
**M.SC(IT) PART-2 (SEM-3)** **PAPER: MITM2103T**

**SOFTWARE ENGINEERING**

UNITNO.1

Center for Distance and Online Education, PunjabiUniversity,Patiala

Lesson No:

1.1 : **Introductory Concepts**

1.2 : **The Software Life Cycle**

1.3 : **Software Process Models**

1.4 : **Project Planning and Organization**

1.5 : **The Requirement Analysis**

1.6 : **Software Design**

1.7 : **DFD and UML**

1.8 : **UML-II**

**(Syllabus)**

## MITM2103T : Software Engineering

**Maximum Marks: 70** **Maximum Time: 3 Hrs.**

**Minimum Pass Marks: 35%**

**Course Objective:** The course is designed to understand the basics of software product development activity and to gain the knowledge of different phases of software development and associated challenges. On completion of this course, the student will be able to

- Understand the traditional approach and models of software development
- Conduct systematic design process using structured and object oriented design methodology
- Create test data to perform testing activity
- Explore tools/techniques to aid the software development

## Course Content

### SECTION A

Software Engineering : History, Definition, Goal; The role of the Software Engineer, The Software Life Cycle, The relationship of Software Engineering to other areas of Computer Science, Classification of Software Qualities, Representative Qualities, Software process models: Waterfall model, prototyping, spiral; Tools and techniques for process modeling, Management of software engineering management functions, project planning and organization.

Requirement Analysis: The requirement process, types of requirements, Characteristics and components of SRS, Data flow Diagrams, Data Dictionary, UML diagrams for specifying behaviors ,metrics, verification of SRS.

Design and Software architecture: The Software design activity and its objectives, Abstraction, Modularity, Coupling-Cohesion criteria, Object-Oriented Design: generalization and specialization, associations and aggregations.

### SECTION B

Coding: Programming standards and procedures, programming guidelines, documentation, and Code verification techniques.

Verification and validation: Approaches to verification, testing goals, principles,

Equivalence class partitioning, Boundary value analysis, mutation testing, graph based testing, cyclomatic complexity, test planning ,automated testing tools, features of Object-Oriented testing.

Software maintenance: The nature of maintenance, maintenance problems, maintenance techniques and tools.

Software re-engineering, reverse engineering, forward engineering: forward Engineering for Object-oriented and client/server architecture, Building blocks for CASE, CASE tools and applications.

**Pedagogy:**

The Instructor is expected to use leading pedagogical approaches in the class room situation, research-based methodology, innovative instructional methods, extensive use of technology in the class room, online modules of MOOCS, and comprehensive assessment practices to strengthen teaching efforts and improve student learning outcomes.

The Instructor of class will engage in a combination of academic reading, analyzing case studies, preparing the weekly assigned readings and exercises, encouraging in class discussions, and live project based learning.

**Text and Readings:** Students should focus on material presented in lectures. The text should be used to provide further explanation and examples of concepts and techniques discussed in the course:


1. Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, " Fundamentals of Software Engineering", 2nd edition Pearson Education. 2003.
2. Shari Lawrence Pfleeger, " Software Engineering : Theory and Practice", 2nd edition, Pearson Education, 2003.
3. P.Jalota, "An Integrated Approach to SoftwareEngineering", Narosa Publications.
4. Roger.S.Pressman," SoftwareEngineering-A practitioner's Approach", 3rd edition,McGraw-Hill.


**Scheme of Examination**

- English will be the medium of instruction and examination.
- Written Examinations will be conducted at the end of each Semester as per the Academic Calendar notified in advance
- Each course will carry 100 marks of which 30 marks shall be reserved for internal assessment and the remaining 70 marks for written examination to be held at the end of each semester.
- The duration of written examination for each paper shall be three hours.

- The minimum marks for passing the examination for each semester shall be 35% in aggregate as well as a minimum of 35% marks in the semester-end examination in each paper.
- A minimum of 75% of classroom attendance is required in each subject.

**Instructions to the External Paper Setter**

The question paper will consist of three Sections: A, B and C. Sections A and B will have four questions each from the respective section of the syllabus and will carry 10.5 marks for each question. Section C will consist of 7-15 short answer type questions covering the entire syllabus uniformly and will carry a total of 28 marks.

**Instructions for candidates**

- Candidates are required to attempt five questions in all, selecting two questions each from section A and B and compulsory question of section C.
- Use of non-programmable scientific calculator is allowed.

**LESSON No. 1.1** **AUTHOR: Dr. VISHAL SINGH**

# Introductory Concepts

**Objectives**



**Objectives:**

In this lesson we will study the meaning of Software engineering, its history, role and importance and the Software life cycle. We will also study the relationship of Software Engineering to other areas of Computer science

## 1.0 Introduction

**Software engineering is a detailed study of engineering to the design, development and maintenance of software. Software engineering is the field of computer science that deals with the building of software systems which are so large or so complex that they are built by a team or teams of engineers.** Usually, these software systems exist in multiple versions and are used for many years. During their lifetime, they undergo many changes-to fix defects, to enhance existing features, to add new features, to remove old features or to be adapted to run in a new environment. It is a detailed study of engineering to the design, development and maintenance of software.

Parnas has defined software engineering as "**multi-person construction of multi-version software.**" This definition captures the essence of software engineering and highlights the differences between programming and software engineering. A programmer writes a complete program, while a software engineer writes a software component that will be combined with components written by other software engineers to build a system. The component one writes may be modified by others; it may be used by others to build different versions of the system long after one has left the project. Programming is primarily a personal activity, while software engineering is essentially a team activity.

**Software Engineering is the use of techniques, methods and methodologies to develop high quality software which is**

- **Reliable**
- **Easy to understand**
- **Useful**
- **Modular**
- **Efficient**
- **Modifiable**
- **Reusable**
- **Good user interface**
- **Well documented**
- **Delivered in cost effective and timely manner**

Software engineering is a profession dedicated to designing, implementing and modifying software so that it is of high quality, affordable, maintainable and fast to build. It is a systematic approach to the analysis, designing, implementation and reengineering of software, that is, the application of engineering to software. Software engineering has made progress since the 1960s. There are standard techniques that are used in the field. But the field is still far from achieving the status of a classic engineering discipline. Many areas remain in the field that are still being taught and practiced on the basis of informal techniques. There are no generally accepted methods even for specifying what a software system should do. In designing an electrical engineering system, such as an amplifier, the system is specified precisely. All parameters and tolerance levels are stated clearly and are understood by the customer and the engineer. In software engineering, we are just

beginning to define both what such parameters ought to be for a software system and-an apparently much harder task-how to specify them.

Furthermore, in classic engineering disciplines, the engineer is equipped with tools and the mathematical maturity to specify the properties of the product separately from those of the design. For example, an electrical engineer relies on mathematical equations to verify that a design will not violate power requirements. In software engineering, such mathematical tools are not well developed. The typical software engineer relies much more on experience and judgment rather than mathematical techniques. While experience and judgment are necessary, formal analysis tools are also essential in the practice of engineering.

## 2.0 Software engineering goals

Stated requirements even for small systems are frequently incomplete when they are originally specified. Besides accomplishing these stated requirements, a good software system must be able to easily support changes to these requirements over the system's life. Thus, a major goal of software engineering is to be able to deal with the effects of these changes. The most often cited software engineering goals are:

**a.** **Maintainability :**It should be possible to easily introduce changes to the software without increasing the complexity of the original system design.

**b.** **Reliability :** The software should prevent failure in design and construction as well as recover from failure in operation. Put another way, the software should perform its intended function with the required precision at all times.

**c.** **Efficiency** :The software system should use the resources that are available in an optimal manner.

**d.** **Understandability :** The software should accurately model the view the reader has of the real world. Since code in a large, long-lived software system is usually read more times than it is written, it should be easy to read at the expense of being easy to write, and not the other way around.

## 3.0 Software engineering principles

In order to attain a software system that satisfies the above goals, sound engineering principles must be applied throughout development, from the design phase to final fielding of the system. These include:

**a.** **Abstraction :** "The essence of abstraction is to extract essential properties while omitting inessential detail." The software should be organized as a ladder of abstraction in which each level of abstraction is built from lower levels. The code is sufficiently conceptual so the user need not have a great deal of technical background in the subject. The reader should be able to easily follow the logical path of each of the various modules. The decomposition of the code should be clear.

**b.** **Information Hiding :** The code should contain no unnecessary detail. Elements that do not affect other portions of the system are inaccessible to the user, so that only the intended operations can be performed. There are no "undocumented features".

**c.** **Modularity :** The code is purposefully structured. Components of a given module are logically or functionally dependent.

**d.** **Localization :** The breakdown and decomposition of the code is rational. Logically related computational units are collected together in modules.

**e.** **Uniformity :** The notation and use of comments, specific keywords and formatting is consistent and free from unnecessary differences in other parts of the code.

**f.** **Completeness :** Nothing is blatantly missing from any module. All important or relevant components are present both in the modules and in the overall system as appropriate.

**g.** **Confirmability :** The modules of the program can be tested individually with adequate rigor. This gives rise to a more readily alterable system and enables the reusability of tested components.

## 4.0 The Role of Software Engineering in System Design

A software system is often a component of a much larger system. The software engineering activity is therefore a part of a much larger system design activity in which the requirements of the software are balanced against the requirements of other parts of the system being designed. For example, a telephone switching system consists of computers, telephone lines and cables, telephones, perhaps other hardware such as satellites and finally, software to control the various other components. It is the combination of all these components that is expected to meet the requirements of the whole system.

A requirement such as "the system must not be down for more than a second in 20 years" or "when a receiver is taken off the hook, a dial tone is played within half a second" can be satisfied with a combination of hardware, software and special devices. The decision of how best to meet the requirement can only be made by considering many trade-offs. Power plant or traffic monitoring systems, banking systems and hospital administration systems are other examples of systems that exhibit the need to view the software as a component of a larger system.

To do software engineering right, then, requires a broader look at the general problem of system engineering. It requires the software engineer to be involved when the requirements are being developed initially for the whole system. It requires that the software engineer attempt to understand the application area rather than just what abstract interfaces the software must meet. For example, if the system is aimed at users who are going to use primarily a menu system, it is not wise to develop a sophisticated word processor as a component of the system.

Above all, any engineering discipline requires the application of compromise- A classic compromise concerns the choice of what should be done in software and what should be done in hardware. Software implementation offers flexibility, while hardware implementation offers performance. For example, a coin-operated machine that could be built either with several coin slots, one for each type of coin or a single slot, leaving it to software to recognize the different coins- An even more basic compromise involves the decision as to what should be automated and what should be done manually.

## 5.0 A History of Software Engineering

The birth and evolution of software engineering as a discipline within computer science can be traced to the evolving and maturing view of the programming activity. In the early days of computing, the problem of programming was viewed essentially as how to place a sequence of instructions together to get the computer to do something useful. The problems being programmed were quite well understood-for example, how to solve a differential equation. The program was written by, say, a physicist to solve an equation of interest to him or her. The problem was just between the user and the computer-no other person was involved.

As computers became cheaper and more common, more and more people started using them. Higher level languages were invented in the late 1950s to make it easier to communicate with the machine. But still, the activity of getting the computer to do something useful was essentially done by one person who was writing a program for a well-defined task.

It was at this time that "programming" attained the status of a profession: you could ask a programmer to write a program for you instead of doing it yourself- This introduced a separation between the user and the computer. Now the user had to specify the task in a form other than the precise programming notation used before. The programmer then interpreted this specification and translated it into a precise set of machine instructions. This, of course, sometimes resulted in the programmer misinterpreting the user's intentions, even in these usually small tasks.

Very few large software projects were being done at this time-the early 1960s-and these were done by computer pioneers who were experts. For example, the CTSS operating system developed at MIT was indeed a large project, but it was done by highly knowledgeable and motivated individuals.

In the middle to late 1960s, truly large software systems were attempted commercially. The best documented of these projects was the OS 360 operating system for the IBM 360 computer family. The large projects were the source of the realization that building large software systems was materially different from building smaller systems. There were fundamental difficulties in scaling up the techniques of small program development to large software development. The term "software engineering" was invented around this time and conferences were held to discuss the difficulties these projects were facing in delivering the promised products. Large software projects were universally over budget and behind schedule. Another term invented at this time was "software crisis."

It was discovered that the problems in building large software systems were not a matter of putting computer instructions together. Rather, the problems being solved were not well understood, at least not by everyone involved in the project or by any single individual. People on the project had to spend a lot of time communicating with each other rather than writing code. People sometimes even left the project and this affected not only

the work they had been doing but the work of the others who were depending on them. Replacing an individual required an extensive amount of training about the "folklore" of the project requirements and the system design. Any change in the original system requirements seemed to affect many parts of the project, further delaying system delivery. These kinds of problems just did not exist in the early "programming" days and seemed to call for a new approach.

Many solutions were proposed and tried. Some suggested that the solution lay in better management techniques. Others proposed different team organizations. Yet others argued for better languages and tools. Many called for organization-wide standards such as uniform coding conventions. There was no shortage of ideas. The final consensus was that the problem of building software should be approached in the same way that engineers had built other large complex systems such as bridges, refineries, factories, ships and airplanes. The point was to view the final software system as a complex product and the building of it as an engineering job. The engineering approach required management, organization, tools, theories, methodologies and techniques. And thus was software engineering born.

The above history emphasizes the growth of software engineering starting from programming. Other technological trends have also played significant roles in the evolution of the field. The most important influence has been that of the change in the balance of hardware and software costs. Whereas the cost of a computerized system used to be determined largely by hardware costs and software was an insignificant factor, today the software component can account for far more than half of the system cost. The declining cost of hardware and the rising cost of software have tipped the balance further in the direction of software, setting in motion a new economical emphasis in the development of software engineering.

Another evolutionary trend has been internal to the field itself. There has been a growing emphasis on viewing software engineering as dealing with more than just "coding." Instead, the software is viewed as having an entire life cycle, starting from conception and continuing through design, development, deployment and maintenance and evolution. The shift of emphasis away from coding has sparked the development of methodologies and sophisticated tools to support teams involved in the entire software life cycle.

We can expect the importance of software engineering to continue to grow for several reasons. First is economics: in 1985, around $140 billion were spent annually on software worldwide. It is anticipated that software costs will grow to more than -$450 billion worldwide by 1995. This fact alone ensures that software engineering will grow as a discipline. Second, software is permeating our society. More and more, software is used to control critical functions of various machines such as aircrafts and medical devices. This fact ensures the growing interest of society in dependable software, to the extent of legislating specific standards, requirements and certification procedures. No doubt, it will continue to be important to learn how to build better software better.

## 6.0 The Role of Software Engineer

The evolution of the field has defined the role of the software engineer and the required experience and education. **A software engineer must of course be a good programmer, be well-versed in data structures and algorithms and be fluent in one or more programming languages**. These are requirements for "programming-in-the-small," roughly defined as building programs that can be written in their entirety by a single individual. But a software engineer is also involved in "programming-in-the-large," which requires more.

The software engineer must be familiar with several design approaches, be able to translate vague requirements and desires into precise specifications and be able to converse with the user of a system in terms of the application rather than in "computerese.' These in turn require the flexibility and openness to grasp and become conversant with, the essentials of different application areas. The software engineer needs the ability to move among several levels of abstraction at different stages of the project, from specific application procedures and requirements, to abstractions for the software system, to a specific design for the system and finally to the detailed coding level.

Modeling is another requirement. The software engineer must be able to build and use a model of the application to guide choices of the many trade-offs that he or she will face. The model is used to answer questions about both the behavior of the system and its performance. Preferably, the model can be used by the engineer as well as the user. The software engineer is a member of a team and therefore needs communication skills and interpersonal skills. The software engineer also needs the ability to schedule work, both of his or her own and that of others.

As discussed above, a software engineer is responsible for many things. In practice, many organizations divide the responsibilities among several specialists with different titles. For example, an analyst is responsible for deriving the requirements and a performance analyst is responsible for analyzing the performance of a system. A rigid fragmentation of roles, however, is often counterproductive.

## 7.0 The relationship of Software Engineering to other areas of Computer science

Software engineering has emerged as an important field within computer science. Indeed, there is a synergistic relationship between it and many other areas in computer science: these areas both influence and are influenced by software engineering. In the following subsections, we explore the relationship between software engineering and some of the other important fields of computer science.

### 7.1 Programming Languages

The influence of software engineering on programming languages is rather evident. Programming languages are the central tools used in software development. As a result, they have a profound influence on how well we can achieve our software engineering goals. In turn, these goals influence the development of programming languages.

The most notable example of this influence in recent programming languages is the inclusion of modularity features, such as separate and independent compilation and the separation of specification from implementation, in order to support team development of large software. The Ada programming language, for example, supports the development of “packages”-allowing the separation of the package interface from its implementation-and libraries of packages that can be used as components in the development of independent software systems. This is a step towards making it possible to build software by choosing from a catalog of available components and combining them, similarly to the way hardware is built.

In the opposite direction, programming languages have also influenced software engineering. One example is the idea that requirements and design should be described precisely, possibly using a language as rigorous and machine-processable as a programming language. As another example, consider the view that the input to a software system can be treated as a program coded in some "programming" language. The commands that a user can type into a system are not just a random collection of characters; rather they form a language used to communicate with the system. Designing an appropriate input language is a part of designing the system interface.

Most traditional operating systems, such as OS 360, were developed before this view was recognized. As a result, the interface to them-the job control language (JCL) is generally agreed to be extremely complicated to master. On the other hand, more recent operating systems, such as UNIX, really do provide the user with a programming language interface, thus making them easier to learn and use.

One result of viewing the software system interface as a programming language is that compiler development tools-which are quite well developed-can be used for general software development. For example, we can use grammars to specify the syntax of the interface and parser-generators to verify the consistency and unambiguity of the interface and automatically generate the front end for the system.

User interfaces are an especially interesting case because we are now seeing an influence in the opposite direction. The software engineering issues revolving around the user interfaces made possible by the widespread use of graphical bit-mapped displays and mice have motivated programming language work in the area of "visual" or "pictorial languages. These languages attempt to capture the semantics of the windowing and interaction paradigms offered by the new sophisticated graphical display devices.

Yet another influence of the programming language field on software engineering is through the implementation techniques that have been developed over the years for language processing. Perhaps the most important lesson learned has been that formalization leads to automation: stating a formal grammar for a language allows a parser to be produced automatically. This technique is exploited in many software engineering areas for formal specification and automatic software generation.

Another implementation influence is due to the two major approaches to language processing-compilation and interpretation-that have been studied extensively by compiler designers. In general, the interpretive approach offers more run-time flexibility and the compilation approach offers more run-time efficiency. Either is generally applicable to any software system and can therefore become another tool in the software engineer's toolbox. An example of their application outside the programming language area can be seen in the data-base field. The queries posed to a data base may be either compiled or interpreted. Common types of queries are compiled to ensure their fast execution: the exact search path used by the data-base system is determined before any queries are made to the system. On the other hand, since not all types of queries can be predicted by the data-base designer, so-called ad hoc queries are also supported, which require the data base to select a search path at run time. The ad hoc queries take longer to execute but give the user more flexibility.

## 7.2 Operating Systems

The influence of operating systems on software engineering is quite strong primarily because operating systems were the first really large software systems built, and therefore they were the first instances of software that needed to be engineered. Many of the first software design ideas originated from early attempts at building operating systems.

Virtual machines, levels of abstraction, and separation of policy from mechanism are all concepts developed in the operating system field with general applicability to any large software system. For example, the idea of separating a policy that an operating system wants to impose, such as assuring fairness among jobs, from the mechanism used to accomplish concurrency, such as time slicing, is an instance of separating the "what" from the "how"-or the specification from the implementation-and the changeable parts from what remains fixed in a design. The idea of levels of abstraction is just another approach to modularizing the design of a system.

In the opposite direction, the influence of software engineering on operating systems can be seen in both the way operating systems are structured and the goals they try to satisfy. For example, the UNIX operating system attempts to provide a productive environment for software development. A class of tools provided in this environment supports software configuration management-a way to maintain and control the relationships among the different components and versions of a software system. Examples of the influence of software engineering techniques on the *structure* of operating systems can be seen in portable operating systems and operating systems that are structured to contain a small "protected" kernel that provides a minimum of functionality for interfacing with the hardware and a "nonprotected" part that provides the majority of the functionality previously associated with operating systems. For example, the nonprotected part may allow the user to control the paging scheme, which has traditionally been viewed as an integral part of the operating system.

Similarly, in early operating systems, the command language interpreter was an integral part of the operating system. Today, it is viewed as just another utility program. This allows, for example, each user to have a personalized version of the interpreter. On many UNIX systems, there are at least three different such interpreters.

### 7.3 Data Bases

Data bases represent another class of large software systems whose development has influenced software engineering through the discovery of new design techniques. Perhaps the most important influence of the data-base field on software engineering is through the notion of "data independence," which is yet another instance of the separation of specification from implementation. The data base allows applications to be written that use data without worrying about the underlying representation of the data. This independence allows the data base to be changed in certain ways-for example, to increase the performance of the system-without any need to change the applications. This is a perfect example of the benefit of abstraction and separation of Separation of concerns, two key software engineering principles.

Another interesting impact of data-base technology on software engineering is that it allows data-base systems to be used as components of large software systems. Since data bases have solved the many problems associated with the management of concurrent access to large amounts of information by multiple users, there is no need to reinvent these solutions when we are building a software system-we can simply use an existing data-base system as a component.

One interesting influence of software engineering on data-base technology has its roots in early attempts to use data bases to support software development environments. This experience showed that traditional data-base technology was incapable of dealing with the problems posed by software engineering processes. For example, the following requirements are not handled well by traditional data bases: storing large structured objects such as source programs or user manuals; storing large unstructured objects such as object code and executable code; maintaining different versions of the same object; and storing objects, such as a product, with many large structured and unstructured fields, such as source code, object code, and a user manual. Another difficulty dealt with the length of *transactions.* Traditional data bases support short transactions, such as a bank account deposit or withdrawal. Software engineers, on the other hand, need very long transactions: an engineer may require a long compilation to occur on a multimodule system or may check out a program and work on it for weeks before checking it back in. The problem posed for the data base is how to handle the locking of the code during these weeks. What if the engineer wants to work only on a small part of the program? Are all other access to the program forbidden?

There is presently considerable work going on in the data-base area to address such problems, ranging from introducing new models for data bases to adapting current database models.

## 7.4 Artificial Intelligence

Artificial intelligence is another field that has exerted influence on software engineering. The software systems built in the artificial intelligence research community have been among the most complex systems built. But they have been different from other software systems in significant ways. Typically, artificial intelligence systems are built with only a vague notion of how the system is going to work. The term "exploratory development" has been used for the process followed in building these systems.

This process is the opposite of traditional software engineering, in which we go through well-defined steps attempting to produce a complete design before proceeding to coding. These developments have given rise to new techniques in dealing with specifications, verification, and reasoning in the presence of uncertainty. Other techniques advanced by artificial intelligence include the use of logic in both software specifications and programming languages.

The logic orientation seems to be filling the gap between specification and implementation by raising the level of implementation languages higher than before. The logic approach to specification and programming is also called *declarative.* The idea is that we declare the specifications or requirements rather than specifying them procedurally; the declarative description is then executable. Logic programming languages such as PROLOG help us follow this methodology.

Software engineering techniques have been used in newer artificial intelligence systems-for example, in *expert systems.* These systems are modularized, with a clear separation between the facts "known" by the expert system and the rules used by the system for processing the facts-for example, a rule to decide on a course of action. This separation has enabled the building and commercial availability of "expert system shells" that include the rules only. A user can apply the shell to an application of interest by supplying application-specific facts. The idea is that the expertise about the application is provided by the user and the general principles of how to apply expertise to any problem are provided by the shell.

A different kind of symbiosis is currently taking place at the intersection of software engineering and artificial intelligence. Techniques of artificial intelligence are being applied to improve the software engineering tasks. For example, "programming assistants" are being developed to act as consultants to the programmer, watching for common programming idioms or the system requirements. Such "assistants" are also being developed to help in the testing activities of the software development, to debug the software.

The problem of providing interfaces for nonexpert users-for example, through the use of natural language-was first attacked by artificial intelligence. Cognitive models were also used to model the user. These works have influenced the very active area of user-interface design in software engineering.

## 7.5 Theoretical Models

Theoretical computer science has developed a number of models that have become useful tools in software engineering. For example, finite state machines have served both as the basis of techniques for software specifications and as models for software design and structure. Communication protocols and language analyzers are programs that have used finite state machines as their processing model.

Pushdown automata have also been used—for example, for operational specifications of systems and for building processors for such specifications. Interestingly, pushdown automata were themselves motivated by practical attempts to build parsers and compilers for programming languages.

Petri nets are yet another contribution of the theoretical computer science field to software engineering. While Petri nets were initially used to model hardware systems, in recent years they have been applied increasingly in the modeling of software. They are currently the subject of intensive study in many software engineering research organizations.

As another example, denotational semantics-a mathematical theory developed for describing programming language semantics-has been the source of recent developments in the area of specification languages.

Software engineering has also affected theoretical computer science. Algebraic specifications and abstract data type theory are motivated by the needs of software engineering. Also in the area of specifications, software engineering has focused more attention on non-first-order theories of logic, such as temporal logic. Theoreticians used to pay more attention to first-order theories than higher-order theories because the two are equivalent in power but first-order theories are more basic from a mathematical viewpoint. They are not as expressive as higher-order theories, however. A software engineer, unlike a theoretician, is interested both in the power and the expressiveness of a theory. For example, temporal logic provides a more compact and natural style for specifying the requirements of a concurrent system than do first-order theories. The needs of software engineering, therefore, have ignited new interest by theoreticians in such higher-order theories.

## 8.0 The relationship of Software Engineering to Other Disciplines

In the foregoing sections, we examined the relationship between software engineering and other fields of computer science. In this section, we explore how software engineering relates to other fields outside of computer science.

Software engineering need not be practiced in a vacuum. There are many problems that are not specific to software engineering and have been solved in other fields. Their solutions can be adapted to software engineering. Thus, there is no need to reinvent every solution. For example, the fields of psychology and industrial design can guide us in the development of better user interfaces.

## 8.1 Management Science

A large part of software engineering is involved with management issues. Such management has two aspects: technical management and personnel management. The generic issues in any kind of project management include project estimation, project scheduling, human resource planning, task decomposition and assignment, and project tracking. The personnel issues involve hiring personnel, motivating people and assigning the right people to the right tasks.

Management science studies exactly these issues. Many models have been developed that can be applied in software engineering. By looking to management science, we can exploit the results of many decades of study.

In the opposite direction, software engineering has provided management science with a new domain in which to test management theories and models. The traditional management approaches to assembly-line production clearly do not apply to software engineering management, giving rise to a search for more applicable approaches.

## 8.2 Systems Engineering

Systems engineering is the field concerned with studying complex systems. The hypothesis is that certain laws govern the behavior of any complex system composed many components with complex relationships. Systems engineering is useful when you are interested in concentrating on the system as opposed to the individual components. Systems engineering tries to discover common themes that apply to diverse systems-for example, chemical plants, buildings and bridges.

Software is often a component of a much larger system. For example, the software in a factory monitoring system or the flight software on an airplane is just components of more complicated systems. Systems engineering techniques can be applied to the study of such systems. We can also consider a software system consisting of thousands of modules as a candidate complex system subject to systems engineering laws. On the other hand, system engineering has been enriched by expanding its set of analysis models-which were traditionally based on classical mathematics-to include discrete models that have been in use in software engineering.

## 9.0 Summary:

In this lesson we have discussed the meaning of Software engineering, its history, role and importance and the Software life cycle. We have also discussed the relationship of Software Engineering to other areas of Computer science

## 10.0 Self check exercise

Q1: What is Software Engineering? What is the role of Software Engineering in System Design?

Q2: Explain the various goals and principles of Software Engineering.

Q3: Write a detailed note on the history of Software Engineering.

Q4: Explain in detail the relationship of Software Engineering to other areas of Computer science.

## 11.0 Suggested readings

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

LESSON No. 1.2 AUTHOR: Dr. VISHAL SINGH

# The Software Life Cycle

**Objectives**



**Objectives:**

In this lesson we will study the software life cycle, which is a very important concept in the field of software engineering. We will study in detail the classification of software qualities and the representative qualities of the software.

## 1.0 The Software Life Cycle

It is a well defined structured sequence of stages in software engineering to develop the intended software. From the inception of an idea for a software system, until it is implemented and delivered to a customer and even after that, the system undergoes gradual development and evolution. **The software is said to have a life cycle composed of several phases. Each of these phases results in the development of either a part of**

**the system or something associated with the system, such as a test plan or a user manual. In the traditional life cycle model also called the "waterfall model"**, each phase has well-defined starting and ending points, with clearly identifiable deliverables to the next phase. In practice, it is rarely so simple. A sample waterfall life cycle model comprises the following phases.

**Requirements analysis and specification:** Requirements analysis is usually the first phase of a large-scale software development project. It is undertaken after a feasibility study has been performed to define the precise costs and benefits of a software system. The purpose of this phase is to identify and document the exact requirements for the system. Such study may be performed by the customer, the developer, a marketing organization, or any combination of the three. In cases where the requirements are not clear-e.g., for a system that has never been done before-much interaction is required between the user and the developer. The requirements at this stage are in end-user terms. Various software engineering methodologies advocate that this phase must also produce user manuals and system test plans.

**Design and specification:** Once the requirements for a system have been documented, software engineers design a software system to meet them. This phase is sometimes split into two subphases: architectural or high-level design and detailed design. High-level design deals with the overall module structure and organization, rather than the details of the modules. The high-level design is refined by designing each module in detail (detailed design).

Separating the requirements analysis phase from the design phase is an instance of a fundamental "what/how" dichotomy that we encounter quite often in computer science. The general principle involves making a clear distinction between what the problem is and how to solve the problem. In this case, the requirements phase attempts to specify what the problem is. There are usually many ways that the requirements may be met, including some solutions that do not involve the use of computers at all. The purpose of the design phase is to specify a particular software system that will meet the stated requirements. Again, there are usually many ways to build the specified system. In the coding phase, which follows the design phase, a particular system is coded to meet the design specification.

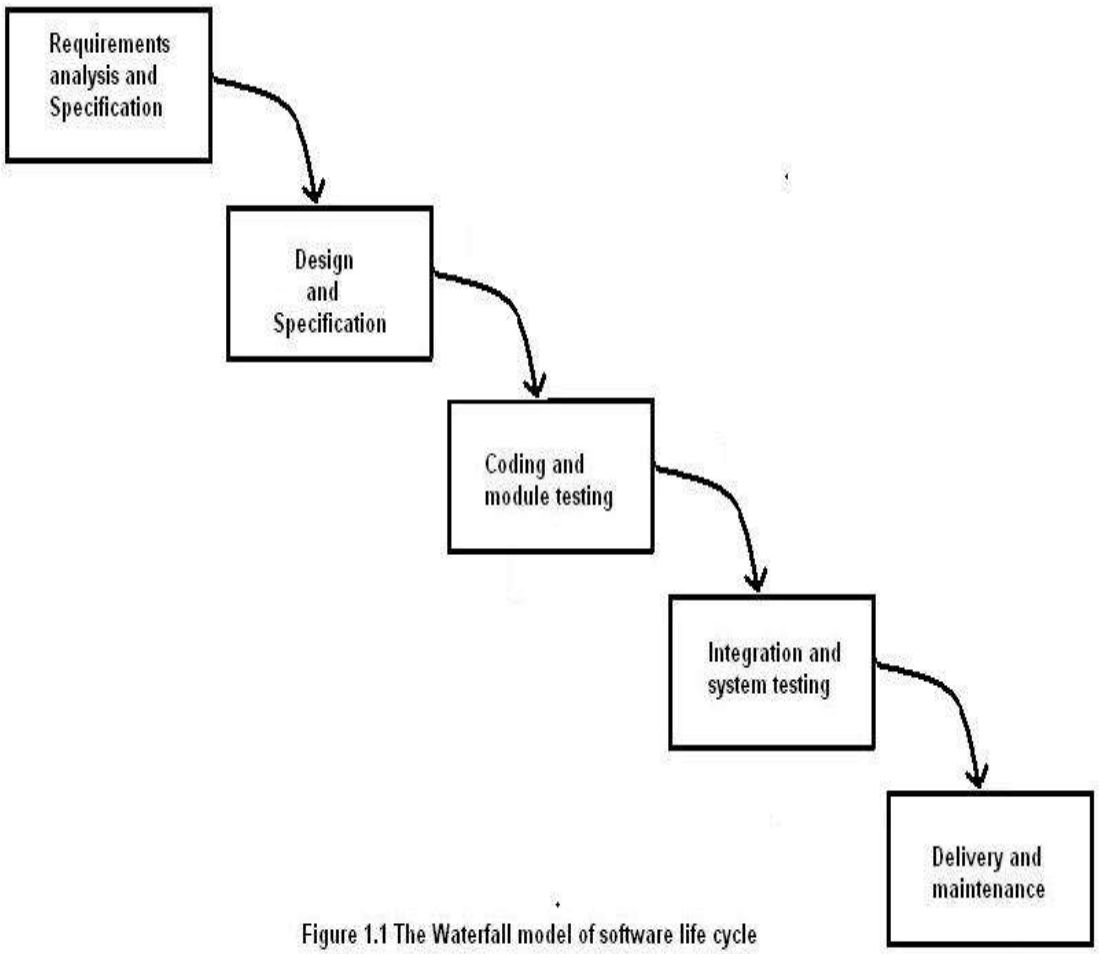


Figure 1.1 The Waterfall model of software life cycle

**Coding and module testing:** This is the phase that produces the actual code that will be delivered to the customer as the running system. The other Phases of the life cycle may also develop code, such as prototypes tests, and test drivers, but these are for use by the developer. Individual modules develops this phase are also tested before being delivered to the next phase.

**Integration and system testing:** All the modules that have been developed before and tested individually are put together integrated-in this phase and tested as a whole system.

**Delivery and maintenance:** Once the system passes all the tests, it is delivered to the customer and enters the maintenance phase. Any modifications made to the system after initial delivery is usually attributed to this phase.

Figure 1.1 gives a graphical view of the software development life cycle that provides a visual explanation of the term "waterfall" used to denote it. Each phase results that "flow" into the next and the process ideally proceeds in an orderly and linear fashion.

A commonly used terminology distinguishes between high phases and low phases of the software life cycle; the feasibility study, requirements analysis and high level design contribute to the former/and implementation-oriented activities contribute to the latter.

