CONTENTS

Chapter 1

INTRODUCTION TO SOFTWARE ENGINEERING

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- Software crisis.
- What is software engineering?
- Different stages of software development process.
- Why Software Development Lifecycles (SDLC) are important?
- Popular Software Development Lifecycles used for developing software.
- Selecting a SDLC for a project.
- Types of software.

1.1 INTRODUCTION

In the current scenario, information systems are important part of any organization. As compared to 1970's and 1980's, they are becoming more and more complex. In the early years i.e., 1940's, software development was not an independent established discipline. Instead it was only an extension of the hardware. Earlier programs were written mostly in assembly language and were not complex. The persons/users who did programming were the one who also executed, tested and fixed the problems/errors in the software.

As the information systems became more and more complex and organizations became more dependent on software, a need was felt to develop the software in a systematic fashion. A survey was conducted by researchers in seventies and it was found that

most of the software used by companies was of poor quality. Also companies were spending most of their time and money in maintaining the software. They found that software product was not same as hardware product as it was to be engineered or developed and it did not wear out. Programmers were clear about civil, mechanical, electrical and computer engineering but it was always a topic of debate that what engineering might mean for software.

Most of the people consider the program and software to be same. But software not only consists of programs but also the supporting manuals.

Software can be defined as a set of instructions which when executed on a computer accepts the inputs and after doing required computations produces the output or result as per users requirements. Additionally it is also accompanied by the user manual so as to understand the features and working of the software.

Hence the software consists of following:

- Source code
- Executables
- User manuals
- Requirement analysis and design documents
- Installation manuals

Some important characteristics of software are:

- As compared to hardware which follows the "bathtub curve" as shown in Fig. 1.1, software does not wear out as over a period of time it will become more reliable.

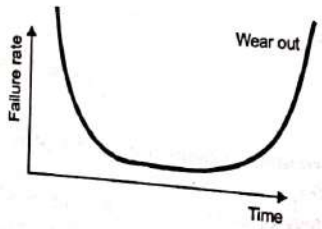


Fig. 1.1 Bath Tub Curve

Instead the software becomes obsolete due to new operating environment, new user requirements etc.. Hence it can only retire but not wear out. So it should follow the curve shown in Fig. 1.2.

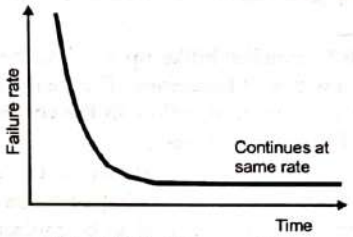


Fig. 1.2 The Software Curve

- Just like hardware or any other engineering product, software is not manufactured. Instead it is developed or engineered. It is built only once and then multiple copies can be produced. On the other hand in other streams every product has to be made from the start using the new components.
- Most software is **custom-built**, rather than being assembled from existing components.
- It is flexible and hence can easily accommodate the changes. If changes are not accommodated in a planned manner it can also led to failure of the product.

1.2 SOFTWARE CRISIS

From 1960's to 1980's , software engineering was spurred by so called ***software crisis.*** A number of large size projects failed called ***software runaways*** because of following reasons:

- Development teams exceeding the budget
- Late delivery of software
- Poor quality
- User requirements not completely supported by the software
- Difficult maintenance
- Unreliable software

Statistics show that only 2% of the projects were used as they were delivered, 3% of the projects used after modifications, 47% of the software was never used only delivered, 19% of the software rejected or reworked and 29% was not even delivered. The problems increased because of increased dependence of business on software and lack of systematic approach to build the software. Developers and researchers realized that development of software was not an easy and straight forward task, instead it required lot of engineering principles.

