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, proce-

dures, 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 1.1 [17].

As Figure 1.1 shows, in the early days, the cost of hardware used to

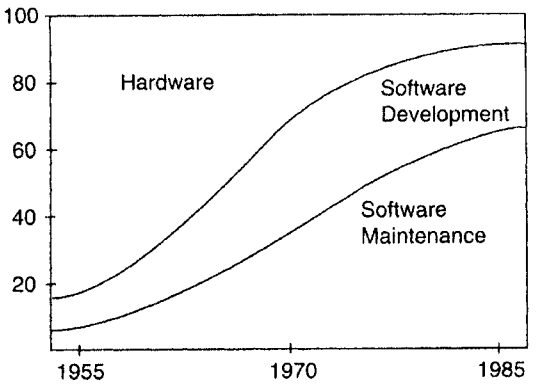

Figure 1.1: Hardware-software cost trend.

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 1.1 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

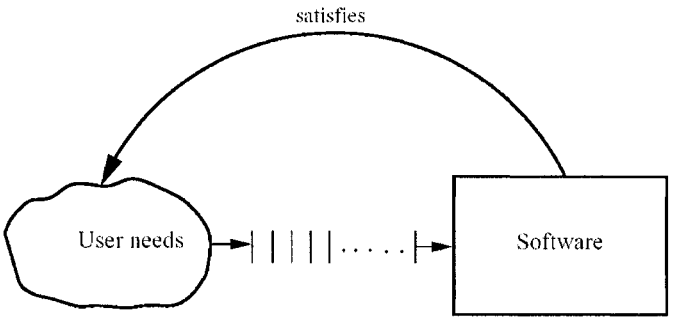

Figure 1.2: Basic problem.

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 1.2.

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 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

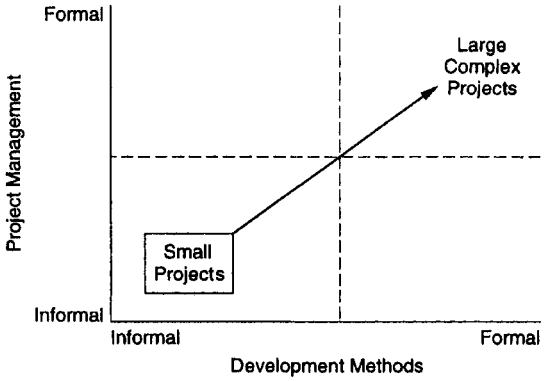

Figure 1.3: The problem of scale.

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 1.3.

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 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

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.

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 is largely labor-intensive. Hence, the cost of a software project

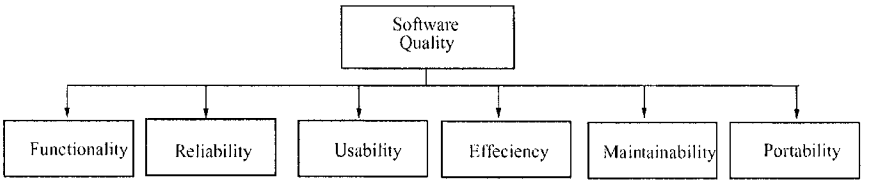

Figure 1.4: Software quality attributes.

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].

According to the quality model adopted by this standard, software quality comprises of six main attributes (called characteristics) as shown in Figure 1.4 [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 enviornments 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 1.5.

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

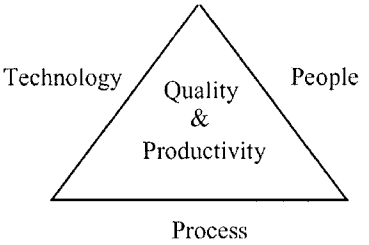

Figure 1.5: The iron triangle.

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 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 1.2.

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. Software engineering!problem domain 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?