As presented here, the phases give a partial, simplified view of the conventional waterfall software life cycle. The process may be decomposed into a different set of phases, with different names, different purposes and different granularity. Entirely different life cycle schemes may even be proposed, not based on a strictly phased waterfall development. For example, it is clear that if any tests uncover defects in the system, we have to go back at least to the coding phase and perhaps to the design phase to correct some mistakes. In general, any phase may uncover problems in previous phases; this will necessitate going back to the previous phases and redoing some earlier work. For example, if the system design phase uncovers inconsistencies or ambiguities in the system requirements, the requirements analysis phase must be revisited to determine what requirements were really intended.

Another simplification in the above presentation is that it assumes that a phase is completed before the next one begins. In practice, it is often expedient to start a phase before a previous one is finished. This may happen, for example, if some data necessary for the completion of the requirements phase will not be available for some time. Or it might be necessary because the people ready to start the next phase are available and have nothing else to do. The research and experience over the past decade have shown that there is a variety of life cycle models and that no single one is appropriate for all software systems.

**The nature and qualities of Software**

The goal of any engineering activity is to build something-a product. The civil engineer builds a bridge, the aerospace engineer builds an airplane and the electrical engineer builds 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.

In some ways software products are similar to other engineering Products and in some ways they are very different. The characteristic that perhaps set software apart from other engineering products the most is that software is malleable. We can modify the product itself-as opposed to its design-rather easily. This makes software quite different from other products such as cars or ovens.

The malleability of software is often misused. While it is certainly possible to modify a bridge or an airplane to satisfy some new need-for example to make the bridge support more traffic or the airplane carry more cargo-such a modification is not taken lightly and certainly is not attempted without first making a design change and verifying the impact of the change extensively. Software engineers, on the other hand, are often asked to perform such modifications on software. Because of its malleability, we seem to think that changing software is easy. In practice, it is not.

We may be able to change the code easily with a text editor, but meeting the need for which the change was intended is not necessarily done so easily- Indeed we need to treat software like other engineering products in this regard: a change in software must be viewed as a

change in the design rather than in the code, which is just an instance of the product. We can indeed exploit the malleability property, but we need to do it with discipline.

Another characteristic of software is that its creation is human intensive: it requires mostly engineering rather than manufacturing. In most other 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. This process has to meet certain criteria to ensure the production of high-quality software.

Any product is expected to fulfill some need and meet some acceptance standards that set forth the qualities it must have. A bridge performs the function of making it easier to travel from one point to another; one of the qualities it is expected to have is that it will not collapse when the first strong wind blows or a convoy of trucks travels across it. In traditional engineering disciplines, the engineer has tools for describing the qualities of the product distinctly from the design of the product. In software engineering, the distinction is not yet so clear. The qualities of the software product are often intermixed in specifications with the qualities of the design.

In this lesson, we examine the qualities that are pertinent to software products and software production processes.

## 2.0 Classification of Software Qualities

There are many desirable software qualities. Some of these apply both to the product and to the process used to produce the product. The user wants the software product to be reliable, efficient and easy to use. The producer of the software wants it to be verifiable, maintainable, portable and extensible. The manager of the software project wants the process of software development to be productive and easy to control.

In this section, we consider two different classifications of software-related qualities: internal versus external and product versus process.

## 2.1 External versus Internal Qualities

We can divide software qualities into external and internal qualities. The external Qualities are visible to the user of the system and the internal qualities are those that concerns the developer of the system. In general the users of the software only care about the external qualities, but it is the internal qualities-which deal largely with the structure of the software-that help developers achieve the external qualities. For example, the internal quality of verifiability is necessary for achieving the external quality of reliability. In many cases, however, the qualities are related closely and the distinction between internal and external is not sharp.

## 2.2 Product and Process Qualities

We use a process to produce the software product. We can also attribute some qualities to the process, although process qualities often are closely related to product qualities. For example, if the process requires careful planning of system test data before any design and development of the system starts, product reliability will increase. Some qualities such as efficiency, apply both to the product and to the process.

It is interesting to examine the word product here. It usually refers to what is delivered to the customer. Even though this is an acceptable definition from the customer's perspective, it is not adequate for the developer who requires a general definition of a software product that encompasses not only the object code and the user manual that are delivered to the customer, but also the requirements, design, source code, test data, etc. With such a definition, all of the artifacts that are produced during the process constitute parts of the product. In fact, it is possible to deliver different subsets the same product to different customers.

For example, a computer manufacturer might sell to a process control company the object code to be installed in the specialized hardware for an embedded application. It might sell the object code and the user's manual to software dealers. It might even sell the design and the source code to software vendors who modify them to build other products. In this case, the developers of the original system see one product, the salespersons in the same company see a set of related products and the end user and the software vendor see still other, different products.

## 3.0 Representative Qualities

In this section, we present the most important qualities of software products and processes. Where appropriate, we analyze a quality with respect to the classifications discussed above.

## 3.1 Correctness, Reliability, and Robustness

The terms "correctness," "reliability," and "robustness" are often used interchangeably to characterize a quality of software that implies that the application performs its functions as expected. At other times, the terms are used with different meanings by different people, but the terminology is not standardized. This is quite unfortunate, because these terms deal with important issues. We will try to clarify these issues below, not only because we need a uniform terminology to be used throughout the book, but also because we believe that a clarification of the terminology is needed to better understand and analyze the underlying issues.

### 3.1.1 Correctness

A program is functionally correct if it behaves according to the specification of the functions it should provide (called functional requirements specifications). It is common

simply to use the term "correct" rather than "functionally correct"; similarly, in this context, the term "specifications" implies "functional requirements specifications." We will follow this convention when the context is clear.

The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. With most current software systems, no such specification exists. If a specification does exist, it is usually written in an informal style using natural language. Such a specification is likely to contain many ambiguities. Regardless of these difficulties with current specifications, however, the definition of correctness is useful. Clearly, correctness is a desirable property for software systems.

Correctness is a mathematical property that establishes the equivalence between the software and its specification. Obviously, we can be more systematic and precise in assessing correctness depending on how rigorous we are in specifying functional requirements. Correctness can be assessed through a variety of methods, some stressing an experimental approach (e.g., testing), others stressing an analytic approach (e.g.. formal verification of correctness). Correctness can also be enhanced by using appropriate tools such as high-level languages, particularly those supporting extensive static analysis. Likewise, it can be improved by using standard algorithms or using libraries of standard modules, rather than inventing new ones.

### 3.1.2 Reliability

Informally, software is reliable if the user can depend on it. The specialized literature on software reliability defines reliability in terms of statistical behavior-the probability that the software will operate as expected over a specified time interval.

Correctness is an absolute quality: any deviation from the requirements makes the system incorrect, regardless of how minor or serious is the consequence of the deviation. The notion of reliability is on the other hand, relative: if the consequence of a software error is not serious, the incorrect software may still be reliable.

Engineering products are expected to be reliable. Unreliable products, in general, disappear quickly from the marketplace. Unfortunately, software products have not achieved this enviable status yet. Software products are commonly released along with a list of "Known Bugs." Users of software take it for granted that "Release 1" of a product is "buggy." This is one of the most striking symptoms of the immaturity of the software engineering field as an engineering discipline.

In classic engineering disciplines, a product is not released if it has "bugs." You do not expect to take delivery of an automobile along with a list of shortcomings or a bridge with a warning not to use the railing. Design errors are extremely rare and worthy of news headlines. A bridge that collapses may even cause the designers to be prosecuted in court.

On the contrary, software design errors are generally treated as unavoidable. Far from being surprised with the occurrence of software errors, we expect them. Whereas with all other products the customer receives a guarantee of reliability, with software we get a disclaimer that the software manufacturer is not responsible for any damages due product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other products.

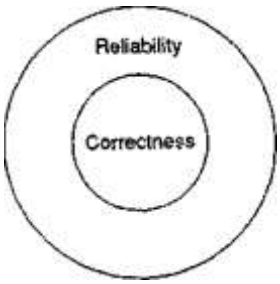



**Figure 2.1** Relationship between correctness and reliability in the ideal case.

### 3.1.3 Robustness

A program is robust if it behaves "reasonably," even in circumstances that were not anticipated in the requirements specification-for example, when it encounters incorrect input data or some hardware malfunction (say, a disk crash). A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct, though, if the requirements specification does not state what the action should be upon entry of an incorrect command. Obviously, robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify -its "reasonable" behavior completely. Thus, robustness would become equivalent to correctness (or reliability, in the sense of Figure 2.1).

## 3.2 Performance

Any engineering product is expected to meet a certain level of performance. Unlike other disciplines, in software engineering we often equate performance with efficiency. We will follow this practice here. A software system is efficient if it uses computing resources economically.

Performance is important because it affects the usability of the system. If a software system is too slow, it reduces the productivity of the users, possibly to the point of not meeting their needs. If a software system uses too much disk space, it may be too expensive to run. If a software system uses too much memory, it may affect the other applications that are run on the same system, or it may run slowly while the operating system tries to balance the memory usage of the different applications.

Underlying all of these statements-and also what makes the efficiency issue difficult-are the changing limits of efficiency as technology changes. Our view of what is

"too expensive" is constantly changing as advances in technology extend the limits. The computers of today cost orders of magnitude less than computers of a few years ago, yet they provide orders of magnitude more power.

Performance is also important because it affects the scalability of a software system. An algorithm that is quadratic may work on small inputs but not work at all for larger inputs. So, there are several ways to evaluate the performance of a system. One method is to measure efficiency by analyzing the complexity of algorithms. An extensive theory exists for characterizing the average or worst case behavior of algorithms, in terms of significant resource requirements such as time and space, or-less traditionally-in terms of number of message exchanges (in the case of distributed systems).

Analysis of the complexity of algorithms provides only average or worst case information, rather than specific information, about a particular implementation. For more specific information, we can use techniques of performance evaluation. The three basic approaches to evaluating the performance of a system are measurement, analysis, and simulation. We can measure the actual performance of a system by means of monitors that collect data while the system is working and allow us to discover bottlenecks in the system. Or we can build a model of the product and analyze it. Or, finally, we can even build a model that simulates the product. Analytic models-often based on queuing theory-are usually easier to build but are less accurate while simulation models are more costly to build but are more accurate. We can combine the two techniques as follows: at the start of a large project, an analytic model can provide a general understanding of the performance-critical areas of the product, pointing out areas where more thorough study is required; then we can build simulation models of these particular areas.

### 3.3 User Friendliness

A software system is user friendly if its human users find it easy to use. This definition reflects the subjective nature of user friendliness. An application that is used by novice programmers qualifies as user friendly by virtue of different properties than an application that is used by expert programmers. For example, a novice user may appreciate verbose messages, while an experienced user grows to detest and ignore them. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing a command.

The user interface is an important component of user friendliness. A software system that presents the novice user with a window interface and a mouse is friendlier than one that requires the user to use a set of one-letter commands. On the other hand, an experienced user might prefer a set of commands that minimize the number of keystrokes rather than a fancy window interface through which he has to navigate to get to the command that he knew all along he wanted to execute.

There is more to user friendliness, however, than the user interface. For example, an embedded software system does not have a human user interface. Instead, it interacts

with hardware and perhaps other software systems. In this case, the user friendliness is reflected in the ease with which the system can be configured and adapted to the hardware environment.

In general, the user friendliness of a system depends on the consistency of its user and operator interfaces. Clearly, however, the other qualities mentioned above-such as correctness and performance-also affect user friendliness. A software system that produces wrong answers is not friendly, regardless of how fancy its user interface is. Also, a software system that produces answers more slowly than the user requires is not friendly even if the answers are displayed in color.

### 3.4 Verifiability

A software is verifiable if its properties can be verified easily. For example, the correctness or the performances of a software system are properties we would be interested in verifying. Verification can be performed either by formal analysis methods or through testing. A common technique for improving verifiability is the use of "software monitors," that is, code inserted in the software to monitor various qualities such as performance or correctness.

Modular design, disciplined coding practices and the use of an appropriate programming language all contribute to verifiability.

Verifiability is usually an internal quality, although it sometimes becomes an external quality also. For example, in many security critical applications, the customer requires the verifiability of certain properties. The highest level of the security standard for a “trusted computer system” requires the verifiability of the operating system kernel.

### 3.5 Maintainability

The term “software maintenance” is commonly used to refer to the modifications that are made to a software system after its initial release. Maintenance used to be viewed as merely “bug fixing”, and it was distressing to discover that so much effort was spent on fixing defects. Studies have shown, however, that the majority of time spent on maintenance is in fact spent on enhancing the product with features that were not in the original specifications or were stated incorrectly there.

There is evidence that maintenance costs exceed 60% of the total costs of the software. To analyze the factors that affect such costs, it is customary to divide software maintenance into three categories: corrective, adaptive and perfective maintenance. Corrective maintenance has to do with the removal of residual errors present in the product when it is delivered as well as errors introduced into the software during its maintenance. Corrective maintenance accounts for about 20 percent of maintenance costs.

Adaptive and perfective maintenance are the real sources of change in software; they motivate the introduction of evolvability as a fundamental software quality and anticipation of change as a general principle that should guide the software engineer. Adaptive maintenance accounts for nearly another 20 percent of maintenance costs while over 50 percent is absorbed by perfective maintenance.

Adaptive maintenance involves adjusting the application to changes in the environment, e.g., a new release of the hardware or the operating system or a new data-base system. In other words, in adaptive maintenance the need for software changes cannot be attributed to a feature in the software itself, such as the presence of residual errors or the inability to provide some functionality required by the user. Rather, the software must change because the environment in which it is embedded changes.

Finally, perfective maintenance involves changing the software to improve some of its qualities. Here, changes are due to the need to modify the functions offered by the application, add new functions, improve the performance of the application, make it easier to use, etc. The requests to perform perfective maintenance may come directly from the software engineer, in order to improve the status of the product on the market, or they may come from the customer, to meet some new requirements.

### 3.5.1 Repairability

A software system is repairable if it allows the correction of its defects with a limited amount of work. In many engineering products, repairability is a major design goal. For example, automobile engines are built with the parts that are most likely to fail as the most accessible. In computer hardware engineering, there is a subspecialty called repairability, availability, and serviceability (RAS).

An analogous situation applies to software: a software product that consists of well-designed modules is much easier to analyze and repair than a monolithic one. Merely increasing the number of modules, however, does not make a more repairable product. We have to choose the right module structure with the right module interfaces to reduce the need for module interconnections. The right modularization promotes repairability by allowing errors to be confined to few modules, making it easier to locate and remove them. Repairability can be improved through the use of proper tools. For example, using a high-level language rather than an assembly language leads to better repairability. Also, tools such as debuggers can help in isolating and repairing errors.

### 3.5.2 Evolvability

Like other engineering products, software products are modified over time to provide new functions or to change existing functions. Indeed, the fact that software is so malleable makes modifications extremely easy to apply to an implementation. There is, however, a major difference between software modification and modification of other engineering products. In the case of other engineering products, modifications start at the design level and then proceed to the implementation of the product. For example, if one

decides to add a second story to a house, first one must do a feasibility study to check whether this can be done safely. Then one is required to do a design, based on the original design of the house. Then the design must be approved, after assessing that it does not violate the existing regulations. And, finally, the construction of the new part may be commissioned.

In the case of software, unfortunately, people seldom proceed in such an organized fashion. Although the change might be a radical change in the application, too often the implementation is started without doing any feasibility study, let alone a change in the original design. Still worse, after the change is accomplished, the modification is not even documented a posteriori; i.e., the specifications are not updated to reflect the change. This makes future changes more and more difficult to apply.

On the other hand, successful software products are quite long lived. Their first release is the beginning of a long lifetime and each successive release is the next step in the evolution of the system. If the software is designed with care and if each modification is thought out carefully, then it can evolve gracefully.

As the cost of software production and the complexity of applications grow, the evolvability of software assumes more and more importance. One reason for this is the need to leverage the investment made in the software as the hardware technology advances. Some of the earliest large systems developed in the 1960s are today taking advantage of new hardware, device and network technologies.

Most software systems start out being evolvable, but after years of evolution they reach a state where any major modification runs the risk of "breaking" existing features. In fact, evolvability is achieved by modularization and successive changes tend to reduce the modularity of the original system. This is even worse if modifications are applied without careful study of the original design and without precise description of changes in both the design and the requirements specification.

Indeed, studies of large software systems show that evolvability decreases with each release of a software product. Each release complicates the structure of the software, so that future modifications become more difficult. To overcome this problem, the initial design of the product, as well as any succeeding changes, must be done with evolvability in mind. Evolvability is one of the most important software qualities, and the principles. Evolvability is both a product- and process-related quality. In terms of the latter, the process must be able to accommodate new management and organizational techniques. changes in engineering education, etc.

### 3.6 Reusability

Reusability is an important factor. In product evolution, we modify a product to build a new version of that same product; in. product reuse, we use it-perhaps with minor changes-to build another product. Reusability appears to be more applicable to software

components than to whole products but it certainly seems possible to build products that are reusable.

A good example of a reusable product is the UNIX shell. The UNIX shell is a command language interpreter; that is, it accepts user commands and executes them. But it is designed to be used both interactively and in "batch." The ability to start a new shell with a file containing a list of shell commands allows us to write programs-scripts-in the shell command language. We can view the program as a new product that uses the shell as a component. By encouraging standard interfaces, the UNIX environment in fact supports the reuse of any of its commands, as well as the shell, in building powerful utilities.

Scientific libraries are the best known reusable components. Several large FORTRAN libraries have existed for many years. Users can buy these and use them to build their own products, without having to reinvent or recode well-known algorithms. Indeed, several companies are devoted to producing just suchlibraries.

### 3.7 Portability

Software is portable if it can run in different environments. The term "environment" can refer to a hardware platform or a software environment such as a particular operating system. With the proliferation of different processors and operating systems, portability has become an important issue for software engineers.

Even within one processor family, portability can be important because of the variations in memory capacity and additional instructions. One way to achieve portability within one machine architecture is to have the software system assume a minimum configuration as far as memory capacity is concerned and use a subset of the machine facilities that are guaranteed to be available on all models of the architecture (such as machine instructions and operating system facilities). But this penalizes the larger models because, presumably, the system can perform better on these models if it does not make such restrictive assumptions. Accordingly, we need to use techniques that allow the software to determine the capabilities of the hardware and to adapt to them. One good example of this approach is the way that UNIX allows programs to interact with many different terminals without explicit assumptions in the programs about the terminals they are using. The X Windows system extends this capability to allow applications to run on any bit-mapped display.

More generally, portability refers to the ability to run a system on different hardware platforms. As the ratio of money spent on software versus hardware increases, portability gains more importance.

### 3.8 Understandability

Some software systems are easier to understand than others. Of course, some tasks are inherently more complex than others. For example, a system that does weather forecasting, no matter how well it is written, will be harder to understand than one that prints a mailing list. Given tasks of inherently similar difficulty, we can follow certain

guidelines to produce more understandable designs and to write more understandable programs.

Understandability is an internal product quality, and it helps in achieving many of the other qualities such as evolvability and verifiability. From an external point of view the user considers the system understandable if it has predictable behavior. External understandability is a component of user friendliness.

### 3.9 Interoperability

Interoperability refers to the ability of a system to coexist and cooperate with other systems-for example, a word-processor's ability to incorporate a chart produced by a graphing package or the graphics package's ability to graph the data produced by a spreadsheet, or the spreadsheet's ability to process an image scanned by a scanner. While rare in software products, interoperability abounds in other engineering products. For example, stereo systems from various manufacturers work together and can be connected to television sets and recorders. In fact, stereo systems produced decades ago accommodate new technologies such as compact discs, while virtually every operating system has to be modified-sometimes significantly before it can work with the new optical disks.

Once again, the UNIX environment, with its standard interfaces, offers a limited example of interoperability within a single environment: UNIX encourages applications to have a simple, standard interface, which allows the output of one application to be used as the input to another.

### 3.10 Productivity

Productivity is a quality of the software production process: it measures the efficiency of the process and as we said before, is the performance quality applied to the process. An efficient process results in faster delivery of the product.

Individual engineers produce software at a certain rate, although there are great variations among individuals of different ability. When individuals are part of a team, the productivity of the team is some function of the productivity of the individuals. Very often, the combined productivity is much less than the sum of the parts. Process organizations and techniques are attempts at capitalizing on the individual productivity of team members.

Productivity offers many trade-offs in the choice of a process. For example, a process that requires specialization of individual team members may lead to productivity in producing a certain product, but not in producing a variety of products. Software reuse is a technique that leads to the overall productivity of an organization that is involved in developing many products, but developing reusable modules is harder than developing modules for one's own use. thus reducing the productivity of the group that is developing reusable modules as part of their product development.

### 3.11 Timeliness

Timeliness is a process-related quality that refers to the ability to deliver a product on time. Historically, timeliness has been lacking in software production processes leading to the "software crisis," which in turn led to the need for-and birth of software engineering itself. Even now, many current processes fail to result in a timely product.

Timeliness by itself is not a useful quality, although being late may preclude market opportunities. Delivering on time a product that is lacking in other qualities, such as reliability or performance is pointless.

Timeliness requires careful scheduling, accurate work estimation and clearly specified and verifiable milestones. All other engineering disciplines use standard project management techniques to achieve timeliness. There are even many computer supported project management tools.

Standard project management techniques are difficult to apply in software engineering because of the difficulty in measuring the amount of work required for producing a given piece of software, the difficulty in measuring the productivity of engineers or even having a dependable metric for productivity and the use of imprecise and unverifiable milestones. Another reason for the difficulty in achieving timeliness in the software process is continuously changing user requirements.

One technique for achieving timeliness is through incremental delivery of the product. Incremental delivery allows the product to become available earlier; and the use of the product helps in refining the requirements incrementally. Outside of software engineering, a classic example of the difficulty in dealing with the requirements of complex systems is offered by modern weapons systems. In several well-publicized cases, the weapons have been obsolete by the time they have been delivered, or they have not met the requirements or in many cases, both. But after ten years of development, it is difficult to decide what to do with a product that does not meet a requirement stated ten years ago. The problem is exacerbated by the fact that requirements cannot be formulated precisely in these cases because the need is for the most advanced system possible at the time of delivery, not at the time the requirements are defined.

Obviously, incremental delivery depends on the ability to break down the set of required system functions into subsets that can be delivered in increments. If such subsets cannot be defined, no process can make the product available incrementally. But a non-incremental process prevents the production of product subsets even if such subsets can be identified. Timeliness can be achieved by a product that can be broken down into subsets and an incremental process.

### 3.12 Visibility

A software development process is visible if all of its steps and its current status are documented clearly. The idea is that the steps and the status of the project are available

and easily accessible for external examination. In many software projects, most engineers and even managers are unaware of the exact status of the project. Some may be designing, others coding and still others testing, all at the same time. This, by itself, is not bad. Yet, if an engineer starts to redesign a major part of the code just before the software is supposed to be delivered for integration testing, the risk of serious problems and delays will be high.

Visibility allows engineers to weigh the impact of their actions and thus guides them in making decisions. It allows the members of the team to work in the same direction, rather than, as are often the case currently, in cross directions. The most common example of the latter situation is as mentioned above, when the integration test group has been testing a version of the software assuming that the next version will involve fixing defects and will be only minimally different from the current version, while an engineer decides to do a major redesign to correct a minor defect. The tension between one group trying to stabilize the software while another person or group is destabilizing lt-unintentionally, of course-is common. The process must encourage a consistent view of the status and current goals among all participants.

Visibility is not only an internal quality; it is also external. During the course of a long project, there are many requests about the status of the project. Sometimes these require formal presentations on the status and at other times the requests are informal. Sometimes the requests come from the organization's management for future planning, and at other times they come from the outside, perhaps from the customer. If the software development process has low visibility, either these status reports will not be accurate, or they will require a lot of effort to prepare each time.

So, visibility of the process requires not only that all process steps be documented, but also that the current status of the intermediate products, such as requirements specifications and design specifications, be maintained accurately; that is, visibility of the product is required also. Intuitively, a product is visible if it is clearly structured as a collection of modules, with clearly understandable functions and easily accessible documentation.

## 4.0 Summary

In this lesson we have studied the software life cycle, which is a very important concept in the field of software engineering. We have also studied the classification of software qualities and the representative qualities of the software.

## 5.0 Self Check Exercise

Q1: Explain in detail the software life cycle.

Q2: What are the various classifications of Software Qualities?

Q3: What are the various representative qualities of the software?

## 6.0 Suggested readings

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

## Software Process Models

**Objectives**



**Objectives:**

In this lesson we will study the various process models such as Waterfall model, Spiral model, Prototyping model etc.

### 1.0 Introduction: SDLC Models

There are various software development approaches defined and designed which are used/employed during development process of software, these approaches are also referred as "**Software Development Process Models**". A software process model is a simplified represenation of a software process

Each process model follows a particular life cycle in order to ensure success in process of software development.

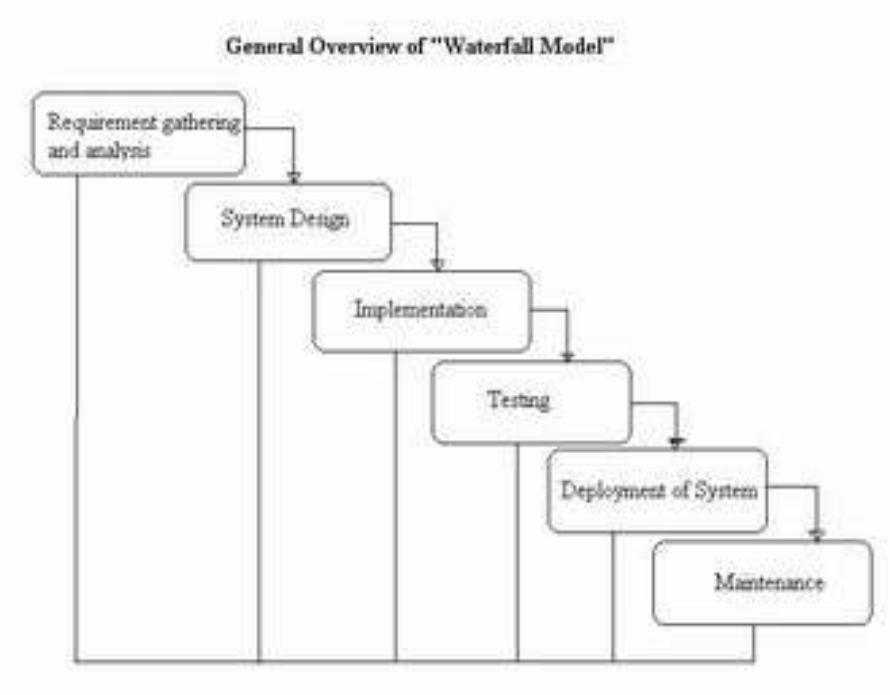
Some of the most commonly used Software developments models are the following:

- **Waterfall model**
- **Spiral model**
- **Prototyping model**
- Iterative model
- RAD model

### 2.0 Waterfall Model

**Waterfall approach was first Process Model to be introduced and followed widely in Software Engineering to ensure success of the project**. In "The Waterfall" approach, the whole process of software development is divided into separate process phases. The phases in Waterfall model are: Requirement Specifications phase, Software Design, Implementation and Testing & Maintenance. All these phases are cascaded to each other so that second phase is started as and when defined set of goals are achieved for first phase and it is signed off, so the name "Waterfall Model". All the methods and

processes undertaken in Waterfall Model are more visible. It refers to a software development model in which phases are performed sequentially with little or no overlap.




The stages of "The Waterfall Model" are:

**Requirement Analysis & Definition**: All possible requirements of the system to be developed are captured in this phase. Requirements are set of functionalities and constraints that the end-user (who will be using the system) expects from the system. The requirements are gathered from the end-user by consultation, these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied. Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model.

**System & Software Design**: Before a starting for actual coding, it is highly important to understand what we are going to create and what it should look like? The requirement specifications from first phase are studied in this phase and system design is prepared. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

**Implementation & Unit Testing**: On receiving system design documents, the work is divided in modules/units and actual coding is started. The system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality; this is referred to as Unit Testing. Unit testing mainly verifies if the modules/units meet their specifications.

**Integration & System Testing**: As specified above, the system is first divided in units which are developed and tested for their functionalities. These units are integrated into a

complete system during Integration phase and tested to check if all modules/units coordinate between each other and the system as a whole behaves as per the specifications. After successfully testing the software, it is delivered to the customer.

**Operations & Maintenance**: This phase of "The Waterfall Model" is virtually never ending phase (Very long). Generally, problems with the system developed (which are not found during the development life cycle) come up after its practical use starts, so the issues related to the system are solved after deployment of the system. Not all the problems come in picture directly but they arise time to time and needs to be solved; hence this process is referred as Maintenance.

**Advantages of waterfall model**

The advantage of waterfall development is that it allows for departmentalization and managerial control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process like a car in a carwash, and theoretically, be delivered on time. Development moves from concept, through design, implementation, testing, installation, troubleshooting and ends up at operation and maintenance. Each phase of development proceeds in strict order, without any overlapping or iterative steps.

**Disadvantages of waterfall model**

The disadvantage of waterfall development is that it does not allow for much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage. Alternatives to the waterfall model include joint application development (JAD), rapid application development (RAD), synch and stabilize, build and fix and the spiral model.

**Waterfall Model Common Error**

**Common Errors in Requirements Analysis**

In the traditional waterfall model of software development, the first phase of requirements analysis is also the most important one. This is the phase which involves gathering information about the customer's needs and defining, in the clearest possible terms, the problem that the product is expected to solve. This analysis includes understanding the customer's business context and constraints, the functions the product must perform, the performance levels it must adhere to and the external systems it must be compatible with. Techniques used to obtain this understanding include customer interviews, use cases, and "shopping lists" of software features. The results of the analysis are typically captured in a formal requirements specification, which serves as input to the next step. Well, at least that's the way it's supposed to work theoretically. In reality, there are a number of problems with this theoretical model and these can cause delays and knock-on errors in the rest of the process. This part discusses some of the more common problems that project managers

Experience during this phase and suggests possible solutions.

**Problem 1: Customers don't (really) know what they want**

Possibly the most common problem in the requirements analysis phase is that customers have only a vague idea of what they need and it's up to you to ask the right questions and perform the analysis necessary to turn this amorphous vision into a formally-documented software requirements specification that can, in turn, be used as the basis for both a project plan and an engineering architecture.

To solve this problem, you should:

- Ensure that you spend sufficient time at the start of the project on understanding the objectives, deliverables and scope of the project.
- Make visible any assumptions that the customer is using and critically evaluate both the likely end-user benefits and risks of the project.
- Attempt to write a concrete vision statement for the project, which encompasses both the specific functions or user benefits it provides and the overall business problem it is expected to solve.
- Get your customer to read, think about and sign off on the completed software requirements specification, to align expectations and ensure that both parties have a clear understanding of the deliverable.

**Problem 2: Requirements change during the course of the project**

The second most common problem with software projects is that the requirements defined in the first phase change as the project progresses. This may occur because as development progresses and prototypes are developed, customers are able to more clearly see problems with the original plan and make necessary course corrections; it may also occur because changes in the external environment require reshaping of the original business problem and hence necessitates a different solution than the one originally proposed.

Good project managers are aware of these possibilities and typically already have backup plans in place to deal with these changes.

To solve this problem, you should:

- Have a clearly defined process for receiving, analyzing and incorporating change requests, and make your customer aware of his/her entry point into this process.
- Set milestones for each development phase beyond which certain changes are not permissible -- for example, disallowing major changes once a module reaches 75 percent completion.
- Ensure that change requests (and approvals) are clearly communicated to all stakeholders, together with their rationale and that the master project plan is updated accordingly.

**Problem 3: Customers have unreasonable timelines**

It's quite common to hear a customer say something like "it's an emergency job and we need this project completed in X weeks". A common mistake is to agree to such timelines before actually performing a detailed analysis and understanding both of the scope of the project and the resources necessary to execute it. In accepting an unreasonable timeline without discussion, you are, in fact, doing your customer a

disservice: it's quite likely that the project will either get delayed (because it wasn't possible to execute it in time) or suffer from quality defects (because it was rushed through without proper inspection).
To solve this problem, you should:

- Convert the software requirements specification into a project plan, detailing tasks and resources needed at each stage and modeling best-case, middle-case and worst-case scenarios.
- Ensure that the project plan takes account of available resource constraints and keeps sufficient time for testing and quality inspection.
- Enter into a conversation about deadlines with your customer, using the figures in your draft plan as supporting evidence for your statements. Assuming that your plan is reasonable, it's quite likely that the ensuing negotiation will be both productive and result in a favorable outcome for both parties.

**Problem 4: Communication gaps exist between customers, engineers and project managers**

Often, customers and engineers fail to communicate clearly with each other because they come from different worlds and do not understand technical terms in the same way. This can lead to confusion and severe miscommunication and an important task of a project manager, especially during the requirements analysis phase, is to ensure that both parties have a precise understanding of the deliverable and the tasks needed to achieve it.
To solve this problem, you should:

- Take notes at every meeting and disseminate these throughout the project team.
- Be consistent in your use of words. Make yourself a glossary of the terms that you're going to use right at the start, ensure all stakeholders have a copy and stick to them consistently.

**Problem 5: The development team doesn't understand the politics of the customer's organization**

The scholars Bolman and Deal suggest that an effective manager is one who views the organization as a "contested arena" and understands the importance of power, conflict, negotiation and coalitions. Such a manager is not only skilled at operational and functional tasks, but he or she also understands the importance of framing agendas for common purposes, building coalitions that are united in their perspective, and persuading resistant managers of the validity of a particular position.
These skills are critical when dealing with large projects in large organizations, as information is often fragmented and requirements analysis is hence stymied by problems of trust, internal conflicts of interest and information inefficiencies.
To solve this problem, you should:

- Review your existing network and identify both the information you need and who is likely to have it.
- Cultivate allies, build relationships and think systematically about your social capital in the organization.

- Persuade opponents within your customer's organization by framing issues in a way that is relevant to their own experience.
- Use initial points of access/leverage to move your agenda forward.

## 3.0 Spiral Model

**The spiral model was defined by Barry Boehm in his 1988 article A Spiral Model of Software Development and Enhancement**. This model was not the first model to discuss iterative development, but it was the first model to explain why the iteration matters. As originally envisioned, the iterations were typically 6 months to 2 years long. Each phase starts with a design goal and ends with the client (who may be internal) reviewing the progress thus far. Analysis and engineering efforts are applied at each phase of the project, with an eye toward the end goal of the project.