Some of the examples of software project failures are listed below:

- In June 1996, Anane 5 launcher broke up and exploded after 40 seconds of take off at an attitude of less than 4 kilometers. The total loss was $ 500 million. It was found that the error was due to overflow in the conversion from a 64 bit floating point number to 16 bit signed integer.
- In early eighties, M/s Sperry Corporations of US was hired by the Internal Revenue Service to automate the processing of income tax forms but as system was inefficient to handle the load it had to be replaced. As a result $ 90 million was spent in addition to the original $ 103 million to enhance the Sperry Corporation equipment. Internal Revenue Service paid $ 40.2 million as interest to customers and $ 21.3 million in overtime wages to its employees.
- Another shocking case of improperly designed software is about Therac-25 a radiation therapy and X-ray machine. This machine killed several patients due to malfunctioning of arrow keys which were not programmed properly by the designers. As a result high dose of radiation was given to patients whereas only low levels were required.
- In early nineties British House of Common Public Accounts Committee, highlighted in a report – an overspend of £4b pounds as a result of project cost over-runs on an annual budget of £8.2b, the software being the major culprit.
- Ministry of Agriculture in UK alone had to undergo a loss of 12 million pounds because of software errors.
- Even the launch of space shuttle Columbia was delayed by three years thus costing millions of dollars.

Developers and researchers found that there was lot of scope for building quality software. As a result of this a need was felt to systematize the development of software. A NATO Science Committee sponsored two conferences in 1968 (in Garnish, Germany) and 1969. Many believed that these conferences marked the official start of the software engineering profession. The conference focused on the design and development of techniques for producing error free software. The concept of software life cycle was also introduced to depict the different stages of software development.

1.3 NO SILVER BULLET

A lot of methodologies and automated tools were introduced to support different stages of software life cycle from 1970's to 1990's. As finding the solution to software crisis was the main objective of the researchers and development organizations, every new tool or technology proposed from 1970's to 1990's was used as a ***silver bullet*** to solve the software crisis. In fact whereas in seventies lot of software engineering principles were introduced, in eighties they were supported by CASE (Computer Aided Software Engineering) tools. Following were touted as silver bullets:

- Structured Programming Languages
- Object Oriented Programming Languages like Java, C++
- Formal methods like Modern Structured Analysis, Object Modeling Technique, Unified Modeling Language
- Process Models like Software Engineering Institute's Capability Maturity Model (CMM), SPICE (Software Process Improvement and Capability Determination),
- Standards representing requirements, design etc.
- Professionalism on the part of team members
- Project management techniques and many more

It was also found by the practitioners that no single silver bullet can solve all the problems of software engineering disciplines. It is only the combination of some of them which can make the projects succeed.

1.4 WHAT IS SOFTWARE ENGINEERING?

Over the years software developers have understood that software development is not merely coding. It is something which starts long before one actually starts programming and continues even after the first version of the software is delivered. Hence it consists of number of activities in addition to programming. Software engineering as a discipline provides methods of systematically developing and maintaining that software.

A number of definitions of software engineering can be found in literature. Some of the definitions are given below:

According to (Pfleeger 87) "Software engineering is a strategy for producing quality software."

Still another definition of software engineering given at NATO Conference is:

"It is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."
Software engineering can also be defined as the application of scientific principles to
(i) Systematic transformation of a problem into a working software solution.
(ii) The subsequent maintenance of that software after delivery until the end of its life.

According to IEEE standard 610.12-1990 software engineering is defined as
1. ***The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software i.e., the application of engineering to software.***
2. ***The study of approaches as in (1).***

In short software engineering as a discipline provides tools and techniques for developing the software in an orderly fashion. The advantages of using software engineering for developing software are:

- Improved quality
- Improved requirement specification
- Improved cost and schedule estimates
- Better use of automated tools and techniques
- Less defects in final product
- Better maintenance of delivered software
- Well defined processes
- Improved productivity
- Improved reliability

1.4.1 How Software Engineering is Different from Other Professions?

The area of computer science is concerned with theory and fundamentals and focuses on developing new algorithms, data structures, languages, tools and techniques. Software engineering uses these concepts and techniques to build the application and hence is concerned with the practicalities of developing the software.

