

Software Engineering Principles

Are you ☐☐☐☐☐☐☐☐ with Onlinevarsity.com?

R G E T R E E S I D

Did you ☐☐☐☐☐☐ this ebook?

D O W N L A D

Do you have a ☐☐☐☐ copy of this ebook?

L G A E L

Are you a victim of ☐☐☐☐☐ ?

P C I A R Y

Answers: REGISTERED, DOWNLOAD, LEGAL, PIRACY

Download a **LEGAL** copy of this ebook
which you can say is **mine**
because **PIRACY** is a **crime**

Software Engineering Principles

© 2014 Aptech Limited

All rights reserved.

No part of this book may be reproduced or copied in any form or by any means – graphic, electronic or mechanical, including photocopying, recording, taping, or storing in information retrieval system or sent or transferred without the prior written permission of copyright owner Aptech Limited.

All trademarks acknowledged.

APTECH LIMITED

Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2014



There are many who firmly believe that Software Engineering as a concept does not work in the rapidly evolving world of the Internet. But the concepts of engineering, of understanding the users needs, designing a simple, complete solution that meets the users needs, coding it and testing it thoroughly are applicable to solutions meant for the internet arena as they are for a main frame solution developed through years of effort.

This book intends to introduce the student to the organized ways in which software companies develop their product. During this module you may be developing requirements, doing design, coding and testing in an informal way. This book shows how these activities are done in a structured way. Carrying out development and other support activities in an organized manner is essential for successful completion of projects.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team, H.O.

Technowise



Are you a

TECHNO GEEK

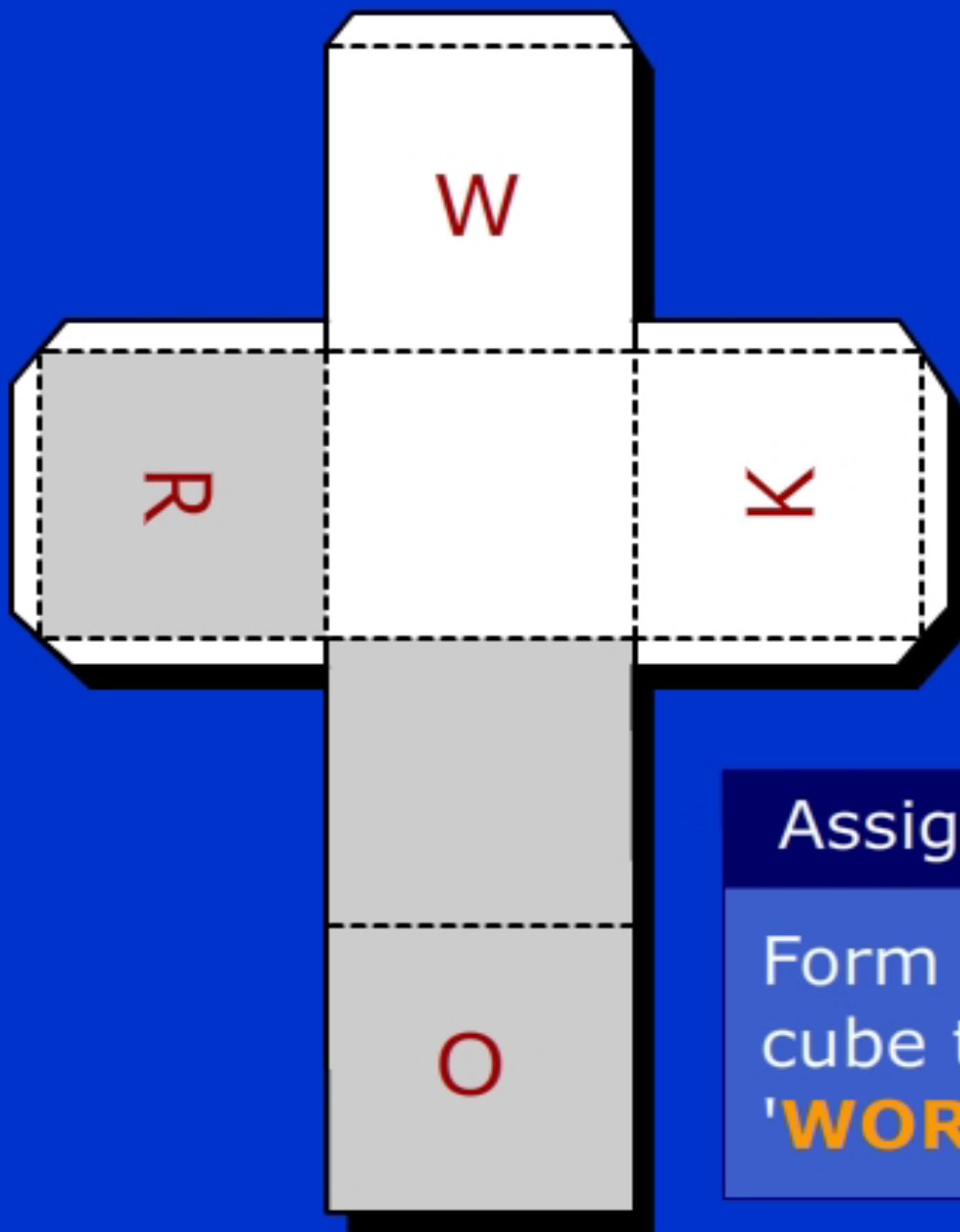
looking for updates?

Login to

www.onlinevarsity.com

CHAPTERS

- 1. Software Characteristics**
- 2. Software Development Processes**
- 3. Software Development Life Cycle**
- 4. Requirements Gathering and Analysis**
- 5. Analyzing the System**
- 6. Designing the System**
- 7. Software Configuration Management**
- 8. Software Maintenance**
- 9. Software Implementation and CASE**
- 10. Object Oriented Software Engineering**
- 11. Case Study**



Assignment

Form the
cube to read
'**WORK**'.

"Practice does not make perfect. Only perfect practice makes perfect."
- Vince Lombardi

For perfection, solve assignments @

www.onlinevarsity.com

Software Characteristics

Objectives

At the end of this chapter, you will be able to:

- *Describe software problems*
- *Describe the characteristics of the software as a process*
- *Describe the characteristics of the software as a product*
- *Explain the need for the Software Engineering discipline*
- *Define Software Engineering Goals*
- *Discuss the Role of a Software Engineer in its current context*

1.1 Introduction

The electronic computers evolved in the 1940s. Hardware posed a major challenge at that time, and all the early efforts were focused on designing the hardware. Hardware was where most technical difficulties existed. However, with the advent of new techniques, the problem subsided. With the availability of cheaper and powerful machines, higher-level languages, and more user-friendly operating systems, the applications of computers grew rapidly. In addition, the nature of software engineering evolved from simple programming exercises to developing software systems, which were much larger in scope, and required great effort by many people.

The techniques for writing simple programs could not be scaled up for developing software systems, and the computing world found itself in the midst of a “software crisis”. At that time, the term **software engineering** was coined in the conferences sponsored by NATO Science Committee in Europe in 1960. The IEEE Glossary of Software Engineering defines software engineering as: “The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.”

Currently, computer systems are used in such diverse areas as business applications, scientific work, video games, air traffic control, and so on. This increase in the use of computers in every field has led to a dramatic increase in the need for software. Furthermore, the complexity of these systems is also increasing – imagine the complexity of the software for aircraft control or a telephone network monitoring system. The complexity has grown at such a pace that one is unable to deal with it. Consequently, many years after the software crisis was first declared, one finds that it has not yet ended. Software engineering is the discipline whose goal is to deal with this problem.

Today, software takes on a dual role. It is a product, and at the same time, the vehicle to deliver a product. As a product, it delivers the computing potential embodied by computer hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is an information transformer – producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia simulation.

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

In this chapter, the major reasons for the software problem and the major problems that software engineering faces are discussed. To gain an understanding of the software, a few characteristics of software as a product and as a process are discussed. We will also learn the role of software engineering in software development.

1.2 The Software Problem

Software is not merely a collection of computer programs. IEEE defines software as a collection of computer programs, procedures, rules, and associated documentation and data. Unlike a program, *the programming system product* is generally not used only by the author of the program, but is used largely by people other than the developers of the system.

In a program, the presence of bug is not usually a major problem, because the author generally uses it; if the program crashes the author will fix the problem and start using it again. These programs are not designed with such issues as portability, usability and reliability in mind.

Software falls under the category of programming systems product. The user may be of different backgrounds and so it is generally provided with a proper user interface. The programs are thoroughly tested before operational use so that there are no bugs left behind. Also, as diverse people having diverse hardware environment use the product, portability is a key issue.

As a thumb rule, a programming systems product costs approximately ten times more than the corresponding program. The software industry is usually interested in these commercial software systems or packages falling under programming systems product. It is also sometimes called *production or industrial quality software*.

The software problems are discussed here in detail.

1.2.1 Cost of Software

The production of software is a labor-intensive activity. Over the past few decades the cost of hardware has consistently decreased. With the advent of newer and faster processors the cost of computing power has decreased. Similarly, the cost per bit of memory decreased more than 50 fold in two decades. On the other hand, the cost of software is increasing. Building and maintaining

it are labor-intensive activities, but delays in delivery can be very costly, and any undetected problems may cause loss of performance and frustrate users.

Figure 1.1 shows that software development and maintenance costs have increased in the last few decades.

The size of software is usually measured in terms of Delivered Lines of Code (DLOC) and productivity is measured in terms of DLOC per person-month. The cost of developing software is generally measured in terms of person-months of effort spent in development.

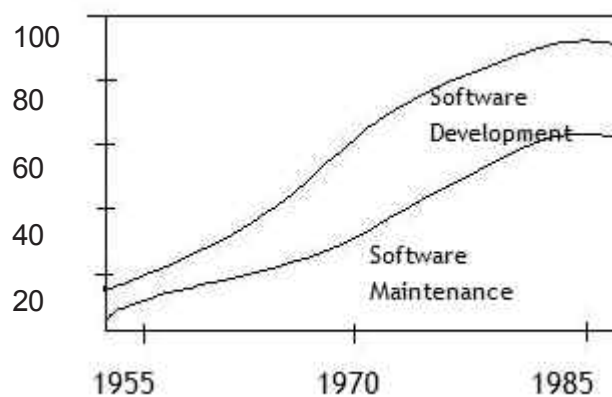


Figure 1.1: Software cost trends

The current productivity in the software industry is usually in the range of 300 to 1000 DLOC per person-month. 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 approximately. This means that with the current productivity figures of the industry, \$8 to \$25 is charged per line of code. Moderately sized projects easily end up with software of 50,000 LOC (Lines of code). With this productivity, such software will cost about \$50 Million and \$12.5 Million.

1.2.2 Reliability

There are many instances quoted about software projects that are behind schedule and have heavy cost overruns. The software industry has gained notoriety for not being able to deliver on time and within budget. For example, a Fortune 500 consumer products company plans to get an information system developed in nine months at the cost of \$250,000. Two years later, after spending \$2.5 million, the job was still not done, and it was estimated that another \$3.6 million would be needed. The project was scrapped (because, the extra cost of \$3.6 million was not worth the returns).

There are *runaway* projects in software industry. These are not projects that are somewhat late or over budget – these are the ones where the budget and schedule are out of control. The problem has become so severe that it has spawned an industry of its own for which there are many consultancy firms that advice how to rein such projects.

The software industry is notorious for not delivering software within schedule and budget and of producing software systems of poor quality. Most of the failures occur due to bugs that get introduced into the software, and as a result do what it is not supposed to do. For example, many banks have lost millions of rupees due to inaccuracies and other problems in their software. The software failure is very different from the failure of mechanical or electrical systems. These systems fail because of aging but software fails due to bugs or errors that get introduced during the design or development process. Hence, even though software may fail after operating correctly for some time, the bug that causes that failure was there from the start.

1.2.3 Change and Rework

Once the software is installed and deployed, it enters the *maintenance* phase. This phase is usually divided into two types:

- **Corrective maintenance:** This type of maintenance is needed to correct or debug some residual errors in the software as and when they are discovered, leading to the software getting changed. Many of these errors surface only after the system has been in operation, sometimes for a long time.
- **Adaptive maintenance:** Software must be often upgraded and enhanced to include more features and provide more services. Once the system has been deployed; the environment in which it operates also changes. The changed software then changes the environment, which in turn requires further change. This phenomenon is called *the law of software revolution*. Maintenance due to this is *adaptive maintenance*.

Maintenance work is based on existing software as compared to development work that creates new software. So maintenance revolves around understanding existing software, and maintainers spent most of their time trying to understand the software they have to modify. The *regression testing* is carried to check whether the system is functioning as before or not, after making necessary modifications. It 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 (changes), and retesting the old parts that were not changed.

One of the biggest problems in software development, particularly for large and complex systems, is that the requirements are not understood. The software does what it is not supposed to do. The requirements are “frozen” when it is believed that they are in good shape, and then the development proceeds. However, as software is being developed the client gets a better understanding of the system, and new requirements are discovered which were not specified earlier.

This leads to *rework*; the requirements, design, and code all have to be changed to accommodate new requirements. It is estimated that rework costs are 30 to 40% of the development cost.

1.3 Software as a Process

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 desired result is high-quality software at low cost. Clearly, a process that does not scale up, that is, cannot handle large software projects or cannot produce good-quality software is not a suitable process.

The fundamental objectives of a process are *optimality* and *scalability*. Optimality means that the process should be able to produce high-quality software at low cost, and scalability means that it should also be applicable for large software projects. To achieve these objectives, a process should have some properties. A few important properties are discussed here.

1.3.1 Predictability

Predictability of a process determines how accurately the outcome of following a process in a project can be predicted before the project is completed. If it is not predictable, the process is of limited use. If the past experiences to control costs and ensure quality are used, a predictable process must be used. With low predictability, the experience gained through projects is of little value. A predictable process is also said to be *under statistical control*.

A process is under statistical control if following the same process produces similar results. This is shown in figure 1.2; the y-axis represents some property of interest (quality, productivity, and so on), and x-axis represents the projects. The dark line is the expected value of the property for this process. Statistical control implies that most projects will be within bound around the expected value.

Statistical control also implies that the value of the property should remain within a particular bound, if the same process is followed. If 20 errors per 100 LOC have been detected in the past for a process, then it is expected that with a high probability, this is the range of errors that will be detected during testing in future projects:

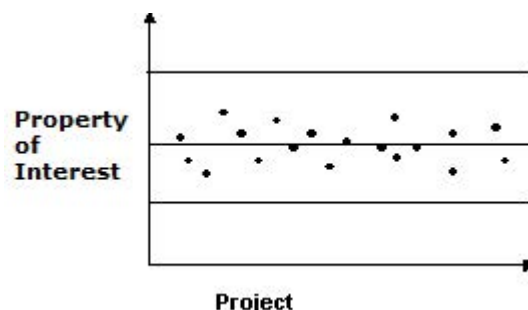


Figure 1.2: Process under statistical control

1.3.2 Support Testability and Maintainability

As seen earlier, maintenance costs generally exceed the development costs during the life of the software. Thus, to reduce the overall cost of software, the goal of development should be to reduce the maintenance effort. According to a survey done by Bell Labs, the effort distribution within phases of a software process is as shown in Table 1.1. Similarly, the distribution of how programmers spend their time on different activities is shown in Table 1.2.

Requirements	10%
Design	20%
Coding	20%
Testing	50%

Table 1.1: Effort Distribution with different phases

Writing Programs	13%
Reading Programs and manuals	16%
Job communication	32%
Others (including personal)	39%

Table 1.2: How programmers spend their time

Table 1.1 shows that most of the effort is spent in software testing, while only a small part is spent in programming. So, 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 design and coding of the software, 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”.

1.3.3 Early Defect Removal and Prevention

Errors can occur at any stage of the development cycle. An example distribution of error occurrences by phase is listed in Table 1.3:

Requirement Analysis	20%
Design	30%
Coding	50%

Table 1.3: Distribution of error occurrences

As can be seen, errors occur throughout the development process. However, the cost of correcting the errors of different phases is not the same, and depends on when the error is detected and corrected.

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 if it were corrected in the requirements phase itself. The reason is fairly obvious. The error in the requirements phase will affect the design and code.

If the error were corrected after coding, both design and code would have to be changed, thereby increasing the cost of correction. So, one can deduce that errors should be detected in the same phase itself in which it has originated, and not wait until testing.

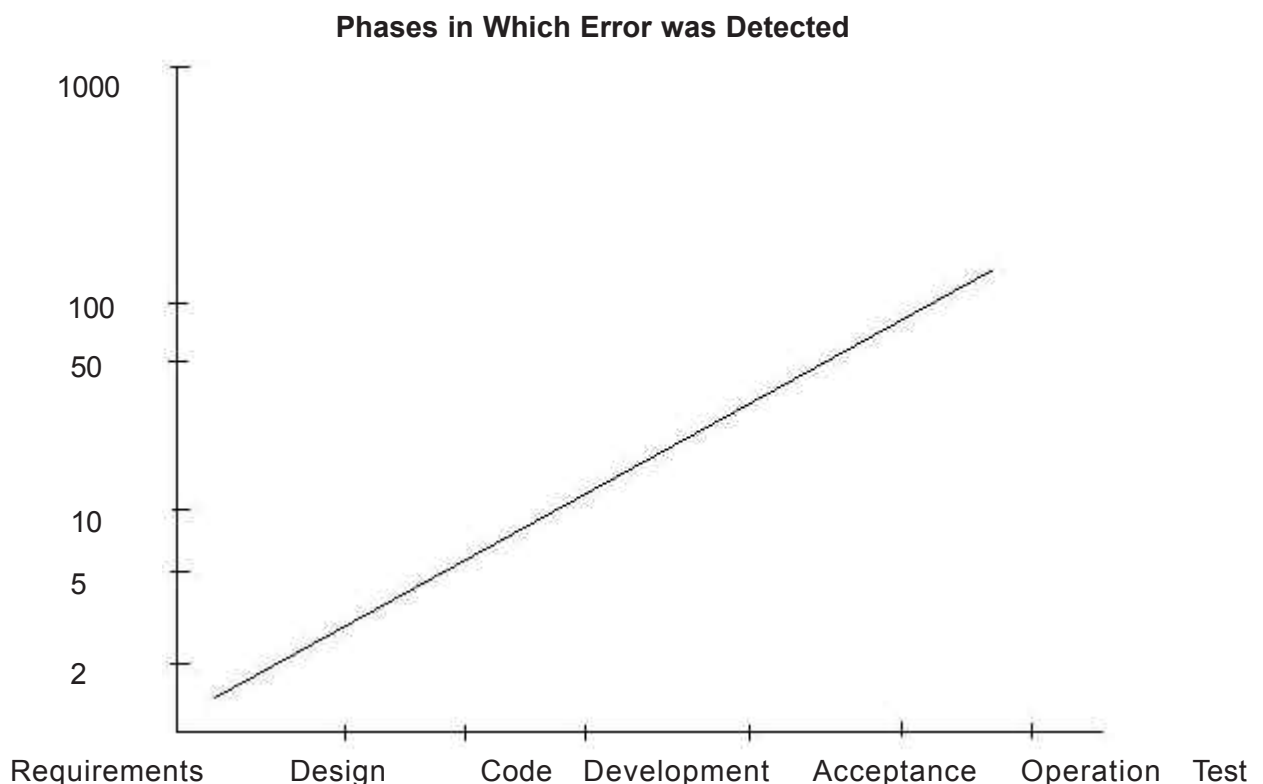


Figure 1.3: Cost of Correcting Errors

1.3.4 Process Improvement

A process is not a static entity. As the cost and quality of the software are dependent on the software process, it should be improved to satisfy goals as quality improvement and cost reduction. The process must learn from the previous experiences. Each project done using the existing process must feed information back to the process itself, which can then use this information for self-improvement.

1.4 Software as a Product

The goal of any engineering activity is to build something- a product. The civil engineer builds a dam, the aerospace engineer makes a plane, and the electrical engineer makes a circuit. The product of software engineering is a “software system”. It is not as tangible as the other products, but it is a product nonetheless. It serves a function.

Software unlike other products is not a physical entity. One cannot touch or feel it to get an idea about its quality. Software is a logical entity, and therefore, it is different from other engineered products.

To gain an understanding of the software, let us examine the characteristics of software that makes it different from other things that human beings build.

➤ **Software is developed or engineered, it is not manufactured**

Software does not automatically roll out of an assembly line. As we will learn later, there exist a number of tools for automating the process, especially for the generation of code; but the development depends on the individual skills and creative abilities of developers. This ability is difficult to specify, difficult to quantify, and even more difficult to standardize.

In most engineering disciplines, the manufacturing process is considered carefully, because it determines the final cost of the product. Also, the process has to be managed closely to ensure that defects are not introduced. The same considerations apply to computer hardware products. For software, on the other hand, manufacturing is a trivial process of duplication. The software production process deals with design and implementation, rather than manufacturing.

➤ **Software is malleable**

The characteristic that sets software apart from other engineering products is that software is malleable. The product itself can be modified (as opposed to its design) rather easily. This makes it quite different from other products as cars and ovens.

The malleability of software is often misused. While it is certainly possible to modify a plane or a bridge to satisfy some new needs – so as to make it support more traffic- but this is not easy to implement. This modification is not taken lightly and it is not directly attempted on the product itself; the design is modified and the impact of change is verified extensively. Software engineers are also often asked to make modifications in their system due to the malleability property. In practice it is not easy.

The code may be changed easily, but meeting the need for which the change was intended is not necessarily done so easily. One should indeed treat software like other engineering products in this regard: a change in software must be viewed as a change in design rather than in code, which is just an instance of the product.

The property of malleability of software can be used to advantage – provided it is done with a lot of discipline; this is where procedures and quality standards for making modifications become important.

➤ **Software does not “wear out”**

The relationship between failure rate and time for hardware is shown in Figure 1.4. As can be seen, the figure is shaped as a bathtub and therefore often called as “bathtub curve”. The relationship depicts that the hardware shows high failure rate in its early life cycle (these failures are due to manufacturing or design defects); defects are corrected, and the failure rate falls to a steady-state level for some period of time.

As time passes by, the failure rate rises again as hardware begins to wear out due to cumulative affects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

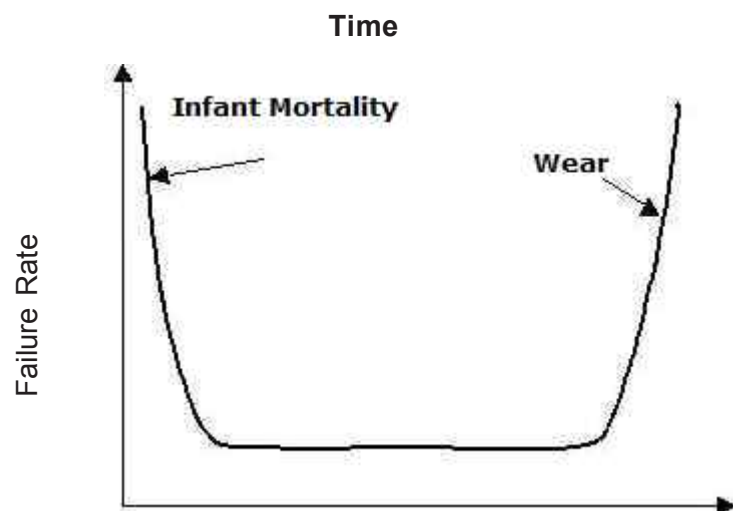


Figure 1.4: Hardware Failure Curve

The curve for software is not the same as shown in Figure 1.4, because software is not susceptible to environmental maladies. In theory, the curve for software failure against time should be as shown in Figure 1.5. The defects in the early stages will cause high failure rates in the early life cycle but as the defects are corrected, the failure rate drops, and the curve flattens as is shown in figure 1.5.

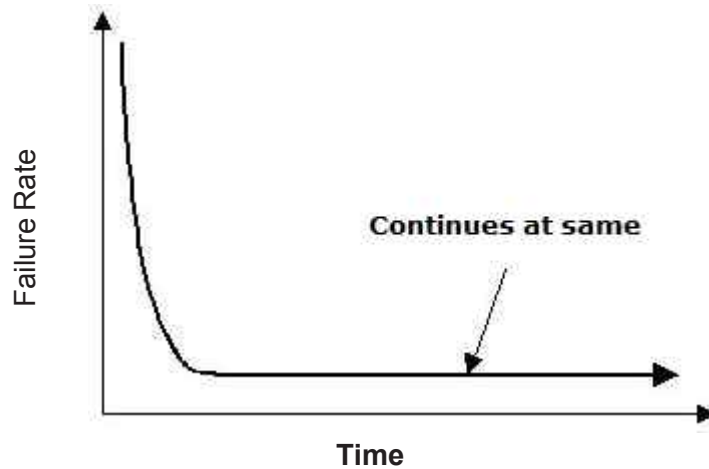


Figure 1.5: Idealized software Failure Curves

The failure rate for software is as shown in Figure 1.6. This contradicts the earlier curve because software undergoes maintenance during its life. As changes are made, new defects are introduced, causing the failure rate to spike. As the defects are corrected, the failure rate drops again. However, before the curve can return to the original steady state, a new change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise; thus software deteriorates due to continuous change requests.

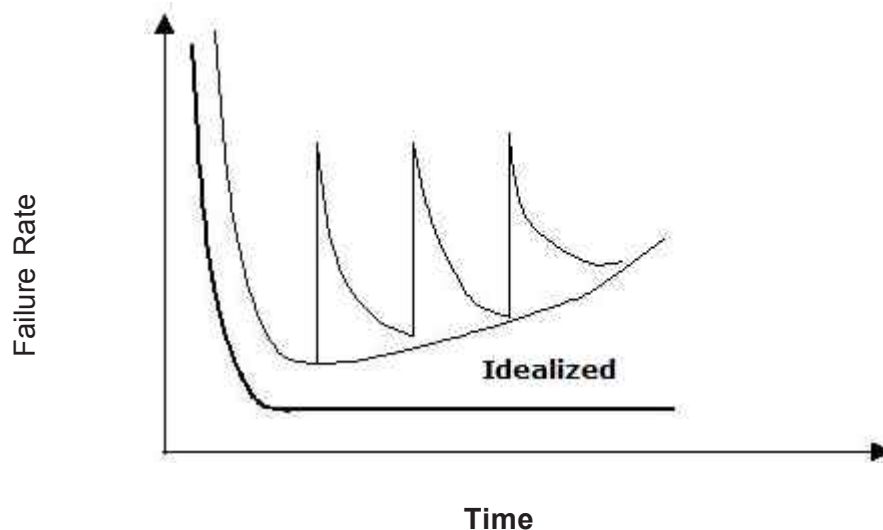


Figure 1.6: Software Failure Curve

Another difference between hardware and software is that when a hardware component wears out, the part is replaced by a spare part. However, no such spare parts exist for software. Therefore, the maintenance of software is much more difficult than hardware.

➤ **Most software are custom-built, rather than being assembled from existing components**

In manufacturing engineered products, a design is first made identifying the various components that go into each. These are then put together as per the original design. This approach affords an enormous amount of flexibility. Any number of people can work independently to produce different components so that the manufacture of one assembly can be entirely independent of another.

Now, if the organization wants to increase production quantities or decrease the production cycle time, it can easily subcontract some of the jobs. The final product merely involves putting together several independently manufactured components. For example, for building hardware, the digital components are assembled to achieve proper functioning. These digital components can be ordered 'off the shelf'.

Till now, software engineers do not have this luxury. Software can be ordered off the shelf, but as a complete unit, not as a software component that can be assembled into software programs. Fortunately, this situation is changing rapidly. The widespread use of object-oriented technology has resulted in the creation of software engineering components. Though the scenario is changing but this is only the beginning of this concept. This concept is usually known as "software reusability".

1.5 The Need for Software Engineering

Most of the problems related with software product development arise because of its inherent characteristics and if we look at other products around us we will find very few similarities with other products that too are designed and developed.

Therefore, comparisons to be made are few, if at all. We must then understand the uniqueness of the problems that are faced in software development so that we can tackle them and find appropriate solutions for them.

The focus on the discipline of software engineering is appreciated more if one realizes the uniqueness of the problems that we are trying to deal with. Once we realize this, it becomes easy to understand the importance of quality of software products.

1.5.1 Factors Influencing the Development of Software and its Quality

Software Engineering can be considered as an answer to many problems that pervade the software development activity. It would be worthwhile to explore some of the problems faced by the industry and then try to understand how the software engineering discipline is equipped to handle them.

Problems faced in the process of software development include:

- Individual ability
- Team communication
- Non-linear estimation
- Change control
- Available project time
- Problem understanding
- Inadequate training
- Lack of management skills
- (Rising) Level of user expectations
- **Individual ability** - There are two aspects of individual ability:
 - Individual competence
 - Knowledge/familiarity with the application area under development

Unlike other professionals, a software person is required to be extremely competent in the technology that is employed for software development (hardware / programming platform), the person must possess good problem solving abilities and also be well versed with the application area that has to be automated. So, whether or not we liked accounts in school, we need to know our debits from credits when working on an accounts system. Lack of knowledge of applications results in low productivity and poor quality.

At most times, the competence level of an individual reflects very heavily in small or medium sized projects, where the number of programmers is few. However, in larger projects, the effects of poor programming are averaged out, as there will also be extremely efficient pieces of code written by the more competent programmers.

Software Engineering provides methods, tools, and procedures, which to some extent, even out the competency levels that exist among individuals and provide standardization. Nevertheless, an individual's competence remains a factor of some importance in the entire process.

➤ **Team communication**

Here, we are dealing with a paradox. It is a well known and proven fact that programmers have a low social need and prefer to work alone, rather than in groups. However, a software project, like all other projects, is teamwork.

Software engineering practices attempt to resolve the situation by providing for bring-your-work-in-the-open techniques like structured walkthroughs and design reviews. Another misconception about team communication is that if we put more people on a project running behind schedule, it will run faster. In fact studies have revealed quite the contrary.

Human beings are not machines and each individual needs to communicate with the other. If there are four individuals on a project, there will be sixteen channels of communication (4x4). If we put two more individuals on this project, then we are not adding two more channels of communication - instead we are adding twenty channels (6x6)!

Notice how the communication complexity increases. We could end up spending more time trying to decipher what others are communicating rather than pushing the project to its stipulated completion date.

Brooks has stated that putting more men on a late project may make it later - and nothing could be closer to the truth. In the words of Brooks, "Men and months are interchangeable commodities only when a task can be partitioned among many workers with no communication among them. This is true of reaping wheat and picking cotton: it is not even approximately true...of programming".

The underlying assumption for this is that men and months are not interchangeable – so that if we will put more men on a project, the time taken to complete a task could increase drastically.

Software engineering does bring to use structured forms of communication and reporting mechanisms, such that each person knows what the other is talking about. So minimum time is spent deciphering communication.

➤ **Non-linear estimation**

Any estimate of how long it will take to develop a piece of software is dependent on two basic facts - the complexity of the software and its size.

One of the methods to grade complexity is by the type of software being written:

- Application (programs written for processing data using high level language)
- System (compilers, assemblers, and so on)
- Utility (real-time process control systems, data communication packages, and so on)

It is an accepted fact that application programs have the highest productivity, and systems programs the lowest. Size and complexity of programs have a non-linear bearing on the effort required to develop them. This means that we cannot make generalizations such as if a software product is twice as large or complex as an existing one, it will take twice as long to develop it. Studies have revealed that it may take 10 times longer or even 100 times longer.

Use of software engineering practices ensures that we do not rely upon ad hoc linear estimations. It treats software time and effort estimation more formally and provides a place for it in the software development life cycle.

➤ **Change control**

The fact that software can be made flexible to suit changing requirements is its strength, but it is a software engineer's nightmare. Requirements can change because of changing business conditions or because of poorly understood requirements. While the first is acceptable, the second is not.

Software engineering provides a formal method for understanding and documenting user requirements such that changes to software for this reason are minimized. It also provides for a formal method of documenting changes in business situations that may trigger changes in the software. First document the change required, and then document the changed piece of software.

Software Management is a full-fledged activity that treats change control and falls under the purview of software engineering.

➤ **Available time**

Staffing a software project is a critical activity, which unfortunately is often ignored or relegated to the background. It is a common occurrence that when estimated time required to develop a piece of software needs 4 months, project managers would like to put 2 programmers to speed up the work or worse still put 4 people to have the job finished in one month.

In such cases, we ignore the communication factor discussed earlier and the fact that each of these 4 people have a learning curve, which by itself may exceed the project time of one month. On the other hand, these four people may have extensive experience of having developed similar systems before, and may be quick to get off the mark and get the job finished in a month; no specific rule is applicable here.

Determining optimum staff levels with respect to the available project time and resources available is an important and difficult aspect of project management, and as an activity it finds an important place in software engineering procedures.

➤ **Understanding the problem**

Understanding user problems is a major problem faced by software engineers, without which it is very difficult to develop quality software.

Software engineering procedures provide various mechanisms (like user interviews and reviews, fact collection techniques, prototyping) and build adequate time for these in the product life cycle.

➤ **Inadequate training**

Most entry-level programmers are not adequately trained to handle the multifarious activities of a software engineering project. Their expertise is limited to coding and related tasks such as maintenance to the extent of fixing “bugs”. They do not communicate with other team members or coordinate project activities. They also do not enjoy working in teams and are unfamiliar with project management techniques. They face difficulty in expressing themselves without using technical jargon, which puts off the user, and makes communication even more difficult.

Software engineering has brought the multi-faceted tasks associated with a software project out of the closet. Techniques of project reviews, structured walkthroughs, and design reviews bring the discussion of various project-related issues into the open. Thus, more and more people on the project are able to learn on-the-job and appreciate the importance of these techniques. Software engineering tools enable unbiased communication. Therefore, communication is standardized to a large extent.

➤ **Lack of management skills**

This problem has arisen because technically competent individuals have been promoted as project managers. They have neither the inclination nor are they trained to perform managerial tasks associated with a project. As discussed earlier, programmers have a low social need (they do not like to interact with other individuals), and software project management is largely a man management issue. The discipline of software engineering has brought the management issue to the fore and focused industry attention on it.

➤ **(Rising) Level of user expectations**

Simply stated, user expectations are in direct proportion to the industry’s proclamation of reliability, quality, safety, and flexibility issues of software. Users will expect the industry to perform to unprecedented quality levels. The industry itself will place demands on members of the software development community for a sterling performance, fuelled by research in newer and better software techniques, tools, and procedures.

These problems pose a challenge to the software engineer whose role is undergoing transformations making it more wholesome than that of a programmer.

The need for software engineering practices is impelling also because of the software “crisis” that the industry faces. The crisis discussed in the next section is the result of problems associated with software development, which in turn arise because of certain inherent characteristics of software that sets it apart from other products.

It is not possible to identify a single, overall goal for software engineering to satisfy all the emerging challenges. However, if we had to identify a single goal for a software engineering project, it would be “**to develop quality software products**”.

A quality software product is developed according to stated requirements, complying with the user's need, is error free, and efficient. However, it will be unfair to single quality as the most important goal of every software project. Though predominantly important, individual projects will have different goals. Therefore, it is very difficult to prescribe a single goal for all software engineering projects.

Goals have to be defined in concrete terms to ensure that every team member understands, adheres, and works towards achieving them. Thus when a project is broken down into phases, goals must be set for every phase, and reviewed during and after the phase is over. These sub-goals of each and every phase must collectively add to the overall goal of the project.

Not only does a software project have several goals, in cases, some conflict each other. An example may be an attempt to make software highly reliable, which often slows down attempts to achieve higher life-cycle productivity, assuming that reliability and productivity are two goals of the project in question. A goal that applies well to one project may not apply so well to another. Thus, in trying to identify goals of a software engineering project, we must identify a set of desirable goals such that it minimizes the conflicts within the goals of a project.

A good example to illustrate this point is an experiment conducted by Weinberg-Schulman several years ago. In this experiment, five teams were given the same program to write with different objectives. Team **A** was asked to complete the assignment with the least possible effort. Team **B** was to minimize the number of statements in the program. Team **C** was asked to minimize the memory used by the program. Team **D** was to produce the clearest possible program, and Team **E** was to produce the clearest possible output.

When results were tallied, it was found that almost every team finished first on their stated objective but lagged on the others.

Therefore, the following lesson can be drawn from this experiment:

- Goals motivate teams to perform
- Goals can conflict with one another – we need to carefully balance them

We set goals for ourselves in our day-to-day lives and need to balance them every now and then. For instance, we are on vacation and have to reach from Point A to Point B in an hour's time. We can cover this distance by train or by road; train ride is more pleasurable, relaxed, scenic, and takes two hours to get there, while the road is crowded and strenuous, but can get us there in 45 minutes. Since we are on vacation, our ultimate objective is enjoyment at every step, so we should take the train to get to our destination. However, time is an issue. Here is the tradeoff. Such tradeoffs come by aplenty in software projects, maintainable code versus user-friendly code versus memory efficient code, to cite a few.

An important point to bear in mind is that a team performs best when given well-stated and clear goals, so it is very important to verbalize or write out the goals of each phase.

Figure 1.7 summarizes the objectives of goals:

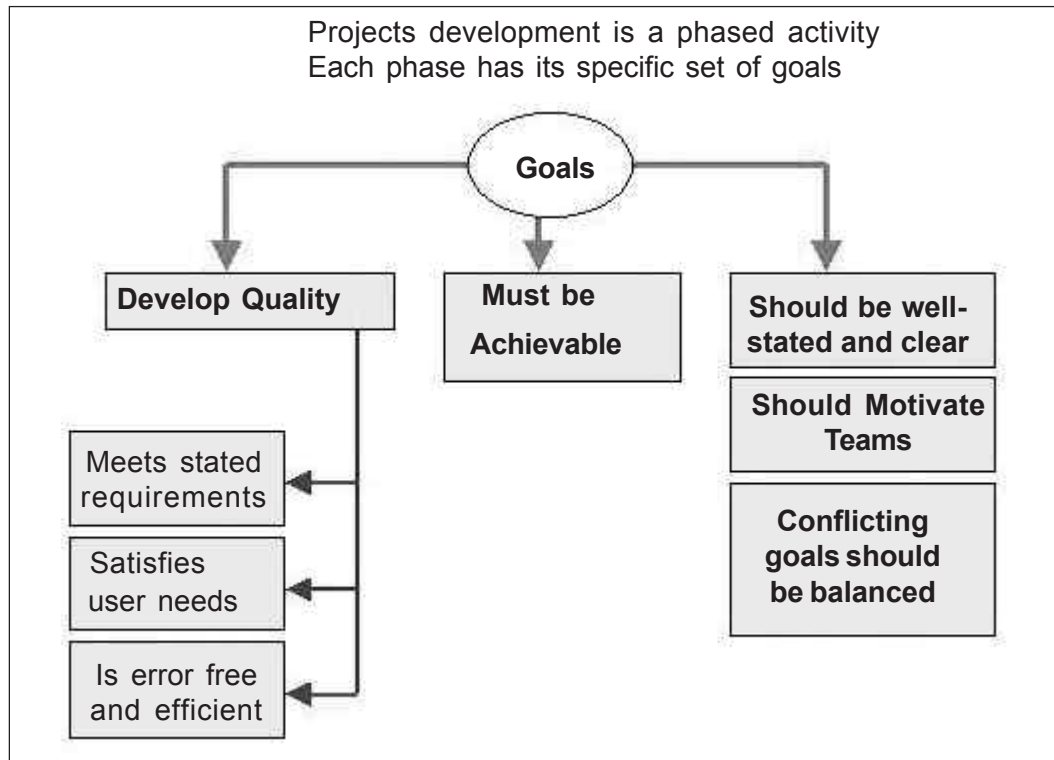


Figure 1.7: Software Engineering Goals

It is essential to remember two important things about goals:

- Two types of goals, quantitative and qualitative may be identified for a project.
- Goals must be identified for every phase in the projects; there can be goals and sub-goals in keeping with activity distribution for every phase.

The type of goals for a software project may be to:

- Keep the programs within 'n' K of core memory
- Develop a product that is easy to modify by members other than those developing it
- Develop the product within 'n' months
- Use the project as a means of educating part of the programming staff in project management techniques

Some of these are goals stated quantitatively while others are stated qualitatively. Also, there are some generic goals of a software engineering product; these are qualities that software must represent. Besides, there are specific goals of every project.

1.6 Role of a Software Engineer

In today's IT world, there is a need for a new type of engineer, who understands both the fundamental concepts of computing, as imparted by computer science, as well as the skill and ability to apply these concepts to the creation of useful, cost-effective software systems.

Apart from being a good programmer, a software engineer must be familiar with various design approaches, be able to translate user requirements into specifications, as well as be able to interact with the users on various areas of an application.