### The Spiral Model

The spiral model, also known as the spiral lifecycle model, is a systems development method (SDM) used in information technology (IT). This model of development combines the features of the prototyping model and the waterfall model. The spiral model is intended for large, expensive and complicated projects. **The steps in the spiral model can be generalized as follows:**

1. The new system requirements are defined in as much detail as possible. This usually involves interviewing a number of users representing all the external or internal users and other aspects of the existing system.
2. A preliminary design is created for the new system.
3. A first prototype of the new system is constructed from the preliminary design. This is usually a scaled-down system and represents an approximation of the characteristics of the final product.
4. A second prototype is evolved by a fourfold procedure: (1) evaluating the first prototype in terms of its strengths, weaknesses, and risks; (2) defining the requirements of the second prototype; (3) planning and designing the second prototype; (4) constructing and testing the second prototype.
5. At the customer's option, the entire project can be aborted if the risk is deemed too great. Risk factors might involve development cost overruns, operating-cost miscalculation or any other factor that could, in the customer's judgment, result in a less-than-satisfactory final product.
6. The existing prototype is evaluated in the same manner as was the previous prototype, and if necessary, another prototype is developed from it according to the fourfold procedure outlined above.
7. The preceding steps are iterated until the customer is satisfied that the refined prototype represents the final product desired.
8. The final system is constructed, based on the refined prototype.
9. The final system is thoroughly evaluated and tested. Routine maintenance is carried out on a continuing basis to prevent large-scale failures and to minimize downtime.

**Applications**

For a typical shrink-wrap application, the spiral model might mean that you have a rough-cut of user elements (without the polished / pretty graphics) as an operable application, add features in phases, and, at some point, add the final graphics. The spiral model is used most often in large projects. For smaller projects, the concept of agile software development is becoming a viable alternative. The US military has adopted the spiral model for its Future Combat Systems program.

**Advantages of Spiral model**

1. Estimates (i.e. budget, schedule, etc.) become more realistic as work progresses, because important issues are discovered earlier.
2. It is more able to cope with the (nearly inevitable) changes that software development generally entails.
3. Software engineers (who can get restless with protracted design processes) can get their hands in and start working on a project earlier.

**Disadvantages of Spiral model**

1. Highly customized limiting re-usability
2. Applied differently for each application
3. Risk of not meeting budget or schedule

## 4.0 Prototype Model

**A prototype is a working model that is functionally equivalent to a component of the product**. In many instances the client only has a general view of what is expected from the software product. In such a scenario where there is an absence of detailed information regarding the input to the system, the processing needs and the output requirements, the prototyping model may be employed.

This model reflects an attempt to increase the flexibility of the development process by allowing the client to interact and experiment with a working representation of the product. The developmental process only continues once the client is satisfied with the functioning of the prototype. At that stage the developer determines the specifications of the client's real needs.

The process of prototyping involves the following steps:

1. **Identify basic requirements**
   Determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.

2. **Develop Initial Prototype**
   The initial prototype is developed that includes only user interfaces.

3. **Review**
   The customers, including end-users, examine the prototype and provide feedback on additions or changes.

4. **Revise and Enhancing the Prototype**

Using the feedback both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps #3 ands #4 may be needed.

**Advantages of Prototyping**

There are many advantages to using prototyping in software development, some tangible some abstract.

**Reduced time and costs**: Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development, the early determination of what the user really wants can result in faster and less expensive software.

**Improved and increased user involvement**: Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications. The presence of the prototype being examined by the user prevents many misunderstandings and miscommunications that occur when each side believe the other understands what they said. Since users know the problem domain better than anyone on the development team does, increased interaction can result in final product that has greater tangible and intangible quality. The final product is more likely to satisfy the users desire for look, feel and performance.

**Disadvantages of Prototyping**

Using, or perhaps misusing, prototyping can also have disadvantages.

**Insufficient analysis**: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.

**User confusion of prototype and finished system**: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to feature creep.

**Developer attachment to prototype:** Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture. (This may suggest that throwaway prototyping, rather than evolutionary prototyping, should be used.)

**Excessive development time of the prototype**: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.

**Expense of implementing prototyping**: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

A common problem with adopting prototyping technology is high expectations for productivity with insufficient effort behind the learning curve. In addition to training for the use of a prototyping technique, there is an often overlooked need for developing corporate and project specific underlying structure to support the technology. When this underlying structure is omitted, lower productivity can often result.

**Best projects to use Prototyping**

It has been argued that prototyping, in some form or another, should be used all the time. However, prototyping is most beneficial in systems that will have many interactions with the users.

It has been found that prototyping is very effective in the analysis and design of on-line systems, especially for transaction processing, where the use of screen dialogs is much more in evidence. The greater the interaction between the computer and the user, the greater the benefit is that can be obtained from building a quick system and letting the user play with it.

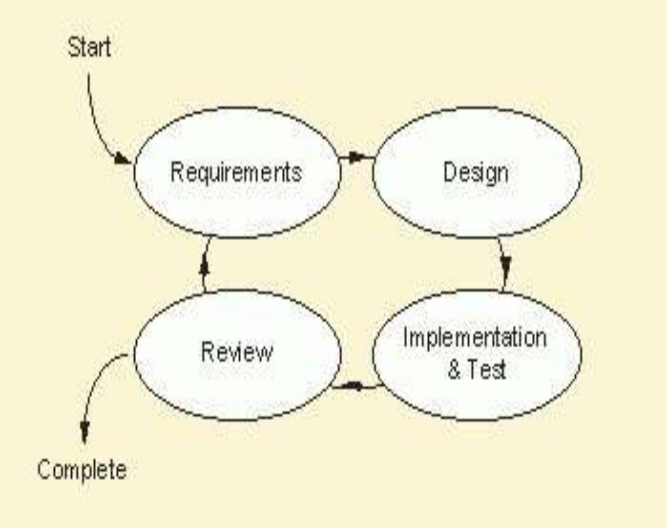
Systems with little user interaction, such as batch processing or systems that mostly do calculations, benefit little from prototyping. Sometimes, the coding needed to perform the system functions may be too intensive and the potential gains that prototyping could provide are too small.

Prototyping is especially good for designing good human-computer interfaces. "One of the most productive uses of rapid prototyping to date has been as a tool for iterative user requirements engineering and human-computer interface design."

**Some other process models:**

**Iterative Model**

An iterative lifecycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model. Consider an iterative lifecycle model which consists of repeating the following four phases in sequence:




A ***Requirements*** phase, in which the requirements for the software are gathered and analyzed. Iteration should eventually result in a requirements phase that produces a complete and final specification of requirements.

A ***Design*** phase, in which a software solution to meet the requirements is designed. This may be a new design or an extension of an earlier design. An ***Implementation and Test*** phase, when the software is coded, Integrated and tested.
A ***Review*** phase, in which the software is evaluated, the current requirements are reviewed and changes and additions to requirements proposed. For each cycle of the model, a decision has to be made as to whether the software produced by the cycle will be discarded, or kept as a starting point for the next cycle (sometimes referred to as incremental prototyping). Eventually a point will be reached where the requirements are complete and the software can be delivered, or it becomes impossible to enhance the software as required, and a fresh start has to be made. The iterative lifecycle model can be likened to producing software by successive approximation. Drawing an analogy with mathematical methods that use successive approximation to arrive at a final solution, the benefit of such methods depends on how rapidly they converge on a solution.

The key to successful use of an iterative software development lifecycle is rigorous validation of requirements and verification (including testing) of each version of the software against those requirements within each cycle of the model. The first three phases of the example iterative model is in fact an abbreviated form of a sequential V or waterfall lifecycle model. Each cycle of the model produces software that requires testing at the unit level, for software integration, for system integration and for acceptance. As the software evolves through successive cycles, tests have to be repeated and extended to verify each version of the software.

**RAD Model**

**Introduction**

**RAD is a linear sequential software development process model that emphasis an extremely short development cycle using a component based construction**

**approach**. If the requirements are well understood and defines and the project scope is constraint, the RAD process enables a development team to create a fully functional system with in very short time period.

**What is RAD?**

RAD (rapid application development) is a concept that products can be developed faster and of higher quality through:

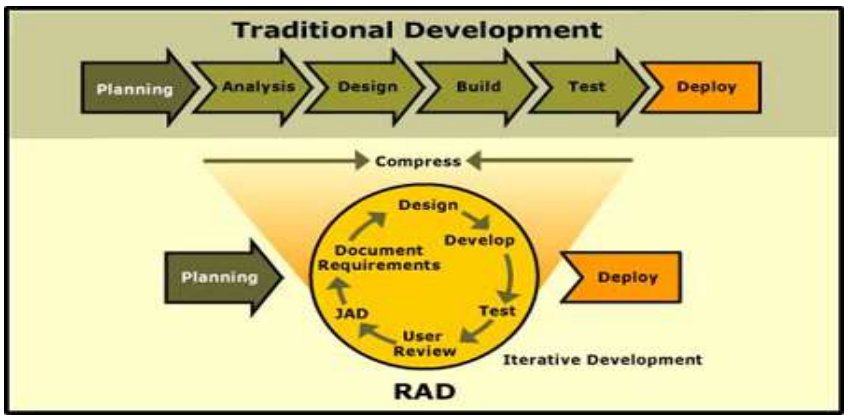
- Gathering requirements using workshops or focus groups
- Prototyping and early, reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version
- Less formality in reviews and other team communication

Some companies offer products that provide some or all of the tools for RAD software development. (The concept can be applied to hardware development as well.) These products include requirements gathering tools, prototyping tools, computer-aided software engineering tools, language development environments such as those for the Java platform, groupware for communication among development members, and testing tools. RAD usually embraces object-oriented programming methodology, which inherently fosters software re-use. The most popular object-oriented programming languages, C++ and Java are offered in visual programming packages often described as providing rapid application development.

**Development Methodology**

The traditional software development cycle follows a rigid sequence of steps with a formal sign-off at the completion of each. A complete, detailed requirements analysis is done that attempts to capture the system requirements in a Requirements Specification. Users are forced to "sign-off" on the specification before development proceeds to the next step. This is followed by a complete system design and then development and testing.

But, what if the design phase uncovers requirements that are technically unfeasible, or extremely expensive to implement? What if errors in the design are encountered during the build phase? The elapsed time between the initial analysis and testing is usually a period of several months. What if business requirements or priorities change or the users realize they overlooked critical needs during the analysis phase? These are many of the reasons why software development projects either fail or don't meet the user's expectations when delivered. RAD is a methodology for compressing the analysis, design, build and test phases into a series of short, iterative development cycles. This has a number of distinct advantages over the traditional sequential development model.




RAD projects are typically staffed with small integrated teams comprised of developers, end users and IT technical resources. Small teams, combined with short, iterative development cycles optimizes speed, unity of vision and purpose, effective informal communication and simple project management.

**RAD Model Phases**

RAD model has the following phases:

1. **Business Modeling:**
   The information flow among business functions is defined by answering questions like what information drives the business process, what information is generated, who generates it, where does the information go, who process it and so on.
2. **Data Modeling:**
   The information collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified and the relation between these data objects (entities) is defined.
3. **Process Modeling:**
   The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.
4. **Application Generation:**
   Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.
5. **Testing and Turn over:**
   Many of the programming components have already been tested since RAD emphasis reuse. This reduces overall testing time. But new components must be tested and all interfaces must be fully exercised.

**Advantages and Disadvantages**

RAD reduces the development time and reusability of components help to speed up development. All functions are modularized so it is easy to work with. For large projects RAD require highly skilled engineers in the team. Both end customer and developer should be committed to complete the system in a much abbreviated time frame. If commitment is

lacking RAD will fail. RAD is based on Object Oriented approach and if it is difficult to modularize the project the RAD may not work well.

## 5.0 Tools and Techniques for process modeling

There are many choices for modeling tools and techniques, once you decide what you want to capture in your process model. The appropriate technique for you depends on your goals and your preferred work style. In particular, your choice for notation depends on what you want to capture in your model. The notations range from textual ones that express processes as functions to graphical ones that depict processes as hierarchies of boxes and arrows to combination of pictures and text that link the graphical depiction to tables and functions elaborating on the high level illustration. Many of the modeling notations can also be used for representing requirements and designs.

In this section the notation is secondary to the type of model and we focus on two major categories, static and dynamic. A static model depicts the process showing that the inputs are transformed to outputs. A dynamic model enacts the process so the user can see how intermediate and final products are transformed over time.

### Static modeling

There are many ways to model a process statistically. In the early 1990s, Lai (1991) develop a comprehensive process notation that is intended to enable someone to model any process at any level of detail. It builds on a paradigm where people perform roles while resources perform activities leading to the production of artifacts. The process model shows the relationship among the roles, activities and artifacts and state tables show information about the completeness of each artifacts at a given time.

In particular, the elements of a process are viewed in terms of seven types:

1. **Activity:** Something that will happen in a process. The element can be related to what happens before and after, what resources are needed, what triggers the activity's start, what rules govern the activity, how to describe the algorithms and lessons learned and how to relate the activity to the project team.
2. **Sequence:** The order of activities. The sequence can be described using triggers, programming constructs, transformations, ordering or satisfaction of conditions.
3. **Process model:** A view of interest about the system. Thus part of the process may be represented as a separate model, either to predict process behavior or to examine certain characteristics.
4. **Resources:** A necessary item a tool or a person. Resources can include equipment, time, office space, people and techniques and so on. The process model identifies how much of each resource is needed for each activity.
5. **Control:** An external influence over process enactment. The controls may be manual or automatic, human or mechanical.
6. **Policy:** A guiding principle. The high level process constraint influences process enactment. It may include a prescribed development process, a tool that must be used or a mandatory management style.
7. **Organization:** The hierarchical structure of process agents, with physical grouping corresponding to logical grouping and related roles. The mapping from physical to

logical grouping should be flexible enough to reflect changes in physical environment.

The process itself has several levels of abstraction, including the software development process that directs certain resources to be used in constructing specific modules as well as generic models that may resemble the spiral or waterfall models, Lai's notation includes several templates such as an Artifact Definition Template, which records information about particular artifacts.

Lai's approach can be applied to modeling software development processes; later in this section, we use it to model the risk involved in development. However to demonstrate its use and its ability to capture many facets of a complex activity, we apply it to a relatively simple but familiar process, driving an automobile. Table contains a description of the key resource in this process, a car.

Other templates define relations, process states, operations, analysis, actions and roles. Graphical diagrams represent the relationship between elements capturing the main relationships and secondary ones. For example figure given below illustrates the process of starting a car. The "initiate" box represents the entrance conditions and the "park" represents an exit condition. The left hand column of a condition box lists artifacts and the right hand column is artifact state.

TABLE Artifact Definition Form for Artifact "CAR" (Lai 1991)

| Name | *Car* | |
|---|---|---|
| Synopsis | *This is the artifact that represents a class of cars.* | |
| Complexity type | *Composite* | |
| Data type | *(car c, user-defined)* | |
| Artifact-state list | | |
| *parked* | ((*state_of*(*car.engine*) = *off*)<br>(*state_of*(*car.gear*) = *park*)<br>(*state_of*(*car.speed*) = *stand*)) | *Car is not moving, and engine is not running.* |
| *initiated* | ((*state_of*(*car.engine*) = *on*)<br>(*state_of*(*car.key_hole*) = *has-key*)<br>(*state_of*(*car-driver*(*car.*)) = *in-car*)<br>*state_of*(*car.gear*) = *drive*)<br>(*state_of*(*car.speed*) = *stand*)) | *Car is not moving, but the engine is running.* |
| *moving* | ((*state_of*(*car.engine*) = *on*)<br>(*state_of*(*car.keyhole*) = *has-key*)<br>(*state_of*(*car-driver*(*car.*)) = *driving*)<br>((*state_of*(*car.gear*) = *drive*) *or*<br>(*state_of*(*car.gear*) = *reverse*))<br>((*state_of*(*car.speed*) = *stand*) *or*<br>(*state_of*(*car.speed*) = *slow*)<br>*or* (*state_of*(*car.speed*) = *medium*) *or*<br>(*state_of*(*car.speed*) = *high*)) | *Car is moving forward or backward.* |
| Subartifact list | | |
| | *doors* | *The four doors of a car* |
| | *engine* | *The engine of a car* |
| | *keyhole* | *The ignition keyhole of a car* |
| | *gear* | *The gear of a car* |
| | *speed* | *The speed of a car* |
| Relations list | | |
| *car-key* | *This is the relation between a car and a key.* | |
| *car-driver* | *This is the relation between a car and a driver.* | |

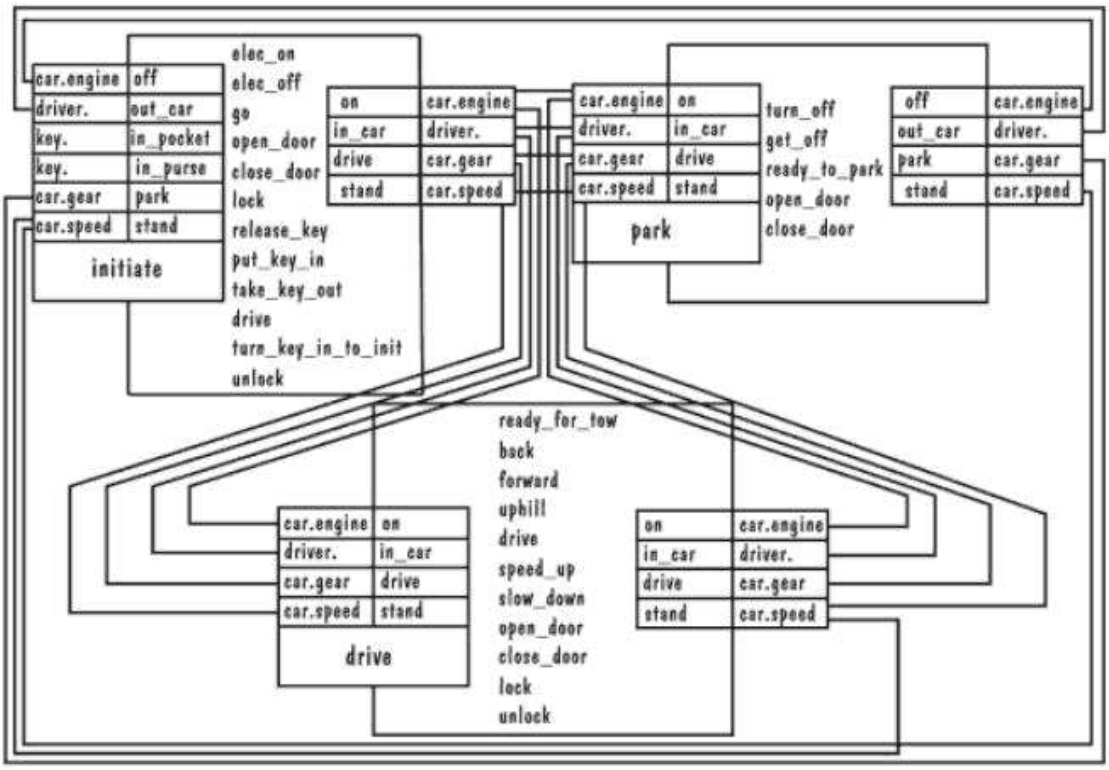
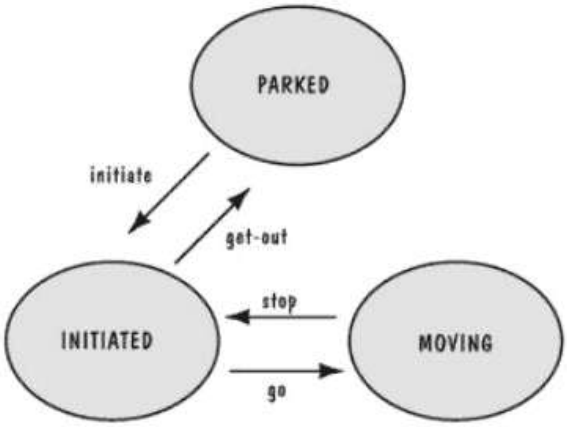



FIGURE The process of starting a car (Lai 1991).

Transition diagram supplement the process model by showing how the states are related to one another. For example, figure illustrates the transitions for a car.

Lai's notation is a good example of how multiple structures and strategies can be used to capture a great deal of information about the software development process but it is also useful in organizing and depicting process information about user requirements as the car example demonstrates.

FIGURE Transition diagram for a car (Lai 1991).



## 6.0 Summary:

In this lesson we have studied the various process models which are used during the development process of software, such as Waterfall model, Spiral model, Prototyping model etc. We have also studied the tools and techniques for process modeling

## 7.0 Self Check Exercise:

Q: Write detailed notes on the following:

- Waterfall model
- Spiral model
- Prototyping model

## 8.0 Suggested readings

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

# Project Planning and Organization

**Objectives**

**1. Introduction**

**2. Project planning**

2.1 Software productivity

2.2 People and productivity

2.3 Cost estimation

**3. Project Control**

3.1 Work breakdown structure

3.2 Gantt charts

3.3 PERT charts

**4. Organization**

4.1 Centralized control team organization

4.2 Decentralized control team organization

4.3 Mixed control team organization

4.4 An assessment of team organization

**5. Summary**

**6. Self Check Exercise**

**7. Suggested Readings**

**Objectives:**

In this lesson we will discuss about the project planning activities like estimation of cost, people and productivity using various models. We will also study various structures for controlling of project.

## 1. Introduction: Management Functions

Management can be defined informally as the art of getting work done through other people. But a more classic definition that allows a systematic study of the subject is the following from Koontz:

> ***The creation and maintenance of an internal environment in an enterprise where individuals, working together in groups, can perform efficiently and effectively toward the attainment of group goals.***

Thus, the main job of management is to enable a group of people to work towards a common goal. Management is not an exact science and the many activities involved in it may be classified according to many different schemes. It is standard, however, to consider management as consisting of the following five interrelated activities, the goal of which is to achieve effective group work:

- **Planning.** A manager must decide what objectives are to be achieved, what resources are required to achieve the objectives, how and when the resources are to be acquired and how the goals are to be achieved. **Planning basically involves determining the flow of information, people and products within the organization.**
- **Organizing. Organizing involves the establishment of clear lines of authority and responsibility for groups of activities that achieve the goals of the enterprise.** Organizing is necessary at the level of a small group, such as a five-person team of software engineers, all the way up to a large corporation composed of several independent divisions. The choice of the organizational structure affects the efficiency of the enterprise. Clearly, the best organizational structure can be devised only when the goals of the enterprise are clear, and this depends on effective planning.
- **Staffing. Staffing deals with hiring personnel for the positions that are identified by the organizational structure**. It involves defining requirements for personnel; recruiting (identifying, interviewing, and selecting candidates); compensating, developing, and promoting employees.
- **Directing.** Directing involves leading subordinates. **The goal of directing is to guide the subordinates to understand and identify with the organizational structure and the goals of the enterprise.** This understanding must be continuously refined by effective and exemplary leadership. Setting examples is especially important in software engineering, where measures of good engineering are lacking. The best training for a beginning engineer is to work alongside a good engineer.
- **Controlling. Controlling consists of measuring and correcting activities to ensure that goals are achieved**. Controlling requires the measurement of performance against plans and taking corrective action when deviations occur.

These general functions of management apply to any management activity, whether in a software engineering organization, an automobile manufacturing plant, or a boy scout group.

## 2. Project planning

The first component of software engineering project management is effective planning of the development of the software. The first step in planning a project is to define and document the assumptions, goals and constraints of the project. A project needs a clear statement of goals to guide the engineers in their daily decision making. Many engineers on typical projects spend many hours discussing alternatives that are known clearly to the project manager, but have not been documented or disseminated properly. The goal of the project planning stage is to identify all the external requirements for and constraints on the project.

Once the external constraints have been identified, the project manager must come up with a plan for how best to meet the requirements within the given constraints. One of the questions at this stage is what process model will serve the project best. Another critical decision is to determine the resources required for the project. The resources include the number and skill level of the people, the amount of computing resources (e.g. workstations, personal computers, and data-base software). Unlike traditional engineering disciplines, where one has to budget for material, the "raw material" used in software is mainly the engineer's brain power. Thus, the cost of a software project is directly proportional to the number of engineers needed for the project. The problem of predicting how many engineers and other resources are needed for a given software project is known as ***software cost estimation.***

Forecasting how many engineers will be needed is a difficult problem that is intimately tied to the problem of how to estimate the productivity of software engineers. There are two parts to the forecasting problem: estimating the difficulty of the task and estimating how much of the task each engineer can solve. Clearly, to estimate the difficulty of the task, one must know what the task is-that is, it is often difficult to specify the software requirement completely.

Incomplete and imprecise requirements hinder accurate cost estimation. The clearer and more complete the requirements are the easier it is to determine the resources required. But even when the requirements are clear, estimating the number of engineers needed is a formidable task with inherent difficulties. The best approach is to develop the resource requirements incrementally, revising current estimates as more information becomes available. We have seen that an appropriate adaptation of the spiral model allows one to start with an estimate in the early planning stages and revise the estimate for the remaining tasks at the conclusion of each iteration of the particular phase of the life cycle.

How long it will take a software engineer to accomplish a given task is primarily a function of the complexity of the problem, the engineer's ability, the design of the software,

and the tools that are available. For example, adding an undo facility to an editor may require adding a new module or a complete redesign, depending on the current design of the editor. Similarly, writing a front-end parser for a system may be a simple task for an engineer who is familiar with parsing technology, but an extremely difficult task for an engineer who is unaware of and thus tries to reinvent, the parsing technology. Finally, writing a compiler with compiler development tools is a lot easier than writing it without them.

We have already observed that management decisions have a strong impact on the technical aspects of a project. We can see another example of this in the interplay between management planning and the entire software life cycle. For example, the choice of an evolutionary process model will impose a different kind of resource planning from a waterfall model. While an evolutionary model allows the manager to do resource planning incrementally as more information becomes available, it also requires more flexibility from the manager. Similarly, an incremental model will affect the resource loading for the different phases of the project, such as design and testing, because the phases are iterated incrementally. In general, there is a strong interplay between management functions and the technical aspects of a software engineering project.

## 2.1 Software Productivity

One of the basic requirements of management in any engineering discipline is to measure the productivity of the people and processes involved in production. The measurements obtained are used during the planning phase of a project as the basis for estimating resource requirements they also form the basis for evaluating the performance of individuals, processes and tools, the benefits of automation, etc. The ability to quantify the impact of various factors on productivity is important if we want to evaluate the many claims made for various "productivity improvement" tools. Indeed, there is an economic need for continued improvements in productivity. Of course, we can be sure that we are making improvements only if we can measure productivity quantitatively. Unfortunately, few, if any, claims made about tools that improve software productivity are based on quantitative measurements.

### 2.1.1 Productivity metrics

We clearly need a metric for measuring software productivity. In manufacturing-dominated disciplines such as automobile or television production, there are simple and obvious metrics to use. For instance, a new manufacturing process that doubles the number of television sets produced per day has doubled the productivity of the factory. By taking into account the costs of the new process, we can determine whether adopting the new process was cost effective.

The situation is not so simple in a design-dominated discipline like software engineering. To begin with, there is the problem that software does not scale up. For example, the typical productivity figure reported for professional software engineers is a

few tens of lines of code per day. If we compare this number with the productivity exhibited on a student project, even beginning students appear to be much more productive than professionals if we simply extrapolate from what they produce on a simple homework project.

There are, however, several obvious reasons why a professional's productivity appears to be less than a student's. First, students work longer hours, especially when a project is due. Second, professionals don't spend the majority of their time in strictly engineering activities: up to half their time may be taken up by meetings, administrative matters, communication with team members, etc. One study showed that up to 40% of a typical workweek is spent on nontechnical work. This implies that the larger a programming team becomes, the lower the individual productivity figures will be. Third, the professional has to meet more quality requirements than the student: reliability, maintainability, performance, etc. Fourth, the inherent complexity of the application affects the programmer's productivity. For example, experience has shown that programmers developing application programs produce as much as three times as many lines per day as programmers working on systems programs.

An ideal productivity metric measure not lines of code but the amount of functionality produced per unit time. The problem, however, is that we have no good way of quantifying the concept of functionality. For example, how can we relate the functionality offered by a payroll system to that offered by an air-traffic monitoring system? We need to either find a measure that somehow takes into account the inherent complexities of the different application areas or develop metrics that are specialized to different application areas. One such metric, developed for information systems, is called function points and is described in the next subsection.

Because of the difficulty of quantifying functionality, the search for productivity metric has for the most part concentrated on the most tangible product of software engineering: the actual code produced by the engineer. Various metrics based on code size have been developed. These will be discussed below, following our discussion of function points.

**Function points**

Function points attempt to quantify the functionality of a software system. The goal is to arrive at a single number that completely characterizes the system and correlates with observed programmer productivity figures. Actually, the function point characterizes the complexity of the software system and thus can be used to forecast how long it will take to develop the system and how many people will be needed to do it.

The function point method was derived empirically, and the limited amount of experimentation to date shows that it applies well in business applications, e.g., information systems. For example, using the function point as productivity metric, it has been possible to quantify improvements in productivity due to such factors as the use of

structured programming and high-level languages. We can measure programmer productivity in terms of the number of function points produced per unit time.

According to the function point method, five items determine the complexity of an application and the function point of given software is the weighted sum of these five items. The weights for these items have been derived empirically and validated by observation on many projects. The items and their weights are shown in Table 1.

The number of inputs and outputs count the distinct number of items that the user Provides to the system and the system provides to user, respectively. In counting the number of outputs, a group of items such as a screen or a file counts as one item-that is the individual items in the group are not counted separately. The number of inquiries is the number of distinct interactive queries made by the user that require specific action by the system. Files are any group of related information that is maintained by the application on behalf of the user or for organizing the application. This item reveals the bias of the method towards business data processing. Finally, the number of interfaces is the number of external interfaces to other systems, e.g., the number of files to be read or written to disk or transmitted or received from other systems. Therefore, if the application reads a file that is produced by another application, the file is counted twice, once as input and once as an interface. If a file is maintained by two different applications, it is counted as an interface for each application. The focus of function point analysis is to measure the complexity of an application based only on the function the application is supposed to provide to users. Therefore, any temporary files produced by the application are not counted in the measure; only those files that are visible or required by the user are so counted.

| **Item** | **Weight** |
|---|---|
| Number of inputs | **4** |
| Number of outputs | **5** |
| Number of inquiries | **4** |

**Table 1 Function point items and weights.**

Once we have a metric, we can use it in many ways. For example, we can compute the productivity of an engineer or a team per month by dividing the function point for an existing software by the number of months it took to develop the software. Or we can divide the amount of money spent to produce the software by the function point to compute how much each function point costs. Or again, we can measure the error rate per function point by dividing the number of errors found by the function point. These function point-related numbers can be recorded and used as bases for future planning or interproject comparisons. If the function point is an accurate description of product complexity, the numbers should be similar across different projects.

**Code size**

Since software engineers are supposed to produce software, the most tangible product of software engineering is the running software that they deliver to their clients. This has led many people to use the size of the software as a measure of productivity. Of course the size of the software is not necessarily an indication of how much effort went into producing it and a programmer that produces twice as many statements as another is not necessarily twice as productive. In general, the size of code is not a measure of any quality: a program that is twice as big as another is not necessarily twice as good in any sense. Nevertheless, the most commonly used metrics to measure productivity are based on code size, for example, in terms of number of source lines produced per unit time

Of course, a code size metric has to be qualified immediately by considering the same issues: Should we count comment lines? Should we count programming language "statements" or simply the number of line feeds? How many times should we count the lines in a file that is "included" several times? And should we count declarations or just executable statements? By choosing answers to these questions, we arrive at a particular productivity metric based on code size.

Two of the most common code size metrics are DSI (delivered source instructions) in which only lines of code delivered to the customer are counted and NCSS (non-commented source statements), in which comment lines are not counted. We will use the generic unit, KLOC (thousands of lines of code), when we do not want to distinguish between the specific metrics.

Lines of code have been used as productivity metric in many organizations. The acceptance of this metric is due to the ease of measuring it and probably also due to the fact that using any metric is better than using nothing at all. But at the same time, we must be cognizant of the many problems associated with the measure. At the heart of the problem is that there is no semantic content to code size; rather, it is merely a function of the form of software. The following observations show some of the deficiencies inherent in the metric.

Consider two programmers faced with the task of programming a module that needs a sort operation internally. One programmer writes his own sort routine and the other uses her time to find out how to use an existing sort routine in the system library. Even though both accomplish the same task, the use of the library routine is generally a better idea for many reasons: over the life of the software, the library routine has less chances of containing errors and is better tested; it allows the programmer to concentrate on the real problems she is trying to solve and perhaps gain better insight into the application rather than code a sorting routine, etc. Yet, the size metric penalizes the use of the library routine because it recognizes the other programmer as more productive! In general, the size-based metric has the unfortunate effect of equating software reuse with lower productivity.

When using lines of code as productivity metric, it is important to know what lines are being counted in order to be able to make accurate comparisons between individuals, projects or organizations. For example, an organization may produce many software tools to support the development of a particular project. These tools may become useful to many other projects in the organization, but are not delivered to the customer as part of the product. Should these programs be counted in the measurements? A commonly adopted convention is to count only the lines of code that are delivered to the customer. To emphasize this decision, the models that use the convention refer to the measure as KDSI-thousands of delivered source instructions.

Using DSI as a measure does not mean that producing internal tools is a worthless activity. In any engineering activity, investments in tools and a support infrastructure are needed to improve productivity. The DSI measure focuses on the objective of the engineering organization and separates the effort involved in building internal tools from that involved in building the product itself. In other engineering disciplines, too, productivity is measured in terms of the final product, not the intermediate "mock-ups."

Another point to consider when comparing the productivity figures reported by different organizations is whether the reported figures include the effect of cancelled projects. For various reasons, many projects are cancelled in organizations before they produce and deliver a product. Whether the lines of code of these cancelled projects are counted in the overall productivity of the organization has a material effect on the productivity figure. Such cancellations also have a secondary impact on the motivational attitude of engineers in the organization and thus affect overall productivity. Whatever criteria are used for measurement, comparison of figures is possible only if the criteria are applied consistently. Since there are no standard metrics, the use and interpretation of available productivity data in the literature must be examined with-care.