Another important area is that of systems engineering. Systems engineering is concerned with all the aspects of computer based systems development and hence includes software, hardware and processes as well. Hence software engineering is a part of systems engineering and focuses on developing the software specification, architectural design, testing and deployment of software part of the system.

1.5 SOFTWARE DEVELOPMENT PROCESS

Developers should be able to deliver good quality software to the end users using a well defined, well managed, consistent and cost-effective process. A software process framework therefore describes the different phases of the project via the activities performed in each phase without telling about the sequence in which these phases or activities will be conducted.

A software life cycle model is a type of process that represents the order also in which the activities will take place. It describes the sequence of activities graphically to build the final product. As different phases may follow one another, repeat themselves or even run concurrently, the sequence may or may not be linearly sequential.

The different phases of software development process are shown in Fig. 1.3. Each of these phases will have different activities to meet its objectives. The process starts by collecting the user requirements and representing them in a suitable form which must be understandable by developers, analyst, users, testers and all other stake holders of the project. This stage is called the requirements analysis stage and the output of this stage

is a document called Requirement Specification Document (RSD) or Software Requirement Specification(SRS). System design stage focuses on high level design of software. In program implementation stage actual coding is done and thereafter product is tested to ensure an error free product. Once the product is delivered, it's maintenance is done by the organization. In rest of the book we will discuss different phases of the software development life cycle in detail.

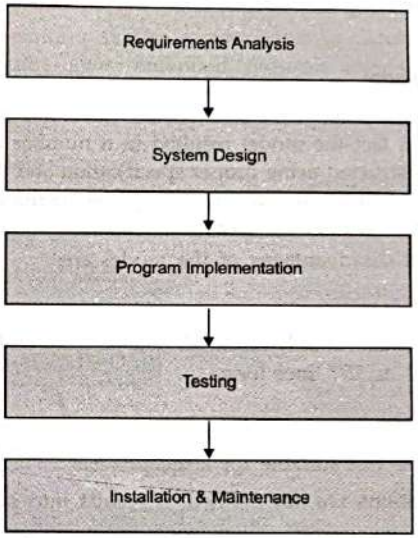


Fig. 1.3 Phases of Software Development Process

1.6 WHY ARE SOFTWARE LIFE CYCLE MODELS IMPORTANT?

The duration of time that begins with conceptualization of software being developed and ends after system is discarded after its usage, is denoted by Software Development Life Cycle (SDLC). A number of software life cycle models have been proposed by the researchers to organize the software engineering activities into phases. While adopting a software process for developing a product, the question which immediately comes to the mind is that whether the process is the right process to be adopted which will ensure a good quality product. It is normally seen that as complexity and size of the project increases, need for a formal process also increases. In order to deliver a high quality software project managers and their team members, end users and other stake holders of the project must agree on common terminology and a software development life cycle model. The model thus selected guides all the stake holders of the project in monitoring their progress. Hence model to be used in any project must be carefully selected. In the next section characteristics of popular software development lifecycle models are highlighted which guide the project members in selecting the appropriate model for their project.

1.7 SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

Some of the widely used, well known software life cycle models are the Waterfall model, V model, Prototyping model, Incremental model, Spiral model etc. Depending upon the scope, complexity and magnitude of the project a particular software life cycle model is selected and this selection of life cycle model significantly contributes towards the successful completion of the project. In the following sections we will be briefly discussing each of these models.

1.7.1 Build and Fix Model

Techniques used in the initial years of software development resulted into the term Build and Fix model. In fact the model resulted in a number of project failures because the product was not constructed using proper specification and design. Instead the product was reworked number of times in order to satisfy the clients as shown in Fig. 1.4. This model has only historical importance now.

The advantages and disadvantages of the model are:

Advantages

- The model is useful only for small size projects, in fact only for programming exercise 100 or 150 lines long.