During interaction with the user, a software engineer must be able to guide the users' choice with regard to the tradeoffs that are likely to be faced with each model. A software engineer should also possess good communication and interpersonal skills. It is important for the engineer to be able to work efficiently with team members.

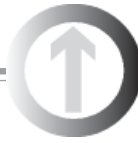
Therefore, a software engineer should be able to move with equal ease through the different stages of a project, such as determination of system requirements, detailed coding levels, implementation, and signoff. Besides technical capabilities, the engineer is also expected to sharpen his managerial skills so that he can participate in a project in a wholesome manner.

However, this should not give the impression that a software engineer on the project is a 'jack of all trades'. So where are the specialists? Of course, there are specialists on a project, analysts for conducting analysis, design specialists for designing a system; but every member should have an all round perspective of the project.

Defining a Software Engineer:

"A Software Engineer is a person who applies Software Engineering principles to the process of software development."

As Software Engineering is applicable to all phases of software development, a Software Engineer can be an analyst, a designer, a programmer, tester or a project manager.




SUMMARY

- Software cost forms the major component of a computer system's cost. Software is currently extremely expensive to develop and is often unreliable.
- Software is not just a set of computer programs but comprises of programs and associated data and documentation. Each of these items is a part of the software engineering process. The main problems for software development currently are: high cost, low quality, and frequent changes causing change and rework.
- Software has become a limiting factor in the evolution of computer-based systems. The intent of software engineering is to produce a framework for building higher quality software.



CHECK YOUR PROGRESS

1. The size of software is usually measured in terms of _____.
2. IEEE defines software as a collection of computer programs, procedures, rules, and associated documentation and data. **[True/False]**
3. The fundamental objectives of a process are _____ and _____.
4. _____ of a process determines how accurately the outcome of following a process in a project can be predicted before the project is completed.
5. The characteristic that sets software apart from other engineering products is that software is malleable. **[True/False]**
6. A quality software product is one that is developed according to stated requirements, complying with the user's need, is error free, and efficient. **[True/False]**
7. Software is malleable in nature. **[True/False]**



Objectives

At the end of this chapter, you will be able to:

- *Describe the importance of a phased approach to software development*
- *List various phases involved in software development process*
- *Describe how the various phases are organized into processes*

2.1 Introduction

In the previous chapter, we looked at the inherent complexities of software and the problems faced by software organizations in attempting to develop software that meets user requirements effectively and efficiently. Software organizations use the discipline of Software Engineering to develop methods and procedures for software development that can be scaled up for large systems, and that can be used to consistently produce high quality software at low cost, and with a reduced cycle time.

The basic approach used is to separate the developing process of the software from the product under development. The development process controls quality, scalability, consistency and productivity. The software process must include planning in addition to the development processes for project management, to ensure that the cost and the scheduled target are achieved. Software development done in phases ensures better management of the process and achieves consistency.

2.2 A Phased Approach to Software Development

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 primary reason for using a phased process is that it breaks the problem of developing software into successfully performing 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 were tackled together.

Furthermore, a phased software developmental approach allows proper checking of quality and progress at some defined points during the development. In general, any problem solving in software must consist of these activities during the development process: requirement specification, system design, detailed design, coding and testing, and conversion, as shown in Figure 2.1.

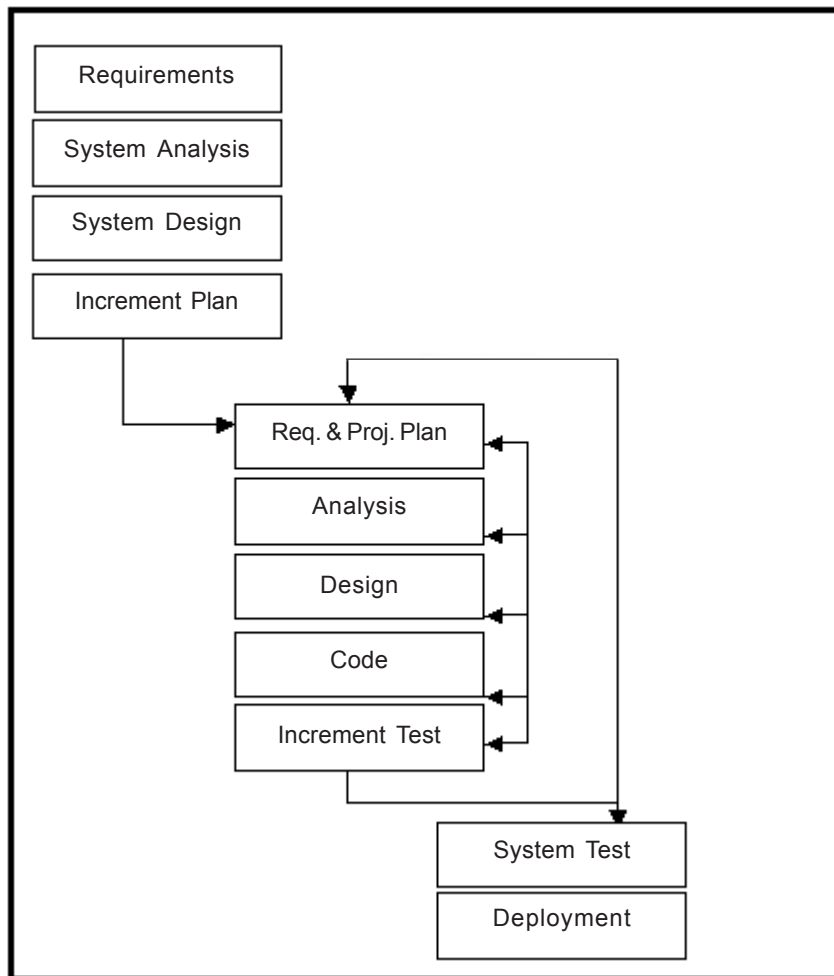


Figure 2.1: Phases in a Process Model

i. Requirement Specifications

Requirements specifications carefully define the objectives of a new or modified system, and list a detailed statement of the functions that the new system must perform.

ii. Analysis

During analysis, the focus is on building models that unambiguously determine the problem for which a software solution is being constructed. During the analysis stage, one should not be concerned with the limitations of the target environment.

iii. Design

During the design stage, the analysis model is adapted such that it serves as the basis for implementation in the target environment. Hence, the design deals with transforming or

refining the analysis model into a design model that determines how one eventually obtains a working system.

iv. **Implementation**

During the implementation stage, the coding of the system is performed.

v. **Testing**

Testing deals with the validation of software at various levels. Testing can be categorized as follows:

- **Unit Testing**

Unit testing deals with testing the smallest units of the designed software.

- **Integration Testing**

Integration testing deals with testing the application that has been (partially) put together by integrating the smallest software units.

- **Validation Testing**

Validation testing tests whether the software functions in a manner expected by the user.

- **System Testing**

System testing deals with different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work should verify that all system elements have been properly integrated and perform allocated functions.

- **Recovery Testing**

Recovery testing involves system tests that force the software to fail in a variety of ways and verifies that recovery is properly performed.

- **Security Testing**

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper usage and unauthorized access. It ensures adherence to restrictions on command usage, access to data, and access requirements according to associated privileges.

- **Stress Testing**

Stress testing deals with executing a system in a manner that demands resources in abnormal quantity, frequency, or volume.

vi. Conversion

During this phase, the old system is substituted by the new system that has been developed. Four approaches can be adopted to achieve this transformation:

- **Parallel Approach**

Both the old and new system are run together until everyone is convinced that the old one can be safely taken out of operation.

- **Direct Cutover Approach**

The old system is discarded and replaced by the new one, all at the same time.

- **Pilot Study Approach**

The new system is introduced into a very limited area of the organization. It is not introduced any further into the rest of the organization as long as the pilot system does not begin running smoothly.

- **Phased Approach**

The new system is introduced in stages. The pilot study approach can be considered as a special case of this approach.

The four approaches are summarized in Table 2.1:

Classical Phases in Software Development
Requirements Specifications
Analysis
Design
Implementation
Testing <ul style="list-style-type: none">• Unit Testing• System Testing• Acceptance Testing
Conversion <ul style="list-style-type: none">• Parallel Strategy Approach• Direct Cutover Approach• Pilot Study Approach• Phased Approach
Maintenance

Table 2.1: Classical Phases in Software Development

vii. Maintenance

During the maintenance stage, the system has to be kept operational. Systematic studies have been conducted to examine the amount of time required for various maintenance tasks. Approximately 20 percent is devoted to debugging or correcting emergency production problems; another 20 percent is concerned with changes in data, files, reports, hardware, or system software. However, 60 percent of all maintenance work consists of making user enhancements, improving documentation, and recoding system components for greater efficiency.

2.3 Software Development Processes

Having discussed the various phases involved in developing software, let us briefly look at how organizations tend to string together these activities to produce “Industrial quality Software”.

A process is a particular method of doing something, generally involving a number of steps or operations.

A software development project will have at least development activities and project management activities. The development processes focus on the activities directly related to the production of software, that is, design, coding, and testing. Project management processes focus on planning, estimating, scheduling, and taking corrective action when things do not go as per plan.

Configuration management processes focus on change and configuration control, so that the correct, tested, and approved versions of software components are put together for delivery to the customer. The process management processes focus on the management and improvement of these processes. Let us look at them in detail.

2.3.1 Development Processes

The development processes specify the major development and quality assurance activities that need to be performed in the project, and hence form the core of the software development processes. The various models proposed are discussed in detail in the next chapter.

➤ A Process Step Specification

Software development activity like any production activity is carried out in a sequence of steps, each step performing a well-defined activity leading to the satisfaction of the project goals, with the output of one step forming the input into the next step.

One of the aims of any process should be to prevent defects from a phase to pass on to the next phase. This requires verification and validation activity at the end of each step. Verification checks the consistency of inputs into a phase, while validation activities check the consistency with user needs. There is a clearly defined output of a phase, which can be verified by some means, and form the input into the next phase. Such outputs of a development process are called work products. These could be requirements documents, design documents, code, prototype, and so on.

As a development process typically contains a sequence of steps, the criteria for exit and entry from the phase are to be defined. These generally depend on the implementation of the process. Besides the verification and validation criteria, the development step needs to produce some information for the management process. Information has to flow from the development process.

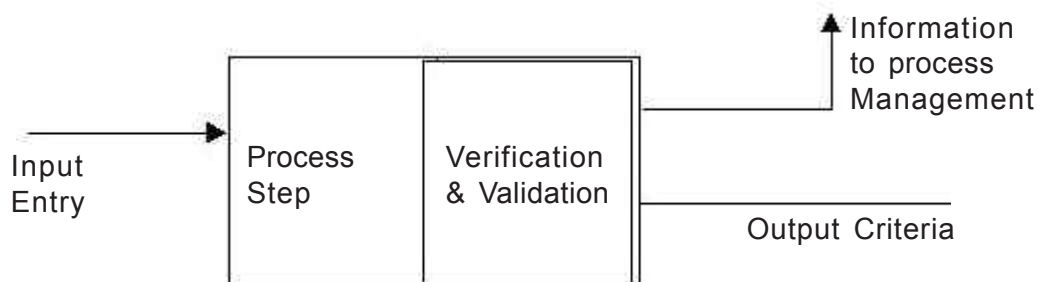


Figure 2.2: A step in a development process

2.3.2 Project Management Processes

The project management component of the software process specifies all the activities that need to be done by the project management to ensure that the cost and quality objectives are met. Project Management is an integral part of software development. To meet the cost, quality and schedule objectives, resources have to be properly allotted to each activity for the project, the progress of different activities has to be monitored, and corrective action taken, if needed. The processes focus on issues like planning a project, estimating resource, and schedule and monitoring and controlling the project. The basic task is to plan the detailed implementation of the process for a particular project and ensure that the plan is followed.

2.3.3 Software Configuration Management (SCM) Processes

Throughout development, software consists of a collection of items, such as programs, data and documents that can be easily changed. These changes do happen often. The easily changeable nature of software, and the fact that changes often take place require that changes take place in a controlled manner. Software Configuration Management systematically controls the changes that take place during development.

IEEE defines SCM as the process of:

- Identifying and defining the items in the system
- Controlling the changes of these items throughout their life cycle
- Recording and reporting the status of items
- Changing requests and verifying the completeness and correctness of items.

These processes will have to be considered independent of the development process, because development processes cannot accommodate changes at any time during development.

2.3.4 Process Management Processes

Software processes are also not static entities. Process management processes look at the changes that need to be done to improve the process.

The three important entities that software engineering deals with are processes, projects, and products. The various software processes ensure that the activities required to develop software are carried out in a controlled manner.



SUMMARY

- 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 software development process goes through several phases:
 - Requirement Specifications
 - Analysis
 - Design
 - Implementation
 - Testing
 - Conversion
- A process is a particular method of doing something, generally involving a number of steps or operations.
- The Software development process consists of the following processes:
 - Development Processes
 - Project Management Processes
 - Software Configuration Management (SCM) Processes
 - Process Management Processes



CHECK YOUR PROGRESS

1. The software development phases are performed in an order specified by the _____ model being followed.
2. Unit testing tests the smallest units of designed software. **[True/False]**
3. During the analysis stage one should not be concerned with the limitations of the target environment. **[True/False]**
4. System testing deals with different tests whose primary purpose is to fully exercise the computer-based system. **[True/False]**
5. Stress testing deals with executing a system in a manner that demands resources in abnormal quantity, frequency, or volume. **[True/False]**
6. _____ Management is the discipline of systematically controlling the changes that take place during development.
7. _____ management processes focus on change and configuration control.
8. During _____ phase, the old system is substituted by the new system that has been developed.

Software Development Life Cycle

Objectives

At the end of this chapter, you will be able to:

- *Compare how software development companies organize their development process*
- *Explain the essentials of any process models*
- *Discuss the advantages and disadvantages of various process models*
- *Describe the criteria for choosing the appropriate process models*
- *Describe the process model for the Web and the process technology*

3.1 Introduction

To every user of software, from the humble but ever present MS Word to the most complicated computational algorithm, software has always been represented as a **Black Box**.

The process of developing the black box has been an even bigger mystery. For the laymen, these software products were developed by organizations that were never heard of, may be, five years back. These software factories are and will remain quite different from the conventional smokestack industries one sees. What makes these factories different is entirely dependent on the nature of the product that is developed in these organizations.

The basic difference that makes software different from other products is that it is not a physical entity. Software exists as a logical entity and that starts all the complications.

How does one develop software? How does one manufacture the product designs? We only need to ask a manufacturing manager to set up a factory to “design his products” and watch his reactions to understand the problems a “software factory manager” has to face.

Software Engineering essentially consists of a set of steps that comprises of methods, tools and procedures that assist a Software Factory Manager manufacture software. Depending on the type of software that we plan to develop at our factory, whether it is an office automation product, which is going to be sold in millions or flight simulation software of which only one copy will be made, one needs to identify the approach that is to be used. Based on product generalizations various process models or paradigms are available.

3.2 Essentials of any Process Model

Is there a way in which we can organize our software development into a series of activities to ensure the completion, acceptance or appreciation of our software product by our customer?

Like any successful product, the development effort will have to start from an insight into the customer requirements. These requirements have to be clearly communicated to the personnel involved in developing the product. Some amount of “original thinking” will have to go into designing a solution that will meet the requirements. The design will have to be converted into software that may be bought, and code that may be developed internally. The bits and pieces developed separately will have to be integrated; the final product tested and delivered to the customer. The customer will have to be educated about the product, and if required, the product will have to be modified and maintained.

Let us look at the key activities that have to be part of any software development activity.

3.2.1 Requirements Definition

During the requirements definition activity or phase, a requirements definition team prepares the requirements document and completes a draft of the *functional specifications* for the system. This should identify the intended use and performance expected from the system as visualized by the user. *System Requirements Review (SRR)* can be planned at the end of this activity to evaluate the requirements. An implementation plan can be prepared describing the plan to build the software.

3.2.2 Requirements Analysis

The next activity is requirements analysis, during which the development team classifies each specification and performs functional or object-oriented analysis. Working with the requirements definition team, developers resolve ambiguities, discrepancies, and to-be-determined (TBD) specifications, producing a final version of the functional specifications document and a requirements analysis report. This phase is concluded with a *Software Specifications Review (SSR)*, at which the results of the analysis are presented for evaluation. The base lined functional specifications forms a contract between the requirements definition team and the software development team, and are the starting point for preliminary design.

3.2.3 Preliminary Design

Members of the development team produce a *Preliminary Design Report (PDR)* in which they define the software system architecture and specify the major subsystems, input/output (I/O) interfaces, and processing modes. The *preliminary design review (PDR)*, conducted at the end of this activity provides an opportunity to evaluate the design presented by the development team.

3.2.4 Detailed Design

The system architecture defined during the previous phase is elaborated in greater detail, to the level of subroutines. The development team fully describes user input, system output; I/O files, and inter-module interfaces. The corresponding documentation, including complete baseline diagrams, makes up the *detailed design document*. At the *Critical Design Review (CDR)*, the detailed design is evaluated to determine if the levels of detail and completeness are sufficient for coding to begin.

3.2.5 Implementation

During implementation (code, unit testing, and integration) activity, the development team codes the required modules using the detailed design document. The system grows as new modules are coded, tested, and integrated. The developers also revise and test reused modules and integrate them into the evolving system. Implementation is complete when all code is integrated, and when supporting documents (*system test plan* and *draft user's guide*) are written.

3.2.6 System Testing

This activity involves the functional testing of end-to-end system capabilities according to the system test plan. The development team validates the completely integrated system and produces a preliminary *system description* document. The end of this activity requires successful completion of tests.

3.2.7 Maintenance and Operation

Maintenance and operation are the important and unavoidable activities that begin when acceptance testing ends. The system becomes the responsibility of the maintenance and operation group. The nature and extent of activity during this activity depends on the type of software being developed. For some support software, the maintenance and operation phase may be very active due to changing user needs.

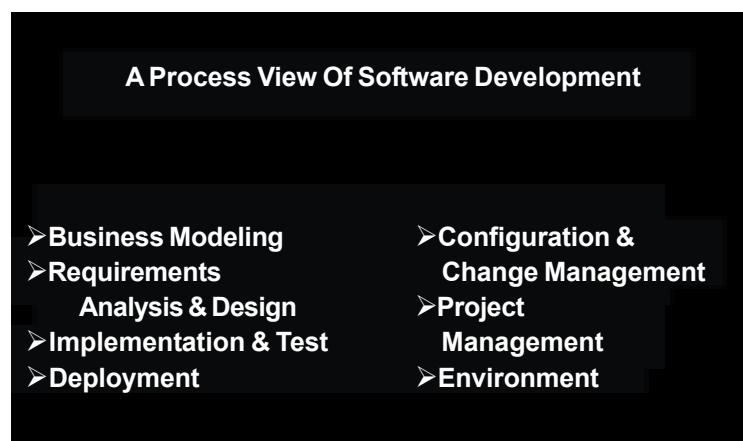


Figure 3.1: A Process view of Software Development

This sequence of activities that are used to deliver the product cannot happen in isolation. There is a set of support activities, that is, Project Management Configuration Management and Data collection (Figure 3.1). The Project Management activity involves planning, scheduling, and monitoring the progress of the project.

The Change and Configuration Management activity ensures that the components of software that are being independently developed are controlled, and the correct versions are assembled together in the software suite. The data collection activities are required for analysis so as to take necessary corrective and preventive action.

All software development organizations will have to carry out these activities if they have to develop a product. The sequence and detailed level of these activities will depend on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required. This has given rise to various process paradigms, which are followed by Software Factories.

Let us evaluate some of the common process models used in industry and their application.

3.3 Process Models

The process models prevalent in the software industry are discussed. These can be broadly categorized into the Linear Process Models and the Evolutionary Process Models.

3.3.1 Linear Process Models

There are several models in this category:

➤ The Waterfall Model

The Waterfall model, sometimes called the linear sequential model, or “classic life cycle” is illustrated in Figure 3.2. This model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and maintenance.

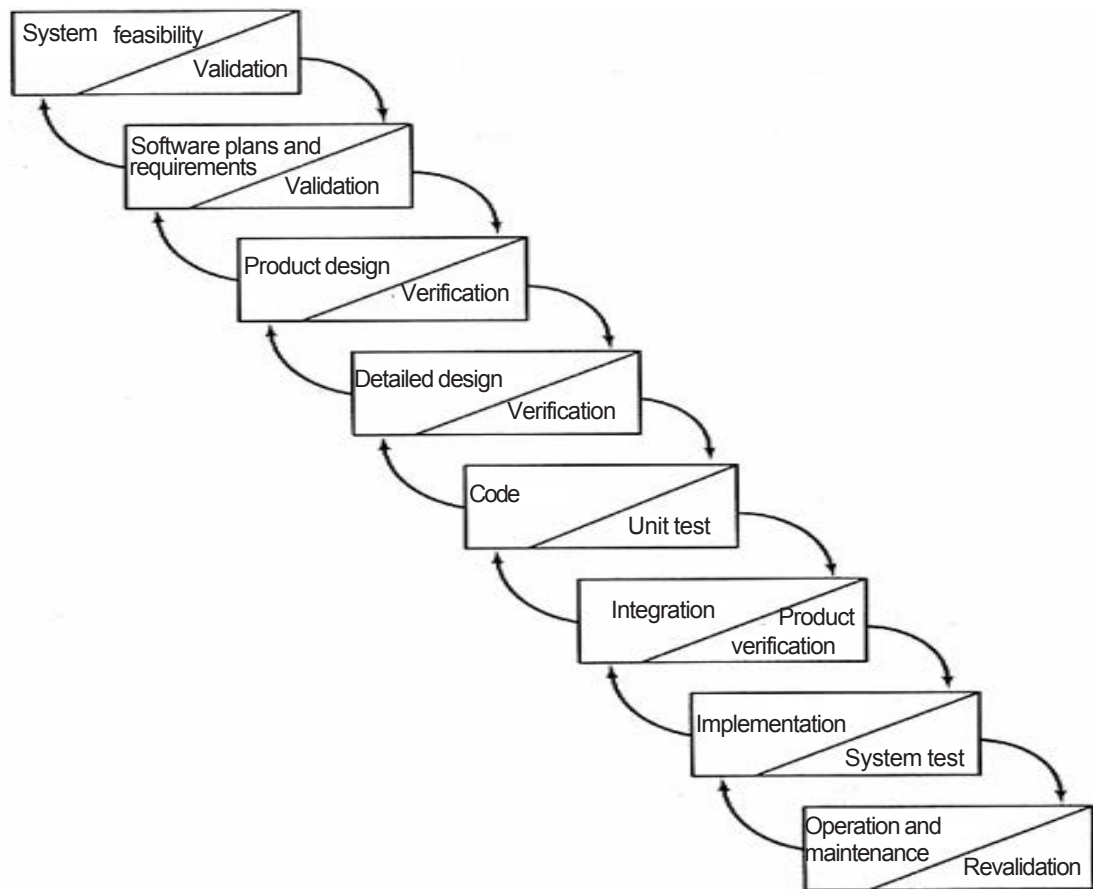


Figure 3.2: The Waterfall Model

➤ **System/Information engineering and modeling**

Software being a part of a larger system (or business), the development process has to first establish the requirements for the entire proposed system, and then identify a part of it to be delivered by the software component. This system view is essential when software must interface with other elements such as hardware, people, and databases. System engineering and analysis encompasses requirements gathering at the system level, with a small amount of top-level analysis and design. A feasibility of the proposed system would have to take place at this stage.

➤ **Software requirements analysis**

The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer (“analyst”) must understand the information domain for the software, as well as the required function, behavior, performance, and interfacing. Requirements for both the system and the software are documented and reviewed with the customer.

➤ **Software design**

It is a multi-step process, focusing on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before code generation begins. Like requirements, the design is documented and becomes part of the software configuration.

➤ **Detailed Design**

During detailed design, the internal logic of each module specified in system design is decided. During this phase, further details of data structures and algorithmic design of each of the modules is specified. The logic of a module is usually specified in a high-level design description language, which is independent of the target language in which the software will eventually be implemented.

Program Design Language (PDL) is one way in which the design can be communicated precisely and completely to whatever degree of detail desired by the designer. It can be used to specify the system design and to include the logic design.

➤ **Coding**

The coding phase translates the design of the system, produced during the design phase, into code in a given programming language. This can be executed by a computer that performs the computation specified by the design. The coding phase affects both the testing and maintenance phase, and aims to implement the design in the best possible manner.

During implementation of this phase, it is essential to keep in mind that the code being written is not only easy to write, but also easy to read and maintain.

All design considerations contain hierarchies. This is a natural way to manage complexity. Coding can either be top-down or bottom-up. In top-down approach, the main module is implemented first, then their subordinates, and so on. In this, implementation starts from the top of the hierarchy and proceeds to the lower level. In a bottom-up implementation, the modules at the bottom of the hierarchy are implemented first, and then the implementation proceeds through the higher levels until it reaches the top.

The advantage of this type of implementation is that the problem is tackled in steps rather than building the whole system in one go. It helps because it is not feasible or desirable to build the whole system in one attempt and then test it. It is better to build parts of the system and test them before putting them together to form the system.

➤ **Testing**

Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, assuring that all statements have been tested, and on the functional externals, that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results. The testing activity is carried out in two phases, the components before integration, and then the integrated product.

➤ **Integration**

Developing a strategy for integration of the components into a functioning whole requires careful planning so that the modules are available for integration when needed. The integrated product can then be tested to demonstrate that the implemented software meets the requirements stated in the requirements document.

➤ **Operations and Maintenance**

Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered; because errors have been encountered, the software must be adapted to accommodate changes in its external environment (for example, a change required because of a new operating system or peripheral device) or because the customer requires functional or performance enhancements. Software maintenance reapplies each of the preceding phases to an existing program rather than a new one.

Each phase in this model ends in a verification and validation activity, the objective of which is to eliminate as many problems as possible in the products of that phase. The linear sequential model is the oldest and the most widely used paradigm for software engineering. For some time the Waterfall model provided personnel in the software field a sequence of common anchor points, a set of milestones around which people can plan, organize, monitor, and control their projects. Many companies, government organizations, and standard groups established interlocking regulations, specifications, and standards based on the model.

Shortfalls

However, criticism of the paradigm has caused even active supporters to question its efficacy. The following problems are sometimes encountered when the linear sequential model is applied:

1. Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.

2. It is often difficult for the customer to state all requirements explicitly. The linear sequential model requires this, and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the program(s) will not be available until late in the project time-span. A major blunder, if undetected until the working program is reviewed can be disastrous.

Developers are often delayed unnecessarily. In an interesting analysis of projects undertaken, it was found that the linear nature of the classic life cycle leads to “blocking states”, in which some project team members must wait for other members of the team to complete dependent tasks. In fact, the time spent in waiting can exceed the time spent on productive work! The blocking states tend to be more prevalent at the beginning and the end of a linear sequential process.

Each of these problems is real. However, the classic life cycle paradigm has a definite and important place in software engineering work. It provides a template into which methods for analysis, design, coding, testing, and maintenance can be placed. The classic life cycle remains the most widely used process model for software engineering. While it does have weaknesses, it is significantly better than a haphazard approach to software development.

➤ The Prototyping Model

This model allows for reduced functionality or limited performance version of the eventual software to be delivered to the user in the early stages of the development process. This helps to make the user requirement more concrete.

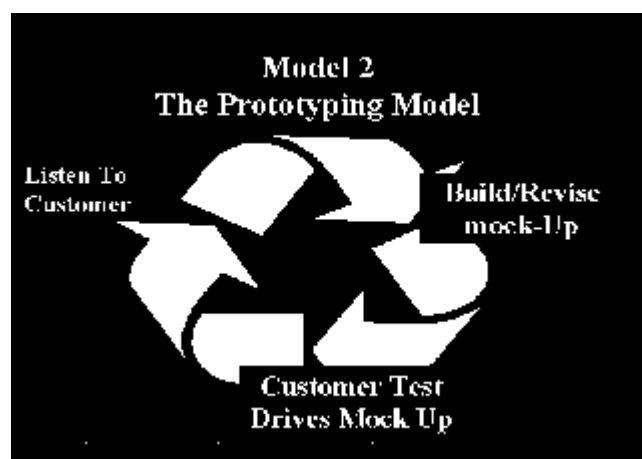


Figure 3.3: A Prototyping Model

A **prototyping model** in Figure 3.3 offers the best approach in cases where the customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In cases where the developer may be unsure of the efficiency of an algorithm, the developer has to clarify requirements. Before proceeding with design and

coding, a throw away prototype is built to give the user a feel of the system. This is used to gather feedback and deliver the final product. The user interacts with the system and specifies the requirements in a more comprehensive manner.

➤ Drawbacks of Linear Models

For a small period in the mid-1970's, it appeared that the software field had found a sequence of common anchor points: a set of milestones around which people could plan, organize, monitor, and control their projects. These were the milestones in the waterfall model, typically including the completion of system requirements, software requirements, preliminary design, detailed design, code, unit test, software acceptance test, and system acceptance test.

Unfortunately, an initial design will likely be flawed with respect to its key requirements. Late discovery of design defects results in costly over-runs and in some cases even project cancellation.

The ideal of a software acceptance test keyed to demonstrating compliance to a set of documented requirements specifications encountered problems as well. Foremost among these was that static tests of input-output transformations provided little insight on whether the software would acceptably support common user task. The acceptance test also failed to cover many off-nominal problem situations subsequently encountered in beta-testing, field-testing, hardware-software integration, or in actual mission operations.

3.3.2 Evolutionary Software Process Models

There is a growing recognition that software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic; tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure; a set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined. In these and similar situations, software engineers need a process model that has been explicitly designed to accommodate a product that evolves over time.

The linear sequential model is designed for straight-line development. In essence, this waterfall approach assumes that a complete system will be delivered after the linear sequence is completed. The prototyping model is designed to assist the customer (or developer) in understanding requirements. In general, it is not designed to deliver a production system. The evolutionary nature of software is not considered in either of these classic software engineering paradigms.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

➤ The Incremental Model

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping. As Figure 3.4 shows, the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces a deliverable “increment” of the software. For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

Incremental Development

.....

Compare waterfall model: Each release is a mini-waterfall

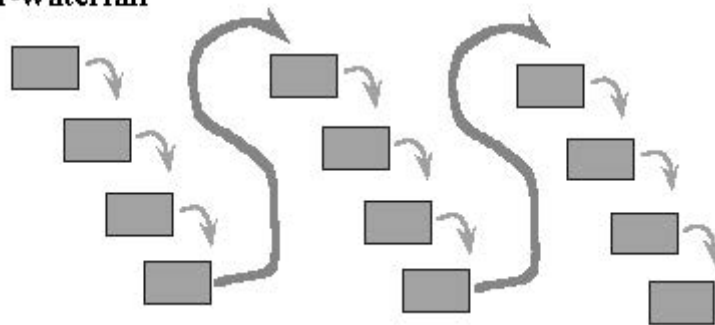


Figure 3.4: Incremental Development

It should be noted that the process flow for any increment can incorporate the prototyping paradigm. When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered.

The customer uses the core product or the product undergoes detailed review. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. However, unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment. Early increments are “stripped down” versions of the final product, but they do provide capability that serves the user and also provide a platform for user evaluation.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project. Earlier increments can be implemented with fewer people. If the core product is well received, then additional

staff (if required) can be added to implement the next increment.

In addition, increments can be planned to manage technical risks. For example, a major system may require the availability of new hardware that is under development and whose delivery date is uncertain. It may be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

Benefits of an Iterative Approach

An iterative approach is generally superior to a linear or waterfall approach for many different reasons:

1. **It tolerates changing requirements.** The reality is that normally requirements change. Requirement change and requirements “creep” have always been a primary source of project trouble, leading to late delivery, missed schedules, unsatisfied customers, and frustrated developers..
2. **Elements are integrated progressively.** Integration is not one “big bang” at the end. The iterative approach is almost a continuous integration. What used to be a long, uncertain, and troublesome time, taking up to 40% of the total effort at the end of a project, is now broken into six to nine smaller integrations that begin with fewer elements.
3. **Risks are mitigated earlier** because integration is generally the only time risks are discovered or addressed. As we unroll the early iterations we go through all core process workflows, exercising many aspects of the project: tools, off-the-shelf software, people skills, and so on. Some perceived risks may prove not to be risks at all, and new, unsuspected risks may emerge.
4. **It allows the organization to learn and improve.** Team members are able to learn along the way, and the various competencies and specialties are more fully employed during the whole lifecycle. Testers start testing early, technical writers write early, and so on. In non-iterative development, the same people will wait to begin their work, make plans, and hone their skills. Training needs, or the need for additional (perhaps external) help is spotted early on during assessment reviews.
5. **It facilitates reuse** because it is partially designed or implemented, instead of identifying all commonality up front. Identifying and developing reusable parts is difficult. Design reviews in early iterations, allows architects to identify unsuspected potential reuse, and develop and mature common code in subsequent iterations.
6. **It results in a more robust product** because we are correcting errors over several iterations. Flaws are detected even in the early iterations as the product moves beyond inception. Performance bottlenecks are discovered when they can still be addressed, instead

of on the eve of delivery. The outcome is higher level of quality.

7. **It tolerates tactical changes in the product.** For example, to compete with existing products. We can decide to release a product with reduced functionality earlier to counter a move by a competitor, or we can adopt another vendor for a given technology.
8. **The process itself can be improved and refined along the way.** The assessment at the end of an iteration not only looks at the status of the project from a product and schedule perspective, but also analyzes what should be changed in the organization, and in the process to perform better in the next iteration.

A customer once said: “With the waterfall approach, everything looks fine until near the end of the project, sometimes up until the middle of integration. Then everything falls apart. With the iterative approach, it is very difficult to hide the truth for very long”. Unfortunately, project managers often resist the iterative approach, seeing it as a kind of endless “hacking”.

9. **Accommodates changes.** It allows us to take into account changing requirements. Users will change their mind along the way.

This is part of human nature. Forcing the users to accept the system as they originally imagine it is wrong. They change their mind because the context changes, they learn more about the environment, the technology, and they see intermediate demonstration of the product, as it is being developed.

10. **It allows technological changes on the way.** If the technology changes or it becomes a standard, or new technology appears, the project can take advantage of it. This is particularly true for platform changes or lower-level infrastructure changes.
11. **Increasing reuse:** It facilitates reuse, because it is easier to identify common parts as they are partially designed or implemented, compared to identifying all commonality up front. It is difficult to identify and develop reusable parts. Design reviews in early iterations allow architects to identify unsuspected, potential reuse, and subsequent iterations allow us to develop and mature this common code.
12. **It is easier to take advantage of commercial-off-the-shelf (COTS) products.** We have several iterations to select them, integrate them, and validate that these fit into the architecture.
13. **Learning:** The developers can learn along the way, and the various competencies and specialties are fully employed during the whole life cycle. Testers start testing early, technical writing starts early, and so on, rather than waiting for a long time, and simply making plans and honing skills. The need for additional training or external help can be detected

in the early iteration assessment reviews.

The process itself can be improved, and refined as it develops. The assessment at the end of iteration not only looks at the status of the project from a product-schedule perspective, but also analyzes what should be changed in the organization, and the process to perform better in the next iteration.

- 14. It results in a more thoroughly tested product.** The critical functions have had many opportunities to be tested over several iterations. The tests themselves, and any test software will have had time to mature, as opposed to tests run once towards the end of the project.

Problems with Incremental Models

The critical milestone in evolutionary development is the initial release: a package of software with sufficient capability to serve as a basis for user exercise, evaluation, and evolutionary improvement. However, this “initial release” milestone frequently has the following problems:

- 1. Inflexible point-solutions**

Frequently, the initial release is optimized for initial demonstration-mode and exploratory-mode success. For example, it may store everything in main memory to provide rapid response time. Then, when users want to transition to large-scale use, the initial point-solution architecture will not scale up.

- 2. High-risk downstream capabilities**

Frequently, the initial release will defer such considerations as security, fault-tolerance, and distributed processing in the interest of providing early functionality and user interface capabilities. The users may like the results, and expect the deferred considerations to be delivered equally rapidly. Usually, this puts the project in big trouble because the initial release’s architecture cannot be easily extended to support the desired security, fault tolerance, distributed processing, or other key considerations.

- 3. Off-target initial release**

Evolutionary developers often begin by saying, “Let us find out what the user needs by building an initial release and seeing what the users want improved.” The lack of initial user activity analysis frequently leads to an initial release, which is so far from user needs that the users never bother learning and using it.

➤ The Spiral Model

These difficulties with the waterfall and evolutionary development models have led to the development and use of a number of alternative process models: risk-driven (spiral), reuse-driven, legacy-driven, demonstration-driven design to cost or schedule, incremental, and hybrid combinations of these with the waterfall or evolutionary development models.

Boehm originally proposed the Spiral model in 1988 as an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. This was seen as an attempt to combine the strengths of the various models. It provides the potential for rapid development of incremental versions of the software. The model also takes into account results of risk assessment activities whose outcome determines taking up of the next phase of the development activity.

This model views software development as a spiral process with the software being developed in a series of incremental releases. Typically, the inner cycles represent the early phases of requirement analysis along with prototyping to refine the requirement definitions, and the outer spirals are progressively representative of the classic software life cycle. The spiral model is divided into a number of framework activities, also called **task regions**. Typically, there are between three and six task regions:

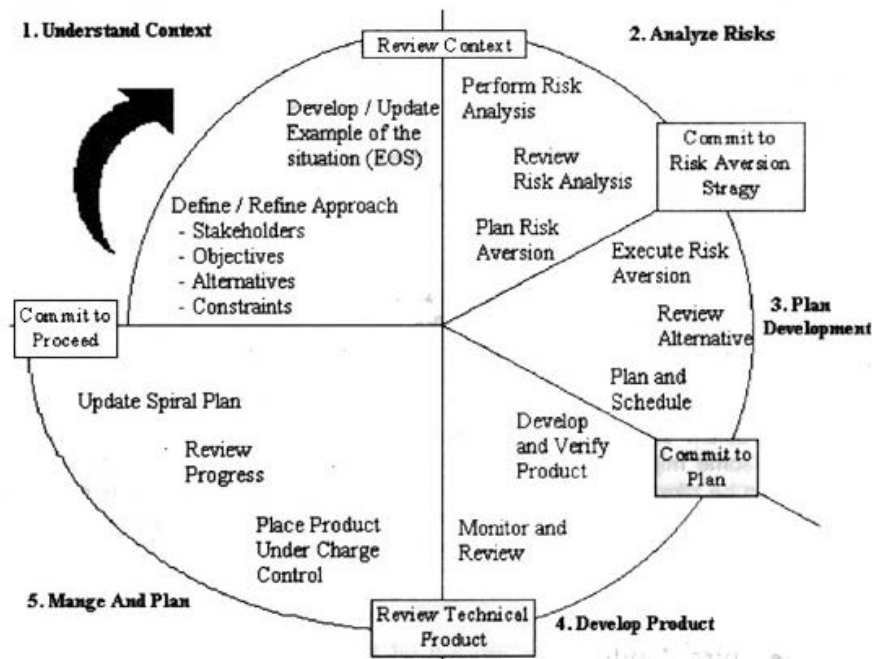


Figure 3.5: Spiral Model

Figure 3.5 depicts a spiral model that contains six task regions:

- **Understanding the Context-** These tasks are required to establish effective communication between developer and customer.

- **Planning-tasks** - They define resources, timelines, and other project related information.
- **Risk analysis-tasks** - These are required to assess both technical and management risks.
- **Engineering-tasks** - These are required to build one or more representations of the application.
- **Construction & release**-These tasks construct, test, install, and provide user support (for example, documentation and training).
- **Customer Evaluation-tasks** – These are required to obtain customer feedback based on evaluation of the software representations created during the engineering stage, and implemented during the installation stage.

Each of the defined six regions has a series of work tasks. These tasks are dependent on the type of project being undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality. In addition, all support activities like Configuration Management, Project Management have to happen.

As this evolutionary process begins, the software engineering team moves around the spiral in a clockwise direction, beginning at the core. The first circuit around the spiral may result in the development of a product specification; subsequent passes around the spiral may be used to develop a prototype, and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

The spiral model is a realistic approach to the development of large-scale systems and software. Since software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism. It also enables the developer to apply the prototyping approach at any stage in the evolution of the product.

It maintains the systematic stepwise approach suggested by the classic life cycle, but incorporates it into an iterative framework that more realistically reflects the real world. The spiral model demands a direct consideration of technical risks at all stages of the project, and if properly applied, should reduce risks before they become problematic.

However, like other paradigms, the spiral model is not a solution to all the problems faced by software developers. It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise, and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur. Finally, the model itself is relatively a new paradigm. It will take a number of years before the efficacy of this important new paradigm can be determined with absolute certainty.