Finally, lines of code are tied to line-oriented languages and are inherently incapable of dealing with the emerging visual languages in which the programmer uses diagrams or screen panels rather than statements.

### 2.1.2 Factors affecting productivity

Regardless of what metric we use for productivity, even if we simply have an intuitive notion, there are factors that affect engineering productivity. One of the important effects of productivity metric is that it allows the benefits of the various factors to be quantified. By identifying the major contributors to productivity, organizations can determine quantitatively whether they can afford to change their practices, that is, whether the improvements in productivity are worth the expenses. For example, will changing to a new programming language or adopting a new development process, or hiring an efficiency expert or giving the engineers a private office, or allowing them to work at home increase productivity sufficiently to justify the expenses?

In one study that used lines of code as a metric, it was found that the single most important factor affecting productivity was the capability of the personnel; half as important, but second on the list, was the complexity of the product, followed by required reliability and timing constraints (i.e., as in real-time software). The least important items on the list were schedule constraints and language experience. The effects of these various factors on productivity are reflected in the "cost driver functions" used in cost estimation models, such as the COCOMO model which we will see later.

Many managers believe that an aggressive schedule motivates the engineers to do a better, faster job. However, experiments have shown that unrealistically aggressive schedules not only cause the engineers to compromise on intangible quality aspects, but also cause schedule delays. A surprising finding was that in a controlled experiment, the subjects who had no schedule at all finished their project first, beating out both the group that had an aggressive schedule and the one that had a relaxed schedule. It has been shown that engineers, in general, are good at achieving the one tangible goal that they have been given: if the primary goal is to finish according to a given time schedule, they usually will-but at the cost of other goals such as clarity of documentation and structure.

A specific example of the effect of overly aggressive scheduling can be seen when design reviews are scheduled far too early in the development cycle. While the manager may want to motivate the engineers to do the job in a shorter time, such premature design reviews force the designer to document only issues that are well understood and deny the reviewers an opportunity to provide useful feedback

There are many intangible factors that affect productivity and reduce the credibility of the published numbers. Examples of these intangible factors are personnel turnover, cancelled projects, reorganizations and restructuring of systems for better quality.

## 2.2 People and Productivity

Because software engineering is predominantly an intellectual activity, the most important ingredient for producing high-quality software efficiently is people. As we mentioned in the last section, experimental evidence shows that the most determinative factor of productivity is the capability of the engineers. Despite the intuitive appeal of this notion and strong supporting empirical evidence, however, many managers and organizations consistently behave as if they did not believe it. Considering the main difference in software engineering competence between the best and the worst engineers and the critical importance of engineering competence in attaining high software productivity, the costs of hiring, retaining and motivating the best engineers can be justified on economic ground.

Yet, a common misconception held by managers, as evidenced in their staffing, planning, and scheduling practices, is that software engineers are interchangeable that is that one software engineer is as productive as another. In fact experiments have revealed a

large variability in productivity between the best, the average and the worst engineers. The worst engineers even reduce the productivity of the team.

Apart from engineering capability, however, because of the strong amount of interaction required among team members, the personalities of the members also should be taken into account. Some engineers function better on centrally controlled teams while others are better suited for teams with decentralized control. In short, engineers are simply not interchangeable due to underlying reasons that have to do with both technical ability and personality.

Another common management practice that fails to recognize the importance of people in the software engineering process is that managers often staff a project with the best available people rather than attempt to find the best people per se. considering the strong impact of personnel capability on software costs; this is a foolish thing to do. In effect, by assigning ill-prepared engineers to a project, the manager is committing the organization to a training period for these engineers. Since this commitment is unintentional and the training haphazard, there is a strong risk that the training will be ineffective and the project unsuccessful or at least late. The way to solve the Problem of finding appropriate people must be faced deliberately: one must schedule the training period as a required task and train the people appropriately, hire outside consultants and control the project closely, perhaps by scheduling frequent incremental deliveries. The point is to recognize the utter importance of key personnel to an intellectual project such as software engineering.

## 2.3 Cost Estimation

Cost estimation is part of the planning stage of any engineering activity. The difference in cost estimation between software engineering and other disciplines is that in software engineering the primary cost is for people. In other engineering disciplines the cost of materials-chips, bricks or aluminum, depending on the activity-is a major component of the cost that must be estimated. In software engineering, to estimate the cost we only have to estimate how many engineers are needed.

Software cost estimation has two uses in software project management. First during the planning stage, one needs to decide how many engineers are needed for the project and develop a schedule. Second, in monitoring the project's progress, one need to access whether the project is progressing according to schedule and take corrective action if necessary. In monitoring progress, the manager needs to ascertain how much work has already been accomplished and how much is left to do. Both of these tasks require a metric for measuring "software work accomplishment."

| Group | Factor |
|---|---|
| Size Attributes | Source Instruction |
| | Object Instruction |
| | Number of routines |
| | Number of data items |
| | Number of output formats |
| | Documentation |
| | Number of personnel |
| Programme Attributes | Type |
| | Complexity |
| | Language |
| | Reuse |
| | Required reliability |
| | Display requirements |
| Computer attributes | Time Constraint |
| | Storage constraint |
| | Hardware configuration |
| | Concurrent h/w development |
| | Interfacing equipment, s/w |
| Personal attributes | Personnel capability |
| | Personnel continuity |
| | Hardware experience |

| | |
|---|---|
| | Application experience |
| | Language experience |
| Project attributes | Tools and techniques |
| | Customer interface |
| | Requirements definition |
| | Requirements volatility |
| | Schedule |
| | Security |
| | Computer access |
| | Travel/rehosting/multi-site |
| | Support software maturity |

Table : 2 Factors used in cost estimation models

| Feature | Mode | | |
|---|---|---|---|
| | Organic | Semidetached | Embedded |
| Organizational understanding of product objectives | Thorough | Considerable | General |
| Experience in working with related software systems | Extensive | Considerable | Moderate |
| Need for software conformance with pre-established requirements | Basic | Considerable | Full |
| Need for software conformance with external interface specifications | Basic | Considerable | Full |
| Concurrent development of associated new hardware and operational procedures | Some | Moderate | Extensive |
| Need for innovative data processing architectures, algorithms | Minimal | Some | Considerable |
| Premium on early completion | Low | Medium | High |
| Product size range | <50 KDSI | <300 KDSI | All sizes |
| EXAMPLES | Batch data reduction | Most transaction processing systems | Large, complex transaction processing systems |
| | Scientific models | New OS, DBMS | Ambitious, very large OS |
| | Business models | Ambitious, inventory, production control | Avionics |
| | Familiar OS, compiler | Simple command-control | Ambitious command-control |

**Table 3** COCOMO Software development modes.

### 2.3.1 Predictive models of software cost

While lines of code are not an ideal metric of productivity, they do seem like an appropriate metric for the total life cycle costs of software. That is the size of an existing piece of software is a good measure of how hard it is to understand and modify that software during its maintenance phase. Furthermore, if we could predict how large a software system was going to be before developing it, that size could be used as a basis for determining how much effort would be required, how many engineers would be needed and how long it would take to develop the software. That is, the size of the software can help us infer not just the initial development cost, but also the costs associated with the later stages of the life cycle.

The majority of software cost estimation methods thus start by predicting the size of the software and using that as input for deriving the total effort required, the required effort during each phase of the life cycle, personnel requirements, etc. The size estimate drives the entire estimation process. Inferring this initial estimate and assessing its reliability can be considerably simpler if the organization maintains a data base of information about past projects.

We can also base the initial estimate on an analytical technique such as function point analysis. We first compute the function point for the desired software and then divide it by the number FP/LOC for the project's programming language to arrive-at a size estimate.

| Development Mode | Nominal effort | Schedule |
|---|---|---|
| Organic | $(PM)_{NOM}=3.2(KDSI)^{1.05}$ | $TDEV=2.5(PM_{DEV}))^{0.38}$ |
| Semidetached | $(PM)_{NOM}=3.0(KDSI)^{1.12}$ | $TDEV=2.5(PM_{DEV}))^{0.35}$ |
| Embedded | $(PM)_{NOM}=2.8(KDSI)^{1.20}$ | $TDEV=2.5(PM_{DEV}))^{0.32}$ |

**Table 4** COCOMO Nominal effort and schedule equations.

Besides basing the estimation of effort on code size, most cost estimation models share certain other concepts. The purpose of a software cost model is to predict the total development effort required to produce a given piece of software in terms of the number of engineers and length of time it will take to develop the software. The general formula used to arrive at the nominal development effort is

$$PM_{initial} = c.KLOC^{k}$$

That is, the nominal number of programmer-months is derived on the basis of the number of lines of code. The constants c and k are given by the model. The exponent k is greater than 1, causing the estimated effort to grow nonlinearly with the code size. To take into account the many variables that can affect the productivity of the project, so-called cost drivers are used to scale this initial estimate of effort. For example, if modern programming practices are being used, the estimate is scaled downward; if there are real-time reliability requirements, the estimate is scaled upward. The particular cost-driver items are determined and validated empirically. In general, the cost drivers can be classified as being attributes of the following items:

- **Product.** For example, reliability requirements or inherent complexity.
- **Computer.** For example, are there execution time or storage constraints?
- **Personnel.** For example, are the personnel experienced in the application area or the programming language being used?
- **Project.** For example, are sophisticated software tools being used?

| Cost Drivers | Ratings | | | | | |
|---|---|---|---|---|---|---|
| | Very low | Low | Nominal | High | Very High | Extra High |
| **Product attributes** | | | | | | |
| Required software reliability | .75 | .88 | 1.00 | 1.15 | 1.40 | |
| Data base size | | .94 | 1.00 | 1.08 | 1.16 | |
| Product complexity | .70 | .85 | 1.00 | 1.15 | 1.30 | 1.65 |
| **Computer attributes** | | | | | | |
| Execution time constraints | | | 1.00 | 1.11 | 1.30 | 1.66 |
| Main storage constraints | | | 1.00 | 1.06 | 1.21 | 1.56 |
| Virtual machine volatility* | | .87 | 1.00 | 1.15 | 1.30 | |
| Computer turnaround time | | .87 | 1.00 | 1.07 | 1.15 | |
| **Personnel attributes** | | | | | | |
| Analyst capability | 1.46 | 1.19 | 1.00 | .86 | .71 | |
| Applications experience | 1.29 | 1.13 | 1.00 | .91 | .82 | |
| Programmer capability | 1.42 | 1.17 | 1.00 | .86 | .70 | |
| Virtual machine experience* | 1.21 | 1.10 | 1.00 | .90 | | |
| Programming language experience | 1.14 | 1.07 | 1.00 | .95 | | |
| **Project attributes** | | | | | | |
| Use of modern programming practices | 1.24 | 1.10 | 1.00 | .91 | .82 | |
| Use of software tools | 1.24 | 1.10 | 1.00 | .91 | .83 | |
| Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 | |

*For a given software product, the underlying virtual machine is the complex of hardware and software (OS, DBMS, etc.) it calls on to accomplish its tasks.

**Table 5:** Efforts multipliers used by COCOMO intermediate model

The particular attributes in each class differ from model to model. For example some models use object code for the size estimate, others use source code, and some both object code and source code. Personnel attributes that can be considered include the capability and continuity (lack of turnover) of the personnel. Table 2 shows the set of factors considered by different models, classified into five different groups of attributes. Size is separated into its own group because of its importance and the different ways in which it is treated in the different models.

The basic steps for arriving at the cost of a proposed software system are the following:

1. Estimate the software's eventual size, and use it in the model's formula to arrive at an initial estimate of effort;
2. Revise the estimate by using the cost driver or other scaling factors given by the model;
3. Apply the model's tools to the estimate derived in step 2 to determine the total effort, activity distribution, etc.

**The best known model for cost estimation today is the Constructive Cost Model, known by its acronym, COCOMO**. COCOMO is actually three different models of increasing complexity and level of detail. We next give an overview of the intermediate COCOMO model and the steps and details involved in the use of such a model.

**COCOMO**

The following fleshes out how the general estimation steps described above apply in the case of COCOMO:

1. The code size estimate is based on delivered source instructions, KDSI. The initial (nominal) development effort is based on the project's development "mode." COCOMO categorizes the software being developed according to three modes; organic, semidetached, and embedded. Table 3 shows how to determine which mode each project falls in. The estimator arrives at the development mode by deciding which entries in the table best characterize the project's features as listed in column 1. The heading on the column that best matches the project is the development mode for the project. For example, flight control software for a new fighter airplane falls into the embedded class, and a standard payroll application falls into the organic class. Study the table carefully to see the effect of the various features on the mode and, therefore, on the development effort.
   Each development mode has an associated formula for determining the nominal development effort based on the estimated code size. The formulas are shown in Table 4. Tables 3 and 4 together can be considered a quantitative summary of a considerable amount of experimental data collected by Boehm over the years.

2. The estimator determines the effort multiplier for the particular project, based on cost-driver attributes. COCOMO uses 15 cost-driver attributes to scale the nominal development effort. These attributes are a subset of the general factors listed in Table 2 and are given in Table 5, along with the multiplier used for each, based on rating the driver for the particular project. There is a guideline for determining how to rate each attribute for the project at hand. The rating ranges from very low to extra high. The multipliers are multiplied together and with the nominal effort derived in step 1 to arrive at the estimate of total effort for the project.

   Table 5 contains a wealth of information. For example, the range of the multipliers for each factor shows the impact of that factor and how much control the manager has over the factor. As an example, the range of analyst capability shows that the difference between using an analyst of very high capability and one of very low capability is a factor of two in the cost estimate. The product attributes, in general, are fixed by the inherent complexity of the product and are not within the control of the manager.

3. Given the estimate of the total effort in step 2, COCOMO allows the derivation of various other important numbers and analyses. For example, Table 4 shows the

formulas, again based on the development mode, for deriving, a recommended length for the project schedule based on the estimate of the total effort for the project.

The COCOMO model allows sensitivity analyses based on changing the parameters. For example, one can model the change in development time as a function of relaxing the reliability constraints or improving the software development environment. Or one can analyze the cost and impact of unstable hardware on a project's software schedule.

Software cost estimation models such as COCOMO are required for an engineering approach to software management. Without such models, one has only judgment to rely on, which makes decisions hard to trust and justify. Worse, one can never be sure whether improvements are being made to the software. While current models still lack a full scientific justification, they can be used and validated against an organization's project data base.

A software development organization should maintain a project data base which stores information about the progress of projects. Such a data base can be used in many ways: to validate a particular cost estimation model against past projects; to calibrate cost-driver or scaling factors for a model, based on an organization's particular environment; as a basis for arriving at an initial size estimate; or to calibrate estimates of effort that are derived from a model.

With the current state of the art of cost estimation modelling, it is not wise to have complete and blind trust in the results of the models, but a project manager would be equally unwise to ignore the value of such tools for a software development organization as a whole. They can be used to complement expert judgment and intuition.

## 3. Project Control

As we said before, the purpose of controlling a project is to monitor the progress of the activities against the plans, to ensure that the goals are being approached and, eventually, achieved. Another aspect of control is to detect, as soon as possible, when deviations from the plan are occurring so that corrective action may be taken. In software engineering, as in any design-dominated discipline, it is especially important to plan realistically-even conservatively-so as to minimize the need for corrective action. For example, while in a manufacturing-dominated discipline it may be justifiable to hire the minimum number of workers necessary and add more employees if production falls below the required level, it has been observed in software engineering that adding people to a project that is late can delay the project even further. This underscores the importance of not only planning, but also controlling, in software engineering. The sooner deviations from the plan are detected, the more it is possible to cope with them.

### 3.1 Work Breakdown Structures

Most project control techniques are based on breaking down the goal of the project into several intermediate goals. Each intermediate goal can in turn be broken down further. This process can be repeated until each goal is small enough to be well understood. We can then plan for each goal individually-its resource requirements, assignment of responsibility, scheduling, etc.

A semiformal way of breaking down the goal is called the **work breakdown structure (WBS).** In this technique, one builds a tree whose root is labelled by the major activity of the project such as "build a compiler." Each node of the tree can be broken down into smaller components that are designated the children of the node. This "work breakdown" can be repeated until each leaf node in the tree is small enough to allow the manager to estimate its size, difficulty and resource requirements. Figure 1 shows the work breakdown structure for a simple compiler development project.

The goal of a work breakdown structure is to identify all the activities that a project must undertake. The tasks can be broken down into as fine a level of detail as is desired or necessary. For example, we might have shown the substructure of a node labelled "design" as consisting of the three different activities of designing the scanner, parser, and code generator. The structure can be used as a basis for estimating the amount of resources necessary for the project as a whole by estimating the resources required for each leaf node.

The work breakdown structure is a simple tool that gives the manager a framework for breaking down large tasks into more manageable pieces. Once these manageable pieces have been identified, they can be used as units of work assignment. The structure can be refined and extended easily by labeling the nodes with appropriate information, such as the planned length of the activity, the name of the person responsible for the activity and the starting and ending date of the activity. In this way, the structure can summarize the project plans.

The work breakdown structure can also be an input into the scheduling process, we will see in the following subsections. In breaking down the work, we are trying decide which tasks need to be done. In scheduling, we decide the order in which to do these tasks. Each work item in the work breakdown structure is associated with an activity to perform that item. A schedule tries to order the activities to ensure their timely completion.

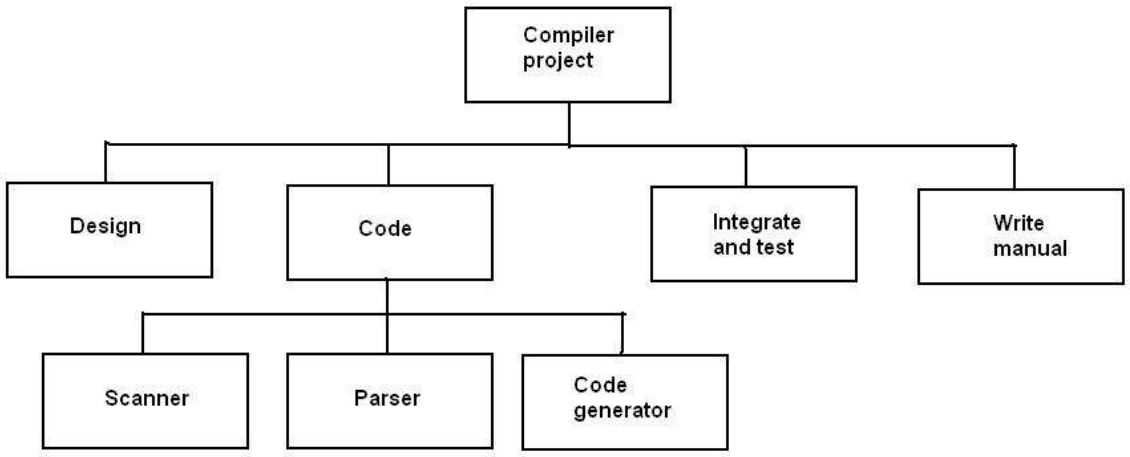



**Figure 1: Work breakdown structure for a complier project**

Two general scheduling techniques are Gantt charts and PERT charts. We will present these in the next two subsections.

### 3.2 Gantt Charts

Gantt charts (developed by Henry L. Gantt) are a project control technique that can be used for several purposes, including scheduling, budgeting and resource planning. **A Gantt chart is a bar chart, with each bar representing an activity**. The bars are drawn against a time line. The length of each bar is proportional to the length of time planned for the activity.

Let us draw a Gantt chart for the tasks identified in the WBS of Figure 1. We estimate the number of days required for each of the six tasks as follows: initial design, 45; scanner, 20; parser, 60; code generator, 180; integration and testing, 90; and writing the manual, 90. Using these estimates, we can draw the Gantt chart of Figure 2 for the compiler project.

A Gantt chart helps in scheduling the activities of a project, but it does not help in identifying them. One can begin with the activities identified in the work breakdown structure, as we did for the compiler example. During the scheduling activity, and also during implementation of the project, new activities may be identified that were not envisioned during the initial planning. The manager must then go back and revise the Breakdown structure and the schedules to deal with these new activities.

The Gantt chart in the figure is actually an enhanced version of standard Gantt charts. The white part of the bar shows the length of time each task is estimated to take. The gray part shows the "slack" time, that is, the latest time by which a task must be finished. One way to view the slack time is that, if necessary, we can slide the white area

over the gray area without forcing the start of the next activity to be delayed. For example, we have the freedom to delay the start of building the scanner to as late as October 17, 1994 and still have it finished in time to avoid delaying the integration and testing activity. The chart shows clearly that the results of the scanner and parser tasks can be used only after the code generator task is completed (in the integration and testing task). A bar that is all white, such as that representing the code generator task, has no slack and must be started and completed on the scheduled dates if the schedule is to be maintained. From the figure, we can see that the three tasks, "design," "code generator," and "integration and testing" have no slack. It is these tasks, then, that determine the total length of time the project is expected to take.

This example shows that the Gantt chart can be used for resource allocation and staff planning. For example, from Figure 2, we can conclude that the same engineer can be assigned to do the scanner and the parser while another engineer is working on the code generator. Even so, the first engineer will have some slack time that we may plan to use to help the second engineer or to get a head start on the integration and testing activity

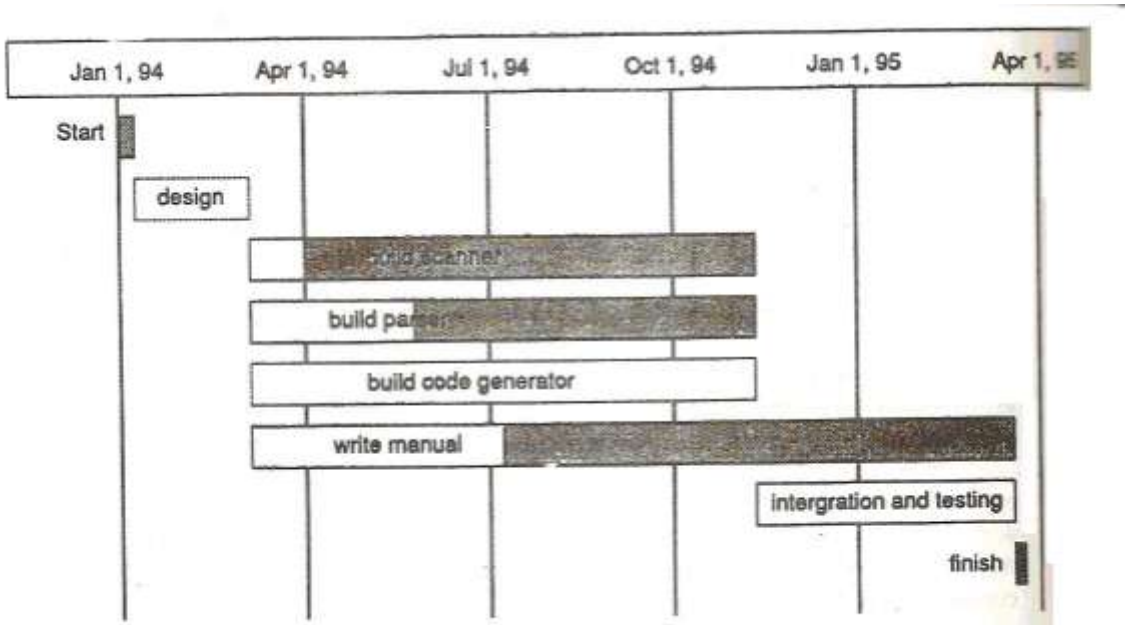



**Figure 2** Gantt chart for a simple compiler project.

Gantt charts can take different forms depending on their intended use. They are best for resource scheduling. For example, if we are trying to schedule the activities of six engineers, we might use a Gantt chart in which each bar represents one of the engineers. In such a chart, the engineers are our resources and the chart shows the *resource loading* during the project. It can help, for example, in scheduling vacation time, or in ensuring that the right number of engineers will be available during each desired period. Figure 3 shows an example. We could label appropriate sections of the bars to show how much time we expect each engineer to spend on each activity (e.g., design and building scanner).

Gantt charts are useful for resource planning and scheduling. While they show the tasks and their durations clearly, however, they do not show intertask dependencies plainly. PERT charts, the subject of the next section, show task dependencies directly.

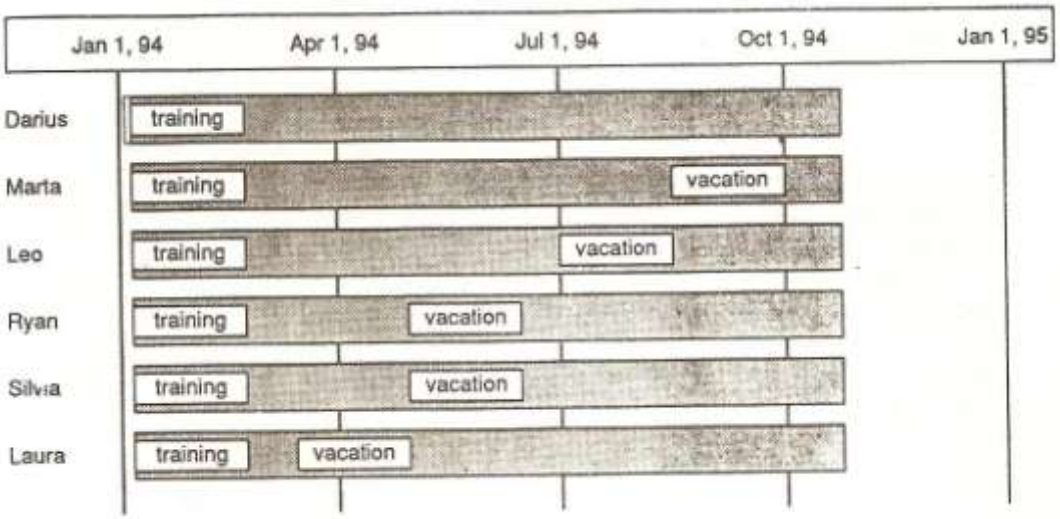



**Figure 3** Gantt chart for scheduling six engineers.

### 3.3 PERT Charts

**A PERT (Program Evaluation and Review Technique) chart is a network of boxes (or circles) and arrows**. There are different variations of PERT charts. Some use the boxes to represent activities and some use the arrows to do so. We will use the first approach here. Each box thus represents an activity. The arrows are used to show the dependencies of activities on one another. The activity at the head of an arrow cannot start until the activity at the tail of the arrow is finished. Just as with the nodes in the work breakdown structure, the boxes in a PERT chart can be decorated with starting and ending dates for activities; the arrows help in computing the earliest possible starting dates for the boxes at their heads. Some boxes can be designated as milestones- A milestone is an activity whose completion signals an important accomplishment in the life of the project. On the other hand, failure to make a milestone signals trouble to the manager and requires an explicit action to deal with the deviation from the schedule.

As with Gantt charts, to build a PERT chart for a project, one must first list all the activities required for completion of the project and estimate how long each will take, then one must determine the dependence of the activities on each other. The PERT chart gives a graphical representation of this information. Clearly, the technique does not help in deciding which activities are necessary or how long each will take, but it does force the Manager to take the necessary planning steps to answer these questions.

Figure 4 shows a PERT chart for the previous compiler project. The information from the work breakdown structure of figure 1 is used to decide what boxes we need. The arrows show the new information that was not available in the work breakdown structure.

The chart shows clearly that the project consists of the activities of initial design, building a scanner, building a parser, building a code generator, integrating and testing these and writing a manual. Recall that the previous estimates for these six tasks were, respectively, 45, 20, 60, 180, 90, and 90 days.

The figure assumes that the project will start on January 1, 1994 (shown underlined). Taking holidays into account (January 1 and 2 are holidays in 1994), the design work will start on January 3, 1994. Since the design activity is estimated to take 45 days, any activity that follows the design may start on March 7, 1994 at the earliest. The dependency arrows help us compute these earliest start dates based on our estimates of the duration of each activity. These dates are shown in the figure. We could also compute the earliest finish dates or latest start dates or latest finish dates, depending on the kind of analysis we want to perform.

The chart shows that the path through the project that consists of the "design," "build code generator," and "integration and testing" activities is the critical path for the project. Any delay in any activity in this path will cause a delay in the entire project. The manager will clearly want to monitor the activities on the critical path much more closely than the other activities.

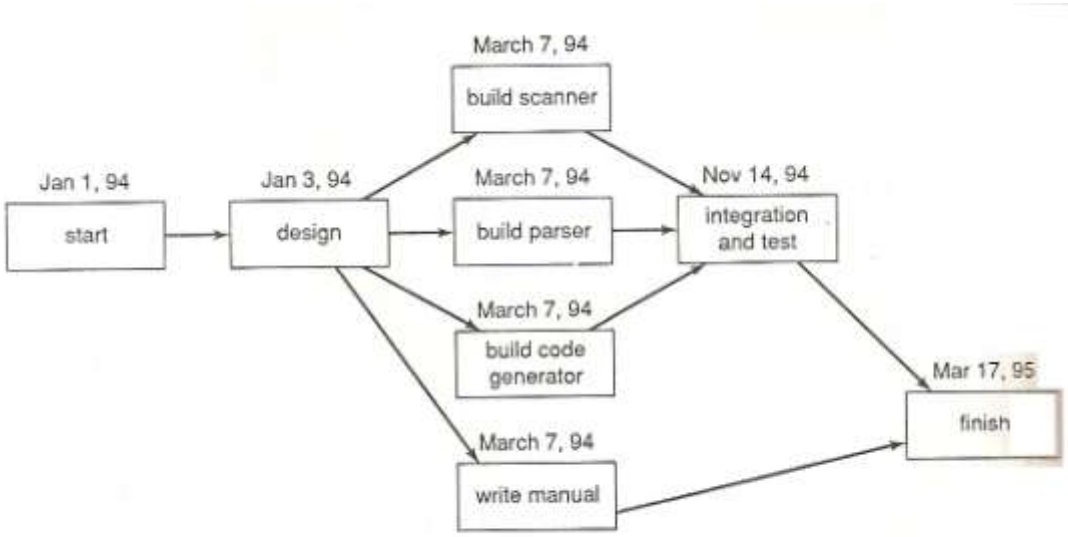



**Figure 4** PERT chart for a simple compiler project. Activities on the critical path are shown in bold.

**Some of the advantages of PERT are as follows:**

- It forces the manager to plan.
- It shows the interrelationships among the tasks in the project and in particular. clearly identifies the critical path of the project, thus helping to focus on it. For example, in the figure, the code generator is clearly the most critical activity in terms of the schedule. The critical path is shown by a dark solid line. We may decide to build a separate subproject for this activity alone, or put our best people

on the project or monitor the critical activity very closely. The fact that the PERT chart has exposed the critical path allows us the opportunity to consider alternative approaches to cope with a potential problem.

- It exposes all possible parallelism in the activities and thus helps in allocating resources.
- It allows scheduling and simulation of alternative schedules.
- It enables the manager to monitor and control the project.

**4. Organization**

The organizing function of management deals with devising roles for individuals and assigning responsibility for accomplishing the project goals. Organization is basically motivated by the need for cooperation when the goals are not achievable by a single human being in a reasonable amount of time. The aim of an organizational structure is to facilitate cooperation towards a common goal. An organizational structure is necessary at any level of an enterprise, whether it is to coordinate the efforts of a group of vice presidents who report to the president of the corporation or to orchestrate the interactions among programmers who report to a common project manager.

Management theorists and practitioners have studied the effects of different organizational structures on the productivity and effectiveness of groups. Because the goal of organization is to encourage cooperation and because cooperation depends substantially on human characteristics, many general organizational rules apply to any endeavor, whether it deals with software production or automobile production.

The task of organizing can be viewed as building a team: given a group of people and a common goal, what is the best role for each individual and how should responsibilities be divided? Analogies with sports teams are illuminating. A basketball team consists of five players on the floor who are playing the game and another perhaps five players who are substitutes. Each player knows his role. There is one ball, and at any one time, only one player can have it. All other players must know their responsibilities and what to expect from the player with the ball. On well-organized teams, the patterns of cooperation and their effects are clearly visible. In poorly organized teams or teams with novice players, the lack of patterns of cooperation is just as clearly visible: when one player has the ball, the other four scream to be passed the ball. The player with the ball, of course, shoots the ball instead of passing it to anyone!

Some of the considerations that affect the choice of an organization are similar to the factors that are used in cost estimation models that we have seen earlier. For example what constitutes an appropriate organization for a project depends on the length of the project. Is it a long-term project or a short, one-shot project? If it is a long-term project, it is important to ensure job satisfaction for individuals, leading to high morale and thus reducing turnover. Sometimes, the best composition for a team is a mix of junior and senior engineers. This allows the junior engineers to do the less challenging tasks and learn from the senior engineers, who are busy doing the more challenging tasks and overseeing the progress of the junior engineers.

Because of the nature of large software systems, changing requirements, and the difficulties of software specification, it has been observed that adding people to a project late in the development cycle leads to further delays in the schedule. Thus, the issue of personnel turnover is a serious one that must be minimized. On a short-term project, personnel turnover is not as important an issue. On a long-term project, it is important to enable junior personnel to develop their skills and gain more responsibility as senior personnel move on to other responsibilities. The trade-offs involved in organizing for a short-term or a long-term project are similar to those involved in organizing a basketball team to win a single game, to be a winner over a single season or to be a consistent winner over many years.

Another issue affecting the appropriate project organization is the nature of the task and how much communication the different team members need to have among themselves. For example, in a well-defined task such as a payroll system in which modules and their interfaces have been specified clearly, there is not much need for project members to communicate among each other, and excessive communication will probably lead to a delay in accomplishing their individual tasks. On the other hand, in a project where the task is not clearly understood, communication among team members is beneficial and can lead to a better solution. Strictly hierarchical organizations minimize and discourage communication among team members; more democratic organizations encourage it.

Software engineering requires not only the application of systematic techniques to routine aspects of the software but also invention and ingenuity when standard techniques are not adequate. Balancing these requirements is one of the most difficult aspects of software engineering management

We can categorize software development team organizations according to where decision-making control lies. A team can have centralized control, where a recognized leader is responsible for and authorized to produce a design and resolve all technical issues. Alternatively, a team organization can be based on distributing decision-making control and emphasizing group consensus. Both of these types of organization, as well as combinations of the two, have been used in practice successfully. The following subsections discuss the two kinds of organization in more detail.