Disadvantages

- The model is not suitable for large projects.
- As specifications are not defined, it results into product full of errors.
- Reworking of product results into increased cost.
- Maintenance of the product is extremely difficult.

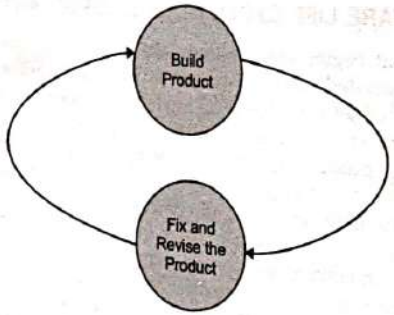


Fig. 1.4 Build and Fix Model

1.7.2 The Waterfall Model

The Waterfall model is one of the most used model of 70's. It was proposed by Royce in 1970 as an alternative to Build and Fix software development method in which code was written and debugged. System was not formally designed and there was no way to check the quality criteria. Different phases of Waterfall model are shown in Figure 1.5.

Given below is a brief description of different phases of Waterfall model.

- ***Feasibility study*** explores system requirements to determine project feasibility. All projects are feasible given unlimited resources and infinite time(Pressman92). Feasibility can be categorized into
  - *Economic feasibility*
  - *Technical feasibility*
  - *Operational feasibility*
  - *Schedule feasibility*
  - *Legal and contractual feasibility*
  - *Political feasibility*

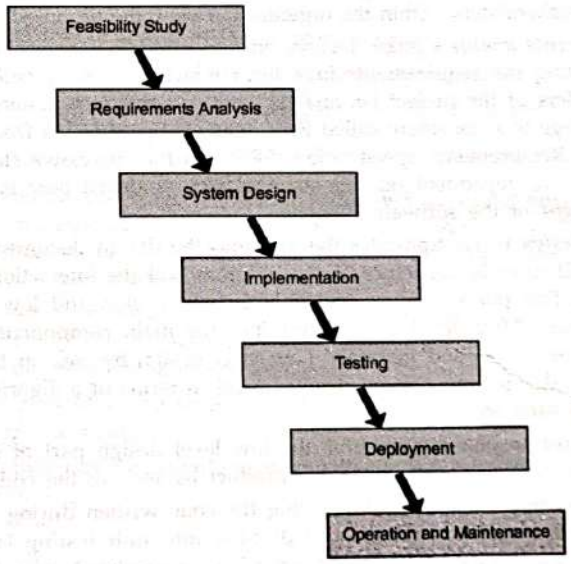


Fig. 1.5 The Waterfall Model

Economic feasibility is also called ***cost-benefit analysis*** and focuses on determining the project costs and benefits. Benefits and costs can be both tangible as well as

intangible. Tangible costs and benefits are the ones which can be easily measured whereas intangible ones cannot be easily measured. Examples of tangible benefits are reduced errors, improved planning and control, reduced costs etc. Intangible benefits include timely information, better resource control, improved information processing, better assets utilization and many more. Similarly tangible costs include cost of hardware and software required in the project, training costs, operational costs and labor costs whereas intangible costs include loss of customer goodwill, decreased operation efficiency and many more. Economic feasibility uses the concept of ***time-value of money(TVM)*** which compares the present cash outlays to future expected returns.

Technical feasibility focuses on organization's ability to construct the proposed system in terms of hardware, software, operating environments, project size, complexity, experience of the organization in handling the similar types of projects and the risk analysis. Operational feasibility deals with assessing the degree to which a proposed system solves business problems. Similarly schedule feasibility ensures that the project will be completed well in time. Legal and contractual feasibility relates to issue like intellectual property rights, copyright laws, labor laws and different trade regulations. Political feasibility finally evaluates how the key stakeholders within the organization view the proposed system(Jeffrey01)