➤ The Component Assembly Model

Component-based development is a variation on general application development, in which:

- The application is **built from discrete executable components**, which are **developed and deployed relatively independently of one another**, potentially by different teams.
- The application may be **upgraded in smaller increments** by upgrading only some of the components that comprise the application.
- Components may be shared between applications, creating opportunities for **reuse**, but also creating **inter-project dependencies**.

One of the various ways of approaching software development is trying to model the real world as closely as possible. The object-oriented paradigm that is used to model real world situations as a series of objects emphasizes the creation of classes that encapsulate both data and the algorithms that are used to manipulate the data.

Object technologies provide the technical framework for a component-based process model for software engineering. If properly designed and implemented, object-oriented classes are reusable across different applications and computer-based system architectures.

The component assembly model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an interactive approach to the creation of software. However, the component assembly model comprises of applications from prepackaged software components, sometimes called “classes”.

The engineering activity begins with the identification of candidate classes. The data that are to be manipulated by the application and the algorithms that will be applied to accomplish the manipulation are examined to identify the required components. Corresponding data and algorithms are packaged into a class.

Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods. The first iteration of the application to be built is then composed, using classes extracted from the library and any new classes built to meet the unique needs of the application. Process flow then returns to the spiral, and will ultimately reenter the component assembly iteration during subsequent passes through the engineering activity.

The component assembly model leads to software reuse, and reusability provides the software engineers with a number of measurable benefits.

Based on reusability studies, reports indicate that component assembly leads to a 70% reduction in development cycle time, a 84% reduction in project cost, and a productivity increase of 26.2, compared to an industry norm of 16.9. Although these results are a function of the robustness of the component library, there is little question that the component assembly model provides significant advantages for software engineers.

3.4 Choosing A Process Model

Software development organizations are increasingly realizing that a core set of tasks need to be done irrespective of the process model they follow. Depending on the time frame available or execution of project, the type of product/project being developed, the detail in which specifications are available, the organizations having previous experience with similar projects, and so on, these tasks are organized in differing sequences.

Organizations choose the process paradigm they want to follow in developing the software. In case different processes are followed, outputs in terms of productivity and quality also vary. Processes tend to become less predictable and past data cannot be used for estimation, or new projects learn from the failures of old projects. Standardized processes are essential for predictability, good estimation capability, and good learning.

Organizations may find it difficult to follow a single standard process for all its projects. One approach is to define a standard process that describes a sequence of activities or a task, but allows for tailoring to suit a particular project. Guidelines for tailoring are provided, and the types of processes that can be followed are restricted to the allowable variations.

3.4.1 Process Tailoring

Tailoring is defined as the process of adjusting the standard process of an organization to obtain a process that is suitable for the particular business or technical needs of a project. Documented guidelines are in many cases provided on the reason of deviation and to what extent deviations are possible from the standard processes so as to achieve the project goals.

Process modification requires that the unique aspects of the project are identified first, and then these features are used to make adjustments to the standard process. Tailoring guidelines try to capture the experience and judgment of people regarding how tailoring can be done effectively. The past experience of people who have used the process becomes available so that even a newer person, who may not have the necessary experience, can select the optimum process for a project.

The process may be tailored with respect to its scope, formality, frequency, and granularity. For each process element that can be tailored, one needs to define the options that are available and the circumstances in which each of these options can be exercised.

Tailoring can happen at a macro (Summary- How certain general activities are performed) level or at a micro level (Detailed –all activities in a process for the various phases and how they can be tailored).

Some of the factors that may need to be considered for tailoring the process include the skill level of the

team, the peak team size, and the criticality of the application.

3.5 The Process Model for the Web

It seems that just about any important product, service, or system - most Web applications (WebApps) fall into this category, is worth engineering. Before we start building it, we should better understand the problem from the user's point of view, design a workable solution, implement it in a solid way, and test it thoroughly. We should also use impeccable project management techniques, control changes as we work, and have a mechanism that ensures quality end result.

Of course, there is another view. Some argue that the application of software engineering principles (even if they are adapted to accommodate the unique characteristics of WebApps) is simply wrong; that software engineering does not really work very well in the fast world of WebApp development.

3.5.1 Web Engineering

Web Engineering is an adaptable and incremental (evolutionary) process. It is populated by a set of framework activities that occur for all business-critical WebApp projects, regardless of the size or complexity. The following framework activities may be considered for Web engineering:

- **Formulation** - Identifies the goals and objectives of the WebApp and establishes the scope for the first increment.
- **Planning** - Estimates overall project cost, evaluates risks, and defines a development schedule for the initial WebApp increment.
- **Analysis** - Establishes requirements for the WebApp and identifies the content that will be incorporated.
- **Modeling** - Incorporates two parallel tasks sequences: content and user interface design. Content design involves the creation or acquisition of content for the WebApp. User interface design involves defining the architecture, navigation, and graphic design, which establish the structure of the WebApp and the flow of the user's interaction with it.
- **Page generation and testing** - Constitute requisite construction activities.
- **Customer evaluation** - Reviews each increment produced as part of the Web engineering process. At this point, changes are requested (scope extensions occur). These changes are integrated into the next path through the incremental process flow.

3.6 The Process Technology

We have discussed process models that organizations can follow at a conceptual level. How does one implement the same in a development organization? To accomplish this, process technology tools have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality. The **Rational Unified Process** is for example, a widely used tool to implement the incremental approach.

Process technology tools allow a software organization to build an automated model of the common process framework, task sets, and umbrella activities discussed earlier. The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that may lead to reduced development time or cost.

Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering tasks defined as part of the process model. Each member of a software project team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted. The process technology tool can also be used to coordinate the use of other computer-aided software engineering tools that are appropriate for a particular work task.



SUMMARY

- Software engineering is a discipline that integrates process, methods, and tools for the development of computer software.
- A number of different process models for software engineering have been proposed, each exhibiting strengths and weaknesses, but all having a series of generic phases in common.
- Depending on the type of development project(s) they may be undertaking, organizations use the discipline provided by these frameworks to successfully achieve project goals.
- Any process model goes through:
 - Requirements definition
 - Requirement analysis
 - Preliminary design
 - Detailed design
 - Coding
 - Testing
 - Integration
 - Operations and maintenance
- The software process models can be broadly categorized into:
 - Linear process models
 - Evolutionary software process models



CHECK YOUR PROGRESS

1. Software has always been represented as ____.
2. Software differs from other products in that it is not a physical entity. **[True/False]**
3. Software exists as a ____ entity.
4. Software Engineering essentially consists of a set of steps that comprises of methods, tools, and procedures that assist a Software Factory Manager to manufacture software. **[True/False]**
5. ____ can be planned at the end of the Requirements Definition activity.
6. The ____ phase concludes with a *Software Specifications Review (SSR)* at which the results of the analysis are presented for evaluation.
7. The **Program Design Language (PDL)** is one way in which design can be communicated precisely and completely to whatever degree of detail desired by the designer. **[True/False]**
8. The incremental model combines elements of linear sequential model (applied repetitively) with the iterative philosophy of prototyping. **[True/False]**
9. The ____ model takes into account results of risk assessment activities whose outcome determines taking up of the next phase of the development activity.
10. The ____ model focuses on the delivery of an operational product with each increment.

ASK to LEARN

Onlinevarsity
your e-way to learning

Questions
in your
mind?



are here to **HELP**

Post your queries in **ASK to LEARN** @

www.onlinevarsity.com

Chapter 4

Requirements Gathering and Analysis

Objectives

At the end of this chapter, you will be able to :

- *Describe the process of System Analysis*
- *Explain how a Feasibility Study is undertaken*
- *Describe the scope and activities of the Requirements Analysis phase of SDLC*
- *Explain the process of Requirements Gathering*
- *Describe the process of Analysis Modeling*
- *Explain the rationale behind Essential Model*

4.1 Introduction

In this chapter on Analysis, we will discuss the various activities related to the Analysis phase. We will understand the right techniques of conducting a feasibility study before starting any kind of development work. The feasibility study findings and information gathered from the customer during the process of requirements gathering forms a baseline for defining the system functionality. Having finalized the system requirements and functionality, we can estimate the efforts and cost involved in developing the proposed system.

The Software Development Lifecycle starts with the **Problem Definition** phase wherein the client's problem is identified and understood. Having identified the problem, it is now time to start developing the system. The process of system development starts with **System Analysis**. It is known that "A System is a collection of interrelated components that work together to achieve some common objective" and "System Analysis is the specification of what the system is required to do".

System Analysis is conducted with the following objectives in mind:

- Identification and analysis of customer needs
- System evaluation for feasibility
- Performing economic and technical analysis

Chapter 4

Requirements Gathering and Analysis

- Allocating functions to manpower, database, hardware, software and other system elements
- Establishing cost and schedule constraints
- Creating a system definition that forms a foundation for all subsequent development activities

System Analysis is an art, and needs immense patience backed by years of experience to get the desired system functionality and performance. Let us learn how to master this art.

4.2 Feasibility Study

There is a very old story about a very capable tailor. Once he used all his skill and insight to stitch a beautiful coat. When a prospective buyer pointed out that the sleeve was rather short, the tailor nonchalantly replied that the coat was so beautiful, surely the gentleman wouldn't mind cutting his hand to fit into it!

Most of us in the computer industry are often accused of having a similar affliction. That automation is indeed the solution to most organizational problems, seems to be a foregone conclusion for many a systems engineer. It is in order to temper such an over enthusiastic prescription of automation as a panacea to all problems that the Software Engineering practice has inbuilt checks and controls. One such first and foremost step is the **Feasibility Study**.

All projects are feasible – given unlimited resources and infinite time! Hence, before proceeding to commit on a large expenditure that is involved in the development of a new system, it is important for any organizations to be reasonably confident that the proposed system is indeed going to deliver the goods and solve the problems on hand within the expected schedules and costs. A feasibility study is a means of enabling this.

A Feasibility study is conducted to find out whether the proposed system will be:

- **Possible** – to build it with the given technology and resources
- **Affordable** – given the time and cost constraints of the organization
- **Acceptable** – for use by the eventual users of the system

Feasibility study provides answers to troublesome questions like – “Can I undertake this project?”, “What are my market constraints?”, “What is competition in this field like?”, “How do I price my product?”, and “What benefits will accrue from this project?” .

4.2.1 Purpose of a Feasibility Study

A feasibility study is initiated by an organization, which feels there is a need for change in the current system of functioning. This need may be felt due to lapses in the current system that may have been brought to light, or the organization may wish to undergo a change in its system in order to introduce better, more efficient systems. A feasibility study is needed to ascertain the pros and cons of instituting a new system.

Chapter 4

Requirements Gathering and Analysis

The purposes of this study are to:

- Determine the need for change within an organization (need analysis)
- Study the effect of the change on the economics of the organization (cost benefit analysis)
- Evaluate various technologies that can be used to implement the suggested change, given the cost and resource constraints of an organization (technical feasibility)
- Evaluate the legal procedures, if any should come into play to implement the suggested change (legal feasibility)
- Evaluate the various alternatives that will be thrown up with regard to resolving the problems of an organization, and recommend the best suited one (evaluating alternatives)

The purpose of a feasibility study is to evaluate alternatives without bias. Therefore, it is possible that the outcome of a feasibility study may be a recommendation to continue with an improved manual system instead of instituting an automated system.

A feasibility study is not justified for projects where benefits out-weigh costs, technical risks are not high, and there are no alternatives to be suggested. For such projects, conducting a feasibility study will add to unnecessary expenditure of time and money for the study itself.

Objective	Chapter
Implementing, Monitoring and Troubleshooting Security	7
Encrypt data on a hard disk by using Encrypting File System (EFS).	8
Implement, configure, manage and troubleshoot policies in a Windows 2000 environment:	13
• Implement, configure, manage and troubleshoot Local Policy in a Windows 2000 environment.	
• Implement, configure, manage and troubleshoot System Policy in a Windows 2000 environment.	
Implement, configure, manage and troubleshoot auditing.	
Implement, configure, manage and troubleshoot local accounts.	
Implement, configure, manage and troubleshoot Account Policy.	
Implement, configure, manage and troubleshoot security by using Security Configuration Tool Set.	

4.2.2 The Stage in which a Feasibility Study is to be Conducted

A feasibility study is conducted to understand the issues that face a project before the project is undertaken.

It is best initiated right at the beginning of a project. It will be the least inexpensive and most useful to do so. If a feasibility study is conducted when the Analysis phase of the project is underway or later, then considerable time and money will have been spent on the project already, and the findings of the feasibility study, if contrary, will be that much more difficult to implement. Feasibility can always be re-evaluated and reassessed at various stages of a project. It is not very often that there are fundamental differences at a later stage, which may result in shelving the entire project.

4.2.3 Different Aspects of Conducting a Feasibility Study

Feasibility and risk analysis are related in many ways. If project risk is great, the feasibility of producing quality software is reduced. The different aspects related to Feasibility Study are:

- 1 Need Analysis
- 2 Economic Feasibility
- 3 Technical Feasibility
- 4 Legal Feasibility
- 5 Evaluation of Alternatives

1. Need Analysis

A *Need Analysis* is conducted with the following objectives in mind:

- **Seeking background information on the organization-** The people conducting the feasibility study need to gather information on the industry to which the organization belongs. They should also try and find out similar cases of automation, the plus and minus points of the same in order to support their position, and also table the weaknesses of other such systems, so that the same may not be repeated.
- **Understanding current issues to be tackled-** This involves a preliminary study to be conducted so that the study team is able to ascertain problem areas and determine the scope of the system.
- **Understanding the user profile -** The feasibility study is an activity that almost never percolates to the grass roots of an organization; it is more or less a senior management issue. So there is a tendency to ignore a new system's impact on the lower levels of an organization chart. A majority of users are placed here. Not only are the users at the lower level of the organization chart more aware of the true problems of the system, they are also the people who would be involved in the day to day running of the new system, when installed. So it is important to understand their aspirations, idiosyncrasies, things that please or displease them.

2. Economic Feasibility

While performing Economic Feasibility, one attempts to weigh the costs of developing and implementing a new system against the benefits to be accrued from having the new system in place.

Several *costs* as well as *benefits* of a system are considered when studying the economic feasibility of a system. The study of costs and benefits is called **Cost Benefit Analysis**.

When benefits outweigh costs, a system is said to be **economically feasible**. It includes both the tangible and the intangible benefits. Tangible benefits can be measured in money value and result in increased revenue and decreased cost. Intangible benefits are difficult to quantify, but their effect is seen in various areas of business performance like:

1. Better market positioning in comparison to the competition
2. Improved service to customers due to correct information on time
3. Improved service to customers results in good word of mouth, which in turn results in more business (a tangible benefit in due course)

The costs associated with most systems fall under the following broad categories:

- **Procurement Costs** – this type of cost deals with the purchase and installation of equipment, cost of setting up (new or modifying an existing one) site to install the equipment, cost of capital that will be spent (interest is calculated on the money that is invested for projects till the project is up and running), and the cost of staff dealing with procurement activities.
- **Start-up Costs** – these costs are incurred in the early days of the project. Cost of hiring and setting up manpower for the project, cost of setting up communication systems (telephone lines etc.), and operating system software are some of the costs that fall under this category
- **Project Costs** – these costs are incurred when the project is underway; once the project is complete and the system up and running none of these costs are relevant.
- **Ongoing / Recurring costs** – this includes site rental, cost of the personnel associated with running of the system, cost of depreciation on hardware, cost of maintenance of hardware, cost of consumables – floppies/tapes, printing stationary etc.

3. Technical Feasibility

It helps in understanding the level and kind of technology needed for a system. It includes functions, performance issues, and constraints that may affect the ability to achieve an acceptable system.

Technical feasibility entails an understanding of the following:

- Different technologies (hardware platform and software environment) involved in the proposed system
- Existing technology levels within the organization

4. Legal Feasibility

Legal feasibility entails copyright violations for systems that have to be developed for the open market, framing of contract for large systems and the violation of terms. There are two aspects to remember about legal feasibility:

- It is not required for all systems, when required it is usually to frame a contract of service between two parties: the consultants hired to develop the system, and the organization for whom the system is to be developed.
- To ascertain legal feasibility, legal experts have to be called in; in many organizations company secretaries can help out. It is beyond the expertise levels of the analyst conducting the study.

5. Evaluation of Alternatives

It includes an evaluation of alternative approaches to the development of a system. The option with the lowest cost and maximum returns is considered the most feasible option. However, a number of qualitative and intangible issues also greatly influence this decision.

4.2.4 Outcome of a Feasibility Study

The outcome of this activity is a **Feasibility Report** that summarizes the findings of the feasibility study and presents the management with various implementation options. Each option is presented along with its costs and benefits and also the constraints of implementing the option.

The intent of this report is to provide the senior management of an organization the economic justification, coupled with implementation constraints for having a new system in place. It is an important input to the management to give the go-on / no-go decision for a project.

The contents of a Feasibility Report includes:

Chapter 4

Requirements Gathering and Analysis

No.	Contents
1.	Introduction <ul style="list-style-type: none">➤ Statement of problem that describes the problem, as understood by the consulting team➤ Implementation environment – for the would-be system➤ Implementation environment – for the would-be system.➤ Constraints on the implementation environment
2.	Management summary and recommendations <ul style="list-style-type: none">➤ Important findings are those findings that are out of the ordinary and have not been unearthed by consultants during discussions with the client, but discovered during preliminary investigations later. It may be necessary to keep such issues in the foreground during the initial discussions between the consultants and the users.➤ Comments on the state of functioning of the current system, the problem areas, problematic attitudes, and lacunae in the system etc are included.➤ Recommendations would be the consultant's view of resolving the issues at hand. The consulting team will recommend an alternative to the existing system with reasons for doing so. Alongside, they will compare the suggested alternative with other alternatives that may have come up while conducting the study. Each recommendation must have a suitable justification.➤ Impact of the recommendations given by the consultant.
3.	Alternatives <ul style="list-style-type: none">➤ Alternative system configurations – Keeping the above mentioned environment in mind, the consulting team could suggest the alternative specific system configuration.
4.	System Description <ul style="list-style-type: none">➤ Abbreviated statement of scope is a very important statement as it has life long bearing on the project. The scope of the system should be stated very clearly; the consultants should spend time on framing this statement. The scope may be phased out, that is to say that a part of the system may be developed in phase 1 and the other in phase 2.➤ Feasibility of allocated elements – The consultants may proceed with assumptions of availability of allocated resources, which may be different from reality.
5.	Cost and benefit analysis <ul style="list-style-type: none">➤ The details of costs and benefits (as discussed above) should be stated here.

6.	Evaluation of technical risk <ul style="list-style-type: none">➤ The details of technical risks as perceived by the consulting team should be outlined here.
7.	Legal ramifications <ul style="list-style-type: none">➤ Details of the legal feasibility as discussed above should be mentioned here.

From the discussion so far, we can gauge the importance of conducting the feasibility study in the right perspective. Its main intent is to provide the senior management of an organization the economic justification, coupled with implementation constraints for having a new system in place. Figure 4.1 summarizes the process of feasibility study and its associated activities:

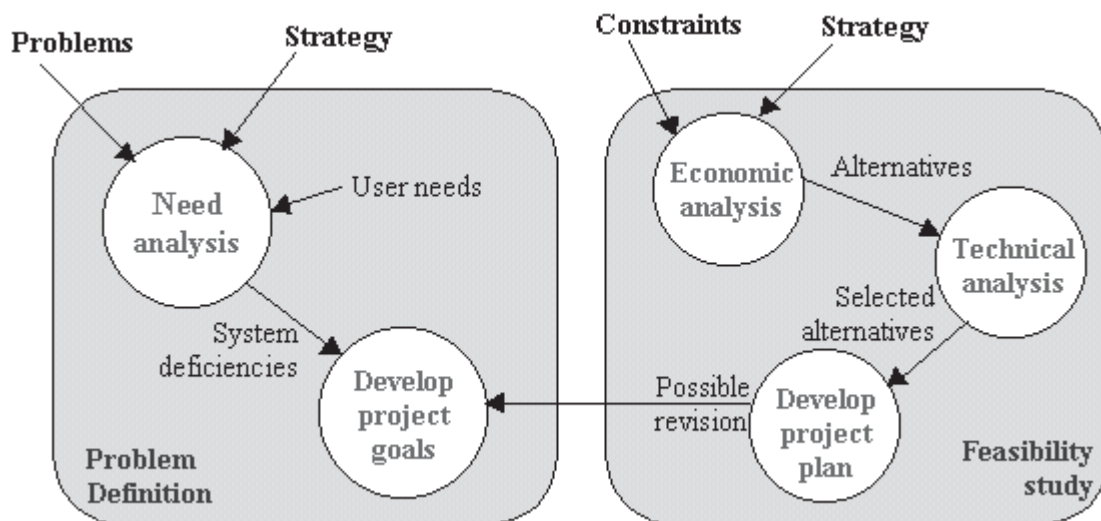


Figure 4.1: Feasibility Study

As seen from figure 4.1, a feasibility study is initiated by an organization when a *need* for change is felt in the current system of functioning. This need is thoroughly analyzed in order to understand the deficiencies of the current system, and identify the objectives and goals of the proposed system. A Project Plan is then prepared taking into consideration the various issues related to economic, technical, and legal feasibility. All possible alternatives are evaluated and a strategy is charted out for further course of action.

4.2.5 Evolving the Project Proposal

From the feasibility report, a project proposal is prepared. It is a proposal from the consulting team after the go ahead for the new system is given, which outlines the characteristics of a system and gives first-cut calendar schedule.

Chapter 4

Requirements Gathering and Analysis

The objectives of preparing a project proposal are:

- Indication of first-cut calendar schedules for the project
- Description of application areas for dealing with identified problems
- Phase wise demarcation of activities
- Outlining hardware, software, manpower costs
- Identification of training needs
- Calculation of total costs of the project
- Enumeration of expected benefits

Based on the above, the management decides whether or not to accept the proposal. Once they decide to accept it, the stage is set for proceeding to the next phase of requirement analysis.

4.3 The Requirement Analysis

Requirements play an essential role in the software-engineering life cycle. One of the most difficult aspects of program development is in getting both the customers and the developers to understand what each of them is trying to say. The problem gets even more complicated when the users themselves do not know the needs and requirements of the system. Many software projects have failed due to certain Requirements Engineering issues, such as poorly documented requirements or requirements that are impossible to satisfy or that which failed to meet user needs.

The identified user requirements are usually informal and vague. The requirement phase translates these informal inputs from the users into a set of formal specified requirements. The process of requirement analysis helps to bridge the communication gap between the customer and the developer. The technical development team works with the customers and system end-users to identify the application domain, functions, services, performance capabilities, hardware constraints, and so on related to the system to be developed.

The two major activities of this phase are:

- **Detailed investigation** is the process of understanding the working of the current system in detail.
- **Analysis or determination of system requirements** is the process of determining the requirements of the new system. These requirements should satisfy client needs.

The Requirements Analysis phase defines **WHAT** a system must do and the Design states **HOW** to do it. In this section, we will learn to analyze that **WHAT**. The **HOW** part will be discussed in a later chapter.

4.3.1 The Requirement Specification Document (RSD)

Once the problem is analysed and the essentials understood, the requirements must be specified in the **Requirement Specification Document (RSD)**.

The contents of the RSD are:

- Functional and performance requirements of a system
- Input-output formats
- Design constraints

After a system has been analysed and its requirements determined, they must be placed in the context of the functions that will be ultimately incorporated in the system. Functional analysis relates to the precise definition of the functions and sub-functions to be performed by the system. The functions have certain performance requirements, as also certain design constraints that have to be borne in mind.

Example: A car is to be developed for the middle-income group of the population. Initial study is conducted with the target group of the population. Requirements state that it must be an affordable vehicle, fuel efficient, easy to run, good braking system for heavy traffic manoeuvres, and compact size for easy parking. The proposed solution should consider:

- *Functions of the car* – power brakes, easy shift floor gear system, power steering
- *Performance requirements* – Fuel-efficient engine to give 'x' miles per gallon
- *Design constraints* – size of the chassis

Based on the above requirements and expected functionality, a suitable car is designed for the middle-income population.

Figure 4.2, summarizes the process of Requirements Analysis. It starts with the process of identifying the user requirements and proposing a solution in the form of **Software Requirements Specification** document or **Project Proposal** document:

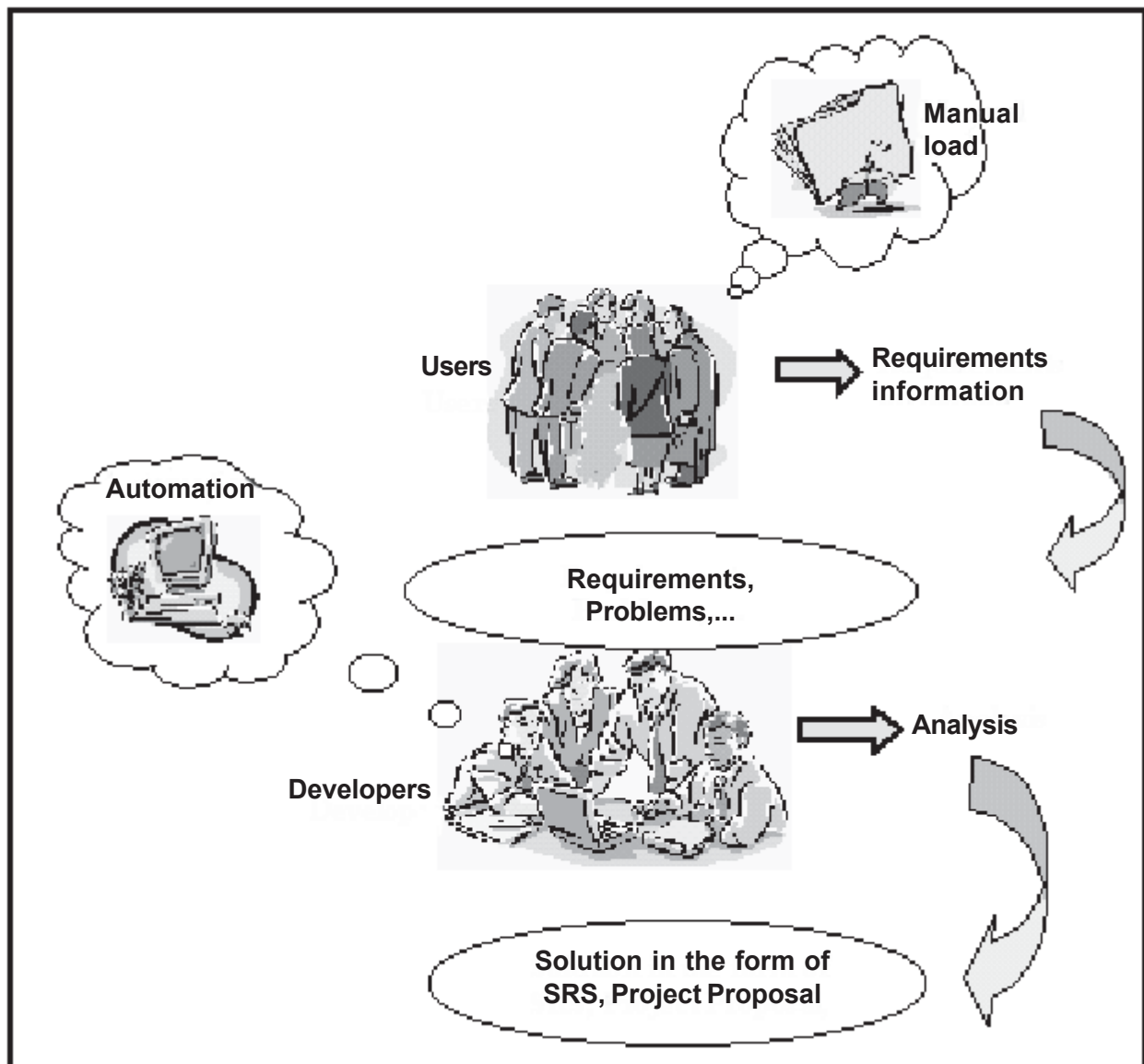


Figure 4.2: The Requirement Analysis process

4.4 The Requirements Gathering

The basic goal of requirement analysis is to **gather requirements**, that is, obtain a clear idea of the client requirements. Requirement gathering involves conducting a preliminary meeting with the client to understand and analyze the client's needs. The different techniques used here are:

- **Interviewing**- It is an oft-used technique when an analyst tries to get information regarding a problem from managers, department heads, and functional heads. The analyst will ask the customer a set of context free questions, which will lead to a basic understanding of their problem and nature of the solution that is desired.

The *context free questions* can be:

- “Whom will the requirements cater to?”
- “What will be the economic benefit if we come out with a successful solution?”
- “Describe the environment in which the solution is to be used.”
- “Are the questions which I have been putting up relevant to the problem that on hand?”

Such questions help to focus on the overall goals, get a better understanding of the problem, the customer, and the effectiveness of the meeting.

- **Questionnaires** - Are used when a wide spectrum of responses is required and the consultants need definite answers to certain questions related to the problem area. For instance, getting responses from workers at a factory site. This is an ideal situation for floating questionnaires as a lot of time can be taken up interviewing such a large group. Moreover, interviews could result in vague and varied responses.
- **Record Reviews** - Is a technique used in all projects. It is only when existing records are reviewed that an analyst can understand the data, its movement, and usage within an organization. This cuts down on the time spent in asking people within the organization basic questions that relate to the day to day functioning of a system.
- **Observation** – Helps the analyst in detecting problems that exist in the current system. Users are either unaware of the problems or they are hesitant to talk about them to outsiders, thinking that they may be misinterpreted. Even when willing to talk about a problem, the user may ignore certain minute and delicate details, which may be of extreme importance to the analyst. Such problems may come to light when an analyst observes the way an organization functions.
- **Facilitated Application Specification Techniques (FAST)** – Is based on the team-oriented approach to requirements gathering wherein a joint team of users and developers work together to identify the problem, propose the probable solution, identify the different strategies and approaches to solve the problem.
- **Quality Function Deployment (QFD)** - Is a quality management technique, which translates the needs of the user into technical requirements for software. QFD emphasizes on understanding what is valuable to the customer and then deploying these valuables throughout the engineering process. According to QFD, there are three types of requirements:
 - Normal Requirements
 - Expected Requirements
 - Exciting Requirements

Chapter 4

Requirements Gathering and Analysis

Once the requirements have been gathered, the development team needs to examine and categorize them. They need to prioritize the requirements and give a solution stating which part of the system should be automated first, so that a major part of the problem area can be taken care of.

4.4.1 Transition from Requirements Gathering to Analysis

The information gathered has to be analyzed. There are some general principles that can be applied to analyze the problem systematically. They are:

- Information domain of the problem needs to be well understood
- The expected system functionality needs to be identified
- The model to be built to depict system information, functionality and behavior must be identified
- The model must be partitioned so as to expose the details in a layered manner
- The analysis process should move from essential information representation to complete system implementation in user environment.

Figure 4.3 explains this process of understanding the problem domain, based on which an Analyst applies his knowledge and experiences to build a suitable model or a view of the proposed system:

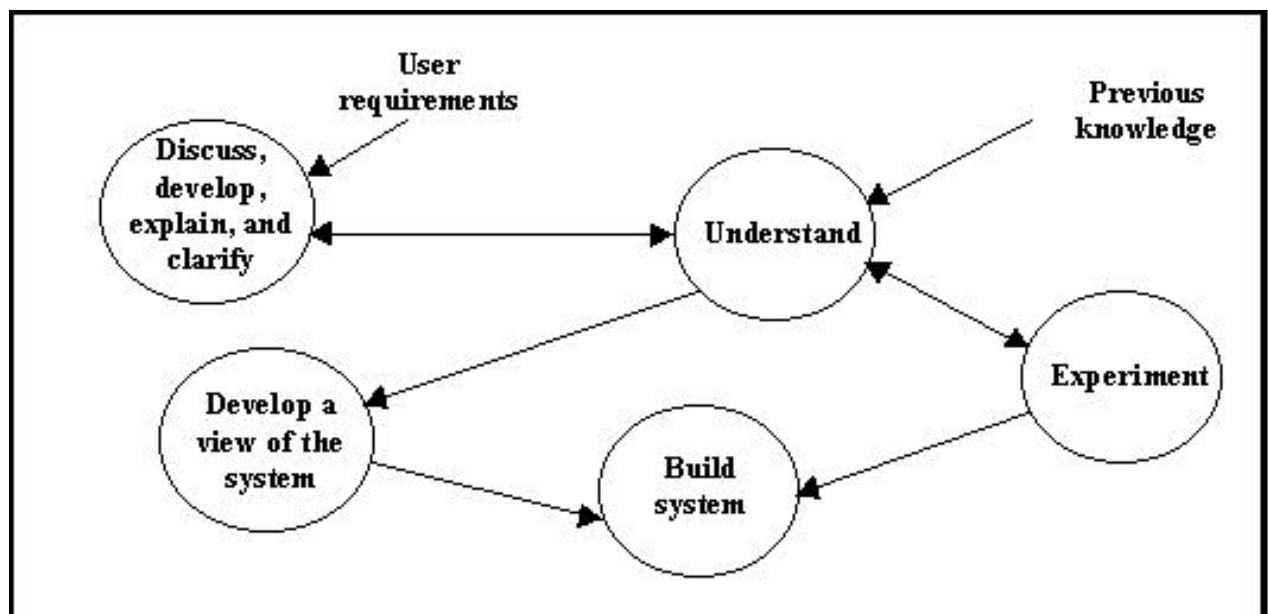


Figure 4.3: Transition from Requirements Gathering to Analysis

A model of the system is created using the different Modeling techniques discussed in the next few sections.

4.5 Analysis Modeling

Models are created to get a better understanding of the actual system to be built. A software model must be capable of modeling the information that software transforms, its functions, and behaviors. The models focus on what the system must do, and not on how it does it. Analysis modeling is the first technical representation of a system. Over the years, various methods have been proposed for analysis modeling but the most dominant ones in use today are:

- **Structured Analysis (SA)**- emphasizes on dividing the programs into independent sections. It is based on function-based decomposition. It focuses on the functions performed by the system and the data consumed by them.
- **Object-oriented Analysis (OOA)** – is the process during which a precise and concise model of the problem in terms of real world objects is developed.

4.6 The Essential Model

After having prepared the analysis models of the proposed system, it is time to evolve this system. For this purpose, it becomes necessary to search and remove the data and process redundancies of the current system to arrive at a solution for the proposed system. It means that we need to develop a model of the new system that the user wants. A model built to show '*what*' the system must do in order to satisfy the user's requirements is termed as the **Essential Model** of the system.

Two components of the Essential Model are:

- Environmental Model
- Behavioural Model

4.6.1 Environmental Model

The starting point for any analysis activity is the identification of the system boundary. A system boundary determines what can be called as a part of the system and what cannot be. It specifies the environment of a system and also entails determining the subsystems that are part of this system, the super-system of which this system is a part, the inputs that the system receives from its environment and the output that it gives to its environment. An analyst must investigate all that falls within the boundary of the system.

For example, if we have to develop a system for New York's International airport, we must remember that it will only be a part of USA's International airport system, which in turn will only be a part of the global airport system. Therefore, this first major model that is to be developed should define the interfaces between the system and the rest of the universe. *It models the outside of the system.*

Figure 4.4 defines the super-system and subsystem environments of a system:

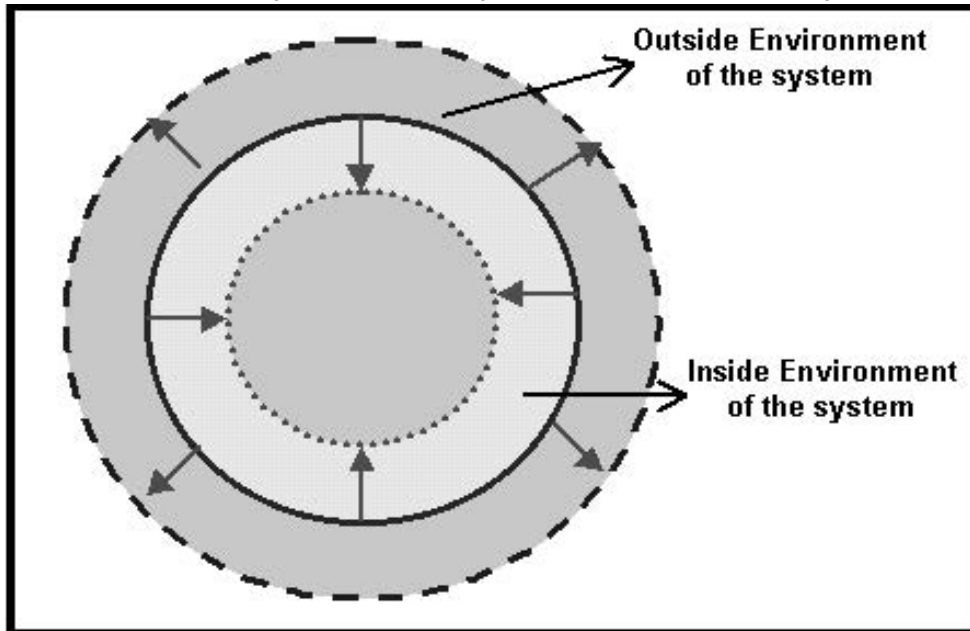


Figure 4.4: Environmental Model

In this context, *the environment model helps us to identify the different relevant external activities.*

The **tools** of the Environmental Model are:

- **Statement of Purpose** - Defines the purpose of a system – why it is being developed, what is it meant to do, and so on. Broadly speaking, it defines the boundaries of a system.
- **Context Diagram** – This outlines:
 - External entities of a system – people, departments or any other entities that a system communicates with
 - Data that a system receives from these external entities
 - Data that a system passes on to these external entities
 - Boundary of a system under study
- **Event List** – This is a list of events that occur outside the system and require a response from the system. This is yet another way of arriving at the data flow diagram for a system.

4.6.2 Behavioral Model

Describes the internal processing of a system. It specifies what processes take place and how one process is related with another within the boundary of the system.

The behavioural model defines:

- The internal functioning of the proposed system (processes as represented in data flow diagrams)
- The inter-relationships between the various entities of the proposed system – broadly speaking, these are the data stores of a system (remember, we are not concerned with the relationships that exist between external entities)
- Data used in the proposed system

The **tools** used to represent the Behavioural Model of a system are:

- Data flow diagrams for the proposed system
- Entity relationship diagrams for the proposed system
- Data dictionary for the proposed system
- Process specifications for the proposed system



SUMMARY

- A System is a collection of interrelated components that work together to achieve some common objective, and System Analysis is the specification of what the system is required to do.
- A Feasibility Study is conducted to find out whether the proposed system are:
 - **Possible** – to build it with the given technology and resources
 - **Affordable** – given the time and cost constraints of the organization
 - **Acceptable** – for use by the eventual users of the system
- A feasibility study is not justified for projects where benefits out-weigh costs, technical risks are not high, and there are no alternatives to be suggested.
- The different aspects related to Feasibility Study are: Need Analysis, Economic Feasibility, Technical Feasibility, Legal Feasibility, and Evaluation of Alternatives.
- Project Proposal is the document that is prepared from the feasibility report, which outlines the characteristics of a system and gives first-cut calendar schedule.
- Once the problem is analysed and the essentials understood, the requirements must be specified in the *Requirement Specification Document*.
- Requirements Gathering involves conducting a preliminary meeting with the client to understand and analyze his needs.
- A software model must be capable of modeling the information that software transforms, its functions and behaviors.
- OOA helps to create a precise and concise model of the problem in terms of real world objects.
- Structured Analysis is a set of techniques and graphical tools that allow us to develop system specifications that are easily understandable by the user.
- A dataflow diagram is a pictorial representation of a system's functions. Functions form part of various processes that are executed for a system.



SUMMARY

- Two components of the Essential Model are:
 - Environmental Model
 - Behavioural Model
- Project sizing refers to estimating the size and complexity of a project.
- Effort estimation refers to estimation of the amount of human effort required to do the project.
- The project size can be measured by using the Program Complexity Method.
- The most commonly used measure of source code program length is the number of lines of code.
- Function Points measure size in terms of the amount of functionality in a system.
- COCOMO can be used to estimate the development effort involved in a project.