### 4.1 Centralized control team organization

Centralized-control team organization is a standard management technique in well understood disciplines. In this mode of organization, several workers report to a supervisor who directly controls their tasks and is responsible for their performance. Centralized control is based on a hierarchical organizational structure in which several supervisors report to a "second-level" manager and so on up the chain to the president of the enterprise. In general, centralized control works well with tasks that are simple enough that the one person responsible for control of the project can grasp the problem and its solution.

One way to centralize the control of a software development team is through a chief programmer team. In this kind of organization, one engineer, known as the chief programmer is responsible for the design and all the technical details of the project. The

chief programmer reports to a peer project manager who is responsible for the administrative aspects of the project. Other members of the team are a software librarian and other programmers who report to the chief programmer and are added to the team on a temporary basis when needed. Specialists may be used by the team as consultants when the need arises. The need for programmers and consultants, as well as what tasks they perform, is determined by the chief programmer, who initiates and controls all decisions. The software library maintained by the librarian is the central repository for all the software, documentation, and decisions made by the team. Figure 5 is a graphical representation of patterns of control and communication supported by this kind of organization.

Chief programmer team organization has been likened to a surgical team performing an operation. During an operation, one person must be clearly in control, and all other people involved must be in total support of the "chief' surgeon; there is no place or time for individual creativity or group consensus. This analogy highlights the strengths of the chief programmer team organization, as well as its weaknesses. The chief programmer team organization works well when the task is well understood, is within the intellectual grasp of one individual and is such that the importance of finishing the project outweighs other factors (such as team morale, personnel development and life cycle costs).

On the negative side, a chief programmer team has a "single point of failure." Since all communication must go through, and all decisions must be made by, the chief programmer, the chief programmer may become overloaded or indeed, saturated. The success of the chief programmer team clearly depends on the skill and ability of the chief programmer and the size and complexity of the problem. The choice of chief programmer is the most important determinant of success of the chief programmer team. On the other hand, since there is a great variability in people's abilities-as much as a 10-to-1 ratio in productivity-a chief programmer position may be the best way to use the rare highly productive engineers.

### 4.2 Decentralized control team organization

In a decentralized-control team organization, decisions are made by consensus and all work is considered group work. Team members review each other's work and are responsible as a group for what every member produces. Figure 6 shows the patterns of control and communication among team members in a decentralized-control organization. The ring like management structure is intended to show the lack of a hierarchy and that all team members are at the same level.

Such a "democratic" organization leads to higher morale and job satisfaction and, therefore, to less turnover. The engineers feel more ownership of the project and responsibility for the problem, leading to higher quality in their work. A decentralized control organization is more suited for long-term projects because the amount of intragroup communication that it encourages leads to a longer development time, presumably accompanied by lower life cycle costs. The proponents of this kind of team organization claim that it is more appropriate for less understood and more complicated problems because a group can invent better solutions than a single individual. Such an

organization is based on a technique referred to as "ego less programming" because it encourages programmers to share and review one another's work.

On the negative side, decentralized-control team organization is not appropriate for large teams, where the communication overhead can overwhelm all the engineers, reducing individual productivity.

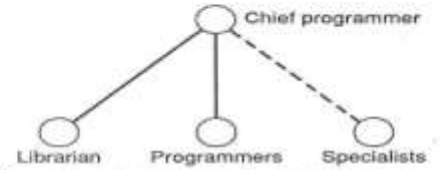



**Figure 5** Patterns of control and communication in a chief programmer team.

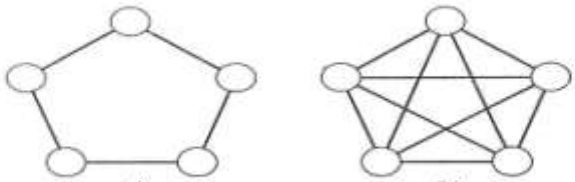



**Figure 6** A decentralized-control team.
**(a)** Management structure. **(b)** Patterns of communication.

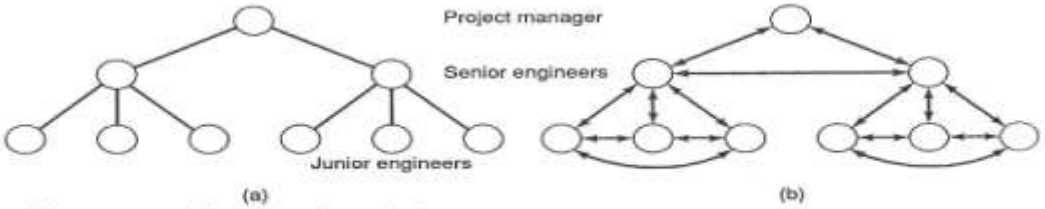



**Figure 7** Mixed-control organizations.
**(a)** Management structure. **(b)** Patterns of communication.

## 4.3 Mixed control team organization

A mixed control team organization attempts to combine the benefits of centralized and decentralized control, while minimizing or avoiding their disadvantages. Rather than treating all members the same as in a decentralized organization or treating a single individual as the chief as in a decentralized organization the mixed organization distinguishes the engineers into senior and junior engineers. Each senior engineer leads a group of junior engineers. The senior engineers in turn report to a project manager.

Control is vested in the project manager and senior programmers, while communication is decentralized among each set of individuals, peers and their immediate supervisors. The patterns of control and communication in mixed-control organizations are shown in Figure 7.

A mixed-mode organization tries to limit communication to within a small group that is most likely to benefit from it. It also tries to realize the benefits of group decision making by vesting authority in a group of senior programmers. The mixed-control

organization is an example of the use of a hierarchy to master the complexity of software development as well as organizational structure.

### 4.4 An assessment of team organization

In the previous subsections, we have presented different ways of organizing software development teams. Each kind of organization discussed in the previous three subsections has its proponents and detractors. Each also has its appropriate place. Experimental assessment of different organizational structures is difficult. It is clearly impractical to run large software development projects using two different types of organization, just for the purpose of comparing the effectiveness of the two structures. While cost estimation models can be assessed on the basis of how well they predict actual software costs, an organizational structure cannot be assessed so easily, because one cannot compare the results achieved with those one would have achieved with a different organization. Experiments have been run to measure the effects of such things as team size and task complexity on the effectiveness of development teams. In the choice of team organization, however, it appears that we must be content with the following general considerations:

- Just as no life cycle model is appropriate for all projects, no team organization is appropriate for all tasks.
- Decentralized control is best when communication among engineers is necessary for achieving a good solution.
- Centralized control is best when speed of development is the most important goal and the problem is well understood.
- An appropriate organization tries to limit the amount of communication to what is necessary for achieving project goals-no more and no less.
- An appropriate organization may have to take into account goals other than speed of development. Among these other important goals are: lower life cycle costs, reduced personnel turnover, repeatability of the process, development of junior engineers into senior engineers and widespread dissemination of specialized knowledge and expertise among personnel.

## 5. Summary

In this lesson we have discussed about various project planning activities like estimation of cost, people and productivity using various models, which are very important for any project. We have also studied various structures for controlling of project.

## 6. Self Check Exercise

1. What is project planning and organization?
2. Discuss the different stages of project planning.
3. What is project control in software engineering? What are the three elements of project control?

## 7. Suggested Readings

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

LESSON No. 1.5 AUTHOR: Dr. VISHAL SINGH

# The Requirement Analysis

**Objectives**



**Objectives:**

In this lesson we will discuss the requirement analysis phase of software engineering along with its use and benefits. We will also study about the software requirement specification in detail.

## 1. Introduction

Before starting to design a software product, it is very important to understand the precise requirements of the customer and to develop them properly. In the past many projects have been suffered because the developers started implementing something without determining whether they were building what the customer wanted. Starting properly documented activities without improperly documented requirements is the biggest mistake that one can commit during the product development. Improperly

documented requirements increase the number of iterative changes required during the life cycle phases and thereby push up the development cost tremendously. They also set the ground for bitter customer development disputes protracted legal battles. Therefore requirement analysis and specification is considered to be a very important phase of software development and has to be undertaken with utmost care. It is the process of determining user expectations for a new or modified product.

## 2. What is Requirements Analysis?

**Requirements Analysis is the process of understanding the customer needs and expectations from a proposed system or application and is a well-defined stage in the Software Development Life Cycle model**. Requirements are a description of how a system should behave or a description of system properties or attributes. It can alternatively be a statement of 'what' an application is expected to do. Given the multiple levels of interaction between users, business processes and devices in global corporations today, there are simultaneous and complex requirements from a single application, from various levels within an organization and outside it as well.

The Software Requirements Analysis Process covers the complex task of eliciting and documenting the requirements of all these users, modeling and analyzing these requirements and documenting them as a basis for system design. A dedicated and specialized Requirements Analyst is best equipped to handle the job. The Requirements Analysis function may also fall under the scope of Project Manager, Program Manager or Business Analyst, depending on the organizational hierarchy. Software Requirements Analysis and Documentation Processes are critical to software project success. Requirements engineering is an emerging field which deals with the systematic handling of requirements.

**Why is Requirements Analysis necessary?**

Studies reveal that inadequate attention to Software Requirements Analysis at the beginning of a project is the most common cause for critically vulnerable projects that often do not deliver even on the basic tasks for which they were designed. There are instances of corporations that have spent huge amounts on software projects where the end application eventually does not perform the tasks it was intended for. Software companies are now investing time and resources into effective and streamlined Software Requirements Analysis Processes as a prerequisite to successful projects that align with the client's business goals and meet the project's requirement specifications.

## 3. Steps in the Requirements Analysis Process

### I. Fix system boundaries

This initial step helps in identifying how the new application integrates with the business processes, how it fits into the larger picture and what its scope and limitations will be.

### II. Identify the customer

In more recent times there has been a focus on identifying who the 'users' or 'customers' of an application are. Referred to broadly as the 'stake holders', these indicate

the group or groups of people who will be directly or indirectly impacted by the new application.

By defining in concrete terms who the intended user is, the Requirements Analyst knows in advance where he has to look for answers. The Requirements Elicitation Process should focus on the wish-list of this defined group to arrive at a valid requirements list.

**III. Requirements elicitation**

Information is gathered from the multiple stakeholders identified. The Requirements Analyst draws out from each of these groups what their requirements from the application are and what they expect the application to accomplish.

Considering the multiple stakeholders involved, the list of requirements gathered in this manner could run into pages. The level of detail of the requirements list is based on the number and size of user groups, the degree of complexity of business processes and the size of the application.

**a) Problems faced in Requirements Elicitation**

- Ambiguous understanding of processes
- Inconsistency within a single process by multiple users
- Insufficient input from stakeholders
- Conflicting stakeholder interests
- Changes in requirements after project has begun

A Requirements Analyst has to interact closely with multiple work-groups, often with conflicting goals, to arrive at a bona-fide requirements list. Strong communication and people skills along with sound programming knowledge are prerequisites for an expert Requirements Analyst.

**b) Tools used in Requirements Elicitation**

Traditional methods of Requirements Elicitation included stakeholder interviews and focus group studies. Other methods like flowcharting of business processes and the use of existing documentation like user manuals, organizational charts, process models and systems or process specifications, on-site analysis, interviews with end-users, market research and competitor analysis were also used extensively in Requirements Elicitation.

However current research in Software Requirements Analysis Process has thrown up modern tools that are better equipped to handle the complex and multilayered process of Requirements Elicitation. Some of the current Requirements Elicitation tools in use are:

- Prototypes
- Use cases
- Data flow diagrams
- Transition process diagrams
- User interfaces

**IV. Requirements Analysis Process**

Once all stakeholder requirements have been gathered, a structured analysis of these can be done after modeling the requirements. Some of the Software Requirements Analysis techniques used are requirements animation, automated reasoning, knowledge-based critiquing, consistency checking, analogical and case-based reasoning.

**V. Requirements Specification**

Requirements, once elicited, modeled and analyzed should be documented in clear, unambiguous terms. A written requirements document is critical so that its circulation is possible among all stakeholders including the client, user-groups, the development and testing teams. Current requirements engineering practices reveal that a well-designed, clearly documented Requirements Specification is vital and serves as :

- Base for validating the stated requirements and resolving stakeholder conflicts, if any
- Contract between the client and development team
- Basis for systems design for the development team
- Bench-mark for project managers for planning project development lifecycle and goals
- Source for formulating test plans for QA and testing teams
- Resource for requirements management and requirements tracing
- Basis for evolving requirements over the project life span

Software requirements specification involves scoping the requirements so that it meets the customer's vision. It is the result of collaboration between the end-user who is often not a technical expert and a Technical/Systems Analyst, who is likely to approach the situation in technical terms.

The software requirements specification is a document that lists out stakeholders' needs and communicates these to the technical community that will design and build the system. The challenge of a well-written requirements specification is to clearly communicate to both these groups and all the sub-groups within.

To overcome this, Requirements Specifications may be documented separately as

- **User Requirements -** written in clear, precise language with plain text and use cases, for the benefit of the customer and end-user
- **System Requirements -** expressed as a programming or mathematical model, addressing the Application Development Team and QA and Testing Team.

Requirements Specification serves as a starting point for software, hardware and database design. It describes the function (Functional and Non-Functional specifications) of the system, performance of the system and the operational and user-interface constraints that will govern system development.

**VI. Requirements Management**

Requirements Management is the comprehensive process that includes all aspects of software requirements analysis and additionally ensures verification, validation and traceability of requirements. Effective requirements management practices guarantee that all system requirements are stated unambiguously, that omissions and errors are corrected and that evolving specifications can be incorporated later in the project lifecycle.

**4. Types of Requirements**

Requirements are categorized in several ways. The following are common categorizations of requirements that relate to technical management:

**Customer Requirements**

Statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints and measures of effectiveness and

suitability (MOE/MOS). The customers are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key customer. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:

- *Operational distribution or deployment:* Where will the system be used?
- *Mission profile or scenario:* How will the system accomplish its mission objective?
- *Performance and related parameters:* What are the critical system parameters to accomplish the mission?
- *Utilization environments:* How are the various system components to be used?
- *Effectiveness requirements:* How effective or efficient must the system be in performing its mission?
- *Operational life cycle:* How long will the system be in use by the user?
- *Environment:* What environments will the system be expected to operate in an effective manner?

**Functional Requirements**

Functional requirements explain what has to be done by identifying the necessary task, action or activity that must be accomplished. Functional requirements analysis will be used as the top level functions for functional analysis.

**Performance Requirements**

The extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success and their relationship to other requirements.

**Design Requirements**

The "build to," "code to," and "buy to" requirements for products and "how to execute" requirements for processes expressed in technical data packages and technical manuals.

**Derived Requirements**

Requirements that are implied or transformed from higher-level requirement. For example, a requirement for long range or high speed may result in a design requirement for low weight.

**Allocated Requirements**

A requirement that is established by dividing or otherwise allocating a high-level requirement into multiple lower-level requirements. Example: A 100-pound item that consists of two subsystems might result in weight requirements of 70 pounds and 30 pounds for the two lower-level items.

**5. What is a Software Requirements Specification?**

**An SRS is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies at a particular point in time (usually) prior to any actual design or development work**. It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

The SRS document itself states in precise and explicit language those functions and capabilities a software system (i.e., a software application, an eCommerce Web site, and so on) must provide, as well as states any required constraints by which the system must abide. The SRS also functions as a blueprint for completing a project with as little cost growth as possible. The SRS is often referred to as the "parent" document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans and documentation plans are related to it.

It's important to note that an SRS contains functional and nonfunctional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues or any other information other than what the development team understands the customer's system requirements to be.

A well-designed, well-written SRS accomplishes four major goals:

- **It provides feedback to the customer.** An SRS is the customer's assurance that the development organization understands the issues or problems to be solved and the software behavior necessary to address those problems. Therefore, the SRS should be written in natural language in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables and so on.
- **It decomposes the problem into component parts.** The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas and helps break down the problem into its component parts in an orderly fashion.
- **It serves as an input to the design specification.** As mentioned previously, the SRS serves as the parent document to subsequent documents, such as the software design specification and statement of work. Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.
- **It serves as a product validation check.** The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

SRSs are typically developed during the first stages of "Requirements Development," which is the initial product development phase in which information is gathered about what requirements are needed--and not. This information-gathering stage can include

onsite visits, questionnaires, surveys, interviews and perhaps a return-on-investment (ROI) analysis or needs analysis of the customer or client's current business environment. The actual specification then is written after the requirements have been gathered and analyzed.

**6. What are the benefits of a Great SRS?**

The IEEE 830 standard defines the benefits of a good SRS:

**Establish the basis for agreement between the customers and the suppliers on what the software product is to do.** The complete description of the functions to be performed by the software specified in the SRS will assist the potential users to determine if the software specified meets their needs or how the software must be modified to meet their needs.

**Reduce the development effort.** The preparation of the SRS forces the various concerned groups in the customer's organization to consider rigorously all of the requirements before design begins and reduces later redesign, recoding and retesting. Careful review of the requirements in the SRS can reveal omissions, misunderstandings, and inconsistencies early in the development cycle when these problems are easier to correct.

**Provide a basis for estimating costs and schedules.** The description of the product to be developed as given in the SRS is a realistic basis for estimating project costs and can be used to obtain approval for bids or price estimates.

**Provide a baseline for validation and verification.** Organizations can develop their validation and Verification plans much more productively from a good SRS. As a part of the development contract, the SRS provides a baseline against which compliance can be measured.

**Facilitate transfer.** The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organization and suppliers find it easier to transfer it to new customers.

**Serve as a basis for enhancement.** Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued production evaluation.

**7. What should the SRS address?**

Again from the IEEE standard:

The basic issues that the SRS writer(s) shall address are the following:

a) **Functionality.** What is the software supposed to do?

b) **External interfaces.** How does the software interact with people, the system's hardware, other hardware and other software?

c) **Performance.** What is the speed, availability, response time, recovery time of various software functions, etc.?

d) **Attributes.** What is the portability, correctness, maintainability, security, etc. considerations?

e) **Design constraints imposed on an implementation.** Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

**8. What are the characteristics of a great SRS?**

Again from the IEEE standard:

An SRS should be

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

**Correct -** This is like motherhood and apple pie. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the specification up to date when you find things that are not correct.

**Unambiguous -** An SRS is unambiguous if, and only if, every requirement stated there in has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.

**Complete -** A simple judge of this is that is should be all that is needed by the software designers to create the software.

**Consistent -** The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.

**Ranked for Importance -** Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful provide this information in the SRS.

**Verifiable -** Don't put in requirements like - "It should provide the user a fast response." Another of my favorites is - "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."

**Modifiable -** Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.

**Traceable -** Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement?

**9. What Kind of Information Should an SRS Include?**

You probably will be a member of the SRS team (if not, ask to be), which means SRS development will be a collaborative effort for a particular project. In these cases, your company will have developed SRSs before, so you should have examples (and, likely, the company's SRS template) to use. But, let's assume you'll be starting from scratch. Several standards organizations (including the IEEE) have identified nine topics that must be addressed when designing and writing an SRS:

1. Interfaces

2. Functional Capabilities
3. Performance Levels
4. Data Structures/Elements
5. Safety
6. Reliability
7. Security/Privacy
8. Quality
9. Constraints and Limitations

But, how do these general topics translate into an SRS document? What, specifically, does an SRS document include? How is it structured? And how do you get started? An SRS document typically includes four ingredients, as discussed in the following sections:

1. A template
2. A method for identifying requirements and linking sources
3. Business operation rules
4. A traceability matrix

**Begin with an SRS Template**

The first and biggest step to writing an SRS is to select an existing template that you can fine tune for your organizational needs (if you don't have one already). There's not a "standard specification template" for all projects in all industries because the individual requirements that populate an SRS are unique not only from company to company, but also from project to project within any one company. The key is to select an existing template or specification to begin with and then adapt it to meet your needs.

In recommending using existing templates, I'm not advocating simply copying a template from available resources and using them as your own; instead, I'm suggesting that you use available templates as guides for developing your own. It would be almost impossible to find a specification or specification template that meets your particular project requirements exactly. But using other templates as guides is how it's recommended in the literature on specification development.

Look at what someone else has done, and modify it to fit your project requirements. Table 1 shows what a basic SRS outline might look like. This example is an adaptation and extension of the IEEE Standard 830-1998:

**Table 1** A sample of a basic SRS outline

| |
|---|
| **1. Introduction** |
| 1.1 Purpose |
| 1.2 Document conventions |
| 1.3 Intended audience |
| 1.4 Additional information |
| 1.5 Contact information/SRS team members |
| 1.6 References |
| **2. Overall Description** |
| 2.1 Product perspective |

2.2 Product functions
2.3 User classes and characteristics
2.4 Operating environment
2.5 User environment
2.6 Design/implementation constraints
2.7 Assumptions and dependencies
**3. External Interface Requirements**
3.1 User interfaces
3.2 Hardware interfaces
3.3 Software interfaces
3.4 Communication protocols and interfaces
**4. System Features**
4.1 System feature A
4.1.1 Description and priority
4.1.2 Action/result
4.1.3 Functional requirements
4.2 System feature B
**5. Other Nonfunctional Requirements**
5.1 Performance requirements
5.2 Safety requirements
5.3 Security requirements
5.4 Software quality attributes
5.5 Project documentation
5.6 User documentation
**6. Other Requirements**
Appendix A: Terminology/Glossary/Definitions list
Appendix B: To be determined

Table 2 shows a more detailed SRS outline

**Table 2** A sample of a more detailed SRS outline

| | |
|---|---|
| **1. Scope** | 1.1 Identification.<br>*Identify the system and the software to which this document applies, including, as applicable, identification number(s), title(s), abbreviation(s), version number(s), and release*<br>*number(s).*<br>1.2 System overview.<br>*State the purpose of the system or subsystem to which this document applies.*<br>1.3 Document overview.<br>*Summarize the purpose and contents of this document.*<br>This document comprises six sections: |

| | |
|---|---|
| | • Scope<br>• Referenced documents<br>• Requirements<br>• Qualification provisions<br>• Requirements traceability<br>• Notes<br>Describe any security or privacy considerations associated with its use. |
| **2. Referenced Documents** | 2.1 Project documents.<br>*Identify the project management system documents here.*<br>2.2 Other documents.<br>2.3 Precedence.<br>2.4 Source of documents. |
| **3. Requirements** | This section shall be divided into paragraphs to specify the Computer Software Configuration Item (CSCI) requirements, that is, those characteristics of the CSCI that are conditions for its acceptance. CSCI requirements are software requirements generated to satisfy the system requirements allocated to this CSCI. Each requirement shall be assigned a project-unique identifier to support testing and traceability and shall be stated in such a way that an objective test can be defined for it.<br>3.1 Required states and modes.<br>3.2 CSCI capability requirements.<br>3.3 CSCI external interface requirements.<br>3.4 CSCI internal interface requirements.<br>3.5 CSCI internal data requirements.<br>3.6 Adaptation requirements.<br>3.7 Safety requirements.<br>3.8 Security and privacy requirements.<br>3.9 CSCI environment requirements.<br>3.10 Computer resource requirements.<br>3.11 Software quality factors.<br>3.12 Design and implementation constraints.<br>3.13 Personnel requirements.<br>3.14 Training-related requirements.<br>3.15 Logistics-related requirements.<br>3.16 Other requirements.<br>3.17 Packaging requirements.<br>3.18 Precedence and criticality requirements. |

| | |
|---|---|
| **4. Qualification Provisions** | To be determined. |
| **5. Requirements Traceability** | To be determined. |
| **6. Notes** | This section contains information of a general or explanatory nature that may be helpful, but is not mandatory.<br>6.1 Intended use.<br>This Software Requirements specification shall...<br>6.2 Definitions used in this document.<br>*Insert here an alphabetic list of definitions and their source if different from the declared sources specified in the "Documentation standard."*<br>6.3 Abbreviations used in this document.<br>*Insert here an alphabetic list of the abbreviations and acronyms if not identified in the declared sources specified in the "Documentation Standard."*<br>6.4 Changes from previous issue.<br>*Will be "not applicable" for the initial issue.*<br>Revisions shall identify the method used to identify changes from the previous issue. |

**Identify and Link Requirements with Sources**

As noted earlier, the SRS serves to define the functional and nonfunctional requirements of the product. Functional requirements each have an origin from which they came, be it a *use case* (which is used in system analysis to identify, clarify and organize system requirements and consists of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal), government regulation, industry standard or a business requirement. In developing an SRS, you need to identify these origins and link them to their corresponding requirements. Such a practice not only justifies the requirement, but it also helps assure project stakeholders that frivolous or spurious requirements are kept out of the specification.

To link requirements with their sources, each requirement included in the SRS should be labeled with a unique identifier that can remain valid over time as requirements are added, deleted or changed. Such a labeling system helps maintain change-record integrity while also serving as an identification system for gathering metrics. You can begin a separate requirements identification list that ties a requirement identification (ID) number with a description of the requirement. Eventually, that requirement ID and description become part of the SRS itself and then part of the Requirements Traceability Matrix, discussed in subsequent paragraphs. Table 3 illustrates how these SRS ingredients work together.

**Table 3** This sample table identifies requirements and links them to their sources

| ID No. | Paragraph No. | Requirement | Business Rule Source |
|---|---|---|---|
| 17 | 5.1.4.1 | Understand/communicate using SMTP protocol | IEEE STD xx-xxxx |
| 18 | 5.1.4.1 | Understand/communicate using POP protocol | IEEE STD xx-xxxx |
| 19 | 5.1.4.1 | Understand/communicate using IMAP protocol | IEEE STD xx-xxxx |
| 20 | 5.1.4.2 | Open at same rate as OE | Use Case Doc 4.5.4 |

**Establish Business Rules for Contingencies and Responsibilities**

"The best-laid plans of mice and men..." begins the famous saying. It has direct application to writing SRSs because even the most thought-out requirements are not immune to changes in industry, market or government regulations. A top-quality SRS should include plans for planned and unplanned contingencies, as well as an explicit definition of the responsibilities of each party, should a contingency be implemented. Some business rules are easier to work around than others, when Plan B has to be invoked. For example, if a customer wants to change a requirement that is tied to a government regulation, it may not be ethical and/or legal to be following "the spirit of the law." Many government regulations, as business rules, simply don't allow any compromise or "wiggle room." A project manager may be responsible for ensuring that a government regulation is followed as it relates to a project requirement; however, if a contingency is required, then the responsibility for that requirement may shift from the project manager to a regulatory attorney. The SRS should anticipate such actions to the furthest extent possible.

**Establish a Requirements Traceability Matrix**

The business rules for contingencies and responsibilities can be defined explicitly within a Requirements Traceability Matrix (RTM) or contained in a separate document and referenced in the matrix, as the example in Table 3 illustrates. Such a practice leaves no doubt as to responsibilities and actions under certain conditions as they occur during the product-development phase.

The RTM functions as a sort of "chain of custody" document for requirements and can include pointers to links from requirements to sources, as well as pointers to business rules. For example, any given requirement must be traced back to a specified need, be it a use case, business essential, industry-recognized standard, or government regulation. As mentioned previously, linking requirements with sources minimizes or even eliminates the presence of spurious or frivolous requirements that lack any justification. The RTM is another record of mutual understanding, but also helps during the development phase.

As software design and development proceed, the design elements and the actual code must be tied back to the requirement(s) that define them. The RTM is completed as development progresses; it can't be completed beforehand (see Table 3).

**What Should I Know about Writing an SRS?**

Unlike formal language that allows developers and designers some latitude, the natural language of SRSs must be exact, without ambiguity and precise because the design specification, statement of work and other project documents are what drive the development of the final product. That final product must be tested and validated against the design and original requirements. Specification language that allows for interpretation of key requirements will not yield a satisfactory final product and will likely lead to cost overruns, extended schedules and missed deliverable deadlines.

Table 4 shows the fundamental characteristics of a quality SRS.

**Table 4** The 10 language quality characteristics of an SRS

| **SRS Quality Characteristic** | **What It Means** |
|---|---|
| **Complete** | SRS defines precisely all the go-live situations that will be encountered and the system's capability to successfully address them. |
| **Consistent** | SRS capability functions and performance levels are compatible, and the required quality features (security, reliability, etc.) do not negate those capability functions. |
| **Accurate** | SRS precisely defines the system's capability in a real-world environment, as well as how it interfaces and interacts with it. This aspect of requirements is a significant problem area for many SRSs. |
| **Modifiable** | The logical, hierarchical structure of the SRS should facilitate any necessary modifications (grouping related issues together and separating them from unrelated issues makes the SRS easier to modify). |
| **Ranked** | Individual requirements of an SRS are hierarchically arranged according to stability, security, perceived ease/difficulty of implementation or other parameter that helps in the design of that and subsequent documents. |
| **Testable** | An SRS must be stated in such a manner that unambiguous assessment criteria (pass/fail or some quantitative measure) can be derived from the SRS itself. |

| | |
|---|---|
| **Traceable** | Each requirement in an SRS must be uniquely identified to a source (use case, government requirement, industry standard, etc.) |
| **Unambiguous** | SRS must contain requirements statements that can be interpreted in one way only. This is another area that creates significant problems for SRS development because of the use of natural language. |
| **Valid** | A valid SRS is one in which all parties and project participants can understand, analyze, accept or approve it. This is one of the main reasons SRSs are written using natural language. |
| **Verifiable** | A verifiable SRS is consistent from one level of abstraction to another. Most attributes of a specification are subjective and a conclusive assessment of quality requires a technical review by domain experts. Using indicators of strength and weakness provide some evidence that preferred attributes are or are not present. |

What makes an SRS "good?" How do we know when we've written a "quality" specification? The most obvious answer is that a quality specification is one that fully addresses all the customer requirements for a particular product or system. That's part of the answer. While many quality attributes of an SRS are subjective, we do need indicators or measures that provide a sense of how strong or weak the language is in an SRS. A "strong" SRS is one in which the requirements are tightly, unambiguously, and precisely defined in such away that leaves no other interpretation or meaning to any individual requirement.

**Table 5** Quality measures related to individual SRS statements

| *Imperatives*: Words and phrases that command the presence of some feature, function, or deliverable. They are listed below in decreasing order of strength. | |
|---|---|
| **Shall** | Used to dictate the provision of a functional capability. |
| **Must or must not** | Most often used to establish performance requirement or constraints. |
| **Is required to** | Used as an imperative in SRS statements when written in passive voice. |
| **Are applicable** | Used to include, by reference, standards or other |

| | |
|---|---|
| | documentation as an addition to the requirement being specified. |
| **Responsible for** | Used as an imperative in SRSs that are written for systems with pre-defined architectures. |
| **Will** | Used to cite things that the operational or development environment is to provide to the capability being specified. For example, The vehicle's exhaust system will power the ABC widget. |
| **Should** | Not used often as an imperative in SRS statements; however, when used, the SRS statement always reads weak. Avoid using Should in your SRSs. |

*Continuances*: Phrases that follow an imperative and introduce the specification of requirements at a lower level. There is a correlation with the frequency of use of *continuances* and SRS organization and structure, up to a point. Excessive use of *continuances* often indicates a very complex, detailed SRS.
The *continuances* below are listed in decreasing order of use within NASA SRSs. Use *continuances* in your SRSs, but balance the frequency with the appropriate level of detail called for in the SRS.

**1. Below:**
**2. As follows:**
**3. Following:**
**4. Listed:**
**5. In particular:**
**6. Support:**

*Directives*: Categories of words and phrases that indicate illustrative information within the SRS. A high ratio of total number of *directives* to total text line count appears to correlate with how precisely requirements are specified within the SRS. The *directives* below are listed in decreasing order of occurrence within NASA SRSs. Incorporate the use of *directives* in your SRSs.

**1. Figure**
**2. Table**
**3. For example**
**4. Note**

*Options*: A category of words that provide latitude in satisfying the SRS statements that contain them. This category of words loosens the SRS, reduces the client's control over the final product and allows for possible cost and schedule risks. You should avoid using them in your SRS. The *options* below are listed in the order

they are found most often in NASA SRSs.

**1. Can**
**2. May**
**3. Optionally**

*Weak phrases*: A category of clauses that can create uncertainty and multiple/subjective interpretation. The total number of *weak phrases* found in an SRS indicates the relative ambiguity and incompleteness of an SRS. The *weak phrases* below are listed alphabetically.

| | | | |
|---|---|---|---|
| **Adequate** | **be able to** | **easy** | **provide for** |
| **as a minimum** | **be capable of** | **effective** | **timely** |
| **as applicable** | **but not limited to** | **if possible** | **tbd** |
| **as appropriate** | **capability of** | **if practical** | |
| **at a minimum** | **capability to** | **normal** | |

*Size:* Used to indicate the *size* of the SRS document, and is the total number of the following:

**1. Lines of text**
**2. Number of imperatives**
**3. Subjects of SRS statements**
**4. Paragraphs**

*Text Structure*: Related to the number of statement identifiers found at each hierarchical level of the SRS and indicate the document's organization, consistency and level of detail. The most detailed NASA SRSs were nine levels deep. High-level SRSs were rarely more than four levels deep. SRSs deemed well organized and a consistent level of detail had *text structures* resembling pyramids (few level 1 headings but each lower level having more numbered statements than the level above it). Hour-glass-shaped *text structures* (many level 1 headings, few a mid-levels, and many at lower levels) usually contain a greater amount of introductory and administrative information. Diamond-shaped *text structures* (pyramid shape followed by decreasing statement counts at levels below the pyramid) indicated that subjects introduced at higher levels were addressed at various levels of detail.

*Specification Depth*: The number of imperatives found at each of the SRS levels of text structure. These numbers include the count of lower level list items that are

| |
|---|
| introduced at a higher level by an imperative and followed by a continuance. The numbers provide some insight into how much of the Requirements document was included in the SRS, and can indicate<br>how concise the SRS is in specifying the requirements. |
| *Readability Statistics*: Measurements of how easily an adult can read and understand the requirements document. Four readability statistics are used (calculated by Microsoft Word). While readability statistics provide a relative quantitative measure, don't sacrifice sufficient technical depth in your SRS for a number.<br>**1. Flesch Reading Ease index**<br>**2. Flesch-Kincaid Grade Level index**<br>**3. Coleman-Liau Grade Level index**<br>**4. Bormuth Grade Level index** |

**10. Conclusion**

There's so much more we could say about requirements and specifications. Hopefully, this information will help you get started when you are called upon--or step up--to help the development team. Writing top-quality requirements specifications begins with a complete definition of customer requirements. Coupled with a natural language that incorporates strength and weakness quality indicators--not to mention the adoption of a good SRS template--technical communications professionals well-trained in requirements gathering, template design, and natural language use are in the best position to create and add value to such critical project documentation.