- ***Requirements analysis*** phase focuses on understanding the problem domain and representing the requirements in a form which are understandable by all the stakeholders of the project i.e. analyst, user, programmer, tester etc. The output of this stage is a document called Requirements Specification Document (RSD) or Software Requirements Specification (SRS). All the successive stages of software life cycle are dependent on this stage as SRS produced here is used in all the other stages of the software lifecycle.
- ***System design*** phase translates the SRS into the design document which depicts the overall modular structure of the program and the interaction between these modules. This phase focuses on the high level design and low level design of the software. High level design describes the main components of a software and their externals and internals. Low level design focuses on transforming the high level design into a more detailed level in terms of a algorithms used, data structures used etc.
- ***Implementation*** phase transforms the low level design part of software design description into a working software product by writing the code.
- ***Testing phase*** is responsible for testing the code written during implementation phase. This phase can be broadly divided into unit testing (tests individual modules), integration testing (tests groups of interrelated modules) and system testing (testing of system as a whole). Unit testing verifies the code against the component's high level and low level design. It also ensures that all the statements in the code are executed at least once and branches are executed in all directions.

Additionally it also checks the correctness of the logic. Integration testing tests the inter modular interfaces and ensures that the module drivers are functionally complete and are of acceptable quality. System testing validates the product and verifies that the final product is ready to be delivered to the customers. Additionally several tests like volume tests, stress tests, performance tests etc., are also done at the system testing level.

- ***Deployment phase*** makes the system operational through installation of system and also focuses on training of user.
- ***Operations and maintenance*** phase resolves the software errors, failures etc., enhances the requirements if required and modifies the functionality to meet the customer demands. This is something which continues throughout the use of product by the customer.

Advantages and disadvantages of the Waterfall model are listed below:

Advantages

- Easy to understand even by non-technical persons i.e., customers.
- Each phase has well defined inputs and outputs e.g., input to system design stage is Requirement Specification Document(RSD) and output is the design document.
- Easy to use as software development proceeds.
- Each stage has well defined deliverables or milestones.
- Helps the project manager in proper planning of the project.

Disadvantages

- The biggest drawback of Waterfall model is that it does not support iteration. Software development on the other hand is iterative i.e., while designing activities are being carried out, new requirements can come up. Similarly while product is being coded, new design and requirement problems can come up.
- Another disadvantage of Waterfall model is that it is sequential in nature. One cannot start with a stage till preceding stage is completed e.g., one cannot start with the system design till all the requirements are understood and represented.
- Users have little interaction with the project team. Their feedback is not taken during development.
- Customer gets opportunity to review the product very late in life cycle because the working version of product is available very late in software development life cycle.
- Model is very rigid because output of each phase is prerequisite for successive stage.
- The Waterfall model also has difficulty in accommodating changes in the product after the development process starts.

- Amount of documentation produced is very high.
- The model in no way supports delivery of system in pieces.
- The model is not suitable for new projects because of uncertainty in the specifications.

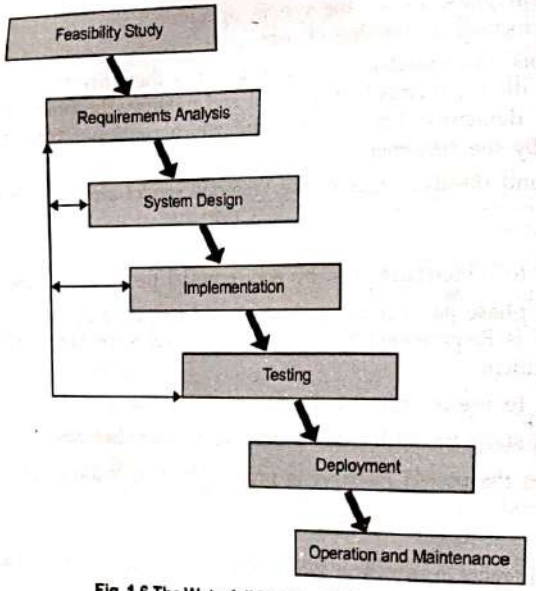


Fig. 1.6 The Waterfall Model with Feedback

Though Waterfall model has been used for large projects in the past, its use must be limited to projects in which requirements are well understood or the company is working on a product of similar kind which it has developed in the past. Modified version of Waterfall model shown in Fig. 1.6 allows feedback to preceding stages and hence is not very rigid. This model clearly shows that the development team can go back to previous phases in order to have better clarity and understanding.