CHECK YOUR PROGRESS


1. A Feasibility Study is conducted to find out whether the proposed systems will be _____, _____ and _____.
2. The three aspects of Need Analysis are _____, _____ and _____.
3. The study of costs and benefits is called _____.
4. When benefits outweigh costs, a system is said to be _____.
5. The _____ Report summarizes the findings of the feasibility study and presents the management with various implementation options.
6. _____ is a quality management technique, which translates the needs of the user into technical requirements for software.
7. _____ Diagram depicts a set of real world entities and the logical relationships among them.
8. _____ represents the software model as a single, large bubble (or process) with input and output data depicted by incoming and outgoing arrows respectively.
9. _____ Model determines what is part of the system and what is not, in other words the system boundary.
10. The tools of the Environmental Model are _____, _____ and _____.
11. The _____ Analysis technique calculates project size in function points, based on the functions that a system performs.
12. The _____ method calculates system size based on the number of lines of code that will be written according to estimates for each of the programs that the system will comprise of.
13. _____ measures project development effort in terms of person-days (man days), person-months or person-years.
14. _____ Method measures project size in terms of Complexity Points.
15. COCOMO classifies software projects into three types in terms of development mode, which are _____, _____ and _____.

g

Balanced Learner-Oriented Guide

@

www.onlinevarsity.com



Objectives

At the end of this chapter, you will be able to:

- *Describe system elements*
- *Discuss Data Modeling*
- *Explain the concepts of cardinality and modality*
- *Discuss function modeling using DFDs*
- *Describe Control Flow Model*
- *Describe the data dictionary*

5.1 Introduction

In this chapter, we shall discuss system engineering, system elements, and function models.

5.2 System Elements

System engineering is a process that focuses not only on software but also on a variety of elements. It analyzes, designs and organizes those elements into a system that can be a product, a service or a technology. When the engineering work deals with a business enterprise, the system engineering process is called information engineering. When the engineering work deals with a product to be built, the process is called product engineering.

System engineering includes a collection of top-down and bottom-up methods that enable us to navigate through the hierarchy shown in Figure 5.1. For instance, a particular top-down method may help us move downwards from business domain towards system elements, where system elements could be hardware, software, people, and such items.

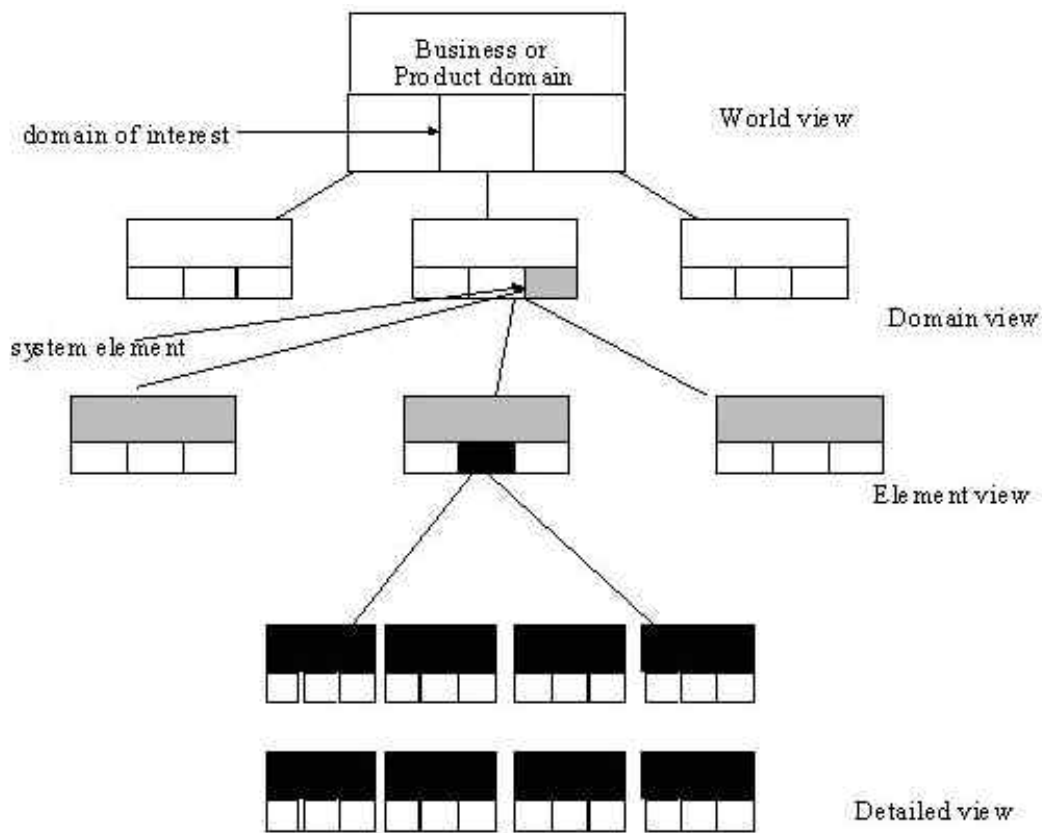


Figure 5.1: System engineering

Source: Software Engineering by Roger Pressman

A complicated characteristic of computer-based systems is that the elements forming a system may itself be a system. Let us consider a “factory automation system”, which is essentially a hierarchy of systems as shown in Figure 5.2. The lowest level of hierarchy shows a numerical control (NC) machine, robots, and data entry devices. Each is a computer-based system in its own right. The elements of the numerical control machine include electronic and electro mechanical hardware (such as processors, sensors and motors), software, people (such as the machine operator), a database (a stored NC program), documentation and procedures. The next level in the hierarchy includes a manufacturing cell, which is a computer-based system that may have elements of its own (such as computers, mechanical equipment), and integrates the lower level system elements such as the NC machine and the robots. Thus, a system element here itself can be a part of a larger system.

The process of system engineering begins with an objective view or “world view”. Using the world view, the product domain is examined and studied to ensure that appropriate business or technology context can be established. Considering the instance of the factory automation system, the world view could depict only the main system, which is the factory automation system. Then, the world view is developed to focus on specific domain of interest. In the factory automation system, there can be three primary domains manufacturing system, inventory system, and information system. Within a specific domain, the need for targeted system elements (data, software, hardware, people) is analyzed.

In this case, the targeted system elements could be material movement system and manufacturing cells for the manufacturing system.

Finally, the analysis, design, and construction of a targeted system element begin. The top of the hierarchy shows a broad general context, but as we proceed further, it gets more detailed.

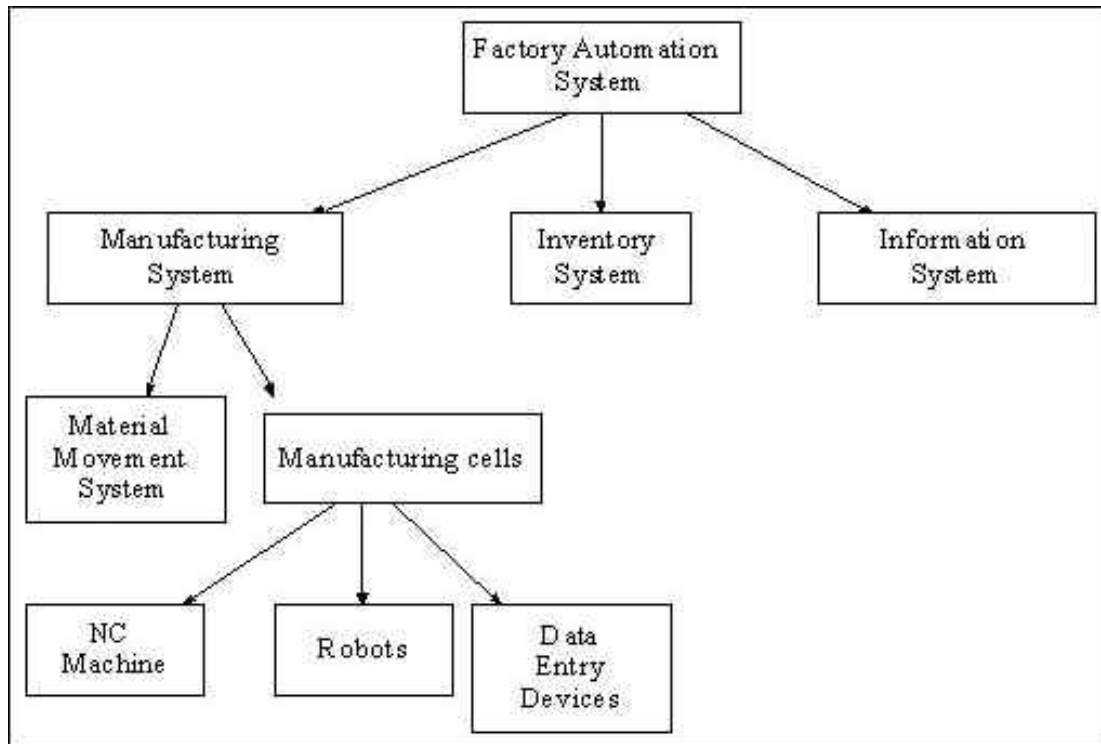


Figure 5.2: The hierarchy

Source: Software Engineering by Roger Pressman

Product engineering is a problem solving activity. The required product data, function and behavior are obtained, analyzed, and allocated to individual engineering components or system elements. The system engineer begins with customer defined objectives and goals for the product and proceeds to allocate requirements to a set of engineering components.

5.3 Data Modeling

Data modeling helps us find answers to questions related to data processing applications such as identifying the primary data objects to be processed by the system, the composition of each data object, and the attributes that describe the object. Data modeling considers data independent of the process that transforms the data. A data model is a formal representation, which hides uninteresting detail, highlights important facts, and gives a better understanding of the system to be built by substituting symbols (wherever necessary) in place of bulky graphical components. By using data models, we can reduce maintenance because of fewer construction errors and better utilization, improve system flexibility, and create greater reusability.

5.3.1 Data Objects or Entities

A representation of information that has a number of different properties or attributes and which can be understood by software is called a **data object**. For instance, **book** can be a data object since it has attributes such as title, edition, price, and author. Any external entity that generates information like an event, a thing, and even an organizational unit can be classified as a data object. Data objects are related to one another. For instance, a **person** can write a **book**. Here, **person** and **book** are data objects and the relationship between them is “*write*”.

Data object description gives information about a data object and all its attributes. A data object does not contain any reference to functions performed on the data, it contains only data.

An entity represents the class of objects that share a distinguishing factor. An entity instance is an occurrence of that entity. For example, ‘Author’ is an entity type while ‘John Grisham’ and ‘Mary Higgins Clarke’ are entity instances.

5.3.2 Attributes

Attributes define the properties of a data object and may serve one of the following purposes:

- Name an instance of the data object
- Describe the instance
- Make references to another instance in another table

One or more attributes must be defined as an identifier or key that will help us locate a particular object.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. For instance, information drawn out for a **student** person such as his name, course enrolled, and fees paid would be inappropriate for a **customer** person. Therefore, information about a person in one context is not always relevant in another context.

Figure 5.3 depicts an everyday example of an object and its characteristics or attributes.

Object: Book



Attributes
Title
Author
Price
Edition
Publisher

Figure 5.3: An object and its characteristics or attributes

5.3.3 Relationships

Data objects are connected to one another in different ways. Let us take the instances of two data objects, **book** and **member**, which belong to a larger system, a library. Relationships between book and member can be explained as illustrated in Figure 5.4.

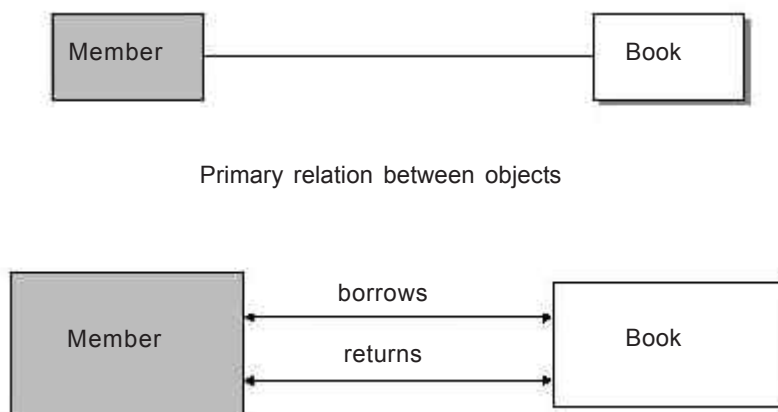


Figure 5.4: The relationship

We can now define a set of object-relationship pairs that explore the particular relationships:

- A member borrows books
- A member returns books

Relationships can be bi-directional. Data objects, attributes and relationships form the basic elements of data modeling and provide the basis for understanding the information domain of a problem.

5.4 Cardinality and Modality

Now, we shall explore the concept of cardinality and modality. These are characteristics of a relationship.

George Tillmann has defined cardinality as:

Cardinality is the specification of the number of occurrences of one object that can be related to the number of occurrences of another object. Cardinality is usually expressed in terms of 'one' or 'many' combinations.

Cardinality defines the maximum number of object relationships that can participate in a relationship.

The following combinations are possible with cardinality:

- **One-to-one** – An occurrence of an object A can interact with one and only one occurrence of object B, and an occurrence of object B can interact with one and only one occurrence of object A.
- **One-to-many** – One occurrence of object A can relate to one or many occurrences of object B, but an occurrence of B can relate to only one occurrence of A. For instance, a class can have many students but a student can attend only one class at a time.
- **Many-to-many** – An occurrence of object A can relate to one or more occurrences of object B. Moreover, an occurrence of object B can relate to one or more occurrences of object A. For instance, one course may be attended by many students and a student can enroll in more than one course.

The most common way to represent cardinality is to use a bar to express one and a trident to represent many.



Figure 5.5: Cardinality, One-to-many relationship

While determining whether or not a particular data object must participate in the relationship, we make use of a concept called *modality*.

The modality of a relationship is zero if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory.

In modality, a bar represents one (mandatory) relationship and a circle represents a modality of zero (optional) relationship.

Both cardinality and modality symbols are depicted in Figure 5.6.

Let us assume a case of a supermarket where a customer may purchase items. At a time, one customer is serviced at the counter and more than one item may be purchased by a customer.

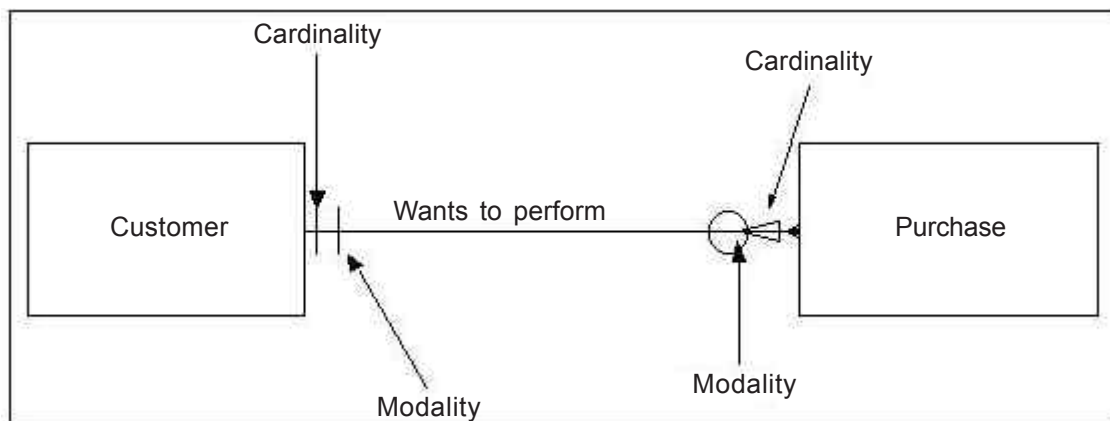


Figure 5.6: Cardinality and Modality

Figure 5.6 shows a one-to-many relationship. A single customer may purchase zero or more items. The first vertical bar and the three-pronged fork indicate cardinality whereas the second vertical bar and the circle indicate modality.

Taking another example, an author will write a book. It cannot be written by itself. Hence, this relationship is mandatory. However, every author need not write a book for some time in future—it may be a break period or an interval for the author during which he does not write any book. So, this relationship becomes mandatory-optional.

This is depicted in Figure 5.7.

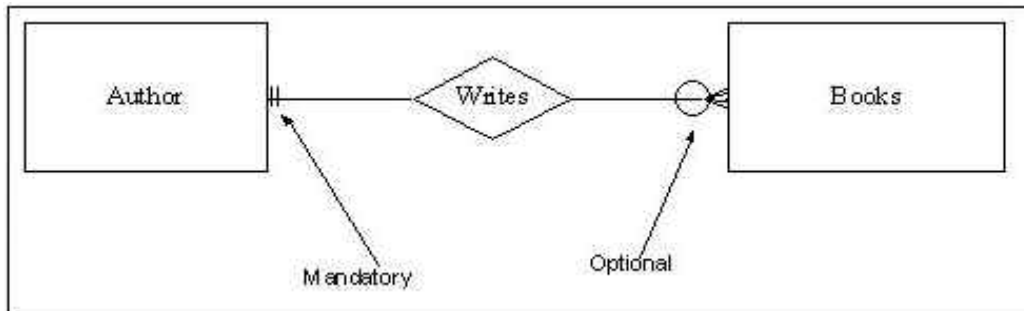


Figure 5.7: The relationship becomes mandatory-optional

5.5 Entity-Relationship Diagrams

Object-relationship pairs can be graphically represented using Entity Relationship (ER) diagrams. Originally proposed by Peter Chan, an ER Diagram (ERD) identifies a number of components such as data objects, attributes, relationships, and a number of type indicators (for cardinality/modality). The main objective of the ERD is to represent data objects and their relationships. ER diagrams use rectangles to represent data objects. A labeled line connecting objects is used to indicate a relationship. A number of special symbols (such as a bar, a circle and a three pronged fork) are used to indicate cardinality and modality between data objects and their relationships. ER diagrams facilitate communication between database designer and end user during requirements analysis.

The term **entity** can be applied to a person, place, thing, object or event. Entity names must be meaningful. The term **relationship** can be applied to verbs used in the description of the system.

Figure 5.8 shows a simple ER diagram. It depicts the relationship between customer (an entity) and product (another entity). A customer places order for a product or a purchase order is placed on a product. This is indicated in the ERD.

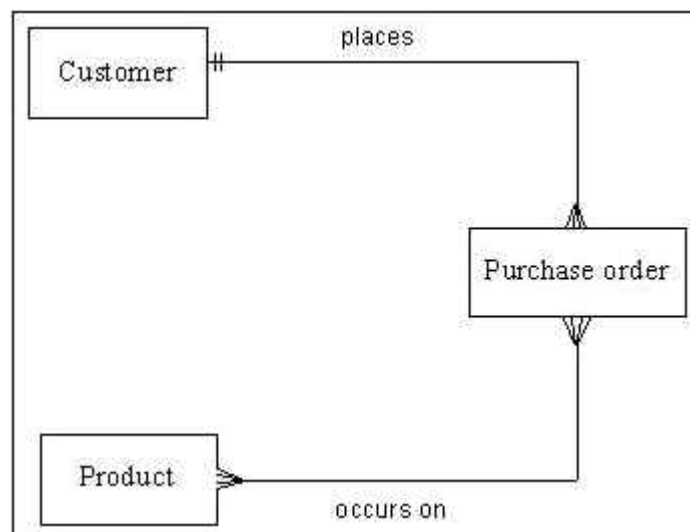


Figure 5.8: A simple ER diagram

5.6 Data Flow Diagrams

A data flow diagram is a well known widely used graphical technique that depicts information flow and functions that are applied as data move from input to output. Data Flow Diagrams (DFD) are also called **data flow graph** or **bubble chart**. DFDs are typically used to represent a system or software at any level of abstraction. They provide mechanisms for function modeling as well as information flow modeling.

Some basic elements within a Data Flow Diagram are:

- **Bubbles:** These are used to represent functions.
- **Arrows:** These are used to represent how data flows. Inward Arrows (going to bubbles) represent input values required by the process represented by the bubble. Outward arrows (moving away from the bubble) represent the results of the function, that is, values.
- **Open boxes:** These are used to represent data stores. Arrows entering open boxes represent data being inserted into the data store. Arrows exiting the open boxes represent data being extracted from the data store.
- **I/O boxes:** These are used to represent data input and output during human-computer interaction.

These elements are illustrated pictorially in Figure 5.9.

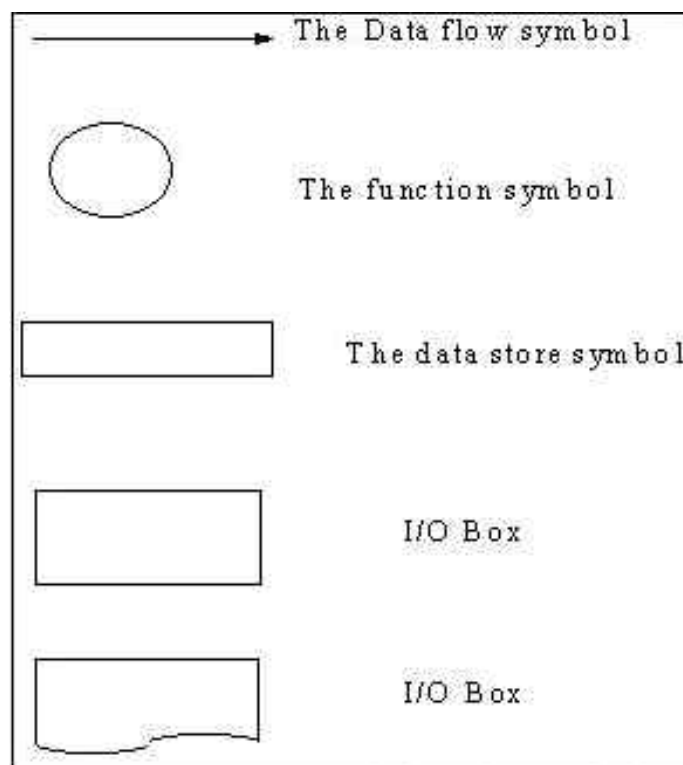


Figure 5.9: The DFD elements

A level 0 DFD is called a fundamental system model or a **context model**. It depicts the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows respectively. When level 0 DFD is expanded, additional processes and information flow paths are represented.

We use a context level diagram or level 0 DFD to implement function modeling and information modeling. Let us take the case of an employee payroll calculation system, which is to be automated. The system will have at least three external entities – accounts department, Human Resources (HR) department and employee. The Accounts department will send the details of various salary related factors such as attendance and overtime done for the particular month to the main process. The HR department will send the employee details such as grade of the employee and leave taken to the main process. The main process will perform the necessary transactions and calculations, and send the updated results to the accounts department and the pay-slip to the employee.

Figure 5.10 shows the DFD for this system:

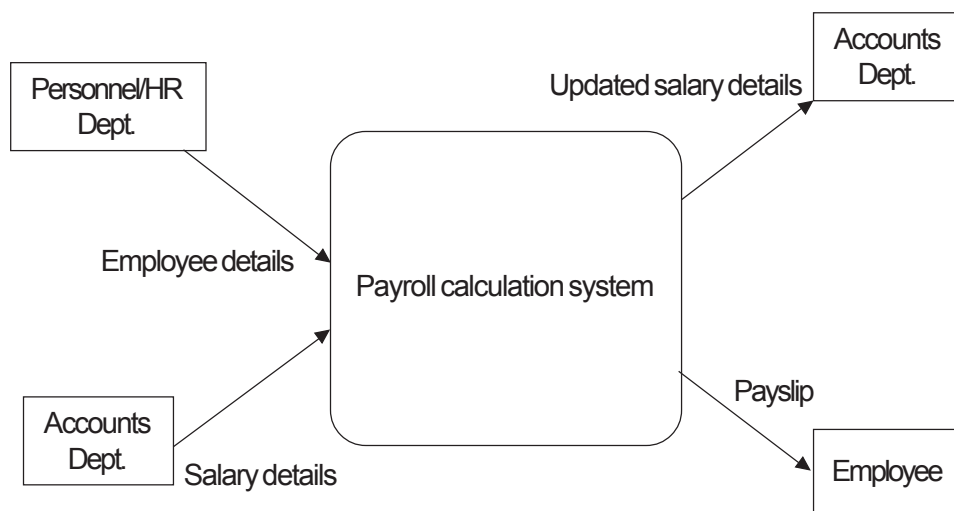


Figure 5.10: The DFD

The DFD can be further exploded (Exploding indicates moving from a higher level to a lower level) to portray the Level 1 DFD as shown in Figure 5.11. Essentially, we are filling more details, adding more processes, and introducing the data stores.

The first level physical DFD for payroll calculation system is shown in Figure 5.11.

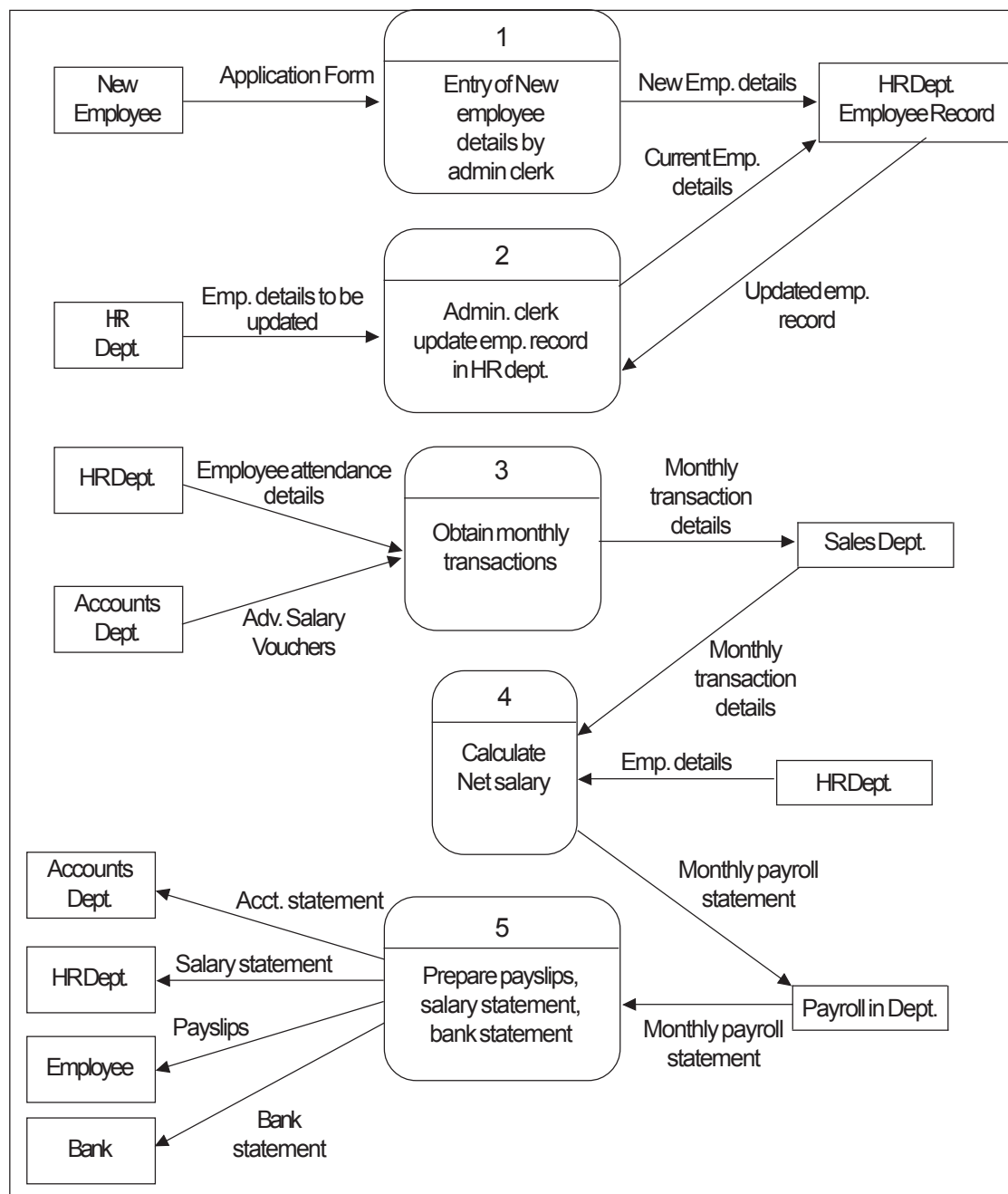


Figure 5.11: The first level physical DFD for payroll calculation system

As we can see in the DFD, the following actions are taking place:

- A new employee who joins the company fills in the appointment form.
- Admin clerk updates this information into the employee record file in the HR department.
- Monthly transaction details are calculated based on attendance, advance salary taken (if any), and various other factors.

- Using all these information, the net salary is calculated.
- Payroll statements, pay slips, bank statements and the like are generated and accordingly distributed to various depts.

General rules to create DFDs are listed as follows:

- Any data flow leaving a process must be based on the data that are input to the process.
- All data flows are named; the name reflects the data flowing between processes, data stores, and sources.
- The only data needed to perform the process should be an input to the process.
- A process should know nothing about any other process in the system. It should depend only on its own input and output.
- Processes are always running, they do not start or stop.

To determine the validity of the DFDs, we need to ask the following questions:

- Are there any unnamed components in the data flow diagram?
- Are there any data stores that are input but never referenced?
- Are there any processes that do not produce output?
- Are there any data stores that are never referenced?
- Is the inflow of data inadequate to perform the process?

If the answer to any of the questions is 'yes', the DFD is not valid.

To summarize, DFDs are an attractive graphical technique for capturing the flow of data and the functions performed within an information system. However, they do not give us precise definitions or semantics. So in case we need a precise and detailed definition, DFDs fall short and cannot fulfill the objective. Assume that we need to test whether the specifications satisfy user expectations. Let us also assume that we need to develop a machine for that purpose. In this case, the development of the machine cannot be based upon a DFD, as it will require more detailed specifications and notations than is provided by a DFD. The syntax of a DFD is sufficient for a precise definition but not for semantics.

5.7 Control Flow

In a number of data processing applications, the data model and the data flow diagram are sufficient to provide meaningful information regarding software requirements. However, there are applications that operate through events rather than data. Such applications produce control information rather than reports and, displays, and process information with heavy concern for time and performance. In these applications, the data flow modeling should be supplemented by control flow modeling.

The Control Flow Diagram (CFD) contains the same processes as DFD but shows the control flow instead of data flow. Instead of representing control processes directly within the flow model, a notational reference to a Control Specification (CSPEC) is used.

The CSPEC represents the behavior of the system in two different ways – a State Transition Diagram (STD), which provides a sequential specification of behavior and a Process Activation Table (PAT), which is a combinatorial specification of behavior. CSPEC describes the behavior of the system but it does not give us any information about the inner working of the processes that are activated because of this behavior.

A sample control flow graph that describes the working of a lamp switch is shown in figure 5.12. A lamp switch can only be in one of the two states at a time - either ON or OFF, so the control flow will represent the state transition from 'ON' to 'OFF' or vice versa.

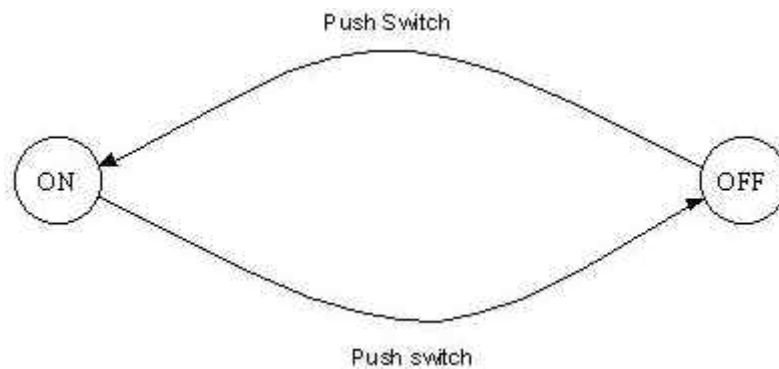


Figure 5.12: A sample control flow graph

5.8 Data Dictionary

In simplest terms, a **data dictionary** is a centralized collection of definitions of all the data flowing to or from data stores and/or processes. A data dictionary defines information elements clearly and explicitly. In large computer-based systems, the data dictionary grows rapidly in size and complexity.

The formal technical definition of data dictionary is as follows:

The data dictionary is an organized listing of all data elements that are pertinent to the system with precise, rigorous, definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and intermediate calculations.

Entries in a data dictionary include the name of the data item, and attributes such as the data flow diagrams, its purpose, where it is used, and where it is derived from.

Though different tools have different formats to represent data dictionaries, the following information is contained by most of them:

- **Name:** This can be the primary name of the data or control item, the data store or an external entity.
- **Alias:** Other names used as nicknames (aliases) for the primary name.
- **Where-used/how-used:** A list of all the processes where the data or control item is used and how it is used.
- **Content description:** A notation that represents content.
- **Supplementary information:** Other information about data types, preset values (if known), restrictions or limitations.

We will now study the detailed classification of the contents of a data dictionary:

- **Data element definitions**
 - **Data element number:** This is used in technical documents. It represents the data element number.
 - **Caption:** This is the unique real life data element name.
 - **Description:** This is a short description of the element in the application domain.
 - **Field names:** These are the names used for the element, generally technical names as designated by a programming language. For instance, *stud_name* and *roll_no* could be two field names.
 - **Code format:** This indicates data type, size and other such parameters of the data elements.
 - **Default value:** This indicates default values that data elements can have.
 - **Database table references:** These indicate references to tables in which the element is used.
 - **Source of the data in the element:** Short description in application domain definitions to describe where the data originates.

- **Table definitions**

Here, various factors related to the data stores are indicated such as table name, owner of the table, list of column names and their details, possible primary and alternate keys, indexes information, and whether duplication is allowed/not allowed.

- **Entity relationship model of data**

ER models were described. At times, they may be part of a data dictionary.

Consider an employee payroll calculation system. Data elements in the data dictionary can be:

- *Name*
- *Description*
- *Aliases*
- *Range and meanings of values*
- *Length*
- *Data type*
- *Editing information*

Aliases can be used to represent fields as in the case of *Emp_code*, which represents *Emp_no*, *Emp_id*, all of which serve to identify the employee using unique means. Descriptions can be included for various fields. Numerical range can be given for fields like *salary* and *age*, and discrete ranges can be provided for values like Marital status -“M”- Married, “U”-Unmarried, “W”-widowed, and “D”-divorced. Data type can indicate alphabetic, numeric, alphanumeric data.

When maintaining these entries in a data dictionary, we must have some form of symbolism to avoid ambiguity and inaccuracy.

DeMarco has proposed a few symbols that can be used to denote entries in a data dictionary. Table 5.1 lists these symbols:

Symbol	Meaning	Explanation
=	is equivalent to	
+	and	
(A)	optional	Indicates that this item may or may not be present.
{B}	iterations of	Indicates that there may be any number of items between zero and infinity. However, it is possible to indicate limits such as 2{B}5. This means that there can be items between two and five.
$\left[\begin{matrix} C \\ D \end{matrix} \right]$	Either C or D but not both.	Indicates an exclusive OR relationship between the two.
?	Query sign	Indicates that this item is a result of a query.

Table 5.1: A sample data dictionary

An example of a data dictionary entry using these symbols would be as shown as follows:

Entry 1

Amount = bill +0{check}1

Entry 2

Bill = date + name of vendor + address of vendor +

$\left[\begin{array}{l} \text{check number} \\ \text{transfer details} \end{array} \right]$

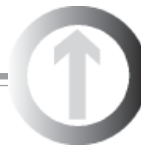
+{invoice no.+ date +invoice amount +(narrative)}+ payment total

Entry 3

Check = ?

The example is based on an organization that pays its vendors by **check** or **bank transfer**. A *bill* is sent accompanied by a **check** when necessary. The first entry is for the information flow *amount*. However, this identifies two major components of the flow- *bill* and **check**. In addition, these should be given entries of their own in the dictionary. In amount, the *bill* is mandatory whereas **check** is optional.

The second entry indicates that the bill includes a date plus name and address of the vendor. It also includes either the **check** number or the transfer details, details of a number of invoices (which may include zero to infinite number), and the total payable amount (payment total). Within the invoice detail, there may or may not be descriptive information (narrative). The entry for check shows a query symbol meaning the *check* details are not fixed.



SUMMARY

- A data model is a formal representation, which hides uninteresting detail, highlights important facts, and gives a better understanding of the system to be built.
- A representation of information that has a number of different properties or attributes and that can be understood by software, is called a data object.
- Attributes define the properties of a data object and may serve one of the following purposes:
 - To name an instance of the data object
 - To describe the instance
 - To make references to another instance in another table
- Relationships are verbs used to indicate actions between any two data objects in the system.
- Cardinality defines the maximum number of object-relationships that can participate in an association or relationship.
- The modality of a relationship is zero if there is no explicit need for the relationship to occur or the relationship is optional. The modality is one if an occurrence of the relationship is mandatory.
- An Entity-Relationship diagram identifies a number of components such as data objects, attributes, relationships and various type-indicators. The main objective of the ERD is to represent data objects and their relationships.
- Data flow diagrams are typically used to represent a system or software at any level of abstraction.
- Data flow diagrams provide mechanisms for function modeling as well as information flow modeling.
- A Control Flow Diagram contains the same processes as the DFD but shows more control flow instead of data flow.
- A data dictionary is a centralized collection of definitions of all the data flowing to or from data stores and/or processes.



CHECK YOUR PROGRESS

1. A _____ is a centralized collection of definitions of all the data flowing to or from data stores and/or processes.
2. In a DFD, _____ is used to represent functions.
3. _____ is the specification of the number of occurrences of one object that can be related to the number of occurrences of another object.
4. The modality of a relationship is _____ if there is no explicit need for the relationship to occur or the relationship is optional.
5. The modality is _____ if an occurrence of the relationship is mandatory.
6. The control specification represents the behavior of the system in two different ways, _____, and _____.

Objectives

At the end of this chapter, you will be able to:

- *Describe the design process and principles of design*
- *Describe the software design concepts*
- *Describe the various design tools:*
 - *Structure Chart*
 - *Structured Flowchart*
 - *Structured English*

6.1 Introduction

In a previous chapter, we discussed system engineering and system elements. We also learned to draw ER diagrams and data flow diagrams. We explored the concepts of cardinality and modality. The term **data dictionary** was defined and its probable contents were listed and explained.

As discussed earlier, one of the most important stages during the Software Development Life Cycle (SDLC) is the **design stage**. The process of design is intricate and complex, and its final goal is to produce a model or a prototype of the system that will eventually result into the proposed system. In this chapter, we will explore the basic concepts and principles that are applicable to software design.

6.2 The Design Process

Design can be defined as the process of applying various techniques and principles to define a device, process or system in sufficient detail to permit its physical realization. Figure 6.1 shows a man in the state of indecision before he begins designing:



Figure 6.1: The design process

Software design is situated at the core of the software engineering process and is always applied irrespective of the software process model used. Each element of the analysis that we explored in the previous chapter (ER diagrams, DFDs, Control Flow, Data dictionary) provides information necessary to create a design model. The design model comprises of data design, architectural design, interface design, and procedural design.

The information domain model created during analysis is transformed using **Data design** into the data structures necessary to implement the software. ER diagrams and data dictionary created during the analysis phase act as a source of information about data objects and relationships, which will form the base for activities in **data design**.

Architectural design defines the relationship among the major structural elements of the program. The modular framework of a program can be derived from the analysis model and the interaction of subsystems defined within the analysis model.

Interface design describes how software communicates with systems that interoperate within it and with people who use it. Data and control flow diagrams provide the information required for interface design.

Component-level design transforms structural elements of the program architecture into a procedural description of software components. PSPEC (Process Specification), CSPEC (Control Specification) and STD (State Transition Diagram) act as sources of information that provide the basis for procedural design.

Design is an iterative process during which requirements are translated into an outline for constructing the software.

The following three characteristics of design have been suggested by McGlaughlin and they serve as a guide for the evaluation of a good design-

- The design must implement all the explicit requirements contained in the analysis model and it must accommodate all the implicit requirements desired by the customer.
- The design must be a guide that is easily understood by those who write code and for those who test and subsequently maintain the software.
- The design should provide a complete picture of the software, address the data, and functional and behavioral domains from an implementation perspective.

Technical criteria to establish a good design can be summarized as follows:

- A design should exhibit a hierarchical organization that makes intelligent use of control among elements of software.
- A design should be modular, which means software should be logically partitioned into elements that perform specific functions and sub-functions.
- A design should contain both data and procedural abstractions.
- A design should lead to interfaces that reduce the complexity of connections between modules and the external environment.
- A design should be derived using a repetitive method that is driven by information obtained during software requirements analysis.

6.3 Design Principles

A software engineer has to abide by certain rules/principles during the design process. These principles are given as follows:

- ***Avoid the tunnel vision***

Tunnel vision denotes a narrow point of view. While designing, the software engineer must take care to see that the approach is of a broad view, consider all alternative approaches and evaluate every approach against the others based on the resources available.

- ***Make the design model traceable to the analysis model***

It is necessary to have means to track the requirements that have been satisfied by the design model because a single element may translate to multiple requirements. In such cases, it may be necessary to go backwards, towards the analysis model to see how this happened.