**11. Summary**

In this lesson we have discussed the requirement analysis phase of software engineering, which is very significant to study the requirements of client. We have also studied about the software requirement specification, which is a legal document signed by both customer and developer before the start of the project.

**12. Self Check Exercise**

1. What is requirements analysis in software engineering? What are its benefits?
2. What are the different ways of analyzing the requirement of the customer?
3. Define SRS. What are the features of SRS? What are the types of SRS in software engineering?

**13. Suggested readings**

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Testing Techniques" by Boris Beizer, Dreamtech Press, New Delhi.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

# Software Design

**Objectives**



**Objectives:**

This lesson provides an introduction to the principles and concepts relevant to the software design. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations and main domains of application of these methods.

## 1.1 Introduction: Software Design

**Software design is a process of defining the architecture, components, interfaces, and other characteristics of a system or component and planning for a software solution.** After the purpose and specifications of software is determined, software developers will design or employ designers to develop a plan for a solution. A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design. Viewed as a process, software design

is the software engineering life cycle activity in which software requirements are analyzed in order to produce a description of the software's internal structure that will serve as the basis for its construction. More precisely, a software design (the result) must describe the software architecture and the interfaces between those components. It must also describe the components at a level of detail that enable their construction.

Software design plays an important role in developing software: it allows software engineers to produce various models that form a kind of blueprint of the solution to be implemented. We can analyze and evaluate these models to determine whether or not they will allow us to fulfill the various requirements. We can also examine and evaluate various alternative solutions and trade-offs. Finally, we can use the resulting models to plan the subsequent development activities, in addition to using them as input and the starting point of construction and testing.

**In a standard listing of software life cycle processes such as IEEE/EIA 12207 Software Life Cycle Processes, software design consists of two activities that fit between software requirements analysis and software construction:**

- **Software architectural design (sometimes called top-level design): describing software's top-level structure and organization and identifying the various components.**
- **Software detailed design: describing each component sufficiently to allow for its construction.**

***Software Design: What is it and why is it important?***

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

The design process is very important. From a practical standpoint, as a labourer, one would not attempt to build a house without an approved blueprint thereby risking the structural integrity and customer satisfaction. In the same manner, the approach to building software products is no different. The emphasis in design is on quality; this phase provides us with representation of software that can be assessed for quality. Furthermore, this is the only phase in which the customer's requirements can be accurately translated into a finished software product or system. As such, software design serves as the foundation for all software engineering steps that follow regardless of which process model is being employed. Without a proper design we risk building an unstable system – one that will fail when small changes are made, one that may be difficult to test; one whose quality cannot be assessed until late in the software process, perhaps when critical deadlines are approaching and much capital has already been invested into the product.

During the design process the software specifications are transformed into design models that describe the details of the data structures, system architecture, interface and components. Each design product is reviewed for quality before moving to the next phase

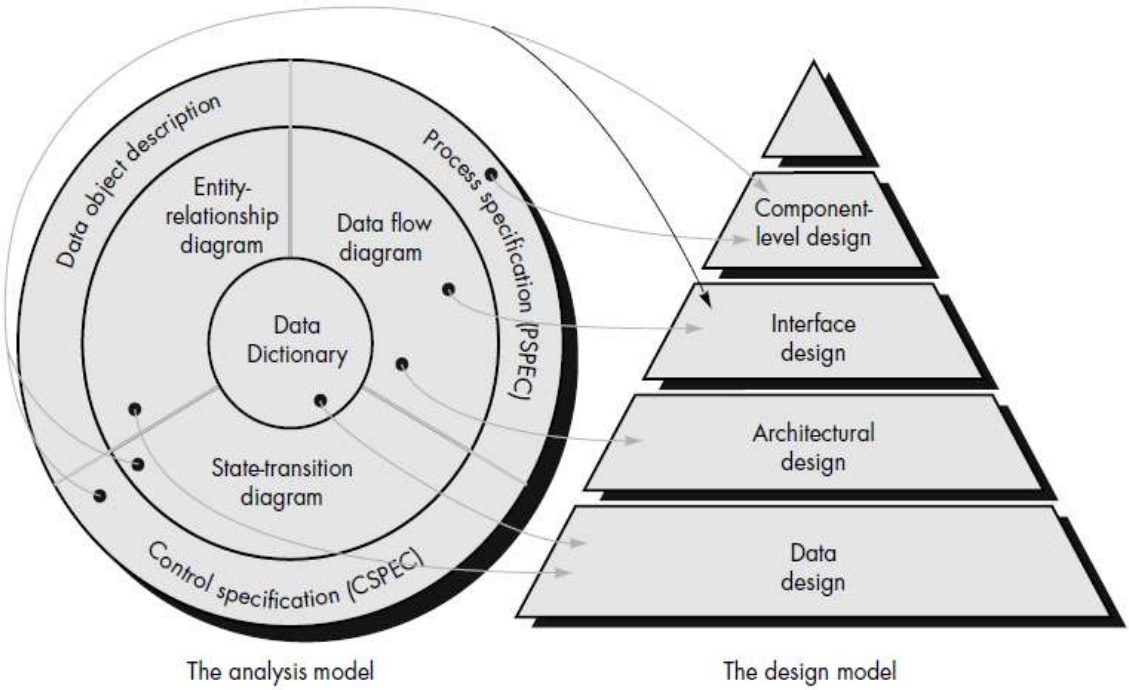
of software development. At the end of the design process a design specification document is produced. This document is composed of the design models that describe the data, architecture, interfaces and components.

At the data and architectural levels the emphasis is placed on the patterns as they relate to the application to be built. Whereas at the interface level, human ergonomics often dictate the design approach employed. Lastly, at the component level the design is concerned with a "programming approach" which leads to effective data and procedural designs.

**Design Specification Models**

- **Data design** – created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software. Part of the data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.
- **Architectural design** - defines the relationships among the major structural elements of the software, the "design patterns" than can be used to achieve the requirements that have been defined for the system and the constraints that affect the way in which the architectural patterns can be applied. It is derived from the system specification, the analysis model and the subsystem interactions defined in the analysis model (DFD).
- **Interface design** - describes how the software elements communicate with each other, with other systems, and with human users; the data flow and control flow diagrams provide much of the necessary information required.
- **Component-level design** - created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the process specification (PSPEC), control specification (CSPEC), and state transition diagram (STD).

These models collectively form the design model, which is represented diagrammatically as a pyramid structure with data design at the base and component level design at the pinnacle. Note that each level produces its own documentation, which collectively form the design specifications document, along with the guidelines for testing individual modules and the integration of the entire package. Algorithm description and other relevant information may be included as an appendix.




**1.2 Design Guidelines**

In order to evaluate the quality of a design (representation) the criteria for a good design should be established. Such a design should:

- exhibit good architectural structure
- be modular
- contain distinct representations of data, architecture, interfaces, and components (modules)
- lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns
- lead to components that exhibit independent functional characteristics
- lead to interfaces that reduce the complexity of connections between modules and with the external environment
- be derived using a reputable method that is driven by information obtained during software requirements analysis

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology and through review.

**1.3 Design Principles**

Software design can be viewed as both a process and a model. "The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. However, it is not merely a cookbook; for a competent and successful design, the designer must use creative skill, past experience, a sense of what makes "good" software and have a commitment to quality.

The design model is equivalent to the architect's plans for a house. It begins by representing the totality of the entity to be built (e.g. a 3D rendering of the house), and slowly refines the entity to provide guidance for constructing each detail (e.g. the plumbing layout). Similarly the design model that is created for software provides a variety of views of the computer software." – adapted from book by R Pressman.

The set of principles which has been established to aid the software engineer in navigating the design process are:

1. **The design process should not suffer from tunnel vision** – A good designer should consider alternative approaches. Judging each based on the requirements of the problem, the resources available to do the job and any other constraints.
2. **The design should be traceable to the analysis model** – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model
3. **The design should not reinvent the wheel** – Systems are constructed using a set of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.
4. **The design should minimise intellectual distance between the software and the problem as it exists in the real world** – That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
5. **The design should exhibit uniformity and integration** – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.
6. **The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered** – Well-designed software should never "bomb". It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
7. **The design should be reviewed to minimize conceptual (semantic) errors** – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax if the design model.
8. **Design is not coding, coding is not design** – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made of the coding level address the small implementation details that enable the procedural design to be coded.
9. **The design should be structured to accommodate change**
10. **The design should be assessed for quality as it is being created**

When these design principles are properly applied, the design exhibits both external and internal quality factors. External quality factors are those factors that can readily be observed by the user, (e.g. speed, reliability, correctness, usability). Internal quality factors relate to the technical quality (which is important to the software engineer) more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

## 1.4 Design Concepts

A set of fundamental software design concepts has evolved over the past four decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps the software engineer to answer the following questions:

- What criteria can be used to partition software into individual components?
- How function or data is structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work and getting it right". Fundamental software design concepts provide the necessary framework for "getting it right."

### 1.4.1 Abstraction

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction **door**.

### 1.4.2 Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth. A program is developed by successively refining levels of procedural detail. A

hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

### 1.4.3 Modularity

The concept of modularity in computer software has been espoused for almost five decades. Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called modules that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables and overall complexity would make understanding close to impossible.

An important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system. Meyer defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

**Modular decomposability.** If a design method provides a systematic mechanism for decomposing the problem into subproblems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

**Modular composability.** If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

**Modular understandability.** If a module can be understood as a standalone unit (without reference to other modules), it will be easier to build and easier to change.

**Modular continuity.** If small changes to the system requirements result in changes to individual modules, rather than system wide changes, the impact of change-induced side effects will be minimized.

**Modular protection**. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Finally, it is important to note that a system may be designed modularly, even if its implementation must be "monolithic." There are situations (e.g., real-time software, embedded software) in which relatively minimal speed and memory overhead introduced by subprograms (i.e., subroutines, procedures) is unacceptable. In such situations, software can and should be designed with modularity as an overriding philosophy. Code may be developed "in-line." Although the program source code may not look modular at first glance, the philosophy has been maintained and the program will provide the benefits of a modular system.

### 1.4.4 Software Architecture

Software architecture alludes to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”. In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design level concepts.

### 1.4.5 Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions or repetition of operations; nor is it necessarily applicable to all architectural styles.

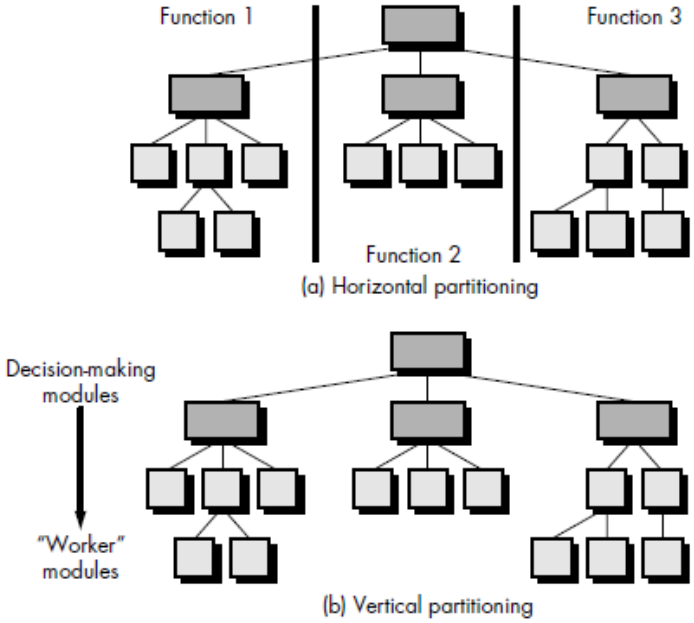
Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the treelike diagram that represents hierarchical control for call and return architectures.

### 1.4.6 Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure, horizontal partitioning defines separate branches of the modular hierarchy for each major program function. Control modules, represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called processing) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

- software that is easier to test
- software that is easier to maintain

- propagation of fewer side effects
- software that is easier to extend




Because major functions are decoupled from one another, change tends to be less complex and extensions to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

*Vertical partitioning* (Figure b), often called *factoring,* suggests that control (decision making) and work should be distributed top-down in the program structure. Top level modules should perform control functions and do little actual processing work. Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

The nature of change in program structures justifies the need for vertical partitioning. Referring to Figure b, it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output. The overall control structure of the program (i.e., its basic behavior is far less likely to change). For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable—a key quality factor.

### 1.4.7 Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture. Data structure dictates the organization, methods of access, degree of associativity and processing alternatives for information. The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A scalar item is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.

When scalar items are organized as a list or contiguous group, a sequential vector is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.

When the sequential vector is extended to two, three and ultimately, an arbitrary number of dimensions, an n-dimensional space is created. The most common n-dimensional space is the two-dimensional matrix. In many programming languages, an ndimensional space is called an array.

Items, vectors, and spaces may be organized in a variety of formats. A linked list is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called nodes) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

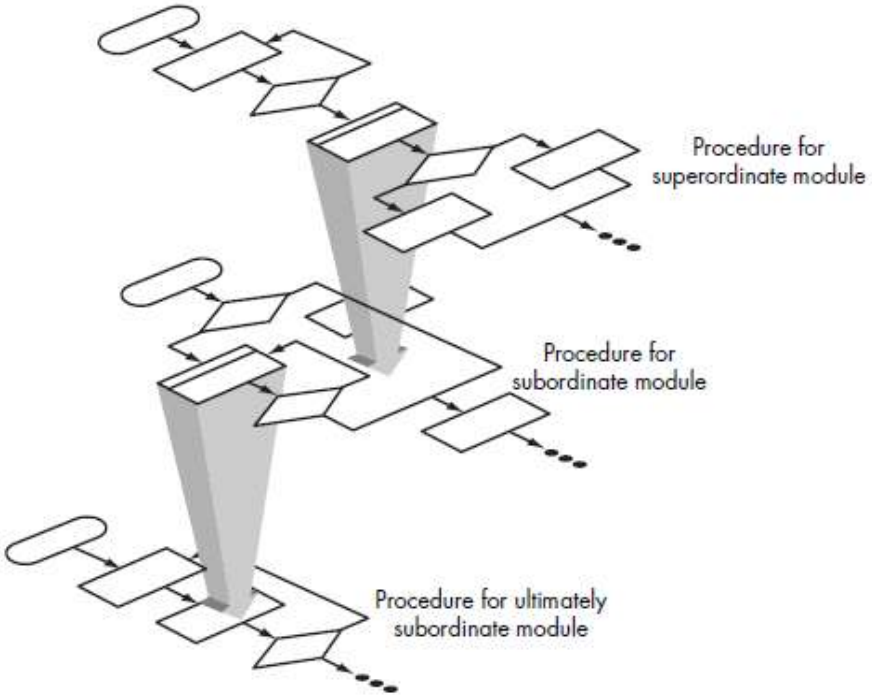
Other data structures incorporate or are constructed using the fundamental data structures just described. For example, a hierarchical data structure is implemented using multilinked lists that contain scalar items, vectors and possibly, n-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and associativity.

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of a **stack** may or may not be specified.

### 1.4.8 Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations and even data organization and structure.

There is, of course, 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. That is, a procedural representation of software is layered as illustrated in Figure below.




### 1.4.9 Information Hiding

The concept of modularity leads every software designer to a fundamental question: "How do we decompose a software solution to obtain the best set of modules?" **The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others."** In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The

use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

## 1.5 Effective Modular Design

All the fundamental design concepts described in the preceding section serve to precipitate modular designs. In fact, modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity, facilitates change (a critical aspect of software maintainability) and results in easier implementation by encouraging parallel development of different parts of a system.

### 1.5.1 Functional Independence

The concept of functional independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In landmark papers on software design Parnas and Wirth allude to refinement techniques that enhance module independence. Later work by Stevens, Myers and Constantine solidified the concept. **Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules**. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design and design is the key to software quality.

Independence is measured using two qualitative criteria: cohesion and coupling. Cohesion is a measure of the relative functional strength of a module. Coupling is a measure of the relative interdependence among modules.

### 1.5.2 Cohesion

**Cohesion is a natural extension of the information hiding concept**. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed coincidentally cohesive. A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is logically cohesive. When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits temporal cohesion.

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bounds. It performs the following tasks: (1) computes supplementary data based on original computed data, (2) produces an error report (with graphical content) on the user's workstation, (3) performs follow-up calculations requested by the user, (4) updates a database, and (5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, procedural cohesion exists. When all processing elements concentrate on one area of a data structure, communicational cohesion is present. High cohesion is characterized by a module that performs one distinct procedural task.

As we have already noted, it is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

### 1.5.3 Coupling

**Coupling is a measure of interconnection among modules in a software structure**. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate through a system.

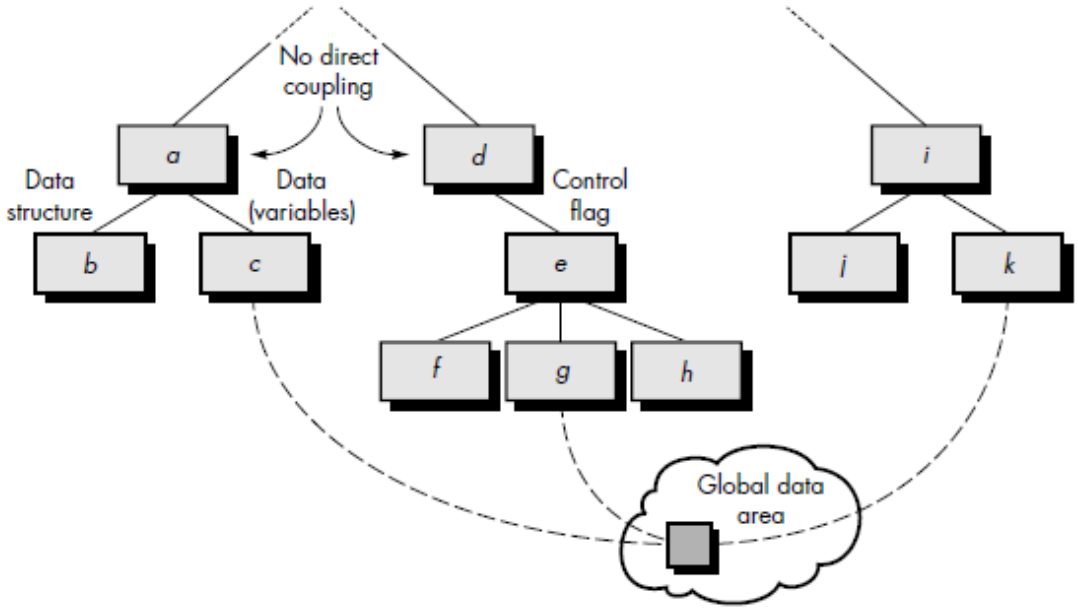



**Figure: Types of Coupling**

Figure provides examples of different types of module coupling. Modules a and d are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module c is subordinate to module a and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called data coupling) is exhibited in this portion of structure. A variation of data coupling, called stamp coupling, is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules b and a. At moderate levels, coupling is characterized by passage of control between modules. Control coupling is very common in most software designs and is shown in Figure where a “control flag” (a variable that controls decisions in a subordinate or superordinate module) is passed between modules d and e. Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. External coupling is essential, but should be limited to a small number of modules with a structure. High coupling also occurs when a number of modules reference a global data area. Common coupling, as this mode is called, is shown in Figure. Modules c, g, and k each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module c initializes the item. Later module g recomputes and updates the item. Let's assume that an error occurs and g updates the item incorrectly. Much later in processing module, k reads the item, attempts to process it and fails, causing the software to abort. The apparent cause of abort is module k; the actual cause, module g. Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software

designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, content coupling, occurs when one module makes use of data or control information maintained within the boundary of another module. Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided. The coupling modes just discussed occur because of design decisions made when structure was developed. Variants of external coupling, however, may be introduced during coding. For example, compiler coupling ties source code to specific (and often nonstandard) attributes of a compiler; operating system (OS) coupling ties design and resultant code to operating system "hooks" that can create havoc when OS changes occur.

## 1.6 Summary

In this lesson we have discussed the principles and concepts relevant to the software design. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

## 1.7 Self Check Exercise

1. What is software design in software engineering? What is objective of software design?
2. Explain various principles of software design in detail.
3. What is cohesion and coupling in software engineering? What is the difference between these two?

## 1.8 Suggested readings

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Testing Techniques" by Boris Beizer, Dreamtech Press, New Delhi.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

LESSON No. 1.7 AUTHOR: Dr. VISHAL SINGH

# DFD and UML

**Objectives**

**1. Introduction**

**2. Data flow diagram**

**3. Data dictionary**

**4. Object Modeling Using UML**

**5. Advantages of OOD**

**Summary**

**Self Check Exercise**

**Suggested Readings**

**Objectives:**

In this lesson we will learn the data flow diagram, data dictionary etc. we will also study about the Unified Modeling Language.

## 1. Introduction

The Data Flow Diagram is also known as a Data Flow Graph or a Bubble Chart. A DFD serve the purpose of clarifying system requirements and identifying major transformations. DFDs show the flow of data through a system. It is an important modeling tool that allows us to picture a system as a network of functional process.

## 2. Data flow diagrams

Data flow diagrams are well known and widely used notations for specifying the functions of an information system. They describe system as collections of data that are manipulated by functions. Data can be organized in several ways: they can be stored in data repositories, they can flow in data flows and they can be transformed to or from the external environment.

One of the reasons for the success of DFD is that they can be expressed by means of an attractive graphical notation that makes them easy to use.

**Symbols used for constructing DFDs**

There are different types of symbols used to construct DFDs. The meaning of each symbol is explained below:

**1. Functional symbol:**

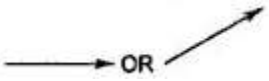
A Function is represented using a circle. This symbol is called a process or a bubble or performs some processing of input data.

**2. External entity**

A square defines a source of destination of system data. External entities represent any entity that supplies or receives information from the system but is not a part of the system.

**3. Data flow symbol**

A directed arc or arrow is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of data flow arrow.




**4. Data store symbol**

A data store symbol is represented using two parallel lines. A logical file can represent either a data store symbol, which can represent either a data structure or a physical file on the disk. Each data store is connected to a process by means of a data flow symbol. The direction of data flow arrow shows whether data is being read from or written into a data store.

**5. Output symbol**

It is used to represent data acquisition and production during human computer interaction.

**Example of DFD**

Figure 1 shows how the symbols can be composed to form a DFD. The DFD describes the arithmetic expression

( a + b ) * ( c + a + d )

Assuming that the data a,b,c and d are read from a terminal and the result is printed. The figure shows that the arrow can be "forked" to represent the fact that the same datum is used in different places.

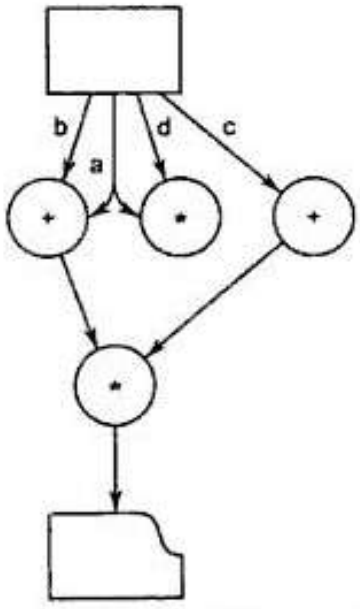



**Figure 1**

Figure 2 describes a simplified information system for a public library. The data and functions shown are not necessarily computer data and computer functions. The DFD describes physical objects, such as books and shelves together with data stores that are likely to be but not necessarily realized as computer files. Getting a book from the shelf can be done either automatically by a robot or manually. In both cases the action of getting a book is represented by a function depicted by a bubble. The figure could even represent the organization of a library with no computerized procedures.

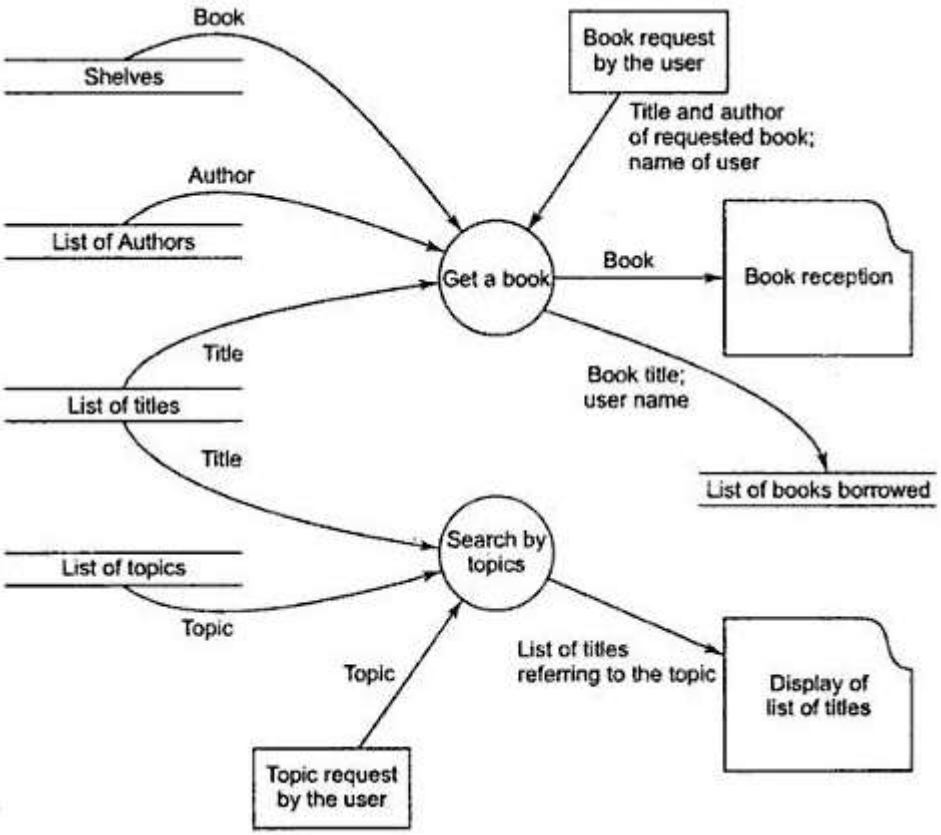



**Figure 2**

Figure describes the fact that in order to obtain a book, the following are necessary: an explicit user request consisting of the title and the name of the author of the book and the users name; access to the shelves that contain the books; a list of authors; and a list of titles. These provide the information necessary to find the book.

**General guidelines and rules for constructing DFD**

The following guidelines will help avoid constructing DFDs that are quite simply wrong or incomplete.

- Remember that a DFD is not a flow chart.
- All names should be unique.
- Processes are always running, they do not start or stop.
- All data flows are named.
- Do numbering of processes
- Avoid complex DFD (if possible)
- The DFD should be internally consistent.
- Keep a note of all the processes and external entities. Give unique names to them. Identify the manner in which they interact with each other.

- Every process should have minimum of one input and one output.
- The direction of flow is from source to destination.
- Only data needed to perform the process should be an input to the process.

**3. Data dictionary**

Every DFD model of a system must be accompanied by a data dictionary. **A data dictionary lists all data items appearing in the DFD model of a system**. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in the DFD model of a system.

A data dictionary lists the purpose of all data items and the definition of all composite data items In terms of their component data items. For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.

**grossPay = regularPay + overtimePay**

For the smallest units of data items, the data dictionary lists their name and their type. The operators using a composite data item can be expressed in terms of their component data items, and these are discussed in the next subsection.

- A data dictionary provides a standard terminology for all relevant data for use by all engineers working in the same project. A consistent vocabulary or data items is very important, since in large projects different engineers have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

For large systems, the data dictionary can be extremely complex and voluminous. Even moderate-sized projects can have thousands of entries in the data dictionary. It becomes extremely difficult to maintain a voluminous dictionary manually. Computer Aided Software Engineering (CASE) tools come handy to overcome this problem. Most CASE tools usually capture the data items appearing in a DFD and automatically generate the data dictionary. These tools also support some query language facility to raise queries about the definition and usage of data items. For example, queries may be formulated either to determine which data item affects which processes, which process affects which data items or to find the definition and usage of specific data items and so on. Query handling is facilitated by storing the data dictionary in a Relational Database Management System (RDBMS).

**4. Object Modelling Using UML**

The object-oriented software development style has become extremely popular. Since its inception in the early 1980s, the object technology has made rapid progress. The development of this technology gathered momentum in the 1990s and is now nearing maturity. In this Lesson, we will review some basic concepts in object-orientation. We will then discuss UML (Unified Modelling Language) which is poised to become a standard in

modelling object-oriented systems.

Before starting our discussion on object modelling using UML, we will first briefly review some basic concepts that are central to the object-oriented approach.

**Overview of Object-Oriented Concepts**

Some important concepts and terms related to the object-oriented approach are pictorially shown in Figure 3. We will first discuss the basic mechanisms shown in the figure, namely objects, classes, inheritance, messages and methods. We will then discuss some key concepts and examine a few related technical terms.

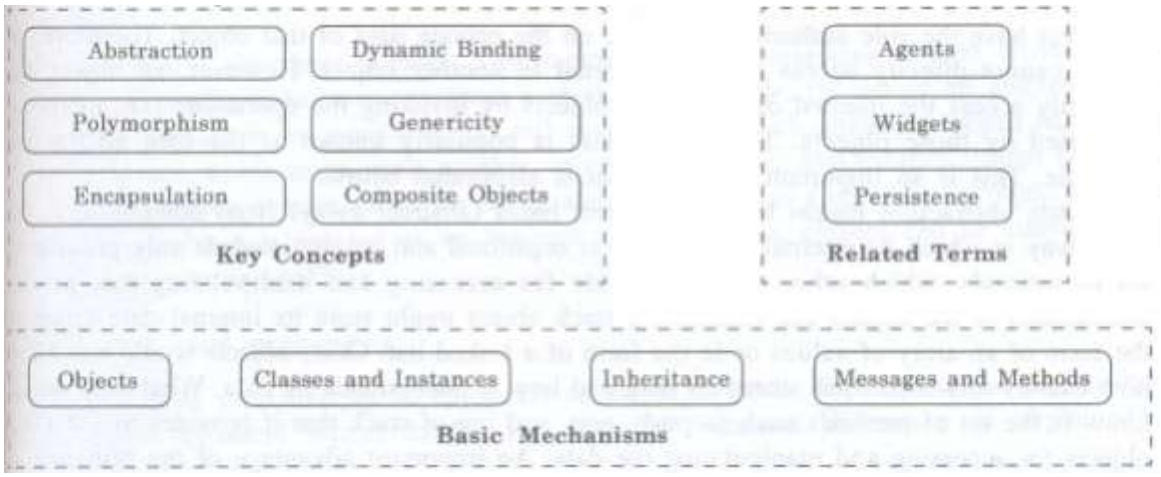



**Figure 3: basic concepts used in object-oriented approach**

**Basic Mechanisms**

The following are some of the basic mechanisms used in the object-oriented paradigm.

**Objects**

In the object-oriented approach, a system is designed as a set of interacting objects, Normally, each object represents a tangible real-world entity such as a library member, an employee, a book, etc. However, at times we would also consider some conceptual entities as objects (e.g. a scheduler, a controller, etc.) to simplify solution to certain problems.

When a system is analyzed, developed and implemented in terms of the natural objects occurring in it, it becomes easier to understand the design and implementation of the system. Each object essentially consists of some data that are private to the object and a set of functions (or operations) that operate on those data (see Figure 4). In fact, the functions of an object have the sole authority to operate on the private data of that object. Therefore, an object cannot directly access the data internal to another object.

However, an object can indirectly access the internal data of other objects by invoking the operations (i.e. methods) supported by those objects. This mechanism is popularly known as the data abstraction principle. This is an important principle that is elaborated below.

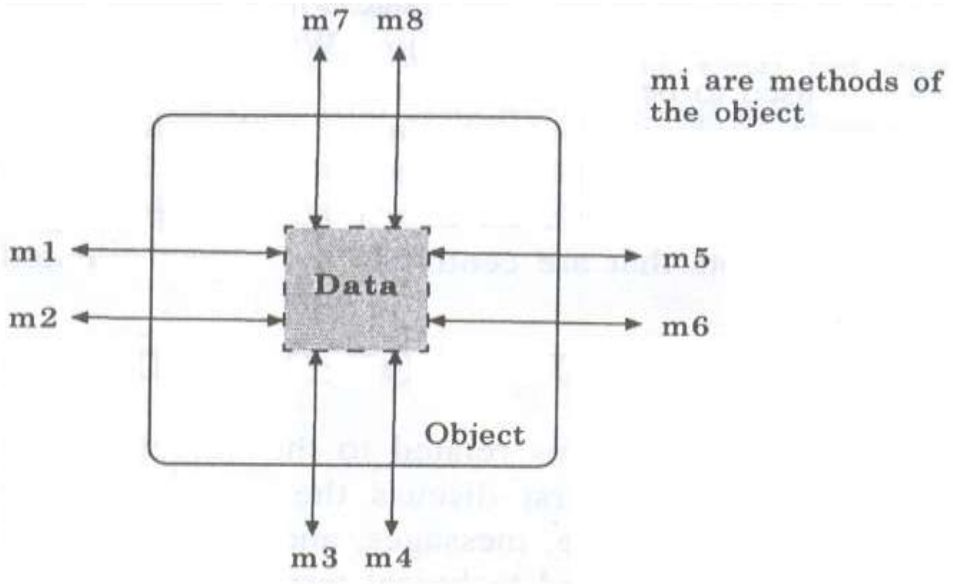



**Figure 4: A model of an object**

**Data abstraction** means that each object hides from other objects the exact way in which its internal information is organized and manipulated. It only provides a set of methods, which other objects can use for accessing and manipulating this private information of the object. For example, a stack object might store its internal data either in the form of an array of values or in the form of a linked list. Other objects would not know how exactly this object has stored its data and how it manipulates its data. What they would know is the set of methods such as push, pop, and top-of-stack that it provides to the other objects for accessing and manipulating the data. An important advantage of the principle of data abstraction is that it reduces coupling among the objects, decreases the overall complexity of a design and helps in maintenance and code reuse.

Each object essentially possesses certain information and supports some operations on this information. For example, in a word processing application, a paragraph, a line, and a page can be objects. A LibraryMember can be an object of a library automation system. The private data of such a library member object can be:

- Name of the member
- Membership number
- Address
- Phone number
- E-mail number
- Date when admitted as a member

- Membership expiry date
- Books outstanding, etc

The operations supported by a LibraryMember object can be:

- Issue-books
- find-books-outstanding
- find-books-overdue
- return-book
- find-membership-details, etc.