The Waterfall model is suited for well understood projects using familiar technology. It can also be used for existing projects if changes to be made are well defined.

1.7.3 The V-Model

This model was developed to relate the analysis and design activities with the testing activities and thus focuses on verification and validation activities of the product. As this model relates the analysis and design phase to testing phase, testing activities are planned early in software lifecycle as shown in the Fig. 1.7.

The dotted lines in the figure 1.7 indicate that the corresponding phases must be carried out in parallel. As in the case of waterfall model, V model should be used, when all the requirements of the project are available in the beginning of the project. The advantages and disadvantages of the model are listed below.

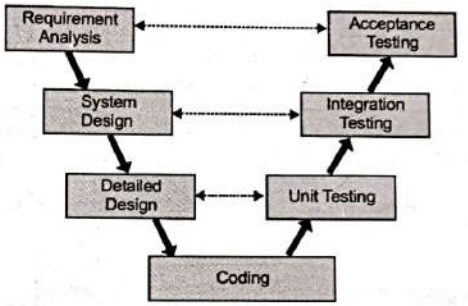


Fig. 1.7 The V Model

Advantages

- The model is simple and easy to use.
- The V model focuses on testing of all intermediate products, not only the final software.
- The model plans for verification and validation activities early in the life cycle thereby enhancing the probability of building an error free and good quality product.

Disadvantages

- The model does not support iteration of phases and change in requirements throughout the life cycle.
- It does not take into account risk analysis.

The V model is used for systems in which reliability is very important e.g., systems developed to monitor the state of the patients, software used in radiation therapy machines.

1.7.4 The Prototype Model

The concept of prototyping is not new in various streams of engineering. A prototype is a partially developed product. Robert T. Futrell and Shafer in their book Quality Software Project Management define prototyping as a process of developing working replica of a system(Robert02). This activity of prototyping now forms the basis of prototype software development life cycle model. Most of the users do not exactly know what they want until they actually see the product. Prototyping is used for developing a mock-up of product and is used for obtaining user feedback in order to refine it further as shown in Fig. 1.8.

In this process model, system is partially implemented before or during analysis phase thus giving the end users an opportunity to see the product early in the life cycle. The process starts by interviewing the users and developing the incomplete high level paper model. This document is used to build the initial prototype supporting only the basic functionality as desired by the user. The prototype is then demonstrated to the user for his/her feedback. After user pinpoints the problems, prototype is further refined to eliminate the problems. This process continues till the user approves the rapid prototype and finds the working model to be satisfactory.

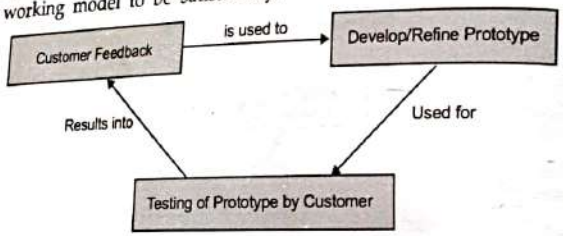


Fig. 1.8 Prototyping Concept

Two approaches of prototyping can be followed:

(i) *Rapid Throwaway Prototyping:* This approach is used for developing the systems or part of the systems where the development team does not have the understanding of the system. The quick and dirty prototypes are built, verified with the customers and thrown away. This process continues till a satisfactory prototype is built. At this stage now the full scale development of the product begins.

(ii) **Evolutionary Prototyping:** This approach is used when there is some understanding of the requirements. The prototypes thus built are not thrown away but evolved with time. The block diagram of the prototype model is shown in Fig. 1.9. The concept of prototyping has also led to the Rapid prototyping model and the Spiral model.