➤ ***Do not reinvent the wheel***

In a software engineering process, time is always short; hence, one should not spend unnecessary time in developing elements that are already available. Instead, one should go in for reusable components that will not only save time but also effort.

➤ ***Minimize the intellectual distance***

Structure of the design should imitate the structure of the problem domain wherever possible. The manner in which the problem is solved should not be too difficult to comprehend or understand.

➤ ***Exhibit uniformity and integration***

A design is uniform if it appears as though it has been developed by one person. There must not be inconsistency in the way it has been designed. The same rules and formats must be followed by all the team members.

➤ ***Structure the design in such a way so as to accommodate change***

The design must be structured so that modifications required at a later stage can be implemented.

➤ ***Structure the design so as to not fail suddenly or abruptly***

A well-designed software system should never fail abruptly but degrade gently. If unusual circumstances occur, they should be taken care of or anticipated in advance such that the program does not fail or terminate abruptly.

➤ ***Assess for quality as it is being created***

Quality assessments must be done as and when the process is going on, and one must not wait till the end of process to check for quality.

➤ ***Review to minimize conceptual errors***

Apart from focusing on elimination of syntax errors, the designer should also see that the major conceptual issues of design like conceptual/semantic errors, elements of consistency and ambiguity are taken care of.

6.4 Design Concepts

Using the design concepts that have evolved over a number of years, we can apply better methods to the software design process.

Some of these concepts are summarized as follows:

➤ **Abstraction**

Abstraction is a process whereby we can identify the important aspects of any event or fact and ignore its details. Abstraction is a separation of concerns wherein we separate the concern of important aspects from the concern of the unimportant details. There can be many levels of abstraction. At the highest level of abstraction, a broader solution is suggested. At lower levels of abstraction, we can find a procedural solution. At the lowest level of abstraction, we find that solutions can be directly implemented without any more developments. Every step in the software engineering process is an improvement in the level of abstraction. **Procedural abstraction** is a named sequence of instructions that has a specific and limited function.

An example of procedural abstraction would be the word **open** for a door. **Open** would imply the following step of procedures (example: walking to a door, reaching out and grasping the knob, turning the knob and pulling the door, and stepping away from the moving door).

Data abstraction is a named collection of data that describes a data object. Data abstraction includes a set of attributes that describe the data object.

Taking to context the procedural abstraction **open**, we can define a data abstraction called **door**. Like any data object, the data abstraction for door will include a set of attributes that describe the door (example: door type, weight and dimensions). Procedural abstraction makes use of the information contained in the attributes of the data abstraction object, **door**.

Control abstraction involves a program control mechanism without specifying internal details.

➤ **Stepwise Refinement**

Refinement is nothing but improvement or enhancement. Each step in refinement is carried out because of some design decisions. **Stepwise Refinement** is a process of elaboration. A statement, which describes function or information (only conceptually) but gives no information of the procedural aspects of the function, can be refined or improved upon to provide more detail as each successive refinement occurs.

In each step of refinement, one or several of the given instructions are decomposed into more detailed instructions. This successive decomposition terminates when all instructions are expressed in terms of any underlying computer or programming language.

Abstraction and refinement together help a designer to create a complete working design model. Abstraction allows a designer to specify procedure and data, and yet, suppress low-level details. Refinement helps the designer to reveal low-level details as the design progresses. Wirth listed the following activities to be a part of stepwise refinement:

- Decomposing design decisions to elementary levels
- Separating design aspects that are not truly interdependent
- Postponing decisions concerning representation details as long as possible
- Carefully demonstrating that each successive step in the refinement process is an expansion of previous steps

The advantages of using stepwise refinement are top-down decomposition, incremental addition of detail and continual verification of consistency.

➤ **Modularity**

Let us assume that we had a bulky task to be done, which involved a lot of effort. Now, instead of approaching the task at one go, if we divide it into smaller sub-tasks and divide it among a number of people, not only can the task be completed faster but the effort taken per person will also be lesser. Similarly, a software project can be broken up into modules, distributed, and later integrated into one system. This attribute of software that allows a program to be intellectually manageable by breaking it up into smaller portions is called **modularity**.

Modularity does not mean breaking up a module into smaller portions indefinitely. This approach will not work because as the number of modules grows, the effort (cost) taken to integrate the modules also grows. A module in the broadest sense could be a subroutine, a function, or a subprogram.

➤ **Software Architecture**

Software Architecture refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. When we talk of architecture, we mean the hierarchical structure of modules, the way they interact with each other, and the formation of data that is to be used by the modules. Software design aims at deriving an architectural interpretation of a system, which will then serve as a framework from which further detailed design activities are conducted.

➤ **Control Hierarchy**

Control hierarchy is also called **program structure** and it signifies the organization of modules and implies a hierarchy of control. Control hierarchy can be depicted using a variety of methods. Control hierarchy represents two subtly different characteristics of the software architecture: **visibility** and **connectivity**. **Visibility** is the set of program components that may be invoked or used as data by a

given component indirectly.

For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All these objects are visible to the module.

Connectivity specifies the set of components that are directly invoked or used as data by a given component directly. For example, a module that directly causes another module to begin execution is said to be connected to it.

➤ **Structural Partitioning**

Both horizontal and vertical partitioning must be done on the program structure. By **horizontal partitioning**, we mean defining separate branches of the modular hierarchy for each major program function.

Three partitions can be defined using horizontal partitioning in the simplest manner: **input**, **data transformation** and **output**. There are several distinct benefits to be achieved by partitioning the architecture horizontally. These benefits are:

- Software becomes easier to test
- Software becomes easier to maintain
- Software becomes easier to extend

Vertical partitioning also known as **factoring** recommends a top-down approach to program architecture. Top-level modules should perform control functions and must be involved with very little processing work. Lower level modules must perform all the tasks related to input, computation, and output. The advantage of vertical partitioning of a program structure is that the software architecture is less likely to be prone to side effects when changes are made and are therefore more maintainable, which is a key quality factor.

➤ **Data Structure**

It is a representation of logical relationship shared among individual elements of data. Data structure determines the organization of data, methods of access, degree of associativity, and processing alternatives for information. Data structures form the building blocks for more sophisticated program structures.

The simplest data structure is a **scalar item**. A scalar item represents a single element of information. When a number of scalar items are ordered as a list or continuous group, we get a **sequential vector**. Vectors are the most commonly used data structures. By using vectors, it is possible to create arrays, which are n-dimensional spaces created by adding dimensions

to a sequential vector. Vectors, scalar items, and arrays may be organized in a variety of techniques. For instance, a **linked list** is a data structure that organizes non-adjacent scalar items, vectors or spaces in a manner, which enables them to be processed as a list. Linked lists have elements called nodes, where each node contains data and a pointer to the address of next node. More sophisticated data structures can be built using these data structures as building blocks. Data structures can be represented at different levels of abstraction. A conceptual model of a data structure, called **stack**, can be implemented either as a vector or as a linked list. Depending upon the level of design detail, the internal workings of stack may or may not be specified.

➤ **Software Procedure**

Software Procedure focuses on processing details of each module individually. Procedure must provide a precise specification of processing including sequences of events, exact decision points, repetitive operations, and even data organization and structure.

There is a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described.

➤ **Information Hiding**

Information hiding implies that each module must be distinguished by design decisions that hide it from others. This means that modules must be designed such that functions and data within one module are inaccessible to those modules that do not need the information. Information hiding defines and enforces access constraints to both procedural details within a module as well as any local data structure used by the module. Information hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with each other.

Structured design involves two important concepts that are based on the theory of abstraction and modularity- **Coupling** and **Cohesion**:

➤ **Coupling**

Coupling refers to the strength of the relationship between modules in a system. The strength of a relationship, that is, coupling, is determined by the data passes between modules and the interdependence between the modules. Good system design implies that interdependence between modules must be minimum. Such modules are called **loosely coupled** modules.

Figure 6.2 depicts the concept of coupling graphically:

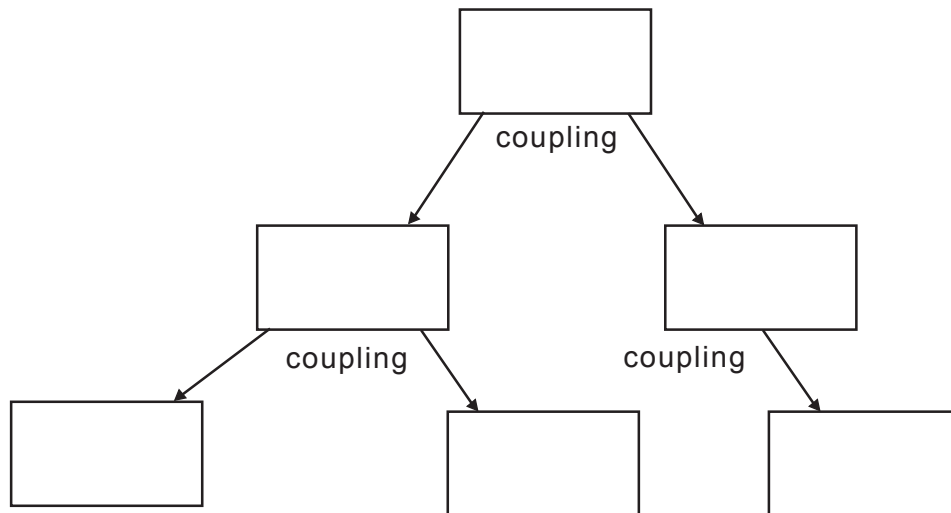


Figure 6.2: The concept of coupling

We may carry out one of the following to minimize interdependence:

- Control the number of parameters exchanged between modules
- Avoid passing unnecessary data to the invoked modules
- Pass data only when needed
- Maintain superior/subordinate relationship between calling and called modules
- Pass data, not control information

Different kinds of **coupling** are:

➤ **Data Coupling** - In this form of coupling, data is passed across modules through parameters. This is the most common form of coupling. In data coupling, the parameters must be elementary forms of data. For example, if we want to calculate interest, we need to pass the following parameters:

- Principal amount
- Rate of interest
- Time period

If these parameters are passed, we can calculate interest. As stated earlier, coupling must be kept to a minimum. However, this is the best form among the various forms of coupling.

Dangers -It is imperative that we remember the following when using data coupling:

- 1) Keep parameters to a manageable number. More than five parameters can be considered unmanageable. If we decouple two modules, but have twenty parameters being passed between them, we have not achieved any decoupling.
- 2) Avoid passing unnecessary parameters. Programmers have the tendency to pass more data than required. This is extra-coupling, which is totally against the objective of minimizing coupling.

- **Content coupling** - This kind of coupling occurs when one module modifies local data or instructions in another module. This form of coupling should never be used. At best, one can call it intertwining and not coupling. It is unstructured programming. If we keep jumping from one module to another and back to where we started, the result could be chaos. This was the style of programming in ancient times, but programmers who have no exposure to structured programming still program this way. In this form of coupling, the interfacing between modules is very weak across modules. Such coupling should be avoided at all costs.
- **Common coupling:** In this kind of coupling, modules are bound together by global data structures. Common coupling refers to the use of global or common systems area, either in RAM or in the disk. This implies that one module writes in an area of the disk and the other module reads from that area.

This form of coupling is avoidable as in many systems the modules become independent of each other over a period of time. The consequences of a change in layout in the originating module do not take into consideration changes to the receiving module. Hence, common or global coupling of data should be avoided.

It however has its utility at the overall system level. System level parameters, which go across modules, are candidates for global coupling. Typical parameters can be company name and address, exchange rates in a banking system, or expense codes in an organization.

Common coupling should be used for system level parameters across modules. It should not be used to process data. Processing data across modules must be done by parameters using data coupling.

- **Control coupling:** This coupling involves passing control flags between modules so that one module controls the sequence of processing steps in another module. This type of coupling is typically used when a module performs a number of functions and only one of them needs to be done at a time.

Let us suppose we have written a common module that adds, modifies, deletes, prints, and displays records from a file. We can only perform one of the five activities at a given time. From the calling module, a flag will have to be passed to indicate the type of activity to be done. The logic can

be applied to an Interest computation module, which calculates simple and compound interest. A flag can be passed to indicate the type of interest to be computed.

If control coupling is to be used, it must be used explicitly. If it is camouflaged with other records or variables, it will be difficult to decipher the logic of the module.

- **Stamp coupling:** This is the same type of coupling as indicated under data coupling; the main difference being that composite or group data is passed across modules in stamp coupling, while elementary data was passed in data coupling.

In the interest computation example, if we pass the entire loan record for computation of interest, then it is stamp coupling as we are passing composite information. The receiving module has to break up composite data into elementary data before processing. Thus, if the layout of the loan record changes, the sending and receiving modules will have to be modified. Coupling is therefore more than data coupling.

Typically, this technique should be used when most of the fields in a record are going to be used, and the overhead of splitting and putting it together may be higher than the benefit of achieving data coupling.

Dangers

We must take care of the following aspects in using stamp coupling:

- 1) Do not use stamp coupling when only a few fields of a record are to be used for processing in the receiving module.
- 2) Do not create an artificial group of elements, which are not related, just to “decrease” coupling.

- ***Cohesion***

Cohesion refers to the strength of the relationship between elements of the same module in a system. Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives an idea to the designer about whether the different elements of a module belong together in the same module.

In that context, it can be viewed as the opposite of coupling. In coupling, we wanted each module to be independent and interact with other modules with the least coupling. However, **in cohesion**, we want all activities within the module to be as tightly connected to each other as possible, which means that cohesion must be as high as possible. The degree of coupling and cohesion between various modules is a measure of the quality of system design.

Figure 6.3 gives a graphical representation of the concept of cohesion:

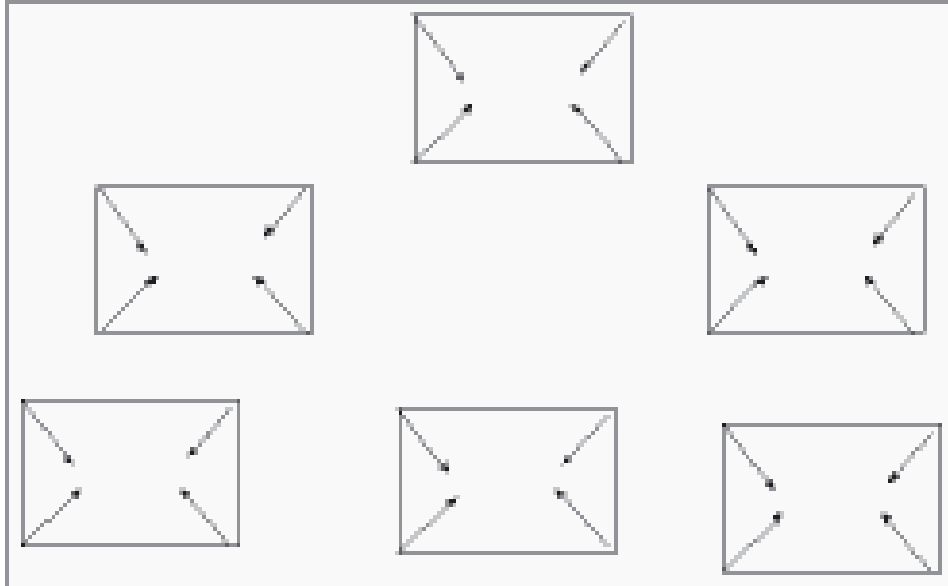


Figure 6.3: The concept of cohesion

Different types of **cohesion** are:

- **Coincidental:** This is the lowest level of cohesion. It occurs when the elements within a module have no apparent relationship to one another. This happens when a large and bulky program is broken down into several smaller sized portions. A module that carried out a set of tasks that relate to each other loosely, if at all, is said to exhibit coincidental cohesion. The modification of such modules may cause other modules to behave incorrectly. It is a bad practice to create a module just to avoid duplicate code, or to break a module into smaller modules to reduce the size of the module.
- **Logical:** Under this type of module division, activities belonging to the same category are grouped together. We can group all reporting or querying activities together. The system is thus divided into logical groups. However, the cohesion is very weak as people working on these modules will not be fully aware of the functionality, and hence there will be more need for the exchange of information. This can lead to loss of productivity. Thus, it is not advisable to logically divide the system into modules. It is better to group reports and queries with their functionality and achieve functional cohesion.
- **Temporal:** In this type of module division, activities occurring in the same time-period are grouped together. All activities occurring at maturity of contracts, contract initiation, or contract settlement can be grouped together as they occur at the same time. The modules are not grouped by functionality of different types of contracts but by different events that occur at time periods. Another example of temporal modularization can be to have modules by end of periods, end of day, end of quarter, or end of year. This is not a good type of modularization, as immense amount of data will have to be exchanged across modules.

- **Communication:** This form of cohesion relates to a situation where all modules share some common data. It is similar to modules having common or global coupling. This form of cohesion has clearly defined boundaries, inputs, and outputs, but suffers from one major weakness. If the common area is impacted, all modules have to be checked again. This is not the most desirable type of cohesion, but will be the most applicable type of module division in cases where the system has a strong central module and many peripheral modules interacting with it.
- **Sequential:** In sequential cohesion, modules are divided into a series of activities such that the output of one module becomes the input to the next module. Thus, the division of modules is well defined, and the inputs and outputs are clearly segregated. If this form of cohesion is coupled with functional cohesion, it is the best form of cohesion.

Typical examples of such divisions can be the processing of stock valuation, which can only be done after stock prices are captured by the system. Thus, stock prices from stock exchange are the inputs to the first module, which produces stock exchange prices on the files for the second module to pick up and value all the stocks on hand.

A good sequentially cohesive module coupled with functional cohesion is the best form of cohesion.

- **Functional:** All activities in the module are functionally related, which means that they are activities involved in solving a similar problem. An interest calculation module, a maturity processing module, and a receivable processing module are examples of modules bound by functionality. As most changes to systems are driven by changes to functional requirements, the chances of a change request affecting more than one module are low if the modules are based on functionality. Thus, the other modules can function independently while one module is being modified. Hence, this is the best form of cohesion.
- **Informational:** Informational cohesion of a module arises when the module contains a complex data structure and several routines to manipulate the data structure. Each routine in the module is functionally bound. Informational cohesion and communicational cohesion are similar in that both refer to a single entity. The difference is that communicational cohesion implies that all code in the module is executed on each invocation of the module, whereas in case of informational cohesion, the code is executed only once.

6.5 Design Tools

Design tools are used to depict or describe the structure of the system.

Commonly used design tools are:

- Structure chart
- Structured Flowchart
- Structured English

The major tool used in Design to depict the structure of the system is the **structure chart**.

6.5.1 Structure Charts

Structure charts are graphical descriptions that show the interaction between modules and the data exchanged between the modules, which interact with one another. It is a widely used tool for designing a modular, top-down system. The logical DFD forms the basis for drawing structure charts. A structure chart differs from a flowchart in two ways: a structure chart has no decision boxes and the sequential ordering of tasks, inherent in a flowchart, can be omitted in a structure chart. Structure charts are developed before code development begins. Procedural or functional logic is not expressed in a structure chart nor is the physical interface between functions depicted in a structure chart. Structure charts identify the data exchanges occurring between individual modules that interact with one another.

A **module** is a set of instructions, which can be invoked by name. It can be a subroutine, procedure, function or even a block. Contents of a module can be referred to collectively under a single name. The name of a module summarizes what the module does. It consists of an action verb and object noun. For instance, we can have a module called “Calculate Medicine Bill”. Here “Calculate” is the action verb and “Medicine Bill” is the object noun. The name only indicates that the bill for medicines is being calculated in this module but the calculation process is not described here.

Symbols and notations used to develop a structure chart are shown in Figure 6.4. Common notations are used to enable uniformity and ease of communication among systems’ developers:

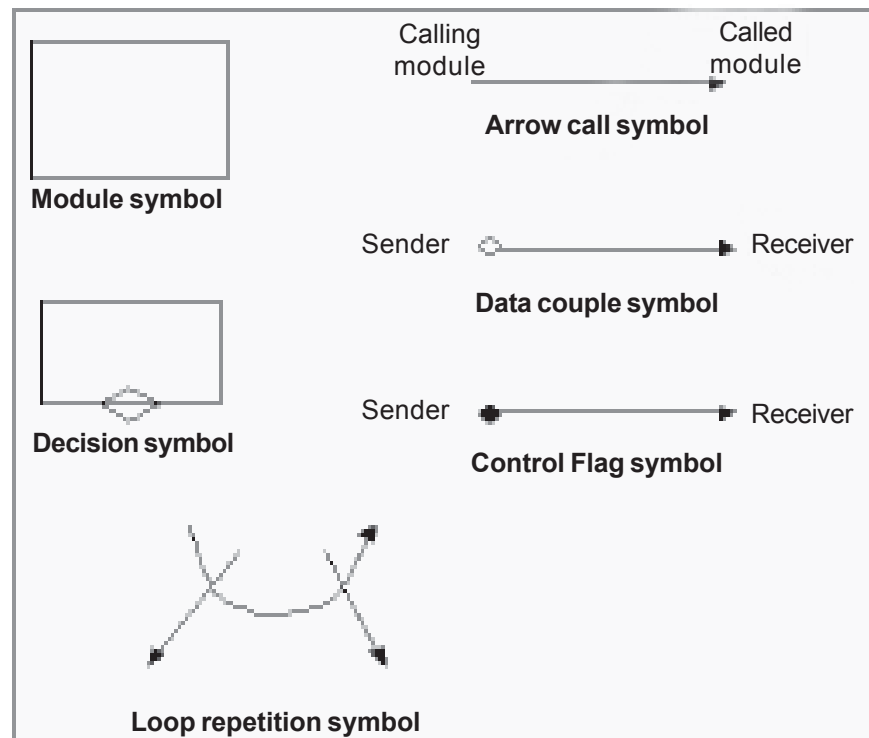


Figure 6.4: A structure chart

- **Rectangle:** These represent modules. The name of the module is written within the rectangle.
- **Arrows:** Arrows indicate that one module is calling another and direction of arrows is from the calling module to the called module. Transfer of information is also depicted using arrows.
- **Small arrows with empty circles:** Also called data couples, these are used to note the passing of data parameters. These indicate data parameters, which are passed down to the calling module or back to the called module.
- **Small arrows with filled circles:** Also called **control flags**, these assist in the control of processing by indicating the occurrence of specialized conditions such as record not existing or reaching the end of file and the like.
- It is easier to make changes in a system if there are fewer control flags and data couples. When the coding is done, it is important to pass as less data couples as possible between modules.
- **Loop:** This indicates that the statements found beneath this symbol are to be repeated a certain number of times.
- **Small diamond:** This appears on the bottom of a rectangle and signifies that only one of the modules beneath the diamond will be executed.

Let us now draw a structure chart for the same example we discussed in the last chapter of the payroll calculation system.

There are five main processes as seen in the DFD. If process 1 and 2 are merged together, we shall have four main processes:

- Maintain Master Employee Pay Details
- Maintain Employee Transaction details
- Calculate Net pay
- Print Reports

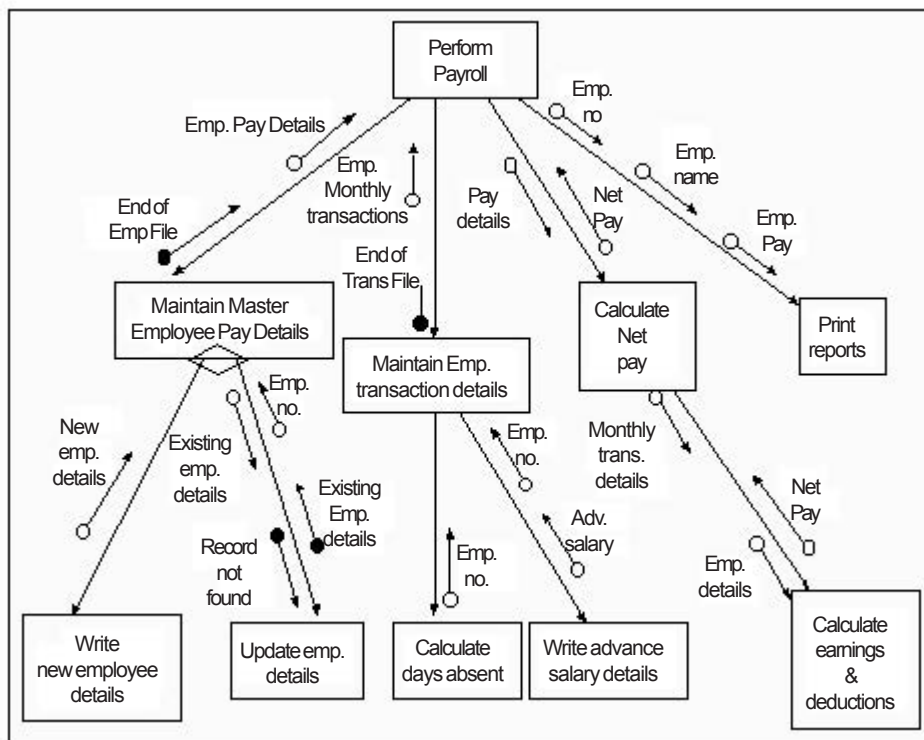


Figure 6.5: A structure chart for the payroll calculation system

The main process in the structure chart is “Perform Payroll”. Each of the modules on the lower levels performs distinct functions. A number of data couple and control flags are used to pass data to and from modules.

6.5.2 Structured English

If we make use of very complex tools to specify the requirements, the client will possibly get confused. At the same time, these requirements cannot be written entirely in a language like English. This is because requirements written in such a language will be vague and unclear. To have the best of both approaches, we use Structured English as a tool, which combines a natural language like English with formal structured syntax.

Structured English can be used to provide a systematic specification for an algorithm. Similar to pseudocode, structured English can be used at any desired level of detail. Structured English should have the following characteristics:

- A predetermined syntax of keywords must provide for all structured constructs, data declarations, and modularity characteristics.
- A free syntax of natural language like English that describes processing features.
- Data declaration facilities that should include both simple (scalar, array) and complex (linked list) data structures.

A sample business procedure written in Structured English is shown as follows:

```
For each MEMBER ID in the MEMBERS file repeat the steps as follows:
```

```
    If    RETURN DATE is greater than DUE DATE  
        Calculate FINE for the number of extra days  
        Print the Fine Bill
```

6.5.3 Structured Flowchart

Flowcharts are the conventional tool to specify and document the sequence of steps carried out to solve a problem. Flowcharts make use of rectangular boxes, arrows, and diamond shaped boxes to depict various elements such as input/output, process, decisions and the like. Unlike conventional flowcharts, structured flowcharts are restricted to compositions of certain basic forms. Structured flowcharts are logically equivalent to pseudocode and are better than pseudocodes in that they express control flow with more clarity.

A few basic flowcharting symbols used to create structured flowcharts are given in Figure 6.6.

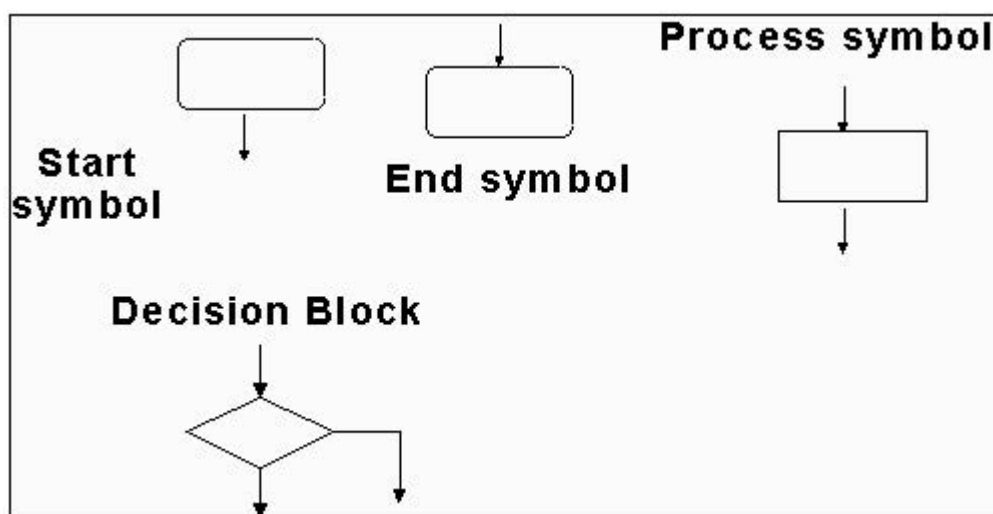


Figure 6.6: The structured flowchart

- The START block represents the beginning of the process. It has only one output and never more than one input.
- The END block represents the end of a process. It has one input and will contain either END or RETURN depending upon its function in the overall process of the flowchart.
- A PROCESS block represents some operation carried out on a data element. A process always has exactly one input and one output.
- A DECISION block is used to represent an if-else decision statement. It can only have two outputs-either Yes or No.

Based on these symbols, further structures used in structured flowcharts can be created. These are depicted in Figure 6.7.

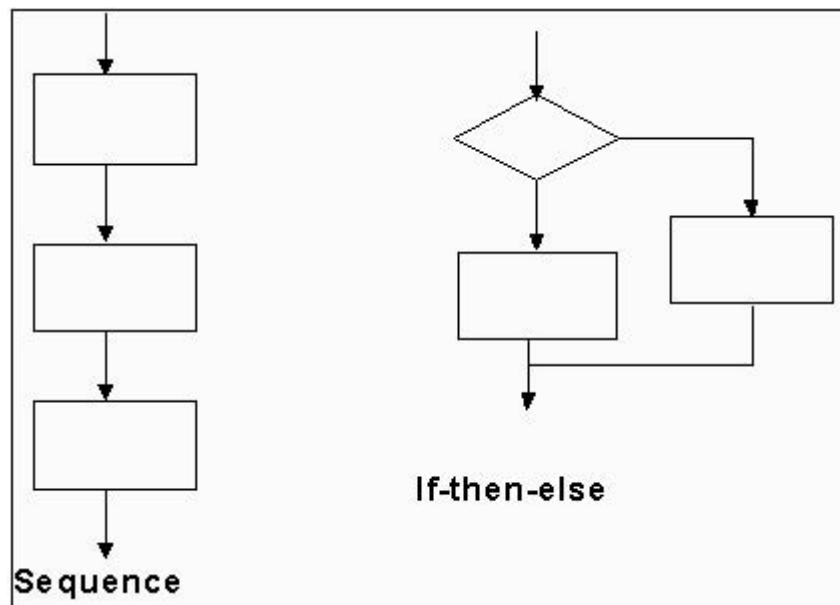


Figure 6.7: The structured flowchart with variation

In a structured flowchart, all the processes and decisions fit into one of a few basic structured elements.

The SEQUENCE process is a series of processes carried out one after the other.

The If-then-else process logically completes the decision block by providing two separate processes, one of which will be carried out eventually. Let us take a look at a sample of a structured flowchart along with its unstructured counterpart so that we can appreciate the difference between the two.

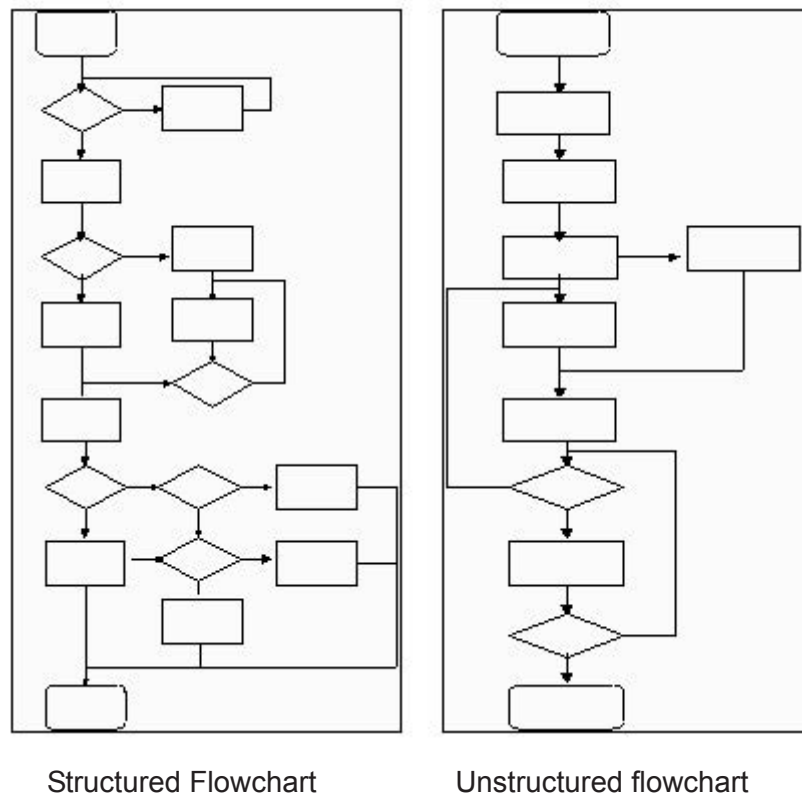


Figure 6.8: The various flowcharts



SUMMARY

- Software design is situated at the core of the software engineering process and is always applied irrespective of the software process model used.
- The design model comprises of data design, architectural design, interface design and procedural design.
- The information domain model created during analysis is transformed by *data design* into the data structures necessary to implement the software.
- Interface design describes how software communicates within itself to systems that interoperate within it and with persons who use it.
- Procedural design transforms structural elements of the program architecture into a procedural description of software components.
- Design is an iterative process during which requirements are translated into an outline for constructing the software.
- Vital design concepts include:
 - Abstraction
 - Stepwise refinement
 - Modularity
 - Software architecture
 - Control hierarchy
 - Structural partitioning
 - Information Hiding
- **Coupling** refers to the strength of the relationship between modules in a system, and is determined by the data passes between modules and the interdependence between the modules.
- **Cohesion** refers to the strength of the relationship between elements of the same module in a system. It represents how tightly bound the internal elements of the module are to one another.
- Standard tools used for design are structure charts, structured English, and structured flowcharts.



CHECK YOUR PROGRESS

1. _____ defines the relationship among the major structural elements of the program.
2. A good design should contain both _____ and _____ abstractions.
3. Structure of the design should imitate the structure of the problem domain wherever possible. **[True/False]**
4. A _____ is a set of instructions, which can be invoked by name.
5. _____ refers to the strength of the relationship between modules in a system while _____ refers to the strength of the relationship between elements of the same module in a system.
6. Structured flowcharts are logically equivalent to pseudocode. **[True/False]**



Visit

Frequently Asked Questions

@

www.onlinevarsity.com

Software Configuration Management

Objectives

At the end of this chapter, you will be able to:

- *Describe baseline concepts*
- *Describe various configuration items*
- *Describe the configuration management process*

7.1 Introduction

Configuration management is the discipline that systematically monitors and controls the changes that take place during the development process within an organization. In simple terminology, software configuration management is a discipline for controlling the evolution of software systems. Software configuration management is an umbrella activity that is applied throughout the software process. The primary goal of SCM is to recognize changes, regulate changes, ensure that change is being properly implemented, and report the changes to those who are interested in them.

Software configuration management defines policies: how the deliverables of the process are stored, accessed and modified, how the different versions of the systems are built, and what authorizations are required to check in and check out the software components in a product database.

The IEEE defines SCM as:

SCM is the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items.

There is a misconception that software maintenance and software configuration management are the same concepts. However, this is not true. Software maintenance activities take place only after the software has been delivered to the customer and put into operation. Software configuration management on the other hand begins right after the project development starts, and ends when the software is being taken out of operation. A software project results in information that may be classified into the following categories of **Source Code** (Programs and executables); **documents** describing computer programs- these may be technical manuals or user manuals; and finally, **data** present within the programs as well as those external to it.

The items that comprise all information produced as part of the software process are collectively called **software configuration**. In other words, product configuration refers not only to the product's constituent components but also to specific versions of the components.

Each individual item is called a **software configuration item**. The number of software configuration items grows with the number of software processes.

7.2 Baselines

Change in the software development life cycle is an inevitable factor but is not always necessary or welcome. Some changes are justifiable.

A **baseline** is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE definition of a baseline is given as follows:

A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development and that can be changed only through formal change control procedure.

Let us take the simple instance of a bookshop where a customer chooses a book and hands it over to an assistant who in turn hands it over to the cashier. The cashier will accept the money and print the bill, get the book packaged/wrapped and give it to the customer. Now, if by mistake the assistant hands the wrong book to the cashier, there are two possibilities. Either he realizes his mistake before the cashier hands over the book to the customer or the customer has already received the wrong book, in which case the assistant must follow a fixed procedure and apologize to the customer and ensure that he receives the correct book.

Before a software configuration item becomes a baseline, change may be made informally and quickly. Once a baseline is made, changes can be made but a specific formal procedure must be applied to evaluate and verify each change.

In this instance of the bookshop, if the assistant realizes his mistake before the cashier prints the bill, it has not yet become a baseline item. Hence, change can be made quickly and informally; the assistant can intimate the cashier and the wrong book can be replaced with the right one, without the customer being aware of the mistake committed. However, once the bill has been printed and the wrong book handed over to the customer, the assistant has no other choice but to follow the standard procedure and apologize to the customer.

With respect to software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items, and the approval of these SCIs that are obtained through a formal technical review. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a project database. When a team member wants to make a modification to an SCI that is baselined, it is copied from the project database into the engineer's private workspace.

7.3 Software Configuration Items

By now, we know that a software configuration item refers to information that is created as part of the software engineering process. An SCI is a document, an entire suite of test cases or a named program component. The following SCIs become the target for configuration management techniques and forms a set of baselines:

1. System specification
2. Software project plan
3. Software requirements specification:
 - a) Graphical analysis models
 - b) Process specification
 - c) Prototypes
 - d) Mathematical specification
4. Preliminary User manual
5. Design specification:
 - a) Data design specification
 - b) Architectural specification
 - c) Module design specification
 - d) Interface design specification
 - e) Object descriptions
6. Source code listing
7. Test specification:
 - a) Test plan and procedure
 - b) Test cases and recorded results
8. Operation and installation manuals

9. Executable program:
 - a) Module executable code
 - b) Linked modules
10. Database description:
 - a) Schema and file structure
 - b) Initial content
11. User manual
12. Maintenance documents:
 - a) Software problem reports
 - b) Maintenance requests
 - c) Engineering change orders
13. Standards and procedures or software engineering

7.4 The Software Configuration Management (SCM) Process

The primary responsibility of software configuration management is the control of change. At the same time, software configuration management is also responsible for the identification of individual SCIs and various versions of the software, auditing of the software configuration to ensure that it has been properly developed, and the reporting of all changes applied to the configuration. Five tasks important to SCM are:

- Identification of objects
- Version control
- Change control
- Configuration auditing
- Reporting

We shall now explore each of these in detail.

7.4.1 Identifying Objects

To control and manage software configuration items, we must name and then organize each item using an object-oriented approach. We can identify two types of objects: **basic objects** and **aggregate objects**. A **basic object** is a unit of text that has been created by a software engineer during analysis, design, coding or testing. For instance, a basic object may be a section of a requirements specification, a source listing for a module, or a group of test cases that are used to implement the code. An **aggregate object** is a collection of basic objects.

Each object possesses a set of distinct features that identify it uniquely. These are:

- Name
- Description
- List of resources
- Realization or implementation

The **name** of an object is a character string that identifies the object explicitly. The **description** is a list of data items that identify the following:

- The SCI type, whether it is a document, program or data, which is represented by the object.
- A project identifier along with change/version information.

A **resource** is an entity that is provided, processed, referenced or otherwise required by the object. Datatypes, specific functions and even variable names are some instances of object resources.

The object identification process must also consider the relationships that exist between named objects. One such relationship is an object that can be identified as **<part-of>** an aggregate object. This relationship defines a hierarchy of objects as can be seen as follows:

```
E-R diagram 2.3 <part-of> data model
Data model <part-of> Design specification
```

This indicates that the ER diagram 2.3 is a part of the data model and that data model itself is part of design specification. Thus, a hierarchy of objects is being specified here.

The **<interrelated>** is another relationship that depicts cross-structural relations:

```
Data model <interrelated> data flow model
Data model <interrelated> test case class x
```

The first case shows an interrelationship between a composite object, while the second case shows the relationship between an aggregate object (data model) and a basic object (test case class x).

The interrelationships between configuration objects can be represented with a Module Interconnection Language (MIL). An MIL describes the interdependencies among configuration objects and enables any version of a system to be constructed automatically. Currently, a number of automated SCM tools are available.

7.4.2 Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software engineering process. We can use an evolution graph to represent different versions of a system. Each node is an aggregate object that is a complete version of the software. Each version of the software is a collection of SCIs. Automated systems are used to create version specifications that are used to create variants or new versions.