The data internal to an object are often called the ***attributes*** of the object and the functions supported by an object are called its ***methods***. A word of caution here-the term object is often used vaguely in the literature and also in this text. Sometimes an object means a single entity, other times it refers to a group of similar objects. In this text, usually the context would resolve the ambiguity, if any.

**Class**

Similar objects constitute a class. This means, objects possessing similar attributes and displaying similar behavior constitute a class. For example, the set of all employees can constitute a class in an employee pay-roll system, since each employee object has similar attributes such as his name, code number, salary, address, etc. and exhibits similar behavior as other employee objects. Once we define a class it serves as a template for object creation. Since each object is created as an instance of some class, classes can be considered as abstract data types (ADTs).

**Methods and messages**

The operations supported by an object are called its methods. Thus operations and methods are almost identical terms, except for a minor technical difference. Methods are the only mean available to other objects for accessing and manipulating the data of another object. The methods of an object are invoked by sending messages to it. The set of valid messages to an object constitutes its protocol.

**Inheritance**

The inheritance feature allows us to define a new class by extending or modifying an existing class. The original class is called the ***base class*** (or superclass) and the new class obtained through inheritance is called the ***derived class*** (or subclass). A base class is a generalization of its derived classes. This means that the base class contains only those properties that are common to all the derived classes. We can also say that each derived class is a specialization of its base class because it modifies or extends the basic properties of the base class in certain ways. Thus, the inheritance relationship can be viewed as a generalization-specialization relationship. Using the inheritance relationship, different classes can be arranged in a class hierarchy (or class tree).

In addition to inheriting all the properties of the base class, a derived class can define

new properties. That is, it can define new data and methods. It can even give new definitions to methods which already exist in the base class. Redefinition of the methods which existed in the base class is called ***method overriding***.

In Figure 5, LibraryMember is the base class for the derived classes Faculty, Student, and Staff. Similarly, Student is the base class for the derived classes Undergrad, postgrad, and Research. Each derived class inherits all the data and methods of the base class. It can also define additional data and methods or modify some of the inherited data and methods. The different classes in a library automation system and the inheritance relationship among them are shown in Figure 5. The inheritance relationship has been represented in this figure using a directed arrow drawn from a derived class to its base class. In Figure 5, the LibraryMember base class might define the data for name, address and library membership number for each member. Though faculty, student and staff classes inherit these data, they might have to redefine the respective issue-book methods because the number of books that can be borrowed and the duration of loan may be different for the different category of library members. Thus, the issue-book method is overridden by each of the derived classes and the derived classes might define additional data max-number-books and max-duration-of-issue which may vary for the different member categories.

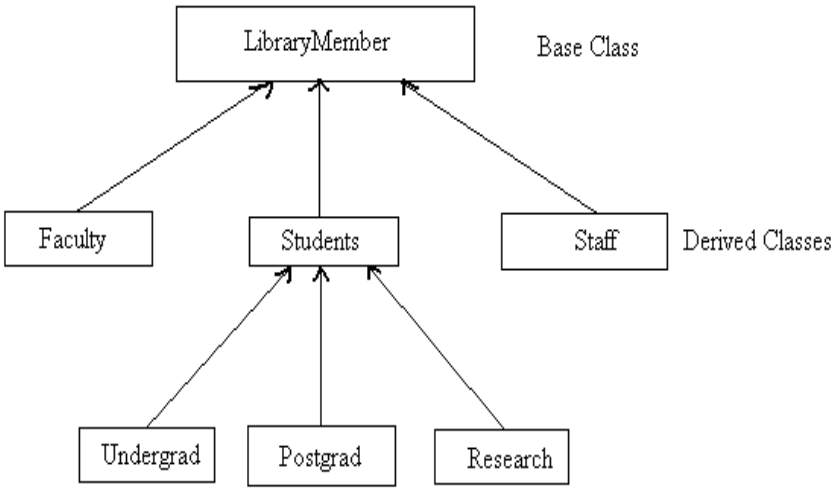



**Figure 5: Library information system example**

Inheritance is a basic mechanism that all object-oriented languages support. In fact, the languages that support classes (ADTs) but do not support inheritance are not called object oriented and instead called object-based languages. But, why do we need the inheritance relationship in the first place? An important advantage of the inheritance mechanism is code reuse. If certain methods or data are similar in a set of classes, then instead of defining these methods and data in each of these classes separately, these methods and data are defined only once in the base class and then inherited by each of its

subclasses. For example, in the Library Information System example of Figure 5, each category of member objects Faculty, Student and Staff need the data member-name, member-address and membership-number and therefore these data are defined in the base class LibraryMember and inherited by its subclasses. Another advantage of the inheritance mechanism is the conceptual simplification that comes from reducing the number of independent features of the classes.

**Multiple inheritance**

Multiple inheritance is a mechanism by which a subclass can inherit attributes and methods from more than one base class. Suppose research students are also allowed to be staff of the institute, then some of the characteristics of the Research class might be similar to the Student class and some other characteristics might be similar to the Staff class. We can represent such a class hierarchy as in Figure 6. Multiple inheritance is represented by arrows drawn from the subclass to each of the base classes. Observe that the class Research inherits features from both the classes, that is, Student and Staff.

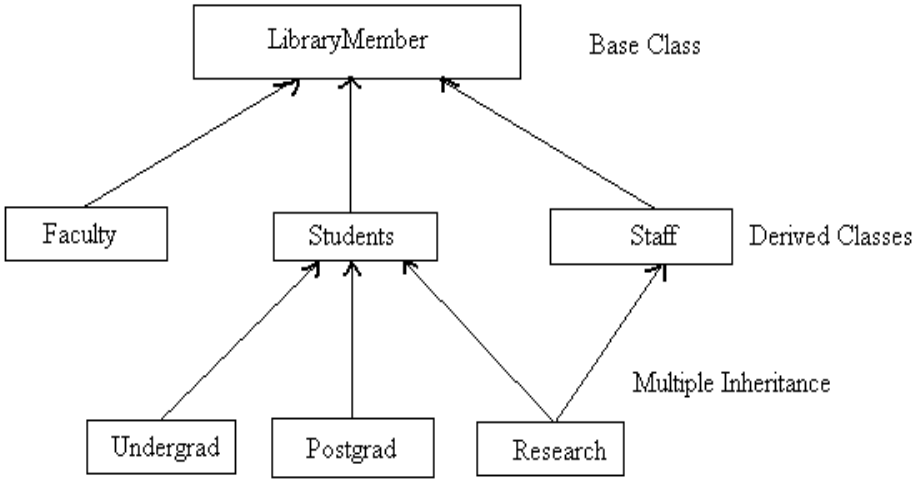



**Figure 6: Library information system example with multiple inheritance**

**Abstract class**

Classes that are not intended to produce instances of themselves are called abstract classes. Abstract classes merely exist so that behavior common to a variety of classes can be factored into one common location, where they can be defined once. This way code reuse can be enhanced and the effort required to develop the software brought down. Abstract classes usually support generic methods, but the subclasses of the abstract classes are expected to provide specific implementations of these methods.

**Key concepts**

We will now discuss some key concepts used in the object-oriented approaches.

**Abstraction**

Let us first examine how exactly the abstraction mechanism works. Abstraction is the selective examination of certain aspects of a problem while ignoring the remaining aspects of the problem. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and to suppress aspects of the problem that are not relevant for the given purpose. Thus, the abstraction mechanism allows us to represent a problem in a simpler way by omitting unimportant details. Many different abstractions of the same problem are possible depending on the purpose for which they are made. Abstraction not only helps the development engineers to understand and appreciate the problem better, it also leads to better comprehension of the system design by the end-users and the maintenance team. Abstraction is supported at two levels in an object-oriented design (OOD).

- A class hierarchy can be viewed as defining an abstraction level, where each base class is an abstraction of its subclasses.
- An object itself can be looked upon as a data abstraction entity, because it abstracts out the exact way in which various private data items of the object are stored and provides only a set of well-defined methods to other objects to access and manipulate these data items.

Abstraction is a powerful mechanism for reducing complexity of software. In fact, it has been shown from data collected from several software development projects that software productivity is inversely proportional to software complexity. Therefore, abstraction can be viewed as a way of increasing software productivity.

**Encapsulation**

The property of an object by which it interfaces with the outside world only through messages is referred to as ***encapsulation***. The data of an object are encapsulated within the methods and are available only through message-based communication. This concept is schematically represented in Figure 7.

**Encapsulation offers three important advantages:**

- It protects an object's variables from corruption by other objects. This protection is provided against unauthorized access and against different types of problems that arise from concurrent access of data such as deadlock and inconsistent values.
- Encapsulation hides the internal structure of an object, making interaction with the object simple and standardized. This facilitates reuse of objects across different projects. Furthermore, if the internal structure or procedures of an object are modified other objects are not affected and this result in easy maintenance and bug correction.

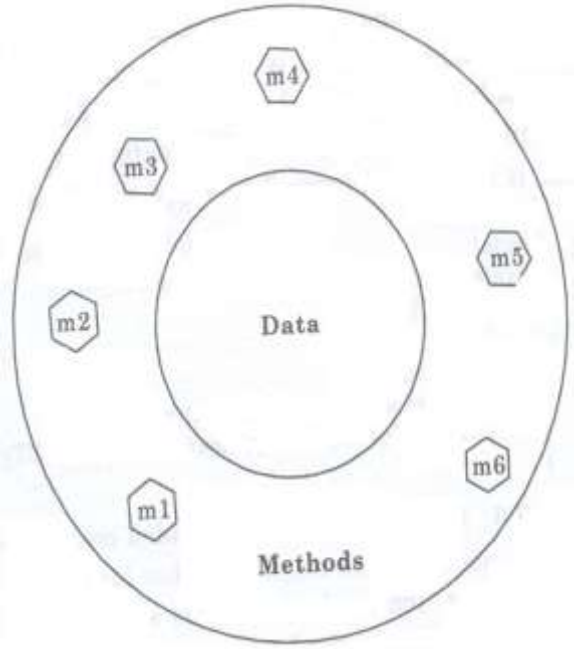



**Figure 7: Schematic representation of the concept of encapsulation**

- Since objects communicate among each other using messages only, they are weakly coupled. The fact that objects are inherently weakly coupled enhances under-standability of design since each object can be studied and understood almost in isolation from other objects

**Polymorphism**

Polymorphism literally means poly (many) morphism (forms). Broadly speaking, polymorphism denotes the following:

- The same message can result in different actions when received by different objects. This is also referred to as *static binding.* This occurs when multiple methods with the same operation name exist.
- When we have an inheritance hierarchy, an object can be assigned to another object of its ancestor class. When such an assignment occurs, a method call to the ancestor object would result in the invocation of the appropriate method of the object of the derived class. Since, the exact method to which a method call would be bound cannot be known at compile time and is dynamically decided at the runtime, this is also known as *dynamic binding.*

An example of static binding is the following. Suppose a class named Circle has three definitions for the create operation. One definition does not take any argument and creates a circle with default parameters. The second definition takes the centre point and radius as its parameters. In this case, the fill style values for the circle would be set to the default "no fill". The third takes the centre point, the radius and the fill style as its input. When the create method is invoked, depending on the parameters given in the invocation, the

appropriate method will be called. If create is invoked with no parameters, then a default circle would be created. If only the centre and radius are supplied, then an appropriate circle would be created with no fill type and so on. A class definition of the Circle class with the overloaded create method is shown in Figure 8. When the same operation (e.g. create) is implemented by multiple methods, the method name is said to be overloaded (Notice the difference between an operation and a method.)

```
class Circle {

    private float x, y, radius;

    private int tillType;

    public create( );

    public create(float x, float y, float centre);

    public create(float x, float y, float centre, int fillType);

    }
```

**Figure 8: Circle class with overloaded create method**

Now let us consider the concept of dynamic binding. Using dynamic binding a programmer can send a generic message to a set of objects which may be of different types (i.e. belonging to different classes) and leave the exact way in which the message would be handled to the receiving objects. Suppose we have a class hierarchy of different geometric objects as shown in Figure 9. Now, suppose the display method is declared in

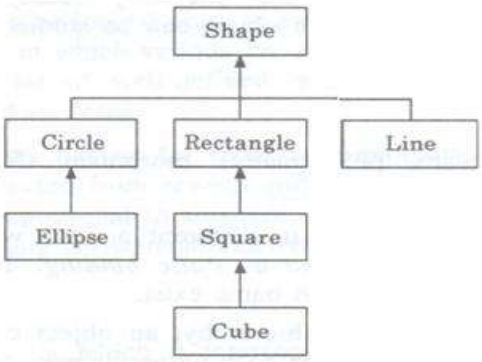



**Figure 9: Class hierarchy of geometric objects**

the shape class and is overridden in each derived class. If the different types of geometric objects making up a drawing are stored in an array of type shape, then a single call to the display method for each object would take care to display the appropriate drawing element. That is the same draw call to a shape object would take care of drawing the

appropriate shape. This is shown in the code segment in Figure 10. The main advantage of dynamic binding is that it leads to elegant programming and facilitates code reuse and maintenance. With dynamic binding, new derived objects can be added with minimal changes to existing objects. We shall illustrate this advantage of polymorphism by comparing the code segments of an object-oriented program with a traditional program for drawing various graphic objects on the screen. Assume that shape is the base class and the classes circle, rectangle and ellipse are derived from it. Now, shape can be assigned any objects of type circle, rectangle, ellipse, etc. But a draw method invocation of the shape object would invoke the appropriate method.

| Traditional code | Object-oriented code |
|---|---|
| If (shape==Circle) then<br>draw_circle ( );<br>else if (shape==Rectangle) then<br>draw_rectangle ( );<br><br>---<br>---<br>--- | Shape.draw ( ); |

**Figure 10: Traditional code and object oriented code using dynamic binding**

It can be easily seen from Figure 10 that because of dynamic binding, the object-oriented code is much more concise and intellectually appealing. Also, suppose in the example program segment, it is later found necessary to handle a new graphics drawing primitive, say ellipse. Then, the procedural code has to be changed by adding a new if then-else clause. However, in case of the object-oriented program, the code need not change, only a new class called ellipse has to be defined.

**Composite objects**

Objects which contain other objects are called ***composite objects***. Containment may be achieved by including the pointer to one object as a value in another object or by creating instances of the component objects in the composite object. Composite objects can be used to realize complex behavior. Composition can occur in a hierarchy of levels. That is, an object contained in another object may itself be a composite object. Typically, the structures that are built up using composite objects are limited to a tree hierarchy, i.e. no circular inclusion relation is allowed. This means that an object cannot contain an object of its own type. It is important to bear in mind the distinction between an inheritance hierarchy of an object class and the containment hierarchy of object instances. The two are not related, the use of inheritance simply allows many different object types to be defined with minimum effort by making use of the similarity of their features. On the other hand, the use of containment allows construction of complex objects.

**Genericity**

Genericity is the ability to parameterize class definitions. For example, while defining a class stack of different types of elements such as integer stack, character stack, and floatingpoint stack, ***genericity*** permits us to define a generic class of type stack and later instantiate it either as an integer stack, a character stack or a floating-point stack as may be required. This can be achieved by assigning a suitable value to a parameter used in the generic class definition.

**Dynamic binding**

In method invocation, static binding is said to occur if the address of the called method is known at compile time. In dynamic binding, the address of an invoked method is known only at the run time. We have already seen that dynamic binding is useful for implementing a type of polymorphism. It is important to remember that dynamic binding occurs when we have some methods of the base class overridden by the derived classes and the instances of the derived classes are stored in instances of the base class.

**Related Technical Terms**

**Persistence**

Objects usually get destroyed once a program finishes execution. Persistent objects are stored permanently. That is, they live across different executions. An object can be made persistent by maintaining copies of it in a secondary storage or in a database.

**Agents**

A passive object is one that performs some action only when requested through invocation of some of its methods. Agents (also called *active objects)* on the other hand monitor events occurring in the application and take action autonomously. Agents are used in applications such as monitoring exceptions. For example, in a database application like accounting, agents may monitor the balance sheet and would alert the

user whenever inconsistencies arise in a balance sheet due to some improper transaction take place.

**Widget**

The term widget stands for *window object.* A widget is a primitive object used in the design of graphical user interface (GUI). More complex graphical user interface design primitives (widgets) can be derived from the basic widget using the inheritance mechanism. A widget maintains internal data such as the geometry of the window, background and foreground colours of the window, cursor shape and size, etc. The methods supported by a widget manipulate the stored data and carry out operations such as resize window, iconify window, destroy window, etc. Widgets are becoming the standard components of GUI design. This has given rise to the technique of component-based user interface development.

## 5. Advantages of OOD

In the last few years that OOD has come into existence, it has found widespread acceptance in the industry as well as in the academics. The main reason for the popularity of OOD is that it holds the following promises:

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability

Out of all these advantages, many people agree that the chief advantage of OOD is improved productivity-which comes about due to a variety of factors, such as:

- Code reuse by the use of predefined class libraries
- Code reuse due to inheritance
- Simpler and more intuitive abstraction, i.e. better organization of inherent complexity
- Better problem decomposition

Compiled results indicate that when companies start to develop software using the object-oriented paradigm, the first few projects incur higher costs than the costs incurred by the traditionally developed projects. After a few projects have been completed, reductions in cost become possible. A functioning and well-established object-oriented methodology and environment is likely to be managed with 20-50% of the cost of traditional development. Having discussed the basic concepts of object-orientation, we now present the Unified Modelling Language (UML).

**Summary**

In this lesson we have discussed all the concepts related to data flow diagram and data dictionary. We have also studied about the Unified Modeling Language and its various designing principles.

**Self Check Exercise**

- What is data flow diagram in software engineering? Explain with the help of examples.
- Discuss various symbols and notations used in DFD?
- Write a detailed note on UML.

**Suggested Readings**

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Testing Techniques" by Boris Beizer, Dreamtech Press, New Delhi.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

LESSON No. 1.8 AUTHOR: Dr. VISHAL SINGH

# UML-II

**Objectives**

**1. Introduction**

**2. Origin**

**3. UML Diagrams**

**4. Use Case Model**

**Summary**

**Self Check Exercise**

**Suggested Readings**

## 1. Introduction

*Unified Modeling Language (UML)*

As the name implies, UML is a modeling language. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create models of systems. Models are very useful in documenting the design and analysis results. Models also facilitate the generation of analysis and design procedures themselves.

However, we must be clear that UML is not a system design or development methodology. Neither is it tied to any specific methodology. It can only be used to document object-oriented analysis and design results obtained using some methodology. In fact, it was one of the objectives of the developers of UML to keep the notations of UML independent of any specific design methodology. In this respect, UML is different from its predecessors (e.g. OMT, Booch's methodology, etc.) where the notations supported by the modelling languages were closely tied to the corresponding design methodologies.

## 2. Origin

In the late 1980s and the early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. It was also not uncommon to

find different project teams in the same organization using different notations for documenting their object-oriented analysis and design results. These diverse notations used to give rise to a lot of confusion.

UML was developed to standardize the large number of object-oriented modelling notations that existed and were used extensively in the early 1990s. The principal ones in use were:

- OMT [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- OOSE [Jacobson 1992]
- Odell's methodology [Odell 1992]
- Shlaer and Mellor methodology [Shlaer 1992]

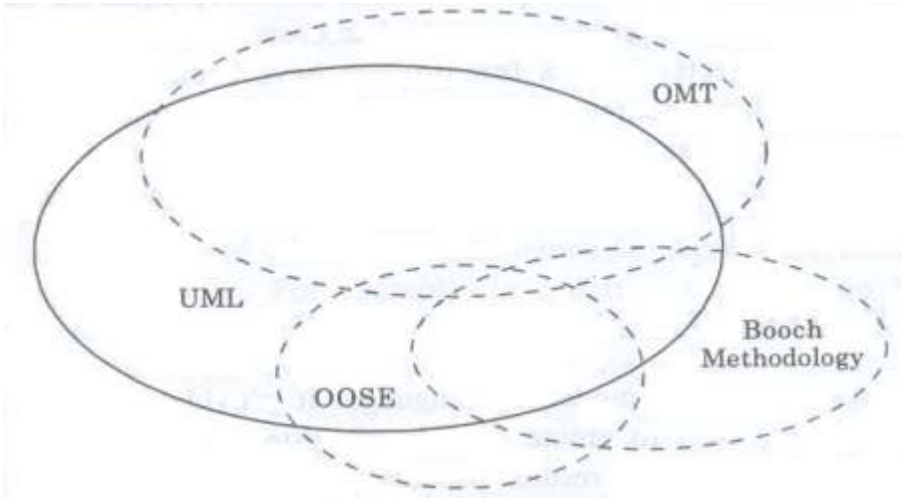



**Figure 1: Schematic representation of the impact of different object modelling techniques on UML**

Needless to say that UML has borrowed many concepts from these modelling techniques. Especially, concepts from the first three methodologies have been heavily drawn upon. The influence of various object modelling techniques on UML is schematically represented in Figure 1. As shown in this figure, OMT has had the most profound influence on UML.

UML was adopted by Object Management Group (OMG) as a de facto standard in 1997. Actually, OMG is not a standard formulating body, but an association of industries which tries to facilitate early formulation of standards. OMG aims to promote consensus notations and techniques with the hope that if the usage becomes widespread, then they would automatically become standards. For more information on OMG, visit www.omg.org.

UML is more complex than its antecedents. This is expected because it is intended to be more comprehensive. We shall see that UML contains an extensive set of notations

and suggest construction of many types of diagrams. UML has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and a strong industry backing have helped UML find widespread acceptance. UML is now being used in a large number of software development projects world-wide. It is interesting to note that the use of UML is not restricted to the software industry alone. As an example of UML's use outside the software development problems, some car manufacturers are planning to use UML for their "build-to-order" initiative.

Many of the UML notations are difficult to draw on paper and are best drawn using a CASE tool such as Rational Rose (visit www.rational.com). Many of the available CASE tools also help in incremental elaboration of the initial object model to final design. Some CASE tools also generate code templates in a variety of languages once the UML models have been constructed.

**What is a model?**

Before we discuss the details of UML, it is important to clearly understand what exactly is meant by a model and why is it necessary to create a model. A model captures aspects important for some applications while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical or program code based. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modelling tool. However, text explanations are often required to accompany the graphical models.

**Why construct a model?**

An important reason behind constructing a model is that it helps manage complexity. Once the models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can be used not only to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification will be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed.

## 3. UML Diagrams

UML can be used to construct nine different types of diagrams to capture different views of a system. Just as a building can be modelled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc. the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

**The UML diagrams can capture the following views of a system:**

- Structural view
- Behavioural view
- Implementation view
- Environmental view

We first provide a brief overview of the different views of a system which can be developed using UML. The diagrams used to realize the important views are discussed in the subsequent sections

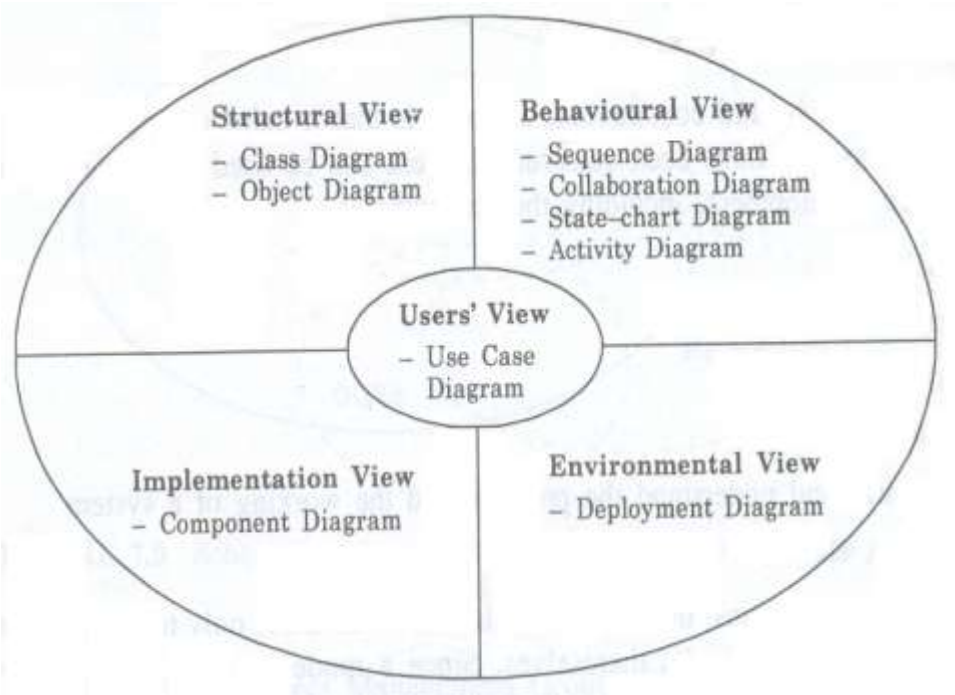



**Figure 2: Different types of diagrams and views supported in UML**

**Users' view**: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of the different system components,

the implementation, etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views.

The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user-centric development style. It is indeed remarkable that even for object-oriented development, we need a functional view.

**Structural view:** The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

**Behavioural view**: The behavioural view captures how objects interact with each other to realize the system behaviour. The system behaviour captures the time-dependent (dynamic) behaviour of the system.

**Implementation view**: This view captures the important components of the system and their dependencies.

**Environmental view**: This view models how the different components are implemented on different pieces of hardware.

For any given problem, should one construct all the views using all the diagrams provided by UML? The answer is "No". For a simple system, the use case model, class diagram, one of the interaction diagrams may be sufficient. For a system in which the objects undergo many state changes, a state chart diagram may be necessary. For a system, which is implemented on a large number of hardware components, a deployment diagram may be necessary. So, the type of models to be constructed depends on the problem at hand. Rosenberg provides an analogy saying that, "Just like you do not use all the words listed in the dictionary while writing a prose, you do not use all the UML diagrams and modelling elements while modelling a system."

## 4. Use Case Model

The use case model for any system consists of a set of "use cases". Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What the users can do using the system?" Thus for the Library Information System (LIS), the use cases could be:

- issue-book
- query-book
- return-book
- create-member
- add-book

The use cases partition the system behaviour into transactions, such that each transaction performs some useful action from the user's point of view. Each transaction may involve either a single message or multiple message exchanges between the user and the system to complete itself.

The purpose of a use case is to define a piece of coherent behaviour without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used, nor the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one main line sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most frequently; occurring sequence of interaction. For example, in the mainline sequence of the withdraw cash the use case supported by a bank ATM would be: you insert the card, enter password, select the amount withdraw option, enter the amount to be withdrawn, complete the transaction and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the account balance. These variations are also called *alternate paths.* A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called *scenarios* or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

The use case model is an important analysis and design artifact. As already mentioned, other UML models must conform to this model in any use case-driven (also called the user. centric) analysis and development approach. It should be remembered that the "use case model" is not really an object-oriented model according to a strict definition of the term., The use case model represents a functional or process model of a system. This is in contrast to all other types of UML diagrams.

Normally, each use case is independent of the other use cases. However, implicit: dependencies among use cases may exist because of dependencies among the use cases at the implementation level resulting from shared resources, objects or functions. For example, in our Library Automation System, renew-book and reserve-book are two independent use cases. But in actual implementation of renew-book, a check is made to see if any book has been reserved by a previous execution of the reserve-book use case. Another example of dependency among use cases is the following. In the Bookshop Automation Software" update-inventory and sale-book are two independent use cases. But during execution of sale-book there is an implicit dependency on update-inventory.

**Representation of Use Cases**

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the

ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modelled (e.g. Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a User with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case.

Both the human users and the external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype «external system».

At this point, it is necessary to explain the concept of a stereotype in UML. One of the main objectives of the creators of the UML was to restrict the number of primitive symbols in the language. A language having a large number of primitive symbols not only becomes difficult to learn but also makes it difficult to use these symbols and understand a diagram constructed by using a large number of basic symbols. The creators of UML introduced a stereotype which when used to annotate a basic symbol can slightly change the meaning of the basic symbol. Just as you stereotype your friends in the class as studious, jovial, serious, etc. stereotyping can be used to give special meaning to any basic UML construct.

**Example 1**

The use case model for the Tic-tac-toe problem is shown in Figure 3. This software has only one use case "play move". Note that we did not name the use case "get-user-move". The name "get-user-move" would be inappropriate because the use cases should be named from the users' perspective.

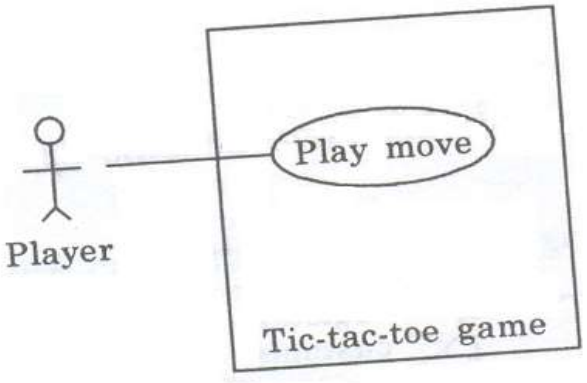



**Figure 3: Use case model for Example 1**

**Text Description**. Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behaviour associated with the use case in terms of the mainline sequence, different variations to the normal behaviour, the system responses associated with the use case, the exceptional conditions that may occur in the behaviour, etc. The behaviour description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the types of information which may be included in a use case text description in addition to the mainline sequence and the alternate scenarios.

**Contact persons**. This section lists the personnel of the client organization with whom the use case was discussed, date and time of the meeting, etc.

**Actors.** In addition to identifying the actors, some information about the actors using this use case which may help in the implementation of the use case may be recorded.

**Pre-condition.** The pre-conditions would describe the state of the system before the use case execution starts.

**Post-condition.** This captures the state of the system after the use case has been successfully completed

**Nonfunctional requirements.** This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

**Exceptions, error situations.** This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

**Sample dialogs**. These serve as examples illustrating the use case.

**Specific user interface requirements.** These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screenshots, interaction style etc.

**Document references**. This part contains references to specific domain-related documents which may be useful to understand the system operation.

**Example 2**

The use case diagram of the Supermarket Prize Scheme is shown in Figure 4.

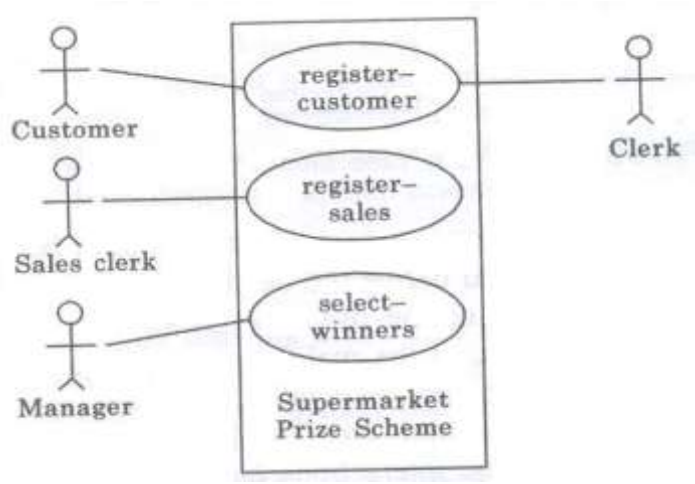



**Figure 4: Use case model for Example 2**

**Text description**

UI: register-customer: Using this use case, the customer can register himself by providing the necessary details.

**Scenario 1: Mainline sequence**

1. Customer: select register customer option
2. System: display prompt to enter name, address and telephone number.
3. Customer: enter the necessary values
4. System: display the generated id and the message that the customer has been successfully registered.

**Scenario 2: At step 4 of the mainline sequence.**

4. System: displays the message that the customer has already registered.

**Scenario 3: At step 4 of the mainline sequence.**

4. System: displays the message that some input information has not been entered. The system displays a prompt to enter the missing values.

U2: register-sales: Using this use case, the clerk can register the details of the purchase made by a customer.

**Scenario 1: Mainline sequence**

1. Clerk: selects' the register sales option.
2. System: displays prompt to enter the purchase details and the id of the customer

3. Clerk: enters the required details.

4. System: displays a message of having successfully registered the sale.

U2: select-winners: Using this use case, the manager can generate the winner list.
**Scenario 1: Mainline sequence**

1. Manager: selects the select-winner option.

2. System: displays the gold coin and the surprise gift winner list.

**Why Develop the Use Case Diagram?**

If you look at the use case diagram, the utility of the use cases represented by the ellipses is obvious. They along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. But what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in selecting and implementing a security mechanism through a login system, so that each actor can invoke only those functionalities to which he is entitled to. Another possible use is in preparing the documentation (e.g. user's manual) targeted to each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

**How to Identify the Use Cases of a System?**

Identification of the use cases involves brainstorming and reviewing the existing requirements specification document. The actor-based method of identifying the use cases is quite popular. This involves first identifying the different types of actors and their usage in the system. For each actor, we identify the functions they initiate or participate in.

**Essential vs. Real Use Case**

Essential use cases are created during the early stage of eliciting the requirements. They are independent of the design decisions and tend to be correct over long periods of time.

Real use cases describe the functionality of the system in terms of its actual current design committed to specific input/output technologies. Therefore, the real use cases can be developed only after the design decisions have been made. Real use cases are a design artifact. However, sometimes organizations commit to development contracts on the basis of, user interface specifications. In such cases, there is no distinction between the essential use case and the real use case.

