient time and attention has gone into the preparation. When preparation is not adequate, the group review is deferred until all participants are fully prepared.

If everything is ready, the group review meeting is held. The moderator is incharge of the meeting and has to make sure that the meeting stays focused on its basic purpose of defect identification and does not degenerate into a general brainstorming session or personal attacks on the author.

The meeting is conducted as follows. A team member (called the *reader*) goes over the work product line by line (or any other convenient small unit), and paraphrases each line to the team. Sometimes no paraphrasing is done and the team just goes over the work product line by line. At any line, if any reviewer has any issue from before, or finds any new issue in the meeting while listening to others, the reviewer raises the issue. There could be a discussion on the issue raised. The author accepts the issue as a defect or clarifies why it is not a defect. After discussion an agreement is reached and one member of the review team (called the *scribe*) records the identified defects in the defect log. At the end of the meeting, the scribe reads out the defects recorded in the

Project	Xxxxxxxx
Work Product Type	Project Plan, V 1.0
Size of Product	14 pages
Review Team	P1, P2, P3, P4
Effort (Person Hours)	
Preparation	Total 10 person-hrs.
Group Review Meeting	10 person-hrs.
Total Effort	**20 person-hrs.**
Defects	
Number of Critical Defects	0
Number of Major Defects	3
Number of Minor Defects	16
Total Number of defects	**19**
Review Status	Accepted
Recommendations for next phase	
Comments	The plan has been well documented and presented

Figure 2.14: Summary report of an inspection.

defect log for a final review by the team members. Note that during the entire process of review, defects are only identified. It is not the purpose of the group to identify solutions—that is done later by the author.

The final defect log is the official record of the defects identified in the inspection and may also be used to track the defects to closure. For analyzing the effectiveness of a review, however, only summary level information is needed, for which a *summary report* is prepared. The summary report describes the work product, the total effort spent and its breakup in the different review process activities, total number of defects found for each category, and size. If types of defects were also recorded, then the number of defects in each category can also be recorded in the summary. A partially filled summary report of review of a project management plan is shown in Figure 2.14 [97].

The summary report is self-explanatory. Total number of minor defects found was 19, and the total number of major defects found was 3. That is, the defect density found is $16/14 = 1.2$ minor defects per page, and $3/14 = 0.2$ major defects per page. From experience, both of these rates are within the range seen in the past; hence it can be assumed that the review was conducted properly. The review team had 4 members, and each had spent 2.5 hours in individual review and the review meeting lasted 2.5

hours. This means that the coverage rate during preparation and review was 14/2.5 = 5.6 pages per hour, which, from past experience, also seems acceptable.

If the modifications required for fixing the defects and addressing the issues are few, then the group review status is "accepted." If the modifications required are many, a follow up meeting by the moderator or a re-review might be necessary to verify whether the changes have been incorporated correctly. The moderator recommends what is to be done. In addition, recommendations regarding reviews in the next stages may also be made (e.g., in a detail design review it may be recommended code of which modules should undergo inspections.)

Rework and Follow Up

In this phase the author corrects all the defects raised during the inspection. The author may redo the work product, if that is what the moderator recommended. The author reviews the corrections with the moderator or in a re-review, depending on the decision of the group review meeting. The scribe ensures that the group review report and minutes of the meetings are communicated to the group review team.

Roles and Responsibilities

The inspection process is a structured process with different people having different responsibilities. The key roles in a group review are those of moderator, reader, scribe, author, and reviewer. These are logical roles and a person can be assigned multiple roles, with the restrictions that the author cannot be the moderator or the reader, and the moderator cannot be the reader. This implies that the minimum size of the group review team is three—the author, the moderator, and the reader. These three people are also reviewers and can assign the role of scribe to someone. The responsibilities of these roles should be clear from the inspection process. Here we briefly summarize the main activities of the moderator and the reviewers.

The moderator perhaps has the most important role during a group review. He has the overall responsibility of ensuring that the review goes well. The moderator should undergo formal training on how to conduct reviews, or should have experience of participating in a few reviews. The responsibilities of the moderator include:

- Schedule the group review meeting
- At the opening of group review meeting ensure that all participants are prepared and have submitted self-preparation log, or reschedule the group review
- Conduct the group review in an orderly and efficient manner
- Ensure that the meeting stays focused on the main task of defect identification
- Track each problem to resolution or ensure that it is tracked by someone else
- Ensure that group review reports are completed