7.4.3 Change Control

Change control combines human procedures and automated tools to provide a mechanism for the control of change. The entire change control process is summarized as follows:

- 1) A change request is submitted and evaluated to measure technical merit, potential side effects, overall impact on other configuration objects, system functions, and the projected cost of change.
- 2) The results of the evaluation are presented in the form of a change report that is used by a Change Control Authority (CCA), a person or group who makes a final decision on the status and priority of the change.
- 3) An Engineering Change Order (ECO) is generated for each approved change. The ECO describes the changes to be made, the constraints that must be respected and the criteria for review and audit.
- 4) The object to be changed is *checked out* of the project database, the change is made and appropriate Software Quality Assurance (SQA) activities are applied.
- 5) The object is then *checked in* to the database and appropriate version control mechanisms are used to create the next version of the software.

These check-in and checkout processes implement two important elements of change control, **access control** and **synchronization control**. Access control governs those team members who have authority to access and modify a configuration object. Synchronization control helps to ensure that parallel changes performed by two different people do not overwrite one another.

7.4.4 Configuration Auditing

These processes discussed help the software developer to maintain proper order and system.

To ensure that change has been properly implemented, we need to implement **formal technical reviews** and **software configuration audit**. The formal technical review focuses on the technical correctness of the configuration object that has been modified. Reviewers assess the SCI to determine consistency with other SCIs, things that may have been left out and the potential side effects.

A software configuration audit complements the formal technical review by assessing a configuration object for characteristics that are generally not considered during review.

7.4.5 Reporting

Configuration status reporting (CSR) is an SCM task that deals with any event - who was responsible for the event, when it occurred, and how will others be affected due to the event. Each time an SCI is assigned a new or updated identification, a CSR entry is made. Each time a change is approved by the CCA, a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an online database so that software developers or maintainers can access change information by keyword category.

Configuration status reporting plays a vital role in the success of a large software development project. When many people are involved, it is likely that there will be a communication gap between them. For instance, two software engineers may attempt to modify the same SCI with different and conflicting actions. CSR helps to eliminate these problems by improving communication among all the team members involved.

7.5 SCM repository

The SCM repository is the heart of any SCM system. It stores all the project objects, each version of every object and the meta-data that describes each version of the objects.

The key requirements for an SCM repository include:

- **Reliability:** The SCM repository stores some of the company's most important assets and the data must be kept in a secure and reliable data repository. The SCM repository must recover quickly and automatically from power interruptions and must support transaction-level recovery from media failures.
- **Scalability:** The SCM repository needs to scale up to very large sizes to meet the needs of large software teams.
- **Availability:** For many development organizations, the SCM repository needs to be available on a non-stop basis as there are often build, test and release management jobs running at all hours.

- **Transparency:** The SCM repository should provide for the data that it manages to be projected into the file system so that users, tools and utilities continue to access and modify the data much as they would without an SCM repository.

7.6 SCM Tools

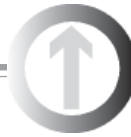
SCM is no longer a manual process, but instead has been automated with SCM tools. Some modern SCM tools that are available in the market today are discussed as follows:

7.6.1 Configuration Management Assistant

Configuration Management Assistant (CMA) is a product of Tartan Laboratories for creating CM systems. It uses an Entity-Relationship-Attribute database. Classes of attributes and relationships define the characteristics of components and the decomposition of a product and interdependencies between components. CMA provides for recording and retrieving descriptions of configurations. It also describes the set of components that record and retrieve information about known consistencies and dependencies between components. CMA predicts completeness, ambiguity and consistency of newly formed configurations.

7.6.2 Adele

Adele is a configuration management system from the University of Grenoble. Its basic features are data modeling, interface checking, representing families of products, configuration building, and workspace control. The Adele database is an Entity relationship that provides for the definition of objects such as interfaces and their realizations, and configurations and families. Objects have attributes that describe their characteristics and the *dep* relation that describes their dependencies, which Adele uses to assist in composing a configuration.



SUMMARY

- Software configuration management is the discipline for systematically monitoring and controlling the changes that take place during the development process within an organization.
- Software configuration management is a discipline that controls the evolution of software systems.
- The primary goal of SCM is to recognize and regulate changes, and ensure that the changes are being properly implemented and reported to those interested.
- Software configuration management begins right after the project is begun and ends when the software is being taken out of operation.
- The items that comprise all information produced as part of the software process are collectively called a **software configuration** and each individual item is called a **software configuration item**.
- A **baseline** is a software configuration management concept that helps us to control change, without seriously impeding justifiable change.
- Before a software configuration item becomes a baseline, changes may be made informally and quickly.
- Once a baseline is made, changes can be made only following a formal procedure.
- With respect to software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that are obtained through a formal technical review.
- Five tasks that are very important to SCM are:
 - Identification
 - Version Control
 - Change control
 - Configuration Auditing
 - Reporting

- The SCM repository is the heart of any SCM system. It stores all the project objects, each version of each object, and the meta-data that describes each version of each object.
- Key requirements for any SCM repository are:
 - Reliability
 - Scalability
 - Transparency
 - Availability



CHECK YOUR PROGRESS

1. A _____ is a unit of text that has been created by a software engineer during analysis, design, coding or testing.
2. A _____ is a software configuration management concept that helps us to control change, without seriously impeding justifiable change.
3. A _____ is an entity that is provided, processed, referenced or otherwise required by the object.
4. The _____ focuses on the technical correctness of the configuration object that has been modified.
5. The primary goal of SCM is to recognize changes, regulate changes, and ensure that the changes are being properly implemented and report the changes to those who may be interested in them. **[True/False]**

Get
WORD WISE



Visit
Glossary@

www.onlinevarsity.com



Objectives

At the end of this chapter, you will be able to:

- *Describe the different types of maintenance*
- *Describe various maintenance issues*
- *Explain the maintenance metrics*
- *Describe the working of a typical maintenance organization*
- *Describe various factors affecting maintenance*

8.1 Introduction

Let us assume that we own an automobile and one fine day it breaks down. The most likely action we would perform is to take it to the garage and get the vehicle checked for possible repair works. What will the automobile engineer do in the garage? The engineer will analyze the problem, identify the impacted items, repair or replace them, and test the car to see if the problem was resolved. A set of documents will give details of the work done by the engineer. This is automobile maintenance. Software maintenance is identical; whenever the user of a software package comes across a problem in the functioning of the software or even an enhancement need; he contacts the Software Maintenance department of the organization. The analyst in the department identifies the impacted items, corrects them, tests the package, and requests the user to check if the package now meets all the requirements.

8.2 Software Maintenance

The process of software engineering does not stop with the delivery and installation of software but also includes maintenance activities. Maintenance activities involve **making enhancements** to the software products developed in the earlier stages of the life cycle, **adapting products** to new environments and **correcting problems**. *Enhancement* of the product may in turn involve providing new operational capabilities, improving the user screens and modes of interaction and improving the performance of the system in general. *Adapting the product* may require moving the project to a different machine or platform, or accommodating new protocols or even new hardware components like an additional disk drive. To *correct problems*, we may need to modify the code and revalidate the software. It has been practically observed that maintenance activities account for a large portion of the system life cycle costs, even as much as 60-70 percent of the total costs set aside for the project. Software maintenance like software development requires a combination of managerial control and technical expertise.

8.3 Types of Maintenance

Though maintenance is a single-phase activity, different tasks are carried out during this phase. Maintenance activities can be classified under four broad categories:

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive maintenance or Reengineering

Let us now discuss each of these categories in detail.

8.3.1 Corrective Maintenance

This type of maintenance activity, also called **breakdown maintenance**, involves testing and diagnosis to determine whether the system is performing exactly as it was supposed to. In case this is not so and the system does not perform as it was expected to perform, then corrective maintenance involves designing and making changes to get the system to do what it was expected to do. Thus, corrective maintenance means fixing bugs or rectifying design flaws, if any. Unless this activity is carried out, further processing on the system will not occur. That is the reason it is called breakdown maintenance, signifying that the system has suffered a breakdown and that immediate rectification is required.

8.3.2 Perfective Maintenance

This type of maintenance activity involves designing and enhancing the system to perform better than it was originally expected to do. At times, the requirements could have been misunderstood or perhaps newer requirements have been suggested. Accommodating this may require that the system be changed and enhanced. This constitutes perfective maintenance.

8.3.3 Adaptive Maintenance

Adaptive maintenance is a subset of perfective maintenance and comes into picture when we wish to take advantage of latest technological advancements. It involves adjusting the application to changes in the environment, such as new hardware equipment and a new operating system.

Adaptive maintenance necessitates changes in the software due to change in the environment or infrastructure and lack of functionality or any bugs in the software.

8.3.4 Preventive Maintenance

Preventive maintenance concerns changing the software in order to make it more maintainable. Preventive maintenance is also called reengineering. Reengineering involves two steps:

- *Reverse engineering*, which involves extracting the functional and design specifications from the code.
- *Forward engineering*, which involves reviewing and improving the functional specifications and design obtained after reverse engineering.

An artist is commissioned to make a portrait of a royal personage. Daily sittings are held in the artist's studio with the royal suitably dressed. About 3-4 months later, the portrait is ready. However, the artist's job is still not over. A number of scenarios can occur here:

- a) There are some visible defects in the portrait. The depiction of the royal in the picture does not exactly correspond with his features. For instance, his nose appears longer in the portrait. The artist has to correct this by making appropriate changes to the picture.
- b) The appearance of the royal has changed (such as growing a moustache where earlier there was none). He wants these changes to be incorporated in the picture.

Is this similar to software maintenance? To what category do each of these cases correspond?

8.4 Categorization of Maintenance Requests

Maintenance requests can arise from users, customers or the technology department itself. Whatever be the source of the maintenance request, the categorization of a maintenance request is generally as follows:

Emergency Fix: The system has come to a grinding halt and needs to be fixed immediately so that business can go on as usual. A car tyre puncture is an analogy of an emergency fix- we need to fix it to move ahead.

- **Urgent Fix:** Needs to be fixed but operations can proceed for few days until it is fixed. An analogy for the urgent fix is a car with a weak battery; this can wait for few days to be corrected.
- **Nice to have fix:** This needs to be done on the system to improve performance or ease operation bottlenecks. Business can proceed without this change in software. Having a stereo in a car can be an example of Nice to have fix.

The priority is fixed by the head of the Maintenance department based on agreed guidelines with the customers. The problem is handled in one of the following ways based on categorization:

- For emergency fixes, a temporary program or data fix has to be done immediately to get the show going. Such emergency operations are carried out with extra caution by ensuring that two people are present when the change takes place, and logging all commands and its results on a file for later analysis by an expert. The temporary fix will have to be followed by a permanent fix after the problem is properly diagnosed.
- For urgent fixes, a team of experts is handed over the problem. They analyze the problem and make an effort estimate to carry out the required modifications. The delivery date is discussed with the customer. If the delivery date is accepted by the customer, work is initiated to make the required modifications. If the customer feels that the delivery date needs to be expedited, it is reviewed to see if a temporary work around is possible until a permanent solution is made available.
- 'Nice to have fixes' are undertaken by the maintenance team whenever they have spare capacity. Otherwise, they are handed over to the deployment team for incorporation in the next software release.

8.5 Maintenance Issues

The main problems encountered in Software Maintenance are as follows:

8.5.1 Dynamic Personnel

The community of software engineers is very dynamic and mobile. The trend of job-hopping is very popular among them and this leads to a variable team of personnel. In such cases, the persons performing maintenance may not be those who developed the software. These persons will lack the in-depth skills required for troubleshooting.

8.5.2 Motivation and Morale

Low morale among maintenance personnel is one of the major issues related to maintenance. A common myth is that maintenance is one of the most boring and dull tasks in the software engineering field. The reality however, is quite the opposite. Maintenance can be quite challenging and interesting. At the same time, prominence is given to the developers in the team rather than maintenance engineers. The work is hard and the pay is low, thus leading to overall dissatisfaction and low morale.

8.5.3 Lack of Documentation

Very often, it so happens that technical documentation is never updated. Regular, timely documentation is necessary, failing which, the maintenance engineers have no clue of the internal design and interface of the system. They may have to resort to guesswork or trial and error methods to work out the problem. This may consume unnecessary time and effort.

8.5.4 Patchy Code

This problem is faced in the maintenance of the software, which is not written as per standards in a structured manner. It becomes extremely difficult to understand the logic of the program. As more and more maintenance is done on the software, the code becomes even patchier.

8.5.5 Outdated or Obsolete Technology

Maintenance is generally done on an older version of the software, which was developed long back. For instance, assume an application is developed in Visual Basic in a 6-month period and undergoes maintenance for the next 2-3 years. During this time, newer technologies like VB.NET would have emerged, making the earlier version of VB obsolete. The software team involved would more likely want to work with emerging technologies than on an obsolete piece of technology.

8.5.6 Round-the-clock Operations

Maintenance is not a nine-to-five desk job but involves round-the-clock assistance. A maintenance engineer will have to work in shifts and be available on call at any time if there is an emergency on the system.

8.6 Maintenance Metrics

Data has to be collected periodically and analyzed to work out if service levels offered are at acceptable level, productivity is good, and down time is minimum. The data must be sent to the head of the maintenance for analysis on a periodic basis, say weekly or monthly. Data analysis should provide quality information to enable objective decision making as described here. In most organizations, the maintenance department signs up Service Level Agreements (SLA) with customers, which is a contract that gives us details of system availability, turnaround for maintenance, and quality of deliverables. These are specified in a quantitative form so that it can be measured against the metrics collected by the maintenance department.

- **System availability:** The data will give details about the percentage time the system was not available, reason for non-availability, whether it was planned or unplanned, and whether an alternate service mode was made available during down-time or not. This data will be compared with the SLA. If it is higher than the SLA, there is cause for alarm.
- **Maintenance Turnaround:** The SLA would have given indicative turnaround times for various categories of maintenance. For example, the SLA can state that an emergency fix will be resolved within two hours of the problem being reported. The periodic reports should give deviations from the SLA and average them over the period. This will indicate if the overall service is at a level of acceptance. Specific instances, which have gone beyond the SLA acceptance criteria must be investigated and corrective action taken to prevent recurrence.
- **Productivity:** The data results in information about the number of errors in each category that were resolved by a software maintenance engineer. As part of the goal setting exercise for individuals,

productivity norms can be set. These can be measured against actual values based on the data from maintenance reports to analyze performance and productivity of various engineers. Based on this data, an appropriate reward and motivation scheme can be worked out.

8.7 A Typical Maintenance Organization

There is no such thing as the right organization structure, organization structure varies from business to business depending upon the needs of the organization. However, some activities will be common in any typical organization:

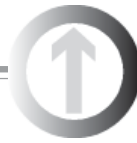
- All maintenance reports are sent to the head of maintenance, called the maintenance manager or maintenance controller.
- The maintenance controller passes the maintenance details to the System Supervisor or Maintenance Project Leader, who is responsible for problems in that area.
- The systems supervisor makes an impact analysis and estimates the effort to make changes.
- These estimates are ratified by the Maintenance manager and passed to the software maintenance personnel for execution as described under “Movement of Configuration Items”. In large organizations, the changes are not approved by the maintenance manager but by a group of people or committee called Change Control Authority Board. This Board intimates the librarian of the changes. The librarian releases the impacted configuration items to the maintenance staff. In small organizations, a few personnel may hold this responsibility along with other responsibilities. In large organizations, they may be full time responsibilities.

8.8 Factors Affecting Maintenance

The maintenance phase is the most expensive phase of the SDLC. This is due to:

- Enhancements, performances, improvements and updates to new revision
- Time spent in understanding the system by new personnel
- Non-availability of proper documentation
- Every new maintenance activity increases the chance of errors

Experts estimate that productivity drops by a factor of 40 in maintenance mode as compared to development mode, when measured in terms of lines of code or function points per day.



SUMMARY

- Maintenance activities involve *making enhancements* to the software products developed in the earlier stages of the life cycle, *adapting products* to new environments and *correcting problems*.
- Maintenance activities account for a large portion of the system life cycle costs, even as much as 60-70 percent of the total costs set aside for the project.
- Corrective maintenance involves testing and diagnosis, then designing and making changes to get the system to do what it was expected to do.
- Perfective maintenance involves designing and enhancing the system to perform better than it was originally expected to do.
- Adaptive maintenance is a subset of perfective maintenance and comes into picture when we wish to take advantage of the latest technological advancements.
- Preventive maintenance is also called reengineering.
- Issues related to software maintenance include:
 - Lack of documentation
 - Dynamic personnel
 - Motivation or morale
 - Patchy code
 - Outdated technology
 - Round-the-clock operations
- Maintenance metrics can be calculated based on the following factors:
 - System availability
 - Maintenance turnaround
 - Productivity



CHECK YOUR PROGRESS

1. _____ maintenance is also called reengineering
2. _____ involves designing and enhancing the system to perform better than it was originally expected to do.
3. Maintenance is not a nine-to-five desk job but involves round-the-clock assistance.
[True/False]
4. _____ of the product involves providing new operational capabilities, improving the user screens and modes of interaction and improving the performance of the system in general.
5. Maintenance is the costliest and longest phase in the SDLC **[True/False]**
6. 'Nice to have fixes' are undertaken by the maintenance team whenever they have spare capacity.
[True/False]

Software Implementation and CASE

Objectives

At the end of this chapter, you will be able to:

- *Analyze why Implementation is an important phase of SDLC*
- *Discuss the activities of Implementation Phase and Implementation Strategies*
- *Discuss the need for Contingency Planning and Post Implementation Maintenance*
- *Learn the meaning of CASE and its evolution*
- *Discuss the classification of CASE*
- *Correlate the use of CASE to software engineering concept*
- *Discuss CASE Toolkits, CASE Methodology companion, CASE Workbench, and Repository features*

9.1 Introduction

In the classic life cycle, a lot of emphasis is laid on all other phases like analysis, design and coding. Implementation is normally relegated to the background and is generally talked about in a rather cursory manner.

One might therefore be tempted to think that implementation is rather a routine and straightforward affair- after all, we only need to take the software that has been thoroughly tested by the development team, install it in appropriate places and make it run.

Life is never quite as simple as that! It is estimated that more than 80% of the problems encountered in software development project are thrown up only at the time of software implementation.

In the implementation phase, a number of activities running in parallel have to culminate. The implementation phase ensures that all these activities are provided converge at the right time and at the right place without any delays on account of unforeseen problems, and that the software begins functioning as it should.

Besides systems and business issues, there are personal issues. Users are apprehensive of computers, 'afraid' of learning technology, fearful of technology savvy juniors, fearful of being termed redundant, and so on. These emotional issues complicate the process of implementing a new system within an organization. Although not beyond redemption, these issues should be accorded due importance and handled accordingly.

Therefore, during the implementation phase, besides physical implementation of hardware and software, the analyst has to more importantly ensure that the new system is implemented in the minds of the users and accepted by them.

9.2 Activities of the Implementation Phase

Unlike the other phases, there are no standard methodologies for implementation. However, **Implementation Planning** is an important activity of the Software Development Life Cycle (SDLC). The specific approach adopted for implementation is largely dependant on the size of organization, the nature of applications and the standard practices that may have been evolved by the project team.

However, despite such individual variations there are some kinds of standard activities to be performed during the implementation process:

- Creation of an Installation Plan
- Implementation of Physical Procedures
- Data Preparation and Conversion
- Conducting User Training
- Running the system

9.2.1 Installation Plan

Preparing a detailed installation plan, is the beginning of the implementation process. The installation plan includes schedules for:

- Purchase of hardware
- Preparation of site – wiring, providing for communication points, air conditioning and so on
- Purchase of environmental software
- Installation of hardware and environmental software
- Installation of the developed software
- Training

Depending upon the implementation strategy adopted, the time at which these have to be installed may also differ. Therefore, a comprehensive installation plan needs to be in place, which gives location wise installation details as relevant to different user groups.

9.2.2 Implementation of Physical Procedures

Physical Procedures are those that an organization will follow side by side with the automated system and which will complement its functioning. These procedures also spell a change from the past and have to be in place before the automated system is implemented. Some of these systems include change in coding schemes or introduction of a coding scheme, switchover to changed input and output document formats, change in sequence of operations and other such.

The successful implementation of systems depends as much on the adaptability of the people using them, as it does on the quality of software, and hence the need for use of Physical Procedures.

Users need to adapt to the new way of functioning. For instance, in the old system of functioning there would have been multiple paper copies of various documents, but in the new system the paper copies have been eliminated. Documents are stored in one place within the computer and retrieved when required. Sometimes users are so used to writing out documents that keying them into the computer is not security enough for them, so many users may want to continue making documents manually too (at least initially). The absence of paper can create a vacuum and the users may need time to get used to this paperless concept. The best way to deal with this kind of user reluctance is to make them aware of the same, and provide training on these changed procedures before implementing the new system.

Physical procedures are an important aspect of implementation because it spells the changeover from old to the new system and requires handling of user training on the same.

Physical procedures are handled by spreading awareness of the new system through well-designed training programmes and handling user queries skillfully and patiently.

9.2.3 Data Preparation and Conversion

Data Preparation is normally the most time consuming and tedious task in the implementation of most systems. During the implementation of a new system, all data backlog has to be transcribed to fit into the new formats and coding schemes, and errors are rigorously checked to ensure no loss of information.

Data conversion is the process of converting data from the old system into a format required by the new, automated system. This usually amounts to creating the master database and entering the master data in the record formats that have been designed for the system, from the documents that exist in the system. As a matter of fact, organizations will not want data entry people running around for data as it shifts their concentration from what they are doing, which may induce errors in the data that is being entered. Data should be consolidated and validated by someone responsible before it is entered into the system.

Data conversion is required even if an automated system already exists. In such case, a program maybe written to read data from the existing format (if the old data format is readable by new software) into the new format. Alternatively, data has to be physically read from printed dumps of the old system master file and entered into the new system. Even if the old data formats are not compatible with the new DBMS formats, a small program can be written in a language like C, which can handle any type of data formats. This may become a cumbersome procedure and should be planned for in advance.

The process of data conversion requires the following:

1. List of all existing files to be converted (for instance, some level of automation existed in the accounts function so some data required for our system's invoice related databases will need to be converted from an existing file format into the new file format)
2. List of new files to be created and data required for it (for example, if an existing system did not have file for storing product discount details, but the new system requires it)
3. List all new documents and procedures that go into use during conversion
4. Identify controls for the conversion process, how to determine that all records have been entered in the new database
5. Prepare conversion schedule
6. Assign responsibility for each task

The importance of data conversion is characterized by the fact that the bulk of master data creation takes place through conversion at this stage. If care is not taken and errors creep in then there will be problems in operating the system with unclean data later. This needs a set of controls to be designed specifically for conversion. Controls like keeping track of the total number of records that have been entered in comparison with the physical number that exists in the old system. We will also need to check that complete data is entered, all field values are entered, the type of value is validated and likewise. This stage has to be planned for and performed in as structured a manner as the other stages of analysis, design and programming. Data conversion seems like an easy task, but it is very difficult to track data from all the user sources to ensure that everything has been entered in the system.

9.2.4 User Training

The successful implementation of any software is dependant in good measure on the quality of training imparted to the users of the system. Different user groups need to be identified.

A *Training Need Analysis* has to be done for each of these groups to find the kind of training that is required for each. Training Needs Analysis can be considered as the process of determining individual or group training needs for an organization. There are usually several types of users within an organization, and all types of users do not need all kind of training.

Usually two types of users can be identified for the system:

- Hands-on users
- End users

The hands-on category needs to know details of each and every input format, data validation, source of data, transactions and their frequency, meaning of error messages, and some amount of troubleshooting. They have to understand the automated system in its totality and the interaction of their application subset with the application superset. The hands-on users in an organization can be the order-entry clerk, the

invoicing clerk or any such person.

End users do not operate the system on a day-to-day basis, but use the information churned out by it. Typically, this category comprises of senior managers who do not operate the system for its transactions but use the query and report modules. The end-users need to understand the overall functioning of the automated system, the meaning of all reports and queries, and understand the information that is presented therein. The end users in a typical organization can be the managers of each of the departments.

Training should be provided at two levels:

- Training for systems operators
- Training for users of the system

If these categories are the same, then the same type of training is imparted. Sometimes, the requirements of various levels of management also decide the type of training to be imparted. For instance; senior management personnel may be trained on the important aspects of the system, while the junior level operations personnel may be given a detailed training.

In the current scenario, most users are also operators so the differentiation between users and operators is fast disappearing. However, it remains relevant for very large systems like airline reservation system. Training for systems operators should emphasise:

- How the equipment has to be used – mounting of tapes and disks, changing printer stationary, copying files (for backup or otherwise)
- Minor troubleshooting – what each error message means and the associated remedial action
- Maintenance activities and their frequency

Training for users should emphasise:

- How to use the equipment – for novice users
- The application – a detailed explanation of the application and what the users can expect from it
- Data related training – various screens, forms, reports and the like
- Addition, deletion and editing of records – how to store new transactions, change previously stored data and delete unwanted data
- How to retrieve information from the system – correlation between users queries and queries designed for the system, formulating queries
- How to utilize the information generated by the system
- Elementary troubleshooting – a user should be able to recognize whether an error is hardware or

software related or caused by an action of the user

9.2.5 Running of the System / Parallel Run

“Parallel Run” signifies the running of two systems together, the old and the new.

In the *parallel method of conversion* the old system is operated along with the new system for a while, until all the teething problems of the new system are sorted out and users are confident of operating the new system.

The period of parallel run varies from organization to organization and from application to application. Although there are no hard and fast rules for this phase, it is attempted to ensure that the following factors are looked into:

- There are no major faults in the package
- Important figures tally
- All modules are used and found satisfactory
- There is fair amount of confidence amongst users in being able to use the software

9.3 Implementation Approval

After the new system has been implemented in the user environment, the users give implementation approval when they are fully satisfied by the installed system. Implementation approval is the approval given by users when user acceptance tests are completed whereby the users have fully tested the system, and uncovered all errors that should have been fixed. This is done before implementation. After implementation, approval is given for the final take over of the system, the sign off of the contract.

9.3.1 Post Implementation Review

Post implementation review is conducted after the system is implemented and conversion is complete. This review should be conducted formally as it enables the analyst / designer and users of the system to understand whether or not the system is working well, and how has it been accepted. The feedback received after the review will be used as inputs to maintain the system later. The post implementation review is the first source of information for maintenance requirements. Maintenance requirements are those areas of the system, which will need regular supervision and intervention of the maintenance team. If left unattended, they may lead to system dysfunction.

Issues that are evaluated during the post implementation review are much the same as were discussed and evaluated at the feasibility stage. Some of these issues are:

- Has the information system reduced / increased the operating cost of an application?

- Does the system provide accurate and up-to-date information in desirable formats?
- Has the new system improved the way systems were operated earlier – taken out redundant tasks, eased data entry and reduced paper work?
- Does the new system need more or less number of users / operators than before?
- Has the new system affected the procedures of the organization – introduced new way of doing things?
- Has the new system improved productivity?
- Has the new system improved service to customers – answering customer queries, servicing the orders, and so on.

Managing each of these activities is critical to the successful implementation of any software. Hence, we shall now discuss the various implementation strategies and the problems encountered during their implementation.

9.4 Implementation Strategies

There is no standard strategy as far as implementation of software is concerned. Some the implementation strategies being commonly used are:

- **Parallel run** -In the parallel method of conversion, the old system is operated along with the new system for a while, till all the teething problems of the new system are sorted out and users are confident of operating the system:
 - Its advantages are that it offers great security and minimizes loss of data due to unknown errors unearthed in the new system. It is the safest way of converting from an old system to a new one.
 - Its disadvantages are that it doubles operating costs for a while. So long as the old system continues to run along with the new, the latter does not get a fair chance of trial.
 - On the one hand the system does not get a fair chance of trial, on the other hand it inspires user confidence when the user sees that results of the new way of processing are the same as that of the manual system.
- **Direct Cutover** - The direct method of implementation means that the old system is replaced with the new and the organization relies on the new system from the beginning itself. A cutover data is decided when the new system takes over and the old system ceases to function:
 - Its main advantage is that the organization immediately starts to see the benefits of the new methods and controls that have been included in the new system.

- The disadvantage of this method is that there is no other system to fall back on, if a problem arises in the new system. Due to this reason, the new system has to be planned, developed and implemented with utmost care.

Direct cutovers are common for systems like hotel reservations and airline reservations. Direct cutover is the best way of introducing a new system into an organization like Materials and Requirements Planning systems. An MRP package is an integrated package, and in most cases requires the organization to adapt to its procedures in a broad sense.

- **Phased Approach** -The phased method of conversion installs a new system throughout an organization in phases. It is best suited to systems that are modular, and a module at a time is developed and installed. So one set of users get to use a part of the system before others:

- Its advantage is that training can be designed for users keeping the function/module in mind.
- Its disadvantage is that a long phase-in period causes unnecessary problems like bad word of mouth by a set of users who have faced problems with a part of the system.

- **Pilot Approach** - The pilot method of conversion means introducing a new system in one part of the organization – a single department or function. Based on the feedback received, changes are made to the system before it can be installed in all parts of the organization. This is most applicable when the system is absolutely new to users:

- Its advantage is that it provides for live testing before implementation.
- Its disadvantage is that if conversion is not handled well and takes too long to be implemented in the whole organization, the users may get the impression that the new system is not sound and is full of errors. Figure 9.1 illustrates the use of various implementation strategies:

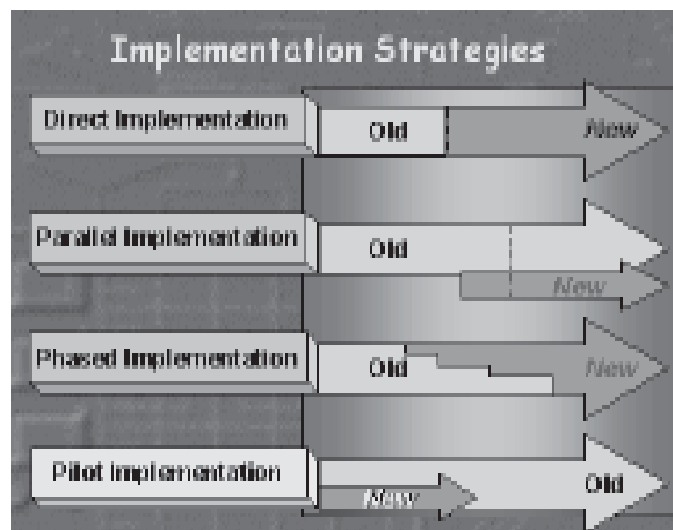


Figure 9.1: Implementation Strategies

In general it can be said that:

- Parallel run is a “play safe” strategy.
- Direct cutover is a clean break from the past and is used for systems that do not need an option of fall back on the old system.
- Phased approach is best used for systems that are clearly modular and can be implemented in an organization module at a time.

9.4.1 Choosing an Implementation Strategy

There is no best strategy as such. The important point to note is that every organization must choose a strategy that is suited to its needs. The strategy for implementation depends on the nature of the application, the user profile of the organization, and a host of other factors discussed as follows:

- **Nature and size of application** - Applications that are modular and each module affects only particular category of users who are best suited to phased implementation. Applications that are being introduced for the first time like an information kiosk, are better suited for direct cutover implementation.
- **Type of organization** - This includes the organization size, culture, and end user profiles. Organizations that are young, small and in the frontline businesses – banking, financial services are more adaptable to change and new system of working, whilst organizations that are larger and old and have experienced years of functioning with manual systems are more difficult to change. For the former it is also the need of the hour, competition pressure from peer organizations, or information on fingertips that propel them into automation very fast.

- **Existence of an internal EDP department** – The EDP staff can oversee the functioning of a new system and handhold their apprehensive colleagues till the latter are completely satisfied with the new system. However, if the EDP staff is cynical of the new system – they have not been consulted on the new system or their suggestions have not been given due consideration, then whatever the implementation strategy they can make things difficult. With the help of internal EDP staff, users can be quick off the mark with the new system and the parallel run tenure can be greatly reduced. Alternatively, the feedback of the ‘phase in’ strategy will be positive or we can even think in terms of having a direct cutover.
- **Number of personnel involved in usage of software** – If the number is significantly large, then a ‘phase in’ strategy can be deployed by user profile – one set of users initiated/trained on the new system at a time.
- **Data volumes** – Play safe when a data volume is large.
- **Location spread of the application** – A widespread organization needs a phase-by-phase implementation strategy.
- **Availability of certain tools** – The tools available in software ensure its security, safety and integrity. The more such tools are available, the sooner one can get off the old system and gain confidence in the new system.
- **Manpower availability** – The implementation process needs manpower. If a system has to be installed in different locations simultaneously, then more than one implementation team that is fully aware of the system is needed. More teams than one are not only difficult to form, but they also need more money. When this is not possible, simultaneous implementation at more than one location should not be taken up.
- **Criticality of the application** – More critical applications need more safety, so a parallel backup system is needed.

9.4.2 Data Conversion Strategies

Apart from the implementation strategies discussed, there are also some Data Conversion strategies that must be taken into account when implementing a new system. Data preparation and conversion is one of the most time consuming activities of implementation. The method adopted for this activity depends on various factors such as:

1. When data is available as part of some existing application

In such cases, the data is generally not re-entered. A set of conversion programs is written to convert the existing data into new formats. This process gets even more complicated if coding schemes and techniques are different. This will entail building up necessary look-up files for code conversion. Conversion Testing is performed to check whether the data conversion has been carried out properly.

2. When application is being automated for the first time

Here, the data entry programs, which are the part of the application itself can be used to enter data. This is a straightforward and foolproof method, because the data entry programs will themselves take care of all the error checking and validations.

3. When data volumes are very high

Screen oriented data entry becomes very time consuming for large volume data. In these cases, it is common to use special data entry packages to facilitate high speeds of data entry by professional data entry operators. A set of programs is then built to validate data thus entered, if necessary.

1. **Suggest the suitable implementation strategy to be adopted in the following situations giving reasons:**
 - a) **An organization is getting connectivity with its worldwide operations through Lotus Notes**
 - b) **A transportation company wants to implement a goods movement and tracking system integrated with a warehouse management system**
 - c) **An organization wants to install a payroll processing system in all its geographically spread offices**

9.4.3 Contingency Planning

A contingency plan is necessary to provide a fallback system in case of a crisis. The fallback system is usually an alternate of the following types:

- Another backup machine
- Leased line
- Service from another location
- Manual system.

Criticality of the application determines contingency plan. A crisis can occur at any stage – after a month of operating a system, after 10 months, after 2 years. Neither is it required nor can we keep running the manual system in parallel for so long. A contingency plan provides the option of falling back upon the manual or alternative system in case of a failure.

For example, take the case of an airline reservation system – in case of a major failure, tickets cannot be held up even for a few hours, so the option of preparing tickets on an alternative network / service provider will have to be put in its contingency plan. The contingency plan is really meant for rare situations when nothing else will work, critical systems have all sorts of backup and do not usually end up stalling just like that

9.4.4 Post Implementation Maintenance

Post implementation maintenance commences when an organization has accepted the system and begun its operations. It is affected through the signing of an Annual Maintenance Contract with the organization that developed the system. If development was done in-house, post implementation maintenance will have to be done by the in-house team.

Post implementation maintenance is necessary for the following reasons:

- Troubleshoot problems that will be encountered by users when they run the system on a day to day basis
- Modify the system when and if bugs are discovered
- Modify the system for changed requirements of the user

9.4.5 Documentation

As in all other phases of SDLC, proper documentation is extremely important in the successful execution of implementation phase and post implementation usage of the software.

Documentation for a system usually consists of:

- **Installation manual** – contains the instructions to install the system, requirements that should be met before particular software is installed and so on.
- **User manual** – detailed instructions to the user for operating the system. It outlines for each task:
 - What the user is expected to do in order to invoke a particular function/module and after it is invoked?
 - Functions/modules that should be invoked and when – for instance, what options to invoke to view data on customer outstanding?
 - How to avoid the errors that occur?
 - Explanation of error messages of the developed system (example - “Product Code not found...please enter again”).
 - Expected result of each function/module.

Chapter 9

Software Implementation and CASE

- **Operations manual** – Outlines various operations issues like how and when to take back up, when to delete and purge data, how the data has been coded, security controls provided in the system, explanation of system level error messages (“insufficient memory”, “insufficient disk space”).

Having understood the various aspects of software implementation, we shall discuss the importance of using CASE TOOLS.

9.5 Introduction to CASE

CASE stands for Computer Aided Software Engineering – literally. In simple layman terms, CASE is a tool, which aids a software engineer to maintain and develop software.

There are variations in what the acronym’s CASE expanded form is – Computer Assisted/Automated Software Engineering. A CASE tool is software that implements the principles of software engineering by automating it. A CASE tool enables people working on a software project to store data about a project - its plan and schedules, to be able to track its progress and make changes easily, analyze and store data about user requirements, store the design of a system – all through automation.

Software engineering is all about automating an application. CASE is about automating those software engineering processes that aid in automation of applications.

Use of tools is not new to the software engineering process. We have the data flow diagram as a process analysis tool, the ERD as a data analysis tool, and the structure chart as a design tool. When we mention a CASE tool today, we necessarily mean an automated version of tools that is an automated data flow diagram, ERD, and the system chart plus synchronization of data across tools of all phases of the systems development life cycle.

9.5.1 Understanding the Case Technology

CASE technology has come a long way since its inception in the early 1980s, when all it had to offer was a set a standalone drawing tools. Today a wide range of case tools is available in the market. Different types of case tools address different areas of software engineering. CASE tool technology can be split into three levels:

- **Production-process support technology.** This range of tools support process activities such as specification, design, implementation and testing.
- **Process Management Technology.** This range of tools support process modeling and process management. This range of tools will interact with ‘Production-process’ support tools for specific process activities.
- **Meta-Case Technology** -These tools are used to create Production-process and Process

Management support tools. Some tools are available but they have not been widely adopted by the industry as they are difficult to use.

A CASE tool can be a simple single tool supporting a specific software engineering activity, or it can be a complex tool consisting of other tools, a hardware platform, a database, an operating system, portability services, a network framework, and many other components. A complex tool is said to be a complete development environment since it provides all kind of support and automation services to software development life cycle. Figure 9.2 illustrates this CASE Environment:

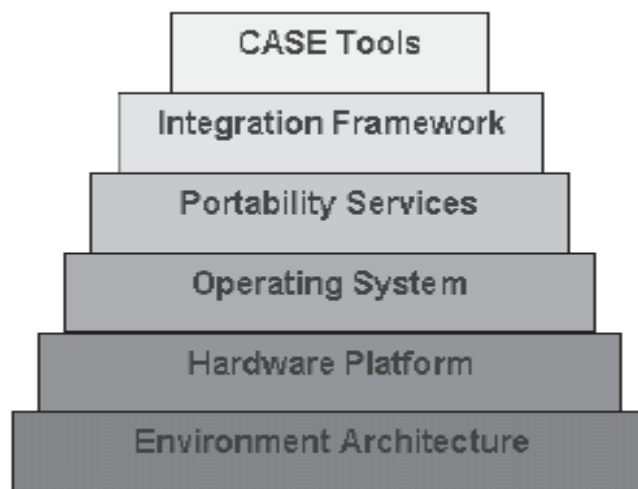


Figure 9.2: Complete CASE environment

9.5.2 The Need for a CASE Tool

The need was felt in the software industry to automate the development process due to the following reasons:

- The size of projects was becoming unwieldy as more and more organizations undertook automation – it was becoming necessary to automate the development process itself.
- Software engineering procedures require systematization of the development process and immense documentation – integration of project management with analysis, design, testing, implementation and so on; only an automated tool can handle the complications and the load.
- “User is king” – user requirements became the fulcrum of automated systems so these had to be tracked very carefully, and changes in requirements applied to the development process meticulously. Tracking is meticulous and easy when recorded and documented well – nothing other than an automated tool can achieve this level of perfection.

Chapter 9

Software Implementation and CASE

- Software engineering supports the principles of reusability, flexibility, maintainability of programs – if the system data is not documented well how will we achieve this?
- “Zero defect” software, failsafe systems, software used in mission critical applications – the development process needs clear understanding and documentation; there is no scope for error.

These points put the development process under a spotlight. How should the development process be managed well? The most fitting answer to this is “**automate it**”. And this is the genesis of CASE.

1. Explain:

- a) Why do cameras come with automatic settings?
- b) Why have graphic artists given up paper and sketch pens in favor of Multimedia technology.
- c) Why have households taken to washing machines and dishwashers?

9.5.3 Evolution of CASE

CASE too has its evolution history. A young concept in an even younger industry, CASE tools of the primitive type was first seen in the early 1980s.