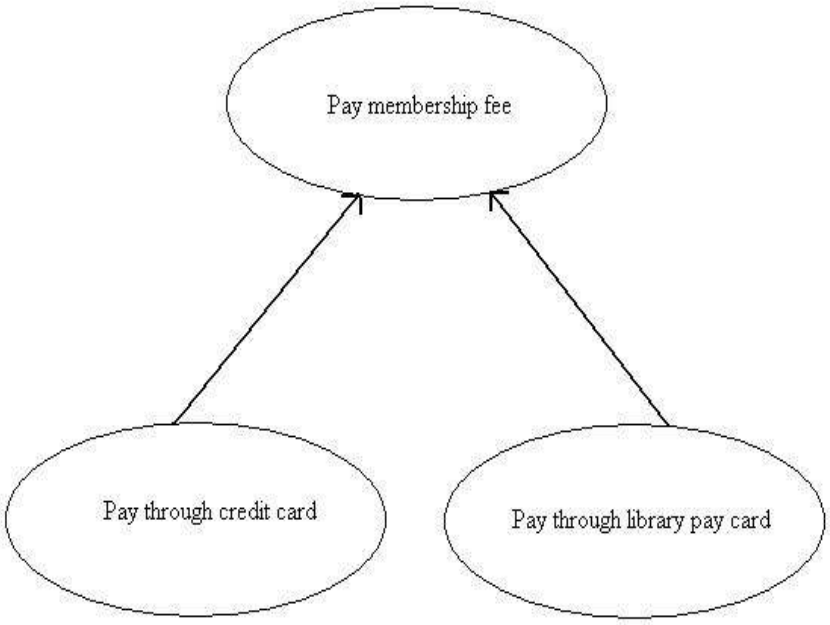
**Factoring of Commonality Among Use Cases**

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases is required under two situations. First, the complex use cases need to be factored into simpler use cases. This would not only make the behaviour associated

with the use case, much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single standard-sized (A4) paper. Secondly, use cases need to be factored whenever there is common behaviour across different use cases, Factoring would make it possible to define such behaviour only once and reuse it wherever, required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here, Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it. UML offers three mechanisms to factor out the use cases which are discussed in the following:

**Generalization**

Use case generalization can be used when you have one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behaviour and meaning of the present use case. The notation is the same too (See Figure 5). It is important to remember that the base and the derived use cases are different from one another and therefore should have separate text descriptions.




**Includes**

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship involves one use case including the behaviour of another use case in its sequence of events and actions. The includes

relationship occurs when you have a chunk of behaviour that is similar across a number of use cases. The factoring of such behaviour will help in avoiding the repetition of the specification and its implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use case into more manageable parts.

As shown in Figure 6, the includes relationship is represented using a predefined stereotype «include». In the includes relationship, a base use case compulsorily and automatically includes the behaviour of the common use case. As shown in Figure 7,

issue-book and renew-book both include check-reservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and independent text description should be provided for it.

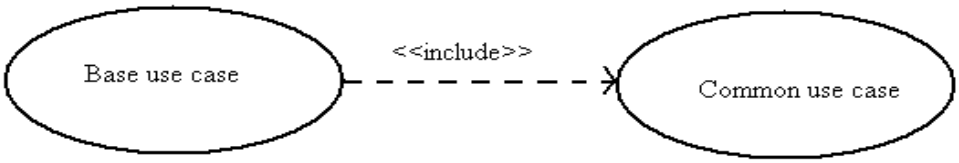



**Figure 6: Representation of Use case inclusion**

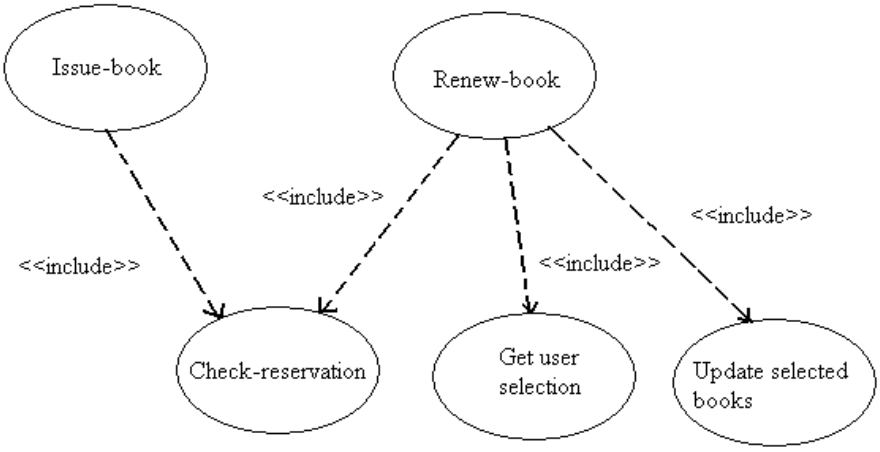



**Figure 7: Paralleling model**

**Extends**

The main idea behind the extends relationship among use cases is that it allows

you to show optional system behaviour. An optional system behaviour is executed only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in Figure 8.

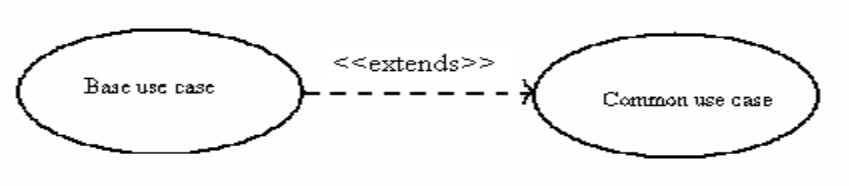



**Figure 8: Example Use case extension**

The extends relationship is similar to generalization. But unlike generalization, the extend use case can add additional behaviour only at an extension point when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.

**Organization of use cases**

When the use cases are factored, they are organized hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in Figure 9, The top-level use cases are superordinate to the refined use cases. The refined use cases are subordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases.

The functionality of the superordinate use cases is traceable to their subordinate use cases. Thus, the functionality provided by the superordinate use cases is composite of the functionality of the subordinate use cases.

In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. In the top-level diagram, only those use cases with which external users interact are shown. The topmost use cases specify the complete services offered by the system to the external users of the system. The subsystem-level use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

**Use Case Packaging**

Packaging is the mechanism provided by UML to handle complexity. When we have too many use cases in the top-level diagram, we can package the related use cases so that

at best 6 or 7 packages are present at the top level diagram. Any modelling element that becomes large and complex can be broken up into packages. Please note that you can put any element of UML (including another package) in a package diagram, The symbol for a package is a folder. Just as you organize a large collection of documents in a folder, you can organize UML elements into packages. An example of packaging use cases is shown in Figure 10.

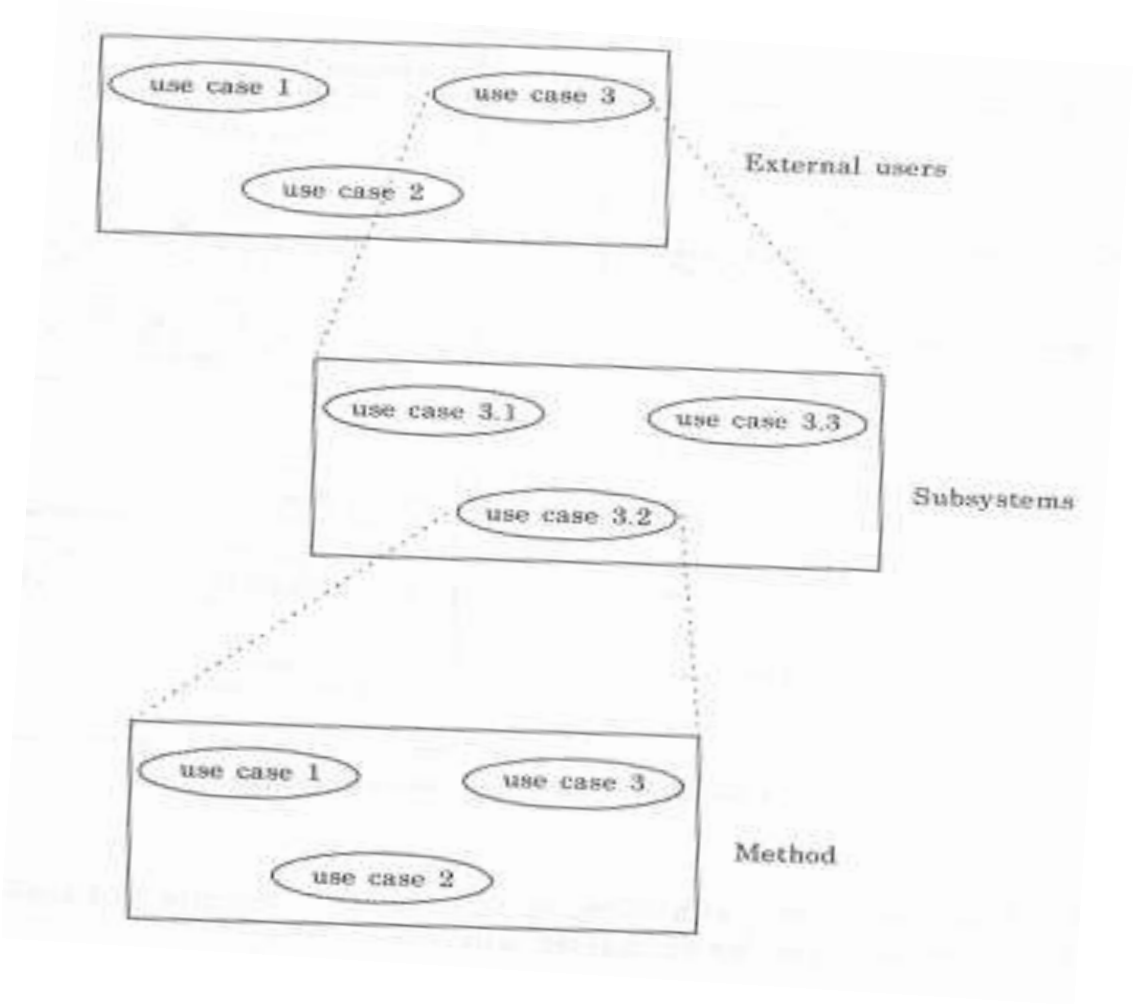



**Figure 9: Hierarchical organization of Use case**

**Class Diagrams**

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system consists of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies. We now discuss the UML syntax for representation of the classes and their relationships.

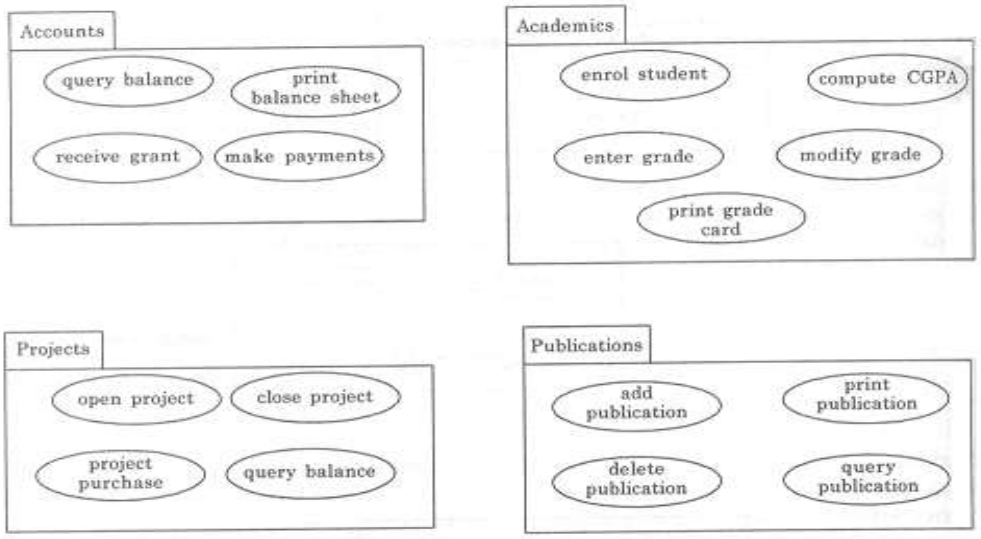



**Figure 10: Use case packaging**

**Classes**

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns. An example of various representations of a class is shown in Figure 11.

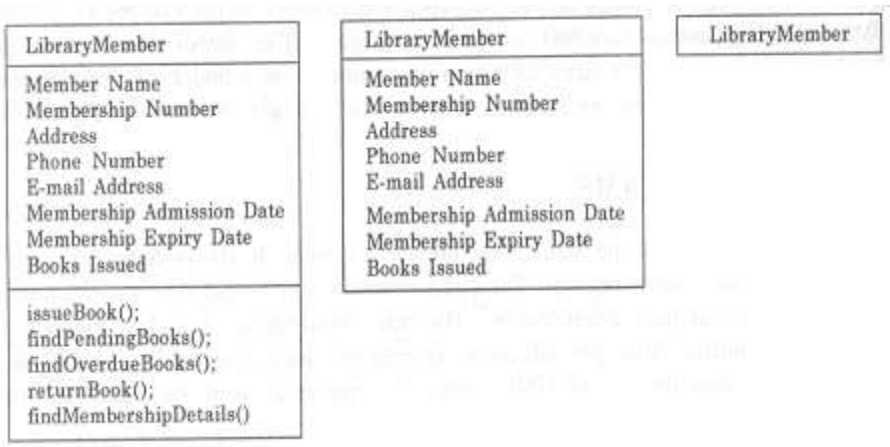



**Figure 11: Different representations of Library Member class**

Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram.

**Attributes**

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names and may optionally contain specification of their type (that is, their class, e.g. Int, Book, Employee, etc.), an initial value, and constraints. Attribute names are written left-justified using plain type letters, and the names should begin with a lowercase letter.

Attribute names may be followed by square brackets containing a multiplicity expression, e.g. sensorStatus[l0]. The multiplicity expression indicates the number of attributes per instance of the class. An attribute without square brackets must hold exactly one value. The type of an attribute is written by appending a colon and the type name after the attribute name, for example, sensorStatus[1]:lnt.

The attribute name may be followed by an equal sign and an initial value that is used to initialize the attributes of the newly created objects, for example, sensorStatus[1]:lnt=O.

**Operation**

The operation names are typically left-justified, in plain type and always begin with a lowercase letter. (Remember that abstract operations are those for which the implementation is not provided during the class definition.) The parameters of a function may have a kind specified. The kind may be "in" indicating that the parameter is passed into the operation, or "out" indicating that the parameter is only returned from the operation, or "inout" indicating that the parameter is used for passing data into the operation and getting result from the operation. The default is "in".

An operation may have a return type consisting of a single return type expression, for example, issueBook(in bookName):Boolean. An operation may have a class scope (i.e. shared among all the objects of the class) and is denoted by underlining the operation name.

Often a distinction is made between the two terms, i.e. operation and method. An operation is something that is supported by a class and invoked by objects of other classes. Please remember that there might be multiple methods implementing the same operation. We pointed out earlier that this is called static polymorphism. The method names can be the same; however, it should be possible to distinguish the methods by examining their parameters. Thus, the terms operation and method are distinguishable only when there is polymorphism.

**Association**

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose Amit has borrowed the book Graph Theory. Here, borrowed is the connection between the objects

Amit and Graph Theory book. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that Library Member borrows Books. Here, borrows is the association between the class LibraryMember and the class Book.

Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship.

Association between two classes is represented by drawing a straight line between the concerned classes. Figure 12 illustrates the graphical representation of the

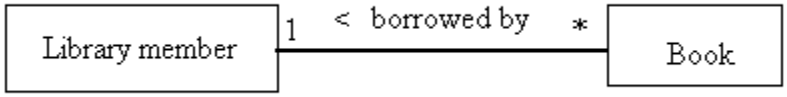



**Figure 12: Association between two classes**

association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with the other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots.

Associations are usually realized by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

**Aggregation**

Aggregation is a special type of association where the involved classes represent a whole part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership.

We have already seen that if an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite

object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as in Figure 13.

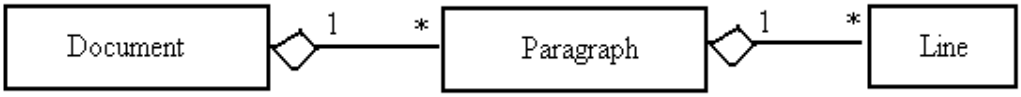



**Figure 13: Representation of Aggregation**

The aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of its own class. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels

**Composition**

Composition is a stricter *form* of aggregation, in which the parts are existence-dependent on the whole. This means that the life of each part is closely tied to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.

A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in Figure 14.

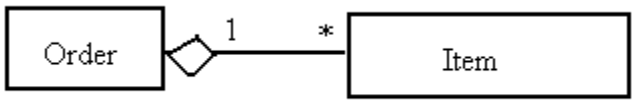



**Figure 14: Representation of Composition**

**Inheritance**

The inheritance relationship is represented by means of an empty arrow pointing from the subclass to the superclass. The arrow may be directly drawn from the subclass to the superclass. Alternatively, the inheritance arrow from the subclasses may be combined into a single line (see Figure 15).

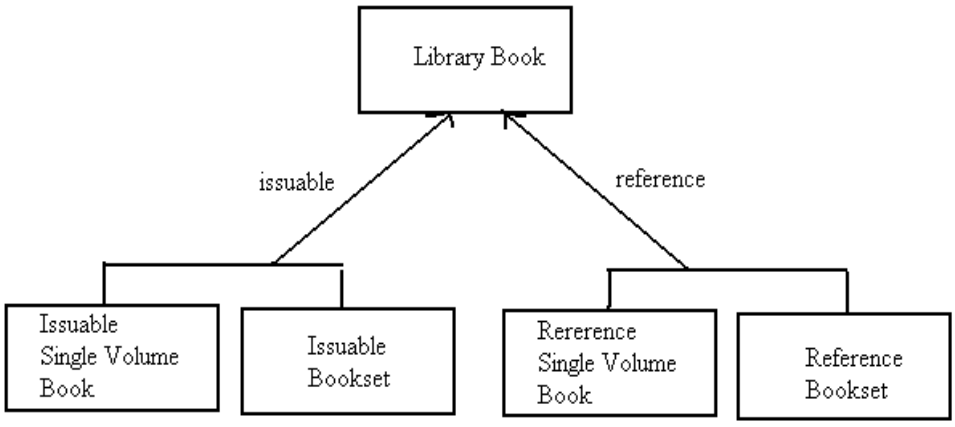



**Figure 15: Representation of inheritance relationship**

The direct arrows allow flexibility in laying out the diagram and can easily be drawn by hand. The combined arrows emphasize the collectivity of the subclass, and that the specialization has been done on the basis of some discriminator. In the example of Figure 15, issuable and reference are the discriminators. The various subclasses of a superclass can then be differentiated by means of the discriminator. The set of subclasses of a class having the same discriminator is called a partition. It is often helpful to mention the discriminators during modelling, as these become documented design decisions.

**Dependency**

Dependency is a form of association between two classes. A dependency relation between two classes shows that a change in the independent class requires a change to be made to the dependent class. Dependencies are shown as dotted arrows (see Figu

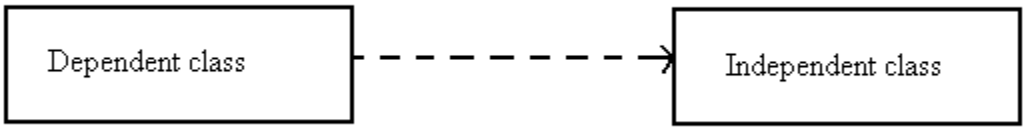



**Figure 16: Representation of dependence between classes**

Dependencies may have various causes. Two important reasons for dependency among classes are the following:

- A class invokes the methods provided by another class.
- A class uses a specific interface of another class. If the properties of the class that provides the interface are changed, then a change becomes necessary in the class that uses that interface.

**Constraints**

Constraints describe a condition or an integrity rule. It can describe the permissible set of values of an attribute, specify the pre- and post-conditions for operations, define a specific order, etc. For example, {Constraint} UML allows you to use any free form expression to describe constraints. The only rule is that they are to be enclosed within braces. Constraints can be expressed using informal English. However, UML also provides Object Constraint Language (OCL) to specify constraints.

**Object diagrams**

Object diagrams, show the snapshots of the objects in a system at a point in time. Since it shows instances of classes, rather than the classes themselves, it is often called the instance diagram. The objects are drawn using rounded rectangles (see Figure 17).

An object diagram may undergo continuous change as the execution proceeds. For example, links may get formed between objects or get broken. Objects may get created and destroyed, and so on. Object diagrams are useful in explaining the working of the system.

**Interaction Diagrams**

Interaction diagrams are models that describe how groups of objects collaborate to realize some behaviour. Typically, each interaction diagram realizes the behaviour of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that anyone diagram can be derived automatically from the other. However, they are both useful. These two actually portray the different perspectives of behaviour of the system and different types of inferences can be drawn from them.

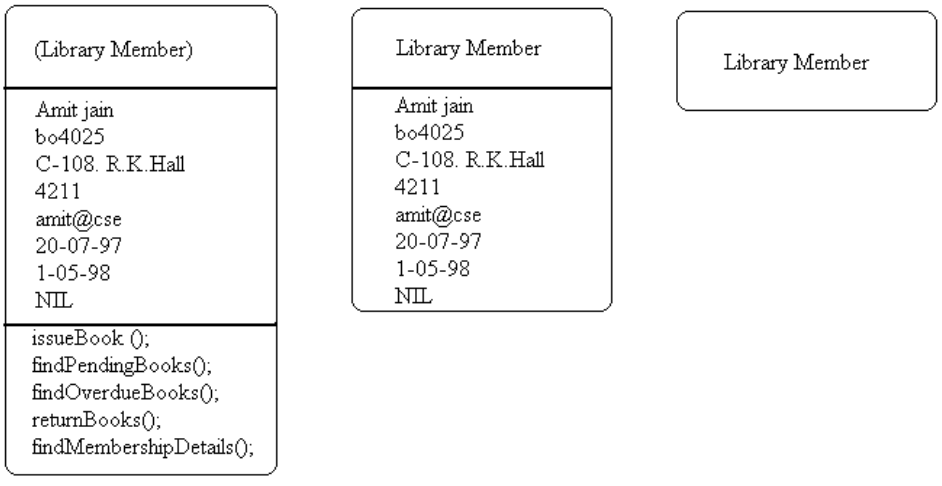



**Figure 17: Different representation of a LibraryMember object**

**Sequence diagram**

A sequence diagram shows interaction among objects as a two-dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class, and both the name of the object and the classes are underlined.

The objects appearing at the top signify that the objects already existed when the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g. a method call), then that object should be shown at the appropriate place on the diagram where it was created.

The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifeline is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifelines of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is labelled with the message name. Some control information can also be included. Two types of control information are particularly valuable.

- A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker (*) shows that the message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, for example, [for every book object].

The sequence diagram for the book renewal use case for the library automation software is shown in Figure 18.

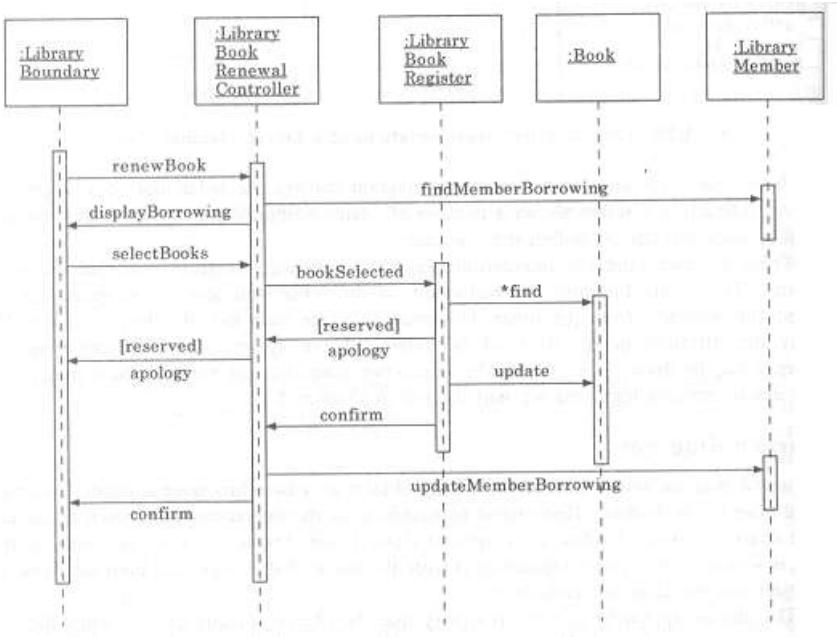


**Figure 18: Sequence diagrams for the renew book use case**

**Collaboration diagram**

A collaboration diagram shows both the structural and behavioural aspects explicitly. This is unlike a sequence diagram which shows only the behavioural aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called a collaborator. The behavioral aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labelled arrow placed near the link.

Messages are prefixed with sequence numbers because that is the only way to describe the relative sequencing of the messages in this diagram.

The collaboration diagram for the example of Figure 18 is shown in Figure 19. A use of the collaboration diagrams in our development process would be to help us determine which classes are associated with which other classes.

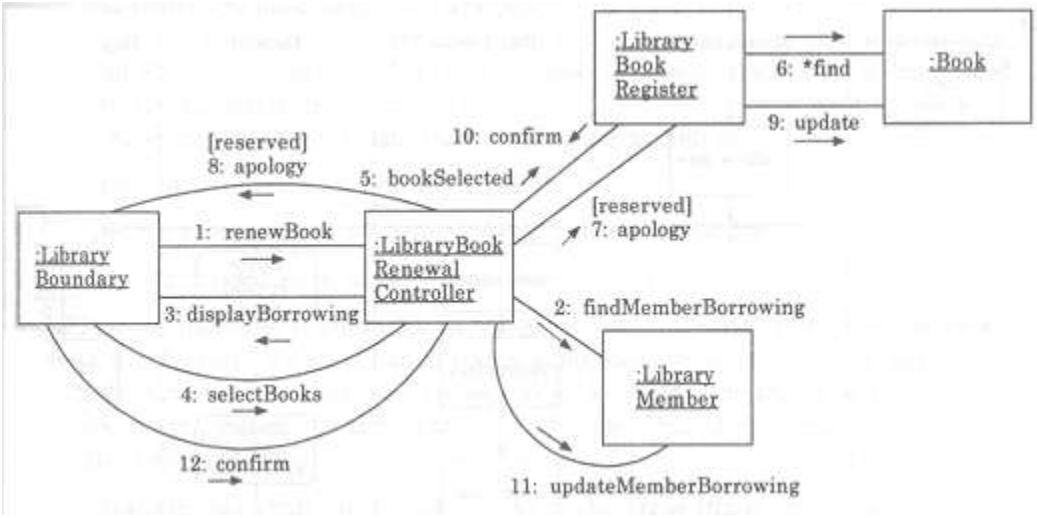



**Figure 19: Collaboration diagrams for the renew book use case**

**Activity Diagrams**

The activity diagram is possibly one modelling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson or Rumbaugh. It is possibly based on the event diagram of Odell [1992] though the notation is very different from that used by Odell.

The activity diagram focuses on representing activities or chunks of processing

which may or may not correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transitions, then these must be identified through conditions.

Activity diagrams are similar to the procedural flow charts. The difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities.

An interesting feature of the activity diagrams is the swim lanes. Swim lanes enable you to group activities based on who is performing them, for example, academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in swim lanes can be assigned to some model elements, for example, classes or some component, etc.

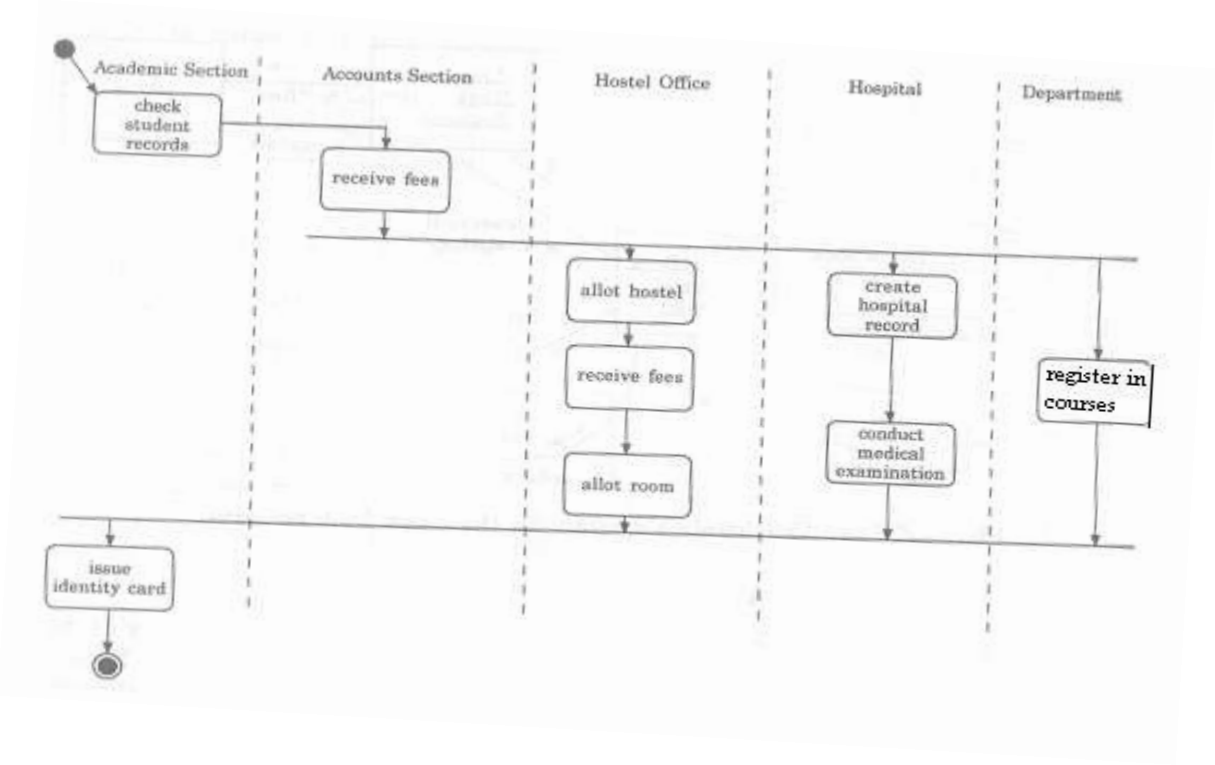



**Figure 20: Activity diagrams for student admission procedure at IIT**

Activity diagrams are normally employed in business process modelling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.

The student admission process in IIT is shown as an activity diagram in Figure 20. This diagram shows the part played by different components of the Institute in the

admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital and the department. After all these activities complete (this synchronization is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

**State Chart Diagram**

A state chart diagram is normally used to model how the state of an object changes in its lifetime. State chart diagrams are good at describing how the behaviour of an object changes across several use case executions. However, if we are interested in modelling some behaviour that involves several objects collaborating with each other, the state chart diagram is not appropriate.

State chart diagrams are based on the finite state machine (FSM) formalism. An FSM consists of a finite number of states corresponding to those of the object being modelled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has since been used for a wide variety of applications. Apart from modelling, it has even been used in theoretical computer science as a generator for regular languages.

A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. .This problem is overcome in UML by using state charts. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called the nested state). Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.

The basic elements of the state chart diagram are as follows:

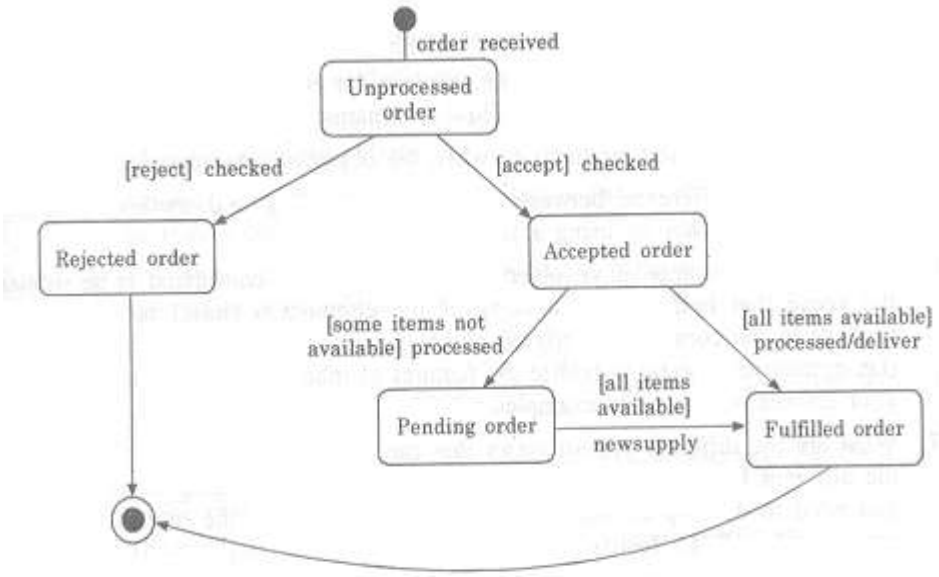
**Initial state**. It is represented as a filled circle.

**Final state.** It is represented by a filled circle inside a larger circle.

**State.** It is represented by a rectangle with rounded corners.

**Transition.** A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is placed along side the arrow. You can also assign a guard to the transition. A guard is a Boolean logic condition. The transition can take place only if the grade evaluates to true. The syntax for the label of the transition is shown in three parts: event[guard]/action.

An example state chart for the order object of the Trade House Automation software is shown in Figure 21.




**Summary**

- In this lesson we have reviewed some important concepts associated with the object-oriented methodology.
- One of the primary advantages of the object-oriented approach is increased productivity of the software development team. The reason why object-oriented projects achieve dramatically higher levels of productivity can be attributed primarily to reuse of predefined classes and partly to reuse due to inheritance, and the conceptual simplicity brought about by the object approach.
- Object modelling plays an important role in analyzing, designing, and under. standing systems. UML has rapidly gained popularity and is poised to become a standard in object modelling.
- UML can be used to construct five different views of a system using nine different kinds of diagrams. However, it is not mandatory to construct all views of a system using all types of diagrams in a modelling effort. The types of models to be constructed depend on the problem at hand.
- We discussed the syntax and semantics of some important types of diagrams which can be constructed using UML.

**Self Check Exercise**

- What is UML used for in software engineering?
- What are the three major elements of UML?
- How many types of Modeling are there in UML?
- What are the principles of modeling in UML?

**Suggested Readings**

"Software Engineering: A Practitioner's Approach" by Roger Pressman, Tata McGraw Hill Publications.

"Software Engineering" by David Gustafson, Schaum's Outlines Series.

"An Integrated Approach to Software Engineering" by Pankaj Jalote.

"Software Engineering" by Ian Sommerville, Pearson Education, Asia.

"Software Engineering Concepts" by Richard Fairley, Tata McGraw Hill Publications.

"Fundamentals of software engineering"by Carlo Ghezi, Mehdi Jazayeri.

"Software engineering: Theory and practice" by Shari Lawrence Pfleeger.

"Fundamentals of Software Engineering" by Rajib Mall., PHI-India.

"Software Testing Techniques" by Boris Beizer, Dreamtech Press, New Delhi.

"Software Engineering" by K.K. Aggarwal, Yogesh Singh, New Age International Publishers.

# Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.

# Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.