The advantages and disadvantages of the prototyping model are listed below:

Advantages

- A partial product is built in the initial stages. Therefore customers get a chance to see the product early in the life cycle and thus give necessary feedback.
- New requirements can be easily accommodated, as there is scope for refinement.
- Requirements become more clear resulting into an accurate product.
- As user is involved from the starting of the project, he tends to be more secure, comfortable and satisfied.
- Flexibility in design and development is also supported by the model.

Disadvantages

- After seeing an early prototype end users demand the actual system to be delivered soon.

- End users may not like to know the difference between a prototype and a well engineered fully developed system.
- Developers in a hurry to build prototypes may end up with sub-optimal solutions.
- If not managed properly, the iterative process of prototype demonstration and refinement can continue for long duration.
- If end user is not satisfied with initial prototype, he may loose interest in the project.
- Poor documentation.

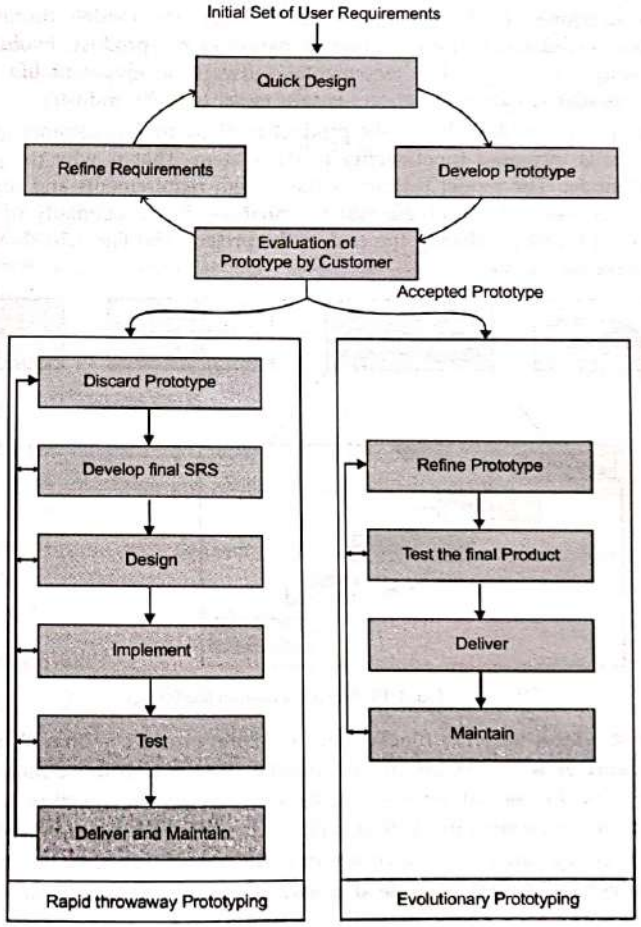


Fig. 1.9 The Prototype Model

Prototype model should be used when requirements of the system are not clearly understood or are unstable. It can also be used if requirements are changing quickly. This model can be successfully used for developing user interfaces, high technology software intensive systems, and systems with complex algorithms and interfaces. It is also a very good choice to demonstrate technical feasibility of the product.

1.7.5 The Incremental Software Development Life Cycle Model

Software like all other complex systems is bound to evolve due to changing business requirements or new requirements coming up. Hence there is a need to have a model which can accommodate the changes in the product. The models discussed earlier do not take into consideration the evolutionary nature of the product. Evolutionary models are also iterative in nature. The incremental software development life cycle model is one of the popular *evolutionary software process model* used by industry.

The incremental model releases the product partially to the customer in the beginning and slowly adds increased functionality to the system. That is why the model is called incremental model. The model prioritizes the system requirements and implements them in groups. Each new release of the system enhances the functionality of the previously released system thereby reducing the cost of the project. The Fig. 1.10 shows the working of the incremental model.