- Early CASE tools had diagramming capabilities – these could aid the drawing of a DFD, ERD and a structure chart and the like. These tools could store basic information about each of these diagrams – names of data flow, entities, processes, modules etc. However, these were still unable to point out missing elements or inconsistencies across diagrams. The diagramming tools could not integrate with the one another and cross-diagram validations were also not possible.
- The next stage in the evolution of CASE, late 1980s, saw the development of tools that not only could store diagram information but also perform cross-diagram checks through a repository. This helped analysts and designers in maintaining consistency in naming and ensuring that no entity or data was missed or duplicated. This level of CASE tool still kept the development process distant from the project management data. The project management data was still only available to the project manager since the tool was not able to store project related information.

Tools conformed to a particular development methodology. Having selected a particular methodology tool, the user had to go by its principles, use the diagrams, symbols and other conventions of the methodology.

- The 1990s stage in the CASE tool evolution saw the integration of project management and system life cycle activities. The tool also included automatic code generation to some extent. The term CASE tool gave way to “CASE Environment”. A CASE environment includes models, tools and techniques that are applicable to software and system engineering on both large and small-scale projects. It is now possible to customize the CASE environment by selecting tools from an available set to suit our need. This means that we do not have to follow a single methodology. The CASE environment is able to seamlessly integrate different tools with no problems at all.

The late 1990's saw the evolution of a CASE framework where different CASE vendor products could be integrated to customize the CASE environment.

9.6 CASE Environment

A CASE Environment consists of a number of CASE tools operating together on the same hardware and software. There are many different classes of CASE environments, which depend on the types of CASE tools used in the environment.

A CASE environment makes system development *economical* and *practical*. The automated tools and environments provide a mechanism for systems personnel to capture, document, and model an information system. This mechanism is provided right from system inception as user requirements to its design and implementation. They provide a project team with powerful analytic tools to ensure consistency, completeness, and conformance to standards.

Definition:

A CASE Environment is a number of CASE tools, which use an integrated approach to support the interactions between the environment's components and the users of the environments.

Figure 9.3 depicts the architecture of a CASE environment:

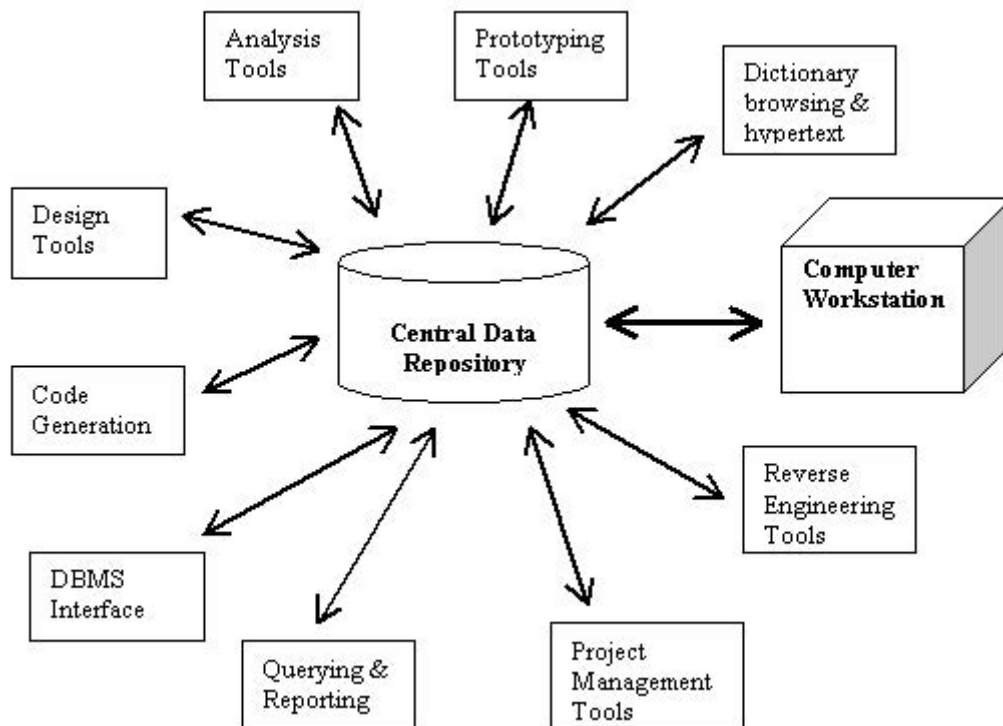


Figure 9.3: CASE Environment

9.7 CASE Terminology

Let us understand some of the important terms associated with Case:

- **CASE Technology**- This software technology provides an automated engineering discipline for software development maintenance and project management, and includes automated structured methodologies and automated tools.
- **CASE Tool** - This automates a particular activity in the software development process – for example, automation of the diagramming activity or automation of the testing process or prototyping
- **CASE Toolkit** - This is a set of tools that are used to integrate a phase of the SDLC like the Analysis or Design phase. An example of the analysis toolkit is Visible Analyst Workbench from Visible Systems Corp. An example of the design toolkit is SQL Design Dictionary from Oracle
- **CASE Workbench** – This is a set of CASE tools integrated together to automate the entire SDLC. An example of the CASE Workbench is Developer 2000 from Oracle
- **CASE Methodology** - This is software engineering methodology that forms the basis of development of a particular system.

- **CASE Methodology Companion** - This is a set of CASE tools that automate tasks in a particular CASE methodology and / or automate the production of documentation and other deliverables required by the methodology. For example – Analyst/Designer Toolkit from Yourdon Inc. that supports the Yourdon structured methodology for system development
- **CASE Hardware** - This is the hardware platform on which a particular CASE tools runs. Like all other software tools, CASE tools too have prerequisites about the hardware that they would run on – machine configuration environmental software and so on.

9.8 Classification of CASE Tools

The CASE Tools range from simple, easy to use tools on a specific hardware environment to those that support highly advanced features and are configurable on a variety of platforms.

CASE tools are classified under four heads:

- Tool
- Toolkit
- Methodology Companion
- Workbench

9.8.1 Tools

Tools are classified according to the activity they support. Different types of CASE tools are:

1. Diagramming tools
2. Prototyping tools
3. Simulation tools
4. Support tools
5. Reverse Engineering tools
6. Re-engineering tools

1. Diagramming Tools

A diagramming tool aids in drawing a particular type of diagram. There are tools for drawing data flow diagrams, Warnier Orr diagram, Structure Charts, Nassi Schneidemann Charts, storing a data dictionary and the like. Figure 9.4 shows a diagramming tool used for drawing a DFD.

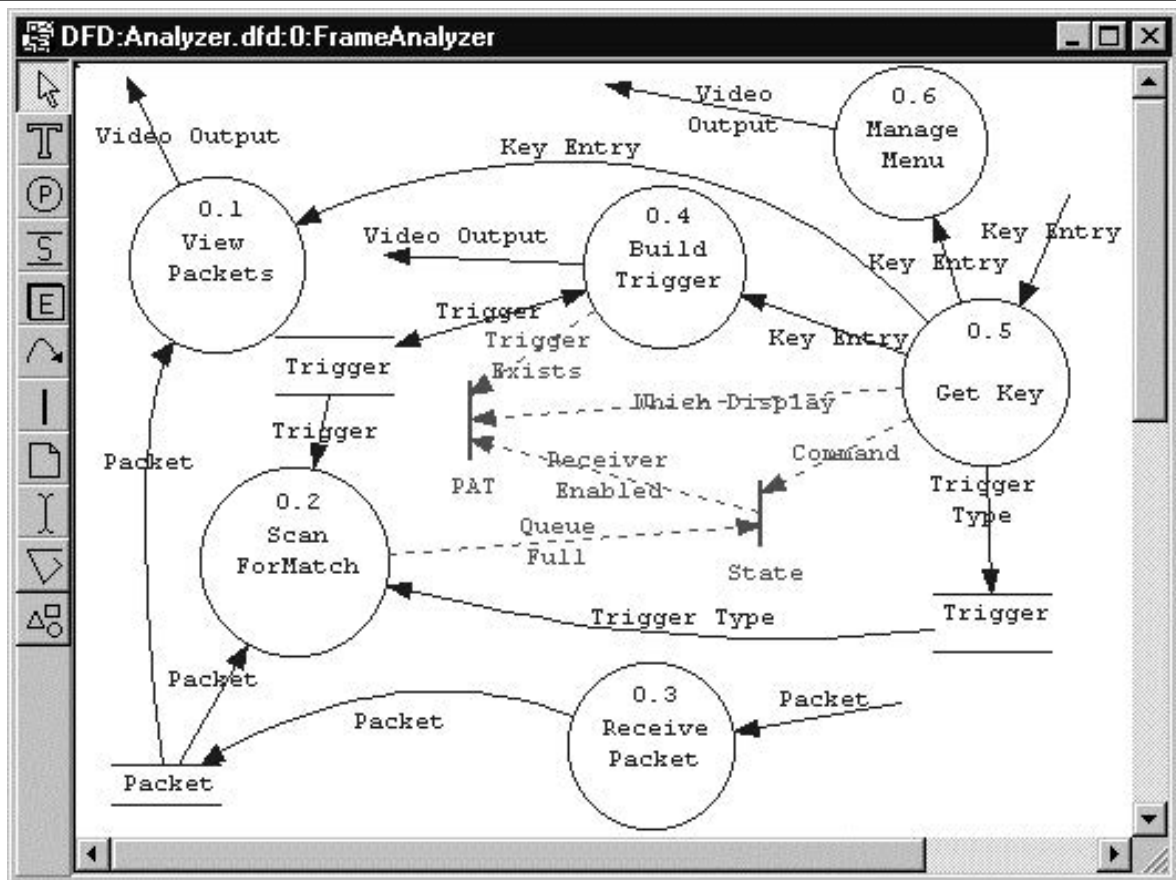


Figure 9.4: Diagramming Tool

Advantages of using diagramming tools are:

➤ **Speed of drawing diagrams:**

- Speeds up the drawing process.
- Making modifications becomes easier (students should recall the number of times they needed to make modifications in their respective systems).
- Drawing diagrams is not everyone's forte – people tend to get bored with this activity and the resultant is an untidy, patchy piece of work.
- Using diagramming tool ensures consistency in use of symbols and standards.
- All project personnel are not skilled to the same level in software engineering procedures – using tools takes care of the variations in skill levels, the tool takes care of the inconsistencies that could have arisen otherwise.
- Using tool results in high productivity gains.

➤ **Print features**

- Scaling print size according to requirement – usually when diagrams are drawn by hand, fitting them on to a particular size of paper is quite tedious, and scaling down or scaling diagrams upward is impossible.
- Printing only part of a diagram – depending on who is viewing the diagram (user, developer or project manager) and for what purpose (user review, project review).
- Creating vertical or horizontal representations

➤ **Ease of Use**

- CASE tool must be easy to install and use – generally they are.
- Complicated software CASE tool software will negate the “productivity gains” advantage.
- Aesthetics should also be built-in to the tool – the GUI environment takes care of that.

➤ **Multiple views** -Different views of diagrams are needed at different times within a project – CASE tools provide that flexibility:

- Overview – when users are reviewing the system
- Detailed view – when project team is reviewing the system

➤ **Flexibility of software:**

- Definition of symbols for a diagram – some tools allow us to define the choice of methodology diagram that we will use for a project. Therefore, we can select a set of symbols for a particular diagram. For example, we may want to use the Gane and Sarson set of symbols for DFDs instead of Yourdon methodology symbols.
- A tool should be able to offer many methodologies – Yourdon or Gane and Sarson data flow drawing techniques, Warnier-Orr charts, Nassi-Schneidermann diagrams – these are tools belonging to different schools of analysis and design but can be used effectively to design and develop a single system

➤ **Integrity Checking** -This is one of the most difficult and tedious tasks when done manually; it is also prone to errors, but the tool can do it to perfection.

- Integrity of individual diagrams as well as across diagrams can be checked.

- Some aspects that need to be tested are:
 - Correlation of input and output – are they related in the concerned system?
 - Checking for existing data in the data dictionary
 - Consistency checks across multiple levels of diagrams
 - Applying individual diagram rules for correctness
- **Availability:**
 - CASE tools are inexpensive.
 - CASE tools are available on many platforms – they do not restrict the user to the choice of single platform only.
 - Product support and training must also be available. There are many tools (restricted versions) available on the Internet.

2. Prototyping Tools

Prototyping is increasingly becoming an important paradigm in software development. The fundamental reason behind any prototyping exercise is that it has to be quick and cost effective. Automated support is therefore a logical fallout. CASE tools can be used to develop system prototypes. Prototypes are developed for two reasons:

- To give the user a feel of the system by showing user interfaces
- To show the user some key functions of a system

User Interface comprises of the following items:

- **Screen painters** – these help to quickly build screens and define data fields with validations that the user can try to get a feel of his future system.
- **Reports generators** - Some report formats can be designed, data can be fed for these reports and they can be printed. Figure 9.5 illustrates a Reports Generator.
- **Menu builders** - these help in building the menu structure of a system through which a user will understand the hierarchy of menus, which gives him a fair idea of the system structure.

Figure 9.5: Reports Generator

3. Simulation Tools

Simulations Tools are to real-time systems what prototypes are to online systems. . In order to arrive at performance of a system we need to simulate the environment in which it will run. Simulation will include getting the time required to enter a transaction, the processing time required for that transaction, time required to print, and update that transaction. A simulation tools helps to create the conditions that a real time system will encounter in its real time operation.

4. Support Tools

CASE Support Tools that apply to all the phases of a system lifecycle, not to just one or two. CASE support tools are tools for the support activities that we have discussed in earlier modules like Quality Assurance and Software Configuration activities. Just like the Quality Assurance or the Software Configuration Management activity, these support tools are also like an umbrella over the software development lifecycle.

Some of the commonly used support tool categories are:

- **Quality Assurance Tools** – it comprises of two types of tools:
 - Tools that ascertain quality based on certain software metrics like defects per function point, or Lines of Code, and other such criteria. So we can record defects of a system as and when they are detected and record them using the tool. The project manager along with his team analyzes these later. Figure 9.6 illustrates the use of a *Function Metrics* tool, which measures the quality of software using the Lines of Code metrics.

- Testing tools that can aid in helping the testing process. We can prepare test cases using this tool and record the test findings too.

Panorama C/C++ Metrics: Standards

Function Metrics	MAX	MIN	WEIGHT
<input type="checkbox"/> Size in lines	200	1	1
<input type="checkbox"/> % code	90	70	1
<input type="checkbox"/> % comments & white spacing	30	10	1
<input type="checkbox"/> Cyclomatic complexity (with case)	25	1	1
<input type="checkbox"/> Cyclomatic complexity (without case)	15	1	1
<input type="checkbox"/> J-complexity0	30	1	1
<input type="checkbox"/> J-complexity1	40	1	1
<input type="checkbox"/> J-complexity1+	40	1	1
<input type="checkbox"/> J-complexity2	40	1	1
<input type="checkbox"/> % segments executed (sc0)	100	90	2
<input type="checkbox"/> % segments executed (sc1)	100	85	2
<input type="checkbox"/> % segments executed (sc1+)	100	85	2
<input type="checkbox"/> % J-coverage	100	85	2
<input type="checkbox"/> % condition true	100	85	2
<input type="checkbox"/> % condition false	100	85	2
<input type="checkbox"/> % condition both	100	85	2
<input type="checkbox"/> % branch	100	85	2
<input type="checkbox"/> Depth Of Inheritance/Class	20	0	1
<input type="checkbox"/> Number of Children/Class	20	0	1
<input type="checkbox"/> Coupling Between Objects/Class	5	0	1
<input type="checkbox"/> Response For a Class	100	0	1
<input type="checkbox"/> Number of Methods/Class	20	0	1
<input type="checkbox"/> Number of Method Users/Class	20	0	1
<input type="checkbox"/> Lines of Code Reused/Class	1000	0	1
<input type="checkbox"/> % of Code Reused/Class	95	0	1

Figure 9.6: Use of a Quality Assurance CASE tool

- **Software Configuration Management tools** – Software Configuration Management deals with the process of tracking and controlling the software development activities. That is, the management of software development projects with respect to issues such as multiple developers working on the same code at the same time, targeting multiple platforms, supporting multiple versions, and controlling the status of code.

A CASE tool should identify each of these configuration items and control versions of various items. Auditing is made easy because all items are listed individually. These tools also help to communicate the changes to all concerned, thereby simplifying monitoring status.

- **Re-engineering Tools** -Existing software systems are re-engineered to make improvements making use of new concepts and technology. For example, the mainframe systems that have been and are being reduced in size use client-server technology. Re-engineering tools are being developed to aid this process. There are two types of re-engineering tools:
 - **Code re-engineering tools** – these take an old program as input and arrive at a better program structure and convert it into structured code.
 - **Data re-engineering tools** – these concentrate on the data structures that a program uses. They look at the existing data structures that was not normalized and efficiently designed in the yesteryear programs and organize it better. These tools break down the data structure generally using graphical notation so that the developer can interactively make changes to the database design
- **Reverse Engineering tools** -Reverse engineering is the process of arriving at system specifications from program code that is not accompanied by any design or specification document. It is software engineering turned upside down. There are two types reverse engineering tools:
 - **Static reverse engineering tools** – the tools in this category use the program source code as an input
 - **Dynamic reverse engineering tools** – these monitor each program as it executes.
- **Documentation Tools** - Documentation is essential for every system. Documentation is also the most tedious part of developing a system. Very few systems have good documentation to support them. Each phase of the SDLC should have its own documentation. Therefore, CASE tools that support a particular phase of the SDLC should also provide documentation support. Documentation that the CASE tool generates should be very concise and precise. It would state all details of the system in a few words. The analyst or designer can then add to this documentation at will.

1. **Get details of at least one CASE tool from the Internet, any magazine/book – state features of the tool, type of tools that form part of the kit, environment in which it will run and methodology that it supports.**

Now that we have looked at the various types of standalone-automated tools, let us try and understand the concept of Toolkits. CASE Toolkits are nothing but CASE tools pertaining to a particular phase of the software development life cycle.

9.8.2 Toolkits

Different types of toolkits being used today are:

1. Analysis toolkits
2. Design toolkits
3. Programming toolkits
4. Project Management toolkits
5. Maintenance toolkits

1. Analysis Toolkits

The analysis tools enable a software engineer to build a model of the system. This model contains a representation of data, function and behavior. An analysis toolkit consists of the following components:

- a) **Prototyping tools** – these tools help to develop a mini model of the system so that a user can be shown a working model of the system.
- b) **Diagramming tools** – these help us in drawing data flow diagrams and entity relationship diagrams.
- c) **Repository-** A repository of a CASE tool is the place where data regarding the diagram objects and their attributes is stored. It is similar to the data dictionary. A repository goes a step further than the data dictionary – since it is an automated version of the data dictionary and therefore easier to access from any tool in an integrated CASE environment, it stores other kind of data too, as explained as follows:

➤ A repository stores information about the system being built which includes:

- Process specifications
- Module specifications
- Data specifications (data elements, data structures)

- Database designs
- Screen and report designs
- A repository can also store project-related information like:
 - Task plans
 - Deliverable documents and document templates (to ease the making of a document)
 - Configuration details (for configuration management)
 - Project management annotations – schedules, time charts and the like

The repository of a CASE environment stores definition of the user's system and the specification of the data model for the user's system, thus going beyond the traditional data dictionary facility.

- d) **Specification Checker** – this parses the diagrams or system specifications to ensure that it conforms to the various syntax rules.

2. Design Toolkits

The Design Tools enable a software engineer to build a model of the system. This model specifies the characterizations of data, architectural, procedural and interface design details. Design toolkits comprise of tools that can:

- Draw structure charts and store related information like a brief description of each module or module connection
- Model data according to rules of normalization
- Generate program specifications (part of detailed design) that is used by programmers
- Generate database schemas – schema is the description of the database (all the data in the database), while sub-schema is a portion of it that is relevant for a particular type of processing. Specific toolkits refer to specific database management systems and can generate database descriptions pertaining to that database only.
- Generate reports related to any of these.

A design toolkit will help in the design phase of the SDLC. If we are working in an integrated CASE environment then all the design-related information will be stored as part of the project repository. This information can be accessed across DFD and ER diagrams. Besides, an integrated environment will enable consistency checks in diagrams across phases.

3. Programming Toolkits

A programming toolkit is a set of programming tools that are compatible and are interfaced with each other to enable construction, testing and documentation of programs as specified in the program specifications. Different categories of programming toolkits are:

- a) **Source Code tools** – these help in writing programs by providing features that will make a program more readable and therefore more maintainable. Source code tools are of two types:
 - **Editing tools** - Editing tool's allows us to enter and edit source code for a program.
 - **Browsing tools** - Browsing tools aid viewing of source code, and making changes across programs.
- b) **Code Generation tools** - The function of a code generator is to convert program specification into program code. Code generators are of two types:
 - **Source Code Generators** – These are more practical as they offer the flexibility of changing the code when required. Source code is not machine dependent and can be ported from one machine to another.
 - **Object Code Generators** - Object code can then be generated for the machine on which it is required to run the program. Object code generators are inflexible as they generate machine specific object code that cannot be ported from one machine to another.

1. Enumerate the benefits accrued from using a code generator.

- c) **Debugging tools** – These help us identify the place where an error exists in a program.

Debugging tools are used to detect errors in the processing logic as well as errors that arise due to the nature of input data. Debugging tools can debug our programs for us in following ways:

- They can provide memory dumps of the state of a program with variable values at a particular time in the processing cycle – when required by the programmer
- They can trace the flow of the program logic as it executes – this is done by displaying the current statement that is under execution in a display area meant for the purpose on the screen. This includes the tracing of control jump from one sub-routine to another
- Similar to flow-trace, debuggers can also display the value of variables as processing continues
- In order to review and understand the processing up to a point (programmers may want to back track and see the sequence of steps, restart execution from a point etc.), debuggers provide the facility of breakpoints and restart. Execution stops or pauses at a breakpoint and restarts from the next statement when the programmer so indicates

- Some debugging tools also come with the facility of listing values to be input for certain variables, a list of which the programmer feeds into the debugging software and is input into the program each time a value for a variable is required.
- d) **Testing tools** - Testing tools aid in the process of testing software. Testing tools are available to support the various techniques of testing, we had white and black-box techniques of testing during the module on Software Quality Assurance. There are two broad category of testing tools available to support the aforementioned techniques of testing:
 - **Static tools** – these are used to test a program when it is not in execution. They merely interpret the source code in terms of the data and control structures and determine the results of different kinds of input to a program. Tools of this category are data flow analyzers and structure checkers.
 - **Dynamic tools**- these operate on a program when it is under execution. They determine the number of loops that are executed in a program run, total number of statements executed, and the path of execution through various decision constructs. Dynamic tools are of two types:
 - *Variable tracers* – help in usage and change in value of variables
 - *Path tracers* – help to determine the execution path, number of statements, and so on

Figure 9.7 illustrates the use of dynamic testing tool, which checks *how many times a particular function was executed and what was its execution time*:

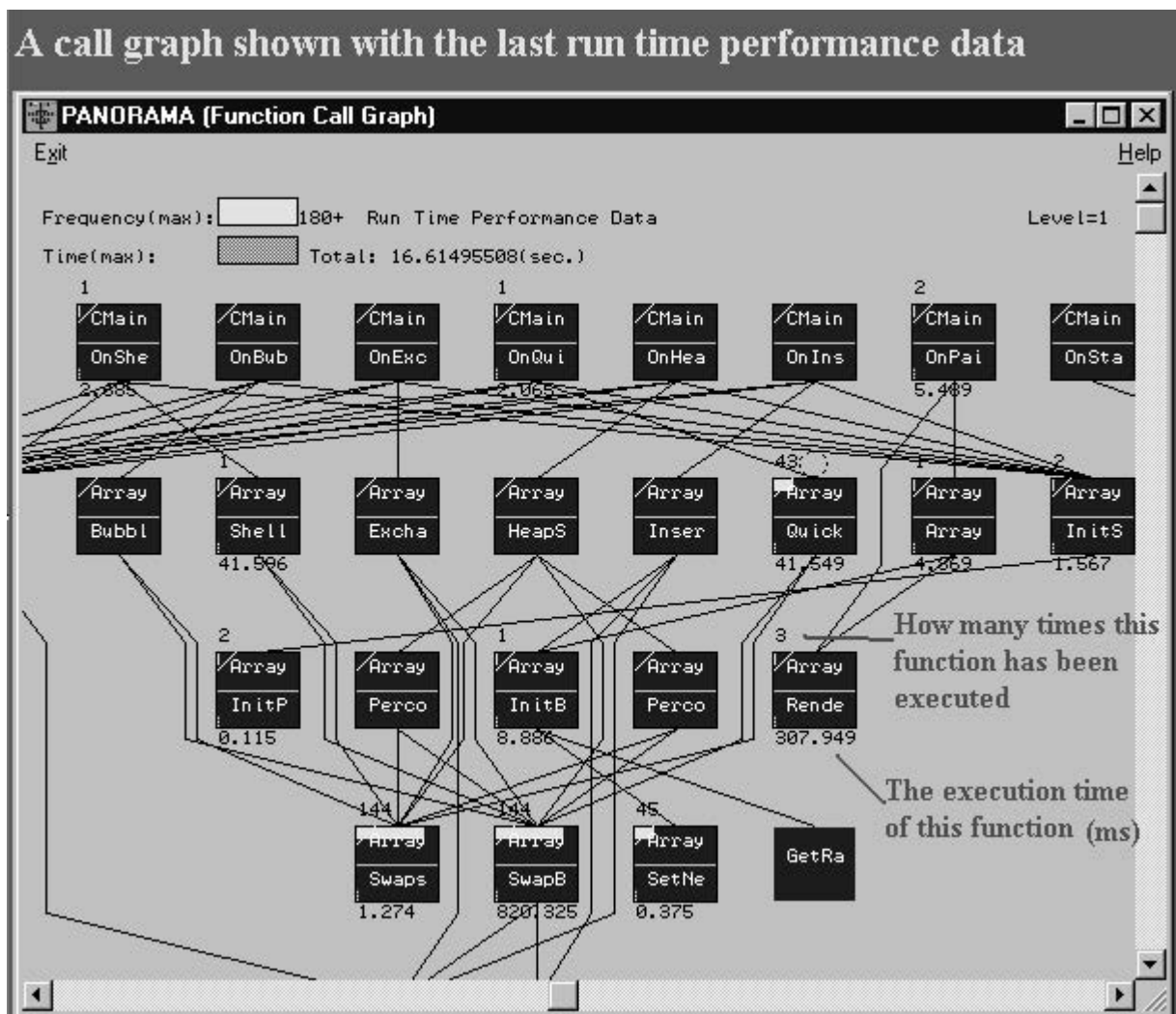


Figure 9.7: Dynamic Testing Tool

4. Project Management Toolkits

Project management tools support project management activities like:

- Estimation
- Planning
- Scheduling and monitoring

The Project Management tools help to choose the appropriate method for project estimation. The project manager prepares the project plan using a project management tool. A project management tool will also support project scheduling and monitoring tools like Gantt chart and PERT chart. For instance, a project manager will plan through the tool as to what resource is required in which phase of the project and

availability of the same. This can then be woven along with the project schedule. If a project is running behind schedule, the project manager can look at the resource plan and see if this delay will cause any resource crunch. If a resource crunch is likely to happen, then it is time to take corrective action on the resource front. Thus, the tool can effect useful project management action.

Schedules are prepared using any of these tools and tracked by maintaining the status of the project, online. Once again, different levels of schedules can be maintained; a macro level schedule is maintained for the project manager, while the team members follow a micro level (almost day to day) schedule.

5. Maintenance Toolkits

Maintenance is a high cost, tedious, error-prone and continuous activity. Maintenance toolkits are therefore very useful in tracking the process and to a certain extent helping programmers in deciphering code written by others, rearranging it more methodically, and coping with changing requirements of a user. Maintenance toolkits fall under three categories:

- **Assessment tools** – these help to assess the extent of maintenance done, that is, the effort expended in maintenance of a system. This is done by assessing the original source code and the changed versions maintained along with the number of changes that took place.
- **Re- and reverse engineering tools** – these have been discussed in the earlier chapter.
- **Migration tools** – these help when the need to move to a new operating environment arises (for example from COBOL to C). These tools help in conversion from the existing system to the new system (read COBOL code and generate C code).

9.8.3 CASE Methodology Companion

A methodology is a set of methods (way of performing various activities, drawing diagrams, which diagrams to draw for what activity etc.) that conform to a particular school of thought. For example, the Gane and Sarson methodology, the James Martin methodology, the RAD methodology used to develop systems. The difference lies in the approach, symbols of diagrams, and diagrams for various activities per se. A methodology provides us with a solution for the entire SDLC, whether it uses five diagrams or ten is a matter of individual methodology procedure. There are certain techniques that are common across various methodologies, like data flow diagram, and structure charts with variations in symbols.

Therefore, when one decides upon a working methodology for an application development, one must also look for a CASE tool that supports this methodology. Such a CASE tool is known as a *methodology companion*.

It is good to have a methodology companion as it supports all that we have been thinking of doing for our project using a particular technique. But it has its limitations. If at a later date we want to

use a different, better technique to perform the same task, we will not be able to do so. Using a methodology companion ties down the user to a particular methodology. Besides, some methodologies are very strong in the analysis phase, but weak in the design phase, or strong in the design phase but weak in the testing phase in terms of techniques that they offer for the phase. So at the end of using a methodology companion, we may find that we have a very good analysis document but a diluted design document.

To overcome this limitation, we have the CASE workbench.

9.8.4 CASE Workbench

A CASE workbench consists of tools that cover and integrate the entire SDLC, not just a single phase (that is how it is different from a toolkit). It also offers us with a choice of tools and techniques to be used for the SDLC, we can select the DFD symbols of one methodology and structure chart symbols of another methodology. A workbench ensures that the output of one phase successfully becomes the input for another phase and that there is no loss of data between two phases; this is done through the repository. Thus, a CASE workbench is strong in all SDLC phases:

The advantages of using a workbench are:

- A workbench takes the project through from its inception to implementation and later adaptive and corrective maintenance.
- A workbench assures smooth flow from one stage of the project to another. Because of which even if project person is not a very good analyst it will not cause any kind of problems in the project flow. This means that a workbench is independent of the skills of people using it.
- Error checking is easier with a workbench as it can be performed across phases using the same tool.

A CASE workbench contains the *Information Repository*, which is a central library (software) that stores information regarding all the phases of the project – analysis, design, programming, testing, implementation and project management. It is a repository that is updated during each phase as the project progresses. Data created in the analysis phase is updated during the design phase and the same is used during code generation and so on. The advantages of a workbench repository are:

- Integration of data across all phases
- Consistent and integrated data
- Sharing of information across team members
- Project management is more efficient and its related data is also stored within the repository

The only disadvantage of a CASE workbench is that it ties the user down to a straight jacketed approach. This means that the user must follow only those techniques that it has to offer in its folds. A Workbench Framework overcomes this disadvantage; in that sense it is an ideal solution.

A framework is a CASE solution that fits together tools from various manufacturers. In that sense, it customizes the CASE environment for us. We can assemble the best tools from various vendors and integrate them through a *Workbench Framework*.

Quite obviously, the Framework supports a number of environments, this is a concept that is best suited to an open systems environment.

Having analyzed the various aspects and components of a CASE tool, it is important to learn about the problems encountered during CASE Implementation. This information has been provided in the Appendix. The appendix also lists the details of various CASE tools being currently used.

9.9 The Future of CASE

In the early eighties, only a handful of organizations used CASE tools. In the late nineties, there were hundreds of CASE vendors in the market. We have come a long way indeed. However, today we are still at a stage where most organizations perceive CASE as being highly desirable though not mandatory. This situation is bound to change very rapidly.

Considering the number of areas being automated, the backlog of applications to be developed, the enormous effort spent in maintaining systems, the increasing focus on quality and productivity, and the dearth of qualified manpower, CASE is in the process of becoming an integral part of every software organization.



SUMMARY

- Implementation is a very important phase of SDLC.
- There are no standard methodologies for implementation.
- The important activities of the implementation phase are:
 - Drawing up an Installation Plan
 - Implementing of Physical Procedures
 - Data Preparation and Conversion
 - Conducting User Training
 - Parallel run of the system
 - Seeking Implementation Approval
- The choice of an appropriate implementation strategy is critical to the successful implementation of any project.
- The implementation is usually followed by post-implementation maintenance phase.
- CASE automates the process of software development.
- The essential components of a CASE are tools, toolkits, methodology companions and workbenches.
- The important toolkits being commonly used today are:
 - Analysis toolkits
 - Design toolkits
 - Programming toolkits
 - Project Management toolkits
 - Maintenance toolkits
- CASE is in the process of becoming an integral part of every software organization.



CHECK YOUR PROGRESS

1. An organization will follow _____ procedures side by side of the automated system, and this will complement its functioning.
2. _____ is the process of converting data from the old system into a format required by the new, automated system.
3. The _____ of conversion means introducing a new system in one part of the organization, a single department or function.
4. A _____ is a number of CASE tools, which use an integrated approach to support the interactions between the environments components and the users of the environments.
5. A _____ is one that aids in drawing a particular type of diagram.
6. A _____ of a CASE tool is the place where data regarding the diagram objects and their attributes is stored.
7. _____ helps in writing programs by providing features that will make a program more readable and therefore more maintainable.
8. _____ helps in assessing the extent of maintenance done, that is, the effort expended in maintenance of a system.
9. A _____ consists of tools that cover and integrate the entire SDLC, not just a single phase.
10. A CASE workbench contains the _____, which is a central library (software) that stores information regarding all the phases of the project.

Chapter 10

Object Oriented Software Engineering

Objectives

At the end of this chapter, you will be able to:

- *Describe the basic Object Oriented concepts*
- *Define the common process framework for OO design*
- *Describe object-oriented project metrics*
- *Estimate, schedule, and track an OO-based project*

10.1 Introduction

The Structured Systems Analysis and Design approach (SSAD) has many drawbacks. The techniques used in SSAD are not very useful in providing a detailed understanding of the system to the user. Even the simplest Data Flow Diagram (DFD) would require some explanation before the user can understand it. With SSAD, the requirements phase does not carry much importance or emphasis. Lastly, most of the modern CASE tools are being built for a newer approach to system analysis and design, the **object-oriented approach**. In this chapter, we will begin with describing the object-oriented viewpoint and then proceed with the concepts of OOA (Object-oriented Analysis) and OOD (Object-oriented Design).

10.2 Object-oriented Basics

Object-oriented design is based on data abstraction; it assumes that every basic system component is a module supporting data abstraction. Simply speaking, object-oriented design is modeled using objects where objects are any entities that have behavior and state. Any real world entity such as a car, a book, and a student can act as an **object**.

Basic building blocks of object-oriented concepts are **classes** and **objects**. A **class** defines a possible set of objects. The **attributes** of an object are the characteristics it possesses and these are defined by its class. Similarly, the operations performed by an object are defined by the class of an object. A **class**, however, is only a template and does not hold any actual data. Therefore, it does not occupy any memory. A class may have an interface that defines accessibility of the parts of a class that are also called members, a class body that implements the operations in the interface, and instance variables that contain the state of an object of that class. If **person** is a class having attributes such as name, age, gender and the like, then instances of class person also called objects can be **student, businessman, or housewife**.

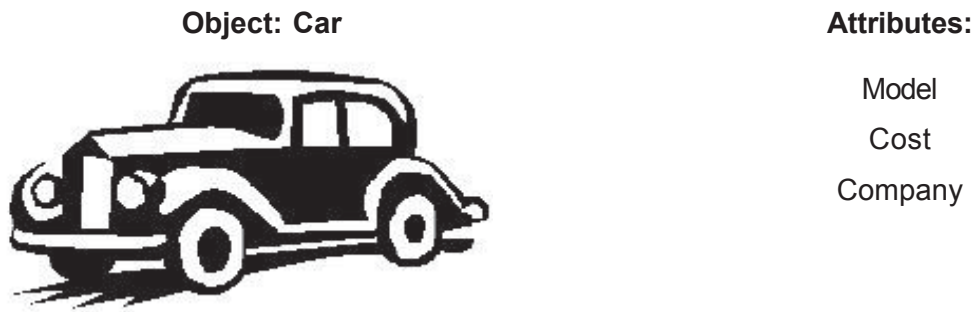


Figure 10.1: Objects and attributes

In Figure 10.1, we depict 'car' as an object. It has model (make), company and cost as its characteristics or attributes. Mercedes, Ford, and Mitsubishi can all be objects for the class **car**.

An object-oriented life-cycle approach applies object-oriented principles to every phase of the development life cycle.

10.2.1 Managing Object-oriented Projects

Management of object-oriented software projects can consist of the following activities:

- Establishing a common process framework
- Using the framework and metrics to develop effort and time estimates
- Specifying work products and milestones that will enable progress to be measured
- Defining checkpoints for quality assurance and control
- Managing the changes that occur as the project progresses
- Track, monitor and control progress

10.3 Common Process Framework

A *common process framework* (CPF) describes an organization's approach to software development and maintenance. The CPF identifies the software engineering standard that is applied to build and maintain software. It also identifies the standards that are applied to the tasks, and milestones and outputs (deliverables) that will be required. The single most important feature of the CPF is that it is adaptable, and hence can meet the individual needs of a project team.

Earlier models used for the system development life cycle assumed that the project activities proceed in a linear sequential fashion. However, this may not always be true. Object-oriented software projects follow an iterative process. In other words, the software evolves through a number of cycles. The common process framework that is used to manage object-oriented software projects must be evolutionary in nature and encourage assembling of components or reuse.

The **Component process assembly** process model is commonly used as an OO process model. It incorporates many of the features of another model called **spiral model**. The **spiral model** is an evolutionary software process model that combines the iterative nature of prototyping with controlled and systematic aspects of linear sequential model. The spiral model is divided into a number of framework activities. Some of these activities are described as follows:

- **Customer communication:** Tasks essential to establish effective communication between software engineer and customer
- **Planning:** Tasks required to define resources schedules and other project related information
- **Risk analysis:** Tasks required to assess both technical and management risks
- **Engineering:** Tasks required to build one or more representations of the application
- **Construction and release:** Tasks required to build, test, install and provide user support (documentation and training)
- **Customer evaluation:** Tasks required to obtain customer feedback based on evaluation of the software developed and installed

The OO process begins with customer communication where problem domain is defined and basic problem classes are identified. The planning and risk analysis activities lay the foundation for the OO project plan. Since OO software process relies on the principle of reuse, classes need not always be built but may be looked up from existing class libraries. If a class does not exist in the library, it will be created, and this newly created class will then be added to the library for future reuse.

10.4 Object-oriented Metrics

The conventional estimation techniques cannot be used for OO projects as the underlying principles used for OO projects are vastly different from that used for conventional projects. Neither FP nor LOC estimation metrics prove effective perfect for OO based projects. Instead, we may use an entirely new set of project metrics for OO based projects. Object-oriented metrics are different because of factors such as localization, encapsulation, information hiding, inheritance, and object abstraction.

Object-oriented project **metrics** are units of measurement that are used to differentiate the following:

- **Products** involved in object-oriented software engineering such as designs, source code, and test cases.
- **Processes** involved in object-oriented software engineering such as the activities of analysis, designing, and coding.
- **People** involved in object-oriented projects such as the efficiency of an individual tester, or the productivity of an individual designer.

Since a class is the basic decomposition unit in OO projects, it is essential to develop a measure of the complexity of the object. The following metrics may also be used:

- **Number of scenario scripts:** Scenario scripts, also called **use cases**, are detailed sequences of steps that describe the interaction between the user and the application. A **use case** is a textual description of a sequence of interactions between an external agent or class of users and the system being designed. Each script is organized into three parts; initiator, action, and participant. An initiator is the person or entity that requests some service. Action is the result of the request, and participants are the persons/entities involved in the action. A typical use case diagram is shown as follows:

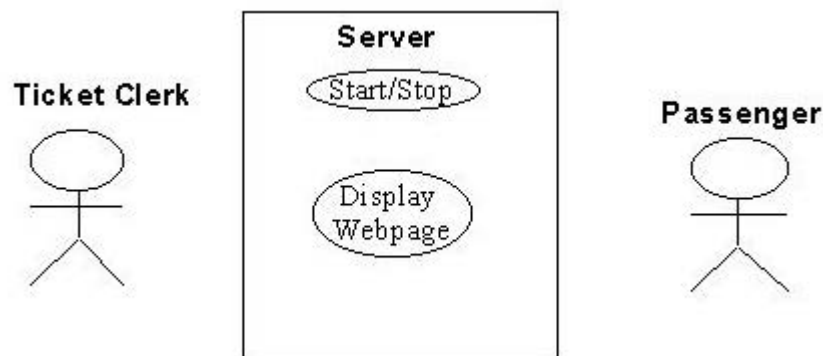


Figure 10.2: Use case diagram

- **Number of key classes:** Key classes are the highly independent components that are defined at the beginning of OOA. The number of such classes indicates the amount of effort required to develop the software, and an indication of the potential amount of reuse to be applied during system development.
- **Number of support classes:** Support classes are necessary to implement the system but are not related to the problem domain. Some instances can be GUI classes, database access related classes and communication classes. Support classes can be created for each key class. The number of support classes indicates the amount of effort required to develop the software and indicates the potential amount of reuse that needs to be applied.
- **Average number of support classes per key class:** In general, key classes are identified and defined in the early stages of the project. Support classes are defined throughout. If the average number of support classes per key class is known for a problem domain, estimating becomes much simpler.
- **Number of subsystems:** A subsystem is an aggregation of classes that support a function visible to end-users of the system.

Deborah Boehm-Davis and Lyle Ross conducted a study for the company GE and compared several development approaches for Ada software (Structured Analysis/Structured Design),

Object-Oriented Design (Booch), and Jackson System Development. They found that the object-oriented solutions, when compared to the other solutions were simpler (using McCabe's and Halstead's metrics), smaller (using lines of code as a metric), and appeared to be better suited to real-time applications and took less time to develop.

Shyam Chidamer and Chris Kemerer developed a metrics suite for object-oriented designs. The six metrics they identified are summarized in Table 10.1.

Metric	Description
Weighted methods per class	This focuses on the complexity and number of methods within a class.
Depth of inheritance tree	This is a measure of how many layers of inheritance make up a given class hierarchy.
Number of children	This is the number of immediate specializations for a given class.
Coupling between object classes	This is a count of the number of other classes to which a given class is coupled.
Response for a class	This is the size of the set of methods that can potentially be executed in response to a message received by an object.
Lack of cohesion in methods	This is a measure of the number of different methods within a class that reference a given instance variable.

Table 10.1: A metrics suite for object-oriented designs

10.5 Estimating an Object-oriented Project

The following approach is recommended by Lorenz and Kidd to be the effective method of deriving an estimation technique:

- Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
- Develop scenario scripts or use cases and determine a count with the help of OOA.
- Determine the number of key classes using OOA.
- Categorize the type of interface for the application and develop a multiplier for support classes.
- Multiply the total number of classes by the average number of work-units per class.
- Cross check the class-based estimate by multiplying the average number of work-units per scenario script or use case.

10.6 Scheduling an Object-oriented Project

Since the process framework for object-oriented projects is iterative in nature, scheduling is a complex process. Some project metrics that may help in scheduling an object-oriented project are:

➤ ***Number of major iterations***

A major iteration means more than 4-5 months of iterative work. Such kind of work will involve going back to the previous stages after each stage is completed, to make improvements. Around 2.5 - 4 months of iterations are easier to track and manage.

➤ ***Number of completed contracts***

A contract is a group of related public responsibilities that are provided by subsystems and classes to their clients. A contract proves to be an excellent milestone.

10.7 Tracking an Object-oriented Project

A recursive/parallel process model framework for an OO project is characterized by task parallelism, which makes project tracking a difficult task. Using the tracking system, managers can establish current performance against schedule and project personnel. A tracking system identifies, estimates, allocates and tracks the software objects.

Major milestones that can be considered completed when the criteria noted have been met are summarized in Table 10.2.

Technical Milestone	Activities
OO analysis completed	The classes and their structures have been defined and reviewed.
	Class elements and functions associated with a class have been defined and reviewed.
	A behavioral model has been created and reviewed.
	Reusable classes have been identified.
	The set of subsystems have been defined and reviewed.
	Classes are allocated to subsystems and reviewed.
	Task allocation has been established and reviewed.

OO design completed	Responsibilities and collaborations have been identified.
	Attributes and operations have been defined and reviewed.
	The messaging model has been created and reviewed.
OO Programming completed	All the new classes have been implemented in code.
	Classes taken from reuse library have been integrated with the system.
	A prototype has been built for the system.
OO Testing	Correctness and completeness of OO analysis and design models have been reviewed.
	Class-responsibility-collaboration has been developed and reviewed.
	Test cases are designed and class-level tests have been conducted for each class.
	System-level tests are completed.

Table 10.2: A tracking system



SUMMARY

- Object-oriented design is based on data abstraction and every basic system component is a module supporting data abstraction.
- A class defines a possible set of objects.
- An object-oriented life-cycle approach applies object-oriented principles to every phase of the development life cycle.
- A **common process framework** (CPF) characterizes an organization's approach to software development and maintenance.
- The **component process assembly** process model is commonly used as an OO process model.
- Object-oriented metrics are different due to factors such as localization, encapsulation, information hiding, inheritance, and object abstraction.
- Metrics used for object-oriented projects are:
 - Number of scenario scripts
 - Number of key classes
 - Number of support classes
 - Average number of support classes
 - Number of subsystems



CHECK YOUR PROGRESS

1. A class does not occupy any memory, only an object does. **[True/False]**
2. A _____ characterizes an organization's approach to software development and maintenance.
3. Project metrics such as FP and LOC that were used for conventional projects also work perfectly fine for object-oriented projects. **[True/False]**
4. A _____ is an aggregation of classes that support a function visible to end-users of the system.
5. A _____ describes an organization's approach to software development and maintenance.

GROWTH
RESearch
OBsERVATION
UPDATES
PARTICIPATION



www.onlinevarsity.com

Objectives

At the end of this chapter, you will be able to:

- *Study the implementation of configuration management in small enterprises*

11.1 Introduction - Business Motivation and Objectives

Very many small software companies, often having started as a one-man enterprise, reach a stage where version/ configuration management and error tracking/ reporting systems becomes a must. This is because as more people get involved in software development, the firm gets more (and more heterogeneous) customers, and the applications themselves no longer require less major changes, but routinized error fixing and minor functionality adjustments. Very often one finds profit margins eroded by unforeseen troubles as lack of version / configuration control, no system for keeping track of errors reported/fixed/tested, and so on.

The Experiment

ICONMAN aims to implement lightweight configuration management procedures for very small enterprises that cannot use the same approaches as larger companies. The procedures will be based on the current software engineering practice of the partners in order to be as simple and unobtrusive as possible. The use of configuration management will also address documentation and error reporting.

There are three very small firms (5 - 10 persons) involved; all of them have a basic product that is delivered in customized versions for different purposes and to different customers. An important element in the experiment is to exchange experiences with different configuration management tools and routines used. The project will define and test a set of software metrics in order to get quantitative information of the impact of the new configuration and error-reporting regime.

Expected Impact and Experience

After the experiment, the three companies will have implemented procedures for revision and configuration management that includes testing of new configurations/ versions of the base product, including both source code and documentation. In addition, there will be procedures to handle all communication with the customers that relates to product development, which includes error reports, requests, and suggestions for changes.

Business Sector	Software Industry
Application Area	Oil-reservoir simulation, statistical analysis, actuarial software
Keywords	Configuration Management, Software metrics
Technologies/Methodologies/Tools	Configuration Management tools

Figure 11.1: Expected Impact and Experience

11.2 Executive Summary

This report is based on work performed in the ICONMAN project during the 18 months (June 1996 – December 1997) of the process improvement experiment. The experiment was sponsored by the CEC under the ESSI Programme.

The project's main goal was to implement configuration management in three small software companies and assess the effect of this effort. The main conclusion from the project is: Implementing configuration management is worthwhile in very small companies. This is based on both qualitative and quantitative measurements and observations.

The companies have with respect to their own judgment successfully implemented routines for configuration management. Even though the process has been more demanding than expected, the

maturing of the system development process was a necessary step in developing the business processes in the companies as a whole.

The main lessons learned were:

1. Configuration management is a complex activity with far reaching consequences for the business as a whole.
2. Implementing configuration management is an iterative process, and requires continuous refinement.
3. In very small companies the introduction of configuration management should be tested in a controlled, but real-life environment.
4. It was difficult to identify quantitative data to measure process performance, but the defined simple metrics were essential for evaluation of the experiment.
5. The existence of an operative configuration management system has shown to make a positive impact on customer relations.

6. The existence of an operative change request database has proven to be valuable in planning product releases.
7. The existence of an operative configuration item library has proven to simplify the process of reconstructing earlier releases, and led to a higher service level for the customer.

11.3 Background Information

In the proposal for the ICONMAN project, all three companies involved identified the control of source code versions of their main products as a major problem of their respective businesses. All companies had a rudimentary in its kernel shared, but nevertheless varying understanding of configuration management. To maximize the outcome of their planned efforts and to gain the topmost synergy effect the three companies decided to apply for a common project and included a request for consultant and research support.

Given the only elementary understanding of configuration management, the sub-contractors after a first assessment of the situation and on the background of their general knowledge about possible solutions proposed to base the configuration management procedures on the SPICE Support Process Category (SUP). The project partners agreed and used a process that is called *SUP-2 – Perform Configuration Management*. The purpose of this process is to establish and maintain the integrity of all the products of a software project throughout the project's software life cycle.

The process is separated into eight tasks:

- SUP 2.1 Establish configuration management library system
- SUP 2.2 Identify configuration items
- SUP 2.3 Maintain configuration descriptions
- SUP 2.4 Manage change requests
- SUP 2.5 Control changes
- SUP 2.6 Build product releases
- SUP 2.7 Maintain configuration item history
- SUP 2.8 Report configuration status

In retrospect, without knowledge of other process descriptions, this can be qualified as a good choice. The support of the consultants saved the companies much work, and the chosen approach showed to be appropriate.

11.3.1 Objectives

The ICONMAN project pursued the following objectives that are described in the project programme:

- Formalize and document internal procedures for configuration management in each company.
- Choose and implement configuration management software.
- Produce an internal “Users Guide” for configuration management describing both the routines and the supporting tools.
- Develop metrics to measure the impact of configuration management.
- Develop a network consisting of the participating software companies and the subcontracted researchers, and utilize this for exchange of experiences.

11.3.2 Involved companies and their roles

Three companies were involved in the experiment. They were all very small and based their business on the development of one main product. Due to a growing demand of variants of these products, each company felt the need to introduce configuration management. Thus, each of the companies set out to perform the same experiment, the introduction of configuration management. The specific characteristics of each organization are presented here:

Event AS delivers systems for stochastic modeling and analysis based on a proprietary statistical method. During the project, 2 employees left the company, thus currently only 2, of which one is the founder of the company, of the original 4 employees work for Event; plans do however exist to expand in the near future.

Technical Software Consultants (TSC) AS’s main product is a technical system simulating oil reservoirs. The company consists of 5 employees, among them the founders of the company, all working as software developers.

Aktuar Systemer AS produces a (family of) system(s) for managing pension funds and their members. Also, Aktuar Systemer AS has 5 employees comprising the founders of the organization. During the project a larger insurance corporation bought the company.

11.3.3 Starting scenario

At the start of the project, all companies had an underdeveloped understanding and only rudimentary routines for configuration management. Due to the small size and limited resources of the companies, a comprehensive, formal assessment of the software practices following the ISO9000/TickIt standard, the Capability Maturity Model or the Bootstrap methodology was not considered. A focused assessment of their software practices and their customer relationships

performed by the Norwegian Computing Center-the project consultants and support researchers - uncovered a number of deficiencies that are explained here. In general, both process and product documentation were largely absent.

Together, the companies developed a strategy to solve these problems. This strategy took into account both the commonalties and the peculiarities of the companies. Again, due to their size and the limited resources, the companies saw a considerable risk that their developers would not adopt rigorous and costly procedures. Thus, the most important precondition for a successful introduction of configuration management was to keep the routines very simple.

The status concerning configuration management in each company before the experiment was as follows:

Event AS - In Event AS, all software development is based on one fundamental software library. Before the ICONMAN project, this library was developed and maintained by one developer. The developers use the library as a resource to develop specific applications.

Simple procedures for control of source code versions, although not documented, have been used since 1991- the beginning of the development of the product, yet before the establishment of the company. Besides this, there were no formal or documented procedures for configuration management, neither for the development process in general nor for quality assurance. Product development is carried out in close co-operation with customers who have received revisions of the software continuously, without any distinction between major releases and minor revisions. Product documentation was rudimentary, both for the developers and users.

There were no formal procedures for handling error reports and requests for changes from the users. The chief developer who is also the founder and manager of the company handled all these. The technical platform consists of Windows 3.11-workstations in a Novell Netware 4.x - based LAN.

TSC AS - TSC's view configuration management and testing as quality assurance measures, and as an important part of their development process. These activities are a prerequisite to keep the code of their product at a stable and at an acceptable level for themselves and their customers.. However, besides some elementary guidelines for version control, no formal or documented routines for change management of their main product, FrontSim were in use. This was the situation both for the source code and the user documentation. Other technical documentation was not produced in the company.

There were some procedures for documentation of code and for the production of user manuals, but no formal procedures to ensure that the production of documentation was in line with the release of code. Informal test routines were applied to test their main product every two weeks. There were no structural system for storing the different versions of test protocols, test procedures, and test data.

Procedures for handling error reports, in both directions, to and from the customer, were found, but they were only partially implemented. There was no formal procedures to handle customer feedback, example, suggestions for changes of the product. The technical environment consists mainly of Unix workstations.

Aktuar Systemer AS - At the beginning of the project, the company had implemented a very rudimentary version control system. Only the current production version and the previous production version of their main product ABAKUS were kept readily at hand. Reconstruction of previous releases was possible, but cumbersome.

Simple forms for error reports and change orders were used. Substitution of these forms with procedures based on E-mail were planned. When the project started, work on procedures for generating production releases had just started. Co-ordination and assignment of priority to various change orders was weak. User documentation and system documentation was limited.

No formal test procedures were implemented except for a limited integration test. The technical platform consists of Windows95-workstations in a Novell Netware 4.x - based LAN.

11.3.4 Work plan

The project was divided in three main phases:

The first phase lasted 6 months. The configuration management procedures, including the management of change requests and metrics to measure the outcome of the experiment were developed. Finally, the appropriate tools were selected and introduced together with the procedures.

A configuration item library and a change request database were established in all companies.

The second phase lasted 7 months. The second phase consisted of (i) training of the developers, (ii) refining the configuration management procedures based on feedback from the developers and (iii) starting the process of collecting metrics data.

The third phase lasted 5 months. Collection of metrics data continued, and the experiment was evaluated.

Although established in the work plan, a clear distinction of the phases other than by production for the defined deliverables as results of each phase was hard to make. Continuous learning and refinement took place throughout the whole project. The different evaluation activities were distributed over the entire experiment.

The milestones of the project were directly related to the project deliverables. This report covers the entire period of the experiment (June 1996 – December 1997) and includes the tasks performed for deliverables D1 – D11 (see Section 11.4). Table 11.1 shows the list of deliverables and Table 11.2 shows the work plan.

Deliverable Reference	Availability: (I)nternal (R)estricted (P)ublic	Description of the deliverable (Title)
D1	I	Report on definition of CM
D2	I	Report on implementation of selected tools
D3	R	Periodic progress report (first 6 months)
D4	R	Report on training on the job
D5	P	Mid-term report
D6	R	Report on first estimate of the defined metrics
D7	I	Report on refined CM procedures
D8	R	Periodic progress reports (6 to 12 months)
D9	P	Final Periodic Progress Report
D10	R	Consolidated cost statement
D11	P	Final Report

Table 11.1: List of deliverables

Description of phases	Jun 96	Jul 96	Aug 96	Sep 96	Oct 96	Nov 96	Dec 96	Jan 97	Feb 97
Definition of CM procedures									
Select tools									
Use tools									
Training on the job									
Refine CM procedures									
Measurement and evaluation									
Reporting and dissemination									

Table 11.2: The work plan of the experiment

Description of phases	Mar 96	Apr 96	May 96	Jun 96	Jul 96	Aug 96	Sep 96	Oct 97	Nov 97	Dec 97
Definition of CM procedures										
Select tools										
Introduce tools										
Training on the job										
Refine CM procedures										
Measurement and evaluation										
Reporting and dissemination										

Table 11.3: The work plan of the experiment (Contd.)

Table 11.3 indicates when project deliverables were scheduled and delivered, and when the various work packages of the experiment took place. The final PPR (D9) and the consolidated cost statement (D10) will be delivered at latest in January 1998; after the Final report has been accepted.

The work plan in Table 11.3 differs from the original one. After negotiations with the CEC concerning the project plan, the start date of the project was moved from April to June 1996. Due to a request for a more detailed measurement programme and a turbulent situation in the company of the prospective project leader, the project was extended from September to December 1997, but still within the original budget.

11.3.5 Expected outcomes

During the ICONMAN project it became apparent that not all the outcomes predicted before the project were of the same relevance to all of the companies. In the list here, we have indicated this. Originally, the following outcomes from the experiment were predicted:

- Improved procedures for handling feedback from customers in combination with proper configuration management were expected to reduce the time spent on correcting errors (Applies to all companies).
- Improved documentation procedures would make it easier to extend the development team if the company's growth continues (Applies to all companies).
- Improved testing procedures would reduce the number of errors in the delivered products (Applies in particular to TSC AS).
- The configuration and change management procedures would allow any developer to work in all areas of the baseline product (Applies in particular to Event AS).

11.4 Work Performed

The following gives the work performed.

11.4.1 Organization

In each company one senior developer acted as a local project leader. A representative of the prime user organization performed the overall project management. The day-to-day management and work of the project was co-ordinated on weekly meetings between the project leaders where both formal and informal matters were discussed. A team of three external consultants supported this work and helped preparing the reports for the CEC (see Section 11.4.4 for more details).

The local project leaders also acted as principal project members and were responsible for the performance of the following tasks:

- Develop and tailor the configuration management routines including the change request procedures based on a common framework (Deliverables 1 and 7).
- Select supporting software tools (Deliverable 2).
- Implement and introduce configuration management routines and supporting tools to the other developers (Deliverable 1, 2 and 7).
- Perform internal training and education (Deliverable 4).
- Collect measurement data (Deliverable 6 and 11).

In each company, at least one additional developer was involved in installing and evaluating the software tools and in collecting data for measurements. All other developers also took part in the project; they adopted and used the new routines and tools, contributed valuable feedback on their feasibility and provided metrics data.

11.4.2 Technical environment

Major emphasis was put on the development of appropriate configuration management routines. The project leaders developed procedures and chose tool environments to support the CM-routines that were introduced in their respective companies.

All companies established a configuration item library and a change request database (see Table 11.4 and 11.5). The selected tools to implement the change item libraries and the change request databases fulfilled the following important criteria: they were easy to integrate into the existing settings and did not require any specific changes in the present environments.

The configuration item libraries contain source code, test data and documentation. Table 11.4 shows that all companies include source code and documentation in their configuration item libraries. In addition TSC keeps test data in the library.

Configuration items **AS TSC EVENT**

Configuration Items	AS	TSC	Events
Source code - Author - Time when the work is carried out	x	x	x
Documentation - Author - Time when the work is carried out	x	x	x
Test data - Registered errors - In what part of the code - Author - Time when the work is carried out	—	x	—

Table 11.4: Contents of the Configuration Item Library for Aktuar Systemer AS (AS), Technical Software Consultants AS (TSC) and Event AS

Event AS and Aktuar Systemer AS chose and adopted Visual Source Safe from Microsoft as the basis for their Configuration Item Library. In TSC, the low level version control system SCCS (included in the UNIX environment) was used to control source code. Necessary shell software was used to make the use of SCCS more user friendly.

The change request database contains data that are related to program changes. The category “change request” contains several types of requests: error reports (data errors, program bugs and strange incidents), customers’ requests for changes, and company internal ideas and suggestions for changes and improvement.

Change requests come in by e-mail, telephone calls, pre-specified forms, in meetings with customers. They are logged, provided with a number, assessed regarding importance and priority, and if accepted, assigned to a developer.

During the life cycle of a change request, new data are added and some are updated, for example, status, finishing date. Table 11.5 shows the information that is stored by the three firms in the change request databases.

Configuration items	AS	TSC	EVENT
Reference no./serial no.	X	X	X
Received by	X	X	X
Assessed by	---	X	---
Request from	X	X	X
Registration date	X	X	X
Description of request	X	X	X
Suggestions for solutions	X ²	X	X
Assigned developer	X	X	X
Status (in life cycle)	X ³	X ⁴	X ⁵
Start date of implementation ⁶	---	---	---
Estimated finishing date	---	X	X
Actual finishing date	X ⁷	X	X
Data sets illustrating the problem	X ⁸	X ⁹	---
Estimated resource usage	---	---	X

Table 11.5: Contents of the Change Request database for the 3 companies

Event AS implemented their change request database using a proprietary system called ProMan (see Deliverable 2), which also functions as a timesheet system making it possible to extract effort data. Aktuar Systemer AS chose TeamWindows from Gupta Corp. to store their change requests. TSC AS chose to integrate the change requests database in the company's existing World Wide Web environment and purchased FrameMaker to generate user-manuals and documents describing CM-procedures.

The routines and tools were tested in a baseline development project in each company. The baseline projects dealt with further development, for example, the next version of the main product of each company. This means that development was not directed towards an explicit external customer, but towards internal deadlines and delivery that was part of the shipment of the official versions to the companies' customers. Both TSC AS and Aktuar Systemer AS had major official releases of their main product during the project period. Event AS developed several new versions of their software library during the project period, they did not however have any major official release of their main product.

11.4.3 Training

Training in all three companies was carried out as training on the job, either on an individual level or at informal internal lectures. Much of the direct education was triggered by developers who had started using the configuration management routines and had concrete questions when trying to accomplish their daily tasks.

This proved to be appropriate for such small companies; as scarce resources did not allow for extensive formal out-of-house-education, the developers acquired the necessary knowledge while contributing to the companies' business goals. In addition, their direct feedback without the extra bureaucratic overhead of formal meetings led to continuous refinement of procedures. The responsibility of the project leaders in their capacity as trainers also comprised the permanent update of the routine descriptions.

The training was performed in two steps:

First step:

- Introduction of configuration management procedures
- Introduction of version control and documentation tools
- Individual education and internal seminar on introducing configuration management.

Second step:

- Internal discussion and seminar on registration of change requests
- Introduction of tools for registration of change requests
- After an initial phase where constant encouragement from the project leaders was needed to take up the procedures, no other major problems have been experienced throughout the project.

11.4.4 Role of the Consultants

A group of researchers from the Norwegian Computing Center (NR) were used by the three companies. The use of NR was covered by a single contract between NR and the prime user Event AS.

The involvement of the consultants was of major importance for the participating companies. They contributed both as mentors and coaches with respect to configuration management. They acted as analysts of the current practice, proposed the underlying SPICE model for the development of the configuration management routines, created awareness concerning the role of configuration management and supported the introduction process by suggesting various possible procedures. The consultants furthermore provided guidance in defining metrics and collecting data. Last, but not least, they carried out a customer study at the initial phase of the experiment and a developer study at the end of the experiment. These studies provided valuable input for the development of the CM-routines and for the assessment of the success of the project as demonstrated here.

The role of the consultants has to be seen especially in the light of the companies' size. For very small organizations with scarce resources and limited possibilities to keep updated with the state-of art, a considerable risk exists in spending time on such a project. The consultants with their appropriate insight in the areas of demand have minimized this risk and have efficiently facilitated the process of acquiring the relevant knowledge and of implementing useful procedures. They also helped in preparing deliverables to the Commission. This increased the quality of the reports to a level acceptable by the Commission.

11.4.5 Phases of the Experiment

The first phase of the project lasted for 6 months. The local project leaders defined in co-operation with the consultants, the configuration management routines, the change request procedures and metrics, and introduced appropriate tools. As a by-product, routine to trace the effort used to perform routine tasks and

to handle requests were also developed. The key deliverable ending this phase was a report describing configuration management routines, a report describing implementation of selected tools, and a periodic progress report (D1-D3).

The second phase (about 7 months) consisted of training-on-the-job during the baseline project (D4), refining the procedures based on the initial experience, and collecting first data, both retrospectively and from the baseline project. The training was carried out as described in Section 11.4.3. The refinement of the configuration management procedures has been carried out based on feedback from the developers. What data to collect was determined with the assistance of the consultants who provided a proposal for each company and explained their suggestions in plenary meetings (see D6).

The three other deliverables for this phase were the mid-term report (D5), the report on refined CM-procedures (D7), and the second periodic progress report (D8).

During the third phase (5 months), the collection of data continued, CM-routines were further refined, and the main dissemination action [3] was conducted. Finally, the metrics data was analyzed.

Quantitative analyses were based on collected data from before and during the project period (see Section 12.5). The qualitative results were derived from interviews with the developers in each company. The key deliverables were the Final Periodic Progress Report (D9), Consolidated costs statement (D10) and this report: the Final Report (D11).

11.4.6 Internal dissemination

The results have been communicated through training and discussions. These discussions under participation of the consultants often gave input to further improvement of the procedures. We can mention the following two examples:

- (1) All companies now have a common kernel of data collected for handling change requests.
- (2) In one of the companies the format of error reports was changed.

11.5 Results and Analysis

All objectives of the experiment as formulated in Section 11.3.1 have been achieved by the three companies:

- Routines for configuration management have been established, implemented and refined.
- Software support in the form of configuration item libraries, change request databases, and systems for logging effort data has been developed and established in all companies.
- Configuration management is now in place. All companies have their own “Users Guide”.
- Metrics have been developed to measure the impact of configuration management.

- A shared understanding between the companies has been developed and is mirrored in a shared kernel of functionality with locally tailored solutions.

The outcomes from the experiment corresponded well with the ones anticipated as outlined in Section 11.3.5.

- All companies believe that improved documentation of the CM-procedures have made it easier to extend the development team. The development teams have not been extended in neither of the companies during the project. However, in interviews the developers claim that the introduction of CM-routines has simplified their daily tasks. This indicates that new employees will integrate into development teams more rapidly.
- Companies have experienced that improved procedures for handling feedback from customers in combination with proper configuration management reduced the time spent on correcting errors and generating production releases (For quantitative measures see Aktuar Systemer in the next section).
- TSC AS have experienced that improved testing procedures did reduce the number of errors in the delivered products.
- In Event AS the configuration and change management procedures now allow any developer to work in all areas of the baseline product.

In the next section, we detail the results from each of the three companies and link them to the outcomes that were expected. Some outcomes not originally recognized are also presented.

11.5.1 Technical

Event AS - During the ICONMAN project, Event AS has established formal configuration management routines to handle change requests and errors reported. From the change request database they established during the project, effort data was collected and used to assess the outcomes that were expected from the experiment.

For Event AS a reduction of the workload and dependence on the senior developer was one of the intended outcomes from the PIE. It was therefore a goal to establish routines, which allowed any developer to work in all areas of the main library. When comparing the quantitative results from Event AS from the early phase of the project with the later phase of the project, it can be seen that the outcome was indeed achieved (see Table 11.6):

- The hours developer B spent on library development, compared to the hours the senior developer spent on library development increased from 8 percent to 16 percent.
- The senior developer spent less on library development in the second period. The “saved” hours were spent on administrating the business, which was one of the positive effects from the experiment.

- The number of new statements per work hour did not change significantly in the two periods, which indicates that efficiency has not decreased even if the senior developer did spend less time on programming in the last period.

Period of time	Activity	New program statements	Hours spent (A)	Hours spent (B)
Jan. 1. to Apr. 30	Library	13972	480	40
	Applic. 1	-200 ¹¹	0	107
	Applic. 2	817	10	92
May 1 to Oct. 15.	Library	8819	339	53
	Applic. 1	524	4.5	204
	Applic. 2	40	5	22.75

Table 11.6: Total increase in program statements and the distribution of workload between the persons A (the senior developer) and B during the period from *January 1- April 30, 1997 and May 1- April 30, 1997.*

Applic.2 is not the same application in the two periods. The results are given in hours, collected from the change request/timesheet database ProMan, which was established during the ICONMAN project. The number of new program statements has been calculated by counting program statements of different versions of either the library or the applications.

Another intended outcome from the PIE was that improved documentation procedures should make it easier to extend the development team if the company's growth continues. To measure potential benefits in this respect, Event AS measured:

- The hours spent on tutoring and assistance provided by A.
- The hours spent by B on the task that necessitated the tutoring in the first place, expecting the number of hours spent on tutoring to decrease.

From Table 11.7 we observe:

- The relative number of hours spent on tutoring and carrying out a task has not changed.
- The number of tutoring hours is constant.

These findings do not support the aim of decreasing tutoring time. Still Event AS finds the result of considerable importance. The result implies that a "known" constant amount of tutoring should be included when planning new projects.

Due to proper measuring, Event AS is confident that the hours spent on tutoring is approximately constant. This information can be actively utilized when planning new projects.

Period of time	Activity	Hours on tutoring (A)	Hours on task (B)
Jan. 1.	Funct.1	5	30
	Funct.2	5	17
to	Funct.3	2	4
	Applic.1	0	107
Apr. 30	Applic.2	0	92
May 1.	Funct.1	5	24
to	Funct.2	4	11.75
	Applic.1	4	204
Oct. 15.	Applic.2	0.75	22.75

Table 11.7: Hours spent by A on tutoring B and the hours spent by B on the task that necessitated the tutoring in the first place, in the period from *January 1- April 30, 1997 and May 1- April 30, 1997.*

Applic.2 is not the same application in the two periods. The results are collected from the change request/timesheet database ProMan, which was established during the ICONMAN project.

The quantitatively measured achievements reported for Event AS may also be due to other factors than the introduction of CM-routines. Person B was newly employed when the ICONMAN project started, and one must expect that some improvements are caused by increasing experience.

TSC (Technical Software Consultants AS) - The introduction of tools/routines for control code management supports the developers in order to maintain correctness and integrity of the software items when they have to make changes in the source code. This was emphasized by the employees in the interviews that took place at the end of the project. It also affected the process of merging changes into the source code in several ways:

- The use of routines and tools for source code management has made carrying out the merging process less time-consuming. Now it takes 20 minutes to merge changes into the code – before it took almost 90 minutes. This can be measured because the developer has to check the whole source code (both before and now) to ensure that the merging process has been carried out properly.
- A graphical interface make the process of merging code more user -friendly and with less chances to make “blunders”, for example, by accidentally deleting/changing previous updates that have been done in the source code.
- The existence of a configuration item library has simplified the process of correcting errors that occur during the merging process.

In TSC, the configuration item library now also contains test data and test results. For the company this means a further improvement of their testing process, which contributes to an increase their software’s quality. Fewer errors have lead to increased customer satisfaction. This has been confirmed by some of the customers. Formalized test routines and a configuration item library for test results have improved the process of testing the baseline product in several ways.

- All test results are stored. This has made it easy to locate the test results that were generated for the different versions/releases of the product.
- It is easy to find out which test results correspond to a certain set of test data. This is important since it is sometimes necessary to change some of the test data, which again will influence the results.
- The amount of time spent on evaluating the test results in the beginning of every week (the testing is carried out in the weekend) has been reduced from 1.5 - 2 hours to about 0.5 hour. This makes it possible to perform additional testing more often without spending extra time.
- The configuration item library makes it easier to ascertain that the company quality claims of the software is good enough to be sent to the customers, that is, if the software has passed all tests with reasonable results.

TSC wanted to find out to what extent they deliver implemented change requests on time, that is, at the estimated date, and whether the delivery rate was improved as a result of the project.

At the initial phase of the ICONMAN project, some baseline measurements were performed. Similar measurement was performed in the rest of the project. The first collection of data was carried out in the period from January through April 1997 - a period of three months. All results are presented in Table 11.8.

Period of time	Status	Fixed requests	Delivery on time	Late delivery	Still within delivery date
Jan. - April	Fixed	11	5	6	
	Open	9		3	3
May - Oct.	Fixed	65	50	15	
	Open	6		3	3

Table 11.8: Number of change requests received and by whom for both periods

In the first period, an average of 7 change requests were received (registered) by TSC each month and in the last period there were an average of 10. The main reasons for this increase in change requests from the first to the second period are:

- Increase in the amount of customers
- More comprehensive use of the software in some companies

- The internal testing has become more systematic

Although the amount of change requests has increased, TSC has managed to increase the number of change requests (here: fixed change requests) delivered in time from 45% to 77%.

Aktuar Systemer AS - In Aktuar Systemer, metrics were identified in order to establish a rough estimate for the amount of resources spent on error correction in Aktuar Systemer, both for planning purposes and as an indicator of the quality level in the systems development process. In addition, the process of generating production releases was addressed, and there was a need to quantify it. The following metrics were chosen:

- Time resources consumed on error correction (hours/error)
- Time resources consumed in generating a production release (hours/release)

No clear trend can be observed regarding time spent on error correction. Regarding time spent on production releases, a significant reduction can be observed.

Error correction

A comparison of measurements performed in 1996 and 1997 shows an increase in time spent on correcting errors (Table 11.11). The 1997-estimate is based on a more reliable data collection than the one in 1996, which was partly based on data we had to reconstruct. We believe an average level of 4.9 hours/error will be confirmed in future measurements, and was derived from the material in the following two tables:

Bug fixing period; programming was carried out.	Number of error reports received	Number of errors rejected or categorised as change of functionality	Number of errors corrected	Comments
10.04.96 - 29.05.96	40	32	8	During this period, prior to their acquisition of a special version of the Abskus system (including source code), one of our customers carried out a short and intense test program.
01.04.97 - 31.10.97	67	12	55	This period of approximately six months is the normal duration for preparing a new release. The testing was performed by various developers.

Table 11.9: Number of errors corrected

Table 11.9 shows the number of errors in the 1996 and 1997-periods. Of the 40 errors reported in the 96-period, only 8 (20%) resulted in programming being performed. This portion increased considerably in the 97-period when 55(82%) error reports involved programming. The reason why 80% of the 1996-error reports were categorized as orders for change of functionality or rejected, can be explained by a weekly co-ordinated test program, where the test-persons received incomplete descriptions of the expected level of functionality. This was corrected in the test program for 1997, and this period therefore constitutes a better basis for evaluation.

Table 11.10 lists the total number of hours spent on error correction in the two periods by developers A, B and C. These hours comprise all time resources spent; like review, analysis, categorization, and eventually programming.

Persons	A	B	C	Total
Total hours spent on error correction in the 1996-period	72.0	13.0	9.5	94.5
Total hours spent on error correction in the 1997-period	192.8	85.2	47.8	325.8

Table 11.10: Overall resource usage

Combining the data in Tables 11.9 and 11.10 gives the results as shown in Table 11.11:

Period	Number of errors	Total hours	Average number of hours pr. error
1996	40	94.5	2.4
1997	67	325.8	4.9

Table 11.11: Resource usage pr. error

A reason for the increase in resource usage may also be found in the strengthened emphasis on test procedures introduced by our new financial owner. This has definitely contributed in uncovering bugs of a more principal character, errors that have existed for a period of time. It is our impression that these errors have consumed a greater part of the time resources spent on error correction.

In addition, the establishment of a Change Request-database has given the process of reporting errors a more formal status. This has tended to increase the number of error reports being logged.

Production releases

The other area that was focused in Aktuar Systemer was time spent on generating production releases. The new CM-routines have transformed an activity that earlier was cumbersome and time-consuming into a routine task performed against the configuration item library (MS Visual Source Safe).

Each new release is now built incrementally during the development cycle as opposed to earlier when everything was done in the end. The reduced resource usage is however not the main advantage of the new procedures. The greatest gain lies in the fact that compiling a release package during the development cycle, and represents a much less error prone method than assembling and compiling everything in the end of the release cycle. Table 11.12 shows the time spent on generation of production releases.

Period	Description	#Hours Dev. A	#Hours Dev. B	Total
1996	Production release	44.5	26.0	70.5
1997- Sept.	Production release	1.0		1.0
1997 - Oct	Alarm release (Bug fix)	20.2	8.0	28.2

Table 11.12: Time spent on the generation of production releases

This autumn we had to produce an additional release with a small number of bug fixes based on the level of source code found in the last production release (Sept.-97), and where only limited testing is performed. In Aktuar Systemer we name this an 'alarm release', since it is forced by the discovery of serious errors in the newly distributed production release. The 28.2 hours spent on this activity comprise both the development of new alarm-routines and the preparation of the release. In the future we expect to prepare an alarm release just as fast as a production release.

11.5.2 Business

All three companies together used a total amount of approximately 200 person-days during the first 12 months of the experiment to introduce configuration management in the respective companies. This included implementing appropriate configuration management routines, selection and installation of tools and training of employees. Most of the time was spent on implementing the CM -routines.

From a business perspective, for us as small companies with short-term commitments, the question arises if the time and costs spent on introducing configuration management is worthwhile as a long-term investment. In the past sections, we have described a number of outcomes from the experiment, both technical and customer related.

In addition, the interaction with our customers has been simplified because historical information is now easily available. This is valid for both the configuration item library and the change request database. Two typical examples are the possibility for reproduction of previous releases for test purposes, and being able to give updated information to customers inquiring about change requests they have sent.

11.5.3 Organization

The introduction of the configuration management routines has led to more systematic work processes in all companies.

In TSC and AS regular follow up meetings on change requests are held. In TSC the results from the weekly testing of the software are also presented at the same meetings. All companies have altered their routines for collection of effort data in order to meet the requirement of the metrics program in the project.

The CM-routines were designed in a way, which would, but did not cause any changes in the overall organizational structure of the companies. After all, this was also not expected, given the small number of employees in each company.

11.5.4 Culture

All three companies are characterized by an entrepreneur atmosphere where responsibility to a certain degree is shared, and where a high mutual appreciation of each other's abilities prevails.

In all companies the employees show an interest in learning something new if this proves helpful in accomplishing their daily tasks. Thus, although not everyone was involved in the development of the CM-routines, after an initial hesitation especially concerning the registration of effort data for the handling of change requests, the routines are now in operation.

In the last stage of the experiment, we have experienced that the routines introduced facilitates the developers' daily tasks and make them more organized. This has been confirmed in the interviews of the consultants with the developers.

11.5.5 Skills

The project has increased the skills of the local project leaders concerning project management when introducing innovations in their organizations. The project leaders also improved their ability to transfer knowledge and to train and educate their colleagues in the companies. The project also showed the importance of reflecting work practices from time to time and supported staff capability to perform such a reflection process. More technically, the staff of the companies in general has now acquired new skills in configuration management as a business and software engineering principle.

The involved software professionals now act more systematically with respect to the daily tasks. As mentioned earlier, at least in two companies this systematic approach is accompanied by a new flexibility allowing developers to work on parts of the product they were not familiar with before the project. Furthermore, the developers have all been introduced to the establishment and use of a metrics programme to gain increased control over the development process.

11.6 Conclusions and Future Actions

This project's main objective was to implement configuration management in three small software development companies and to assess the effect of this measure from a business and a technical point of view. The ultimate goal was to improve the performance of these organizations.

At the end of the project, we conclude that the implementation of configuration management in the three organizations has been successfully completed.

This claim is backed up by results from both qualitative and quantitative measurements. More generally, we insist that configuration management in small companies is worthwhile. The implementation process was however far more demanding than we anticipated.

Configuration management is a complex process and highly integrated with change management, both with respect to error correction and handling customer requests. As such, configuration management in small companies dealing mainly with the development of one product has a large impact on the way the whole business is organized.

Formalized routines for configuration management will help small companies in planning their production process.

For the three companies involved, the project was a first step for the following future actions:

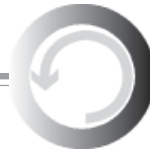
- The use of all routines and tools has to be consolidated, for example:
 - Necessary testing has to be intensified
 - Necessary time logging as to be improved
- Further metrics to extend the control over the development process have to be implemented
- Documentation of the development process has to be further improved

For Event AS, the ICONMAN project has also resulted in a new PIE project with the title “A Statistical Approach to Support Project Estimation and Management”. In this project, the main objective is to perform a statistical analysis of data from the change request database established in the ICONMAN.



SUMMARY

- The ICONMAN project's main objective was to implement configuration management in three small software development companies and to assess the effect of this measure from a business and a technical point of view
- Configuration management is a complex process and highly integrated with change management, both with respect to error correction and handling customer requests
- Implementing configuration management is an iterative process, and requires continuous refinement
- Configuration management in small companies dealing mainly with the development of one product has a large impact on the way the whole business is organized
- Formalized routines for configuration management will help small companies in planning their production process



CHECK YOUR PROGRESS

1. The use of configuration management will also address documentation and error reporting. **[True/False]**
2. The ICONMAN project's main objective was to implement _____ management.
3. The ICONMAN project was divided in three main phases. **[True/False]**
4. One of the objectives of the ICONMAN project was to formalize and document internal procedures for configuration management in each company. **[True/False]**
5. The ICONMAN project involved three companies: Event AS, Technical Software Consultants (TSC), and Aktuar Systemer AS. **[True/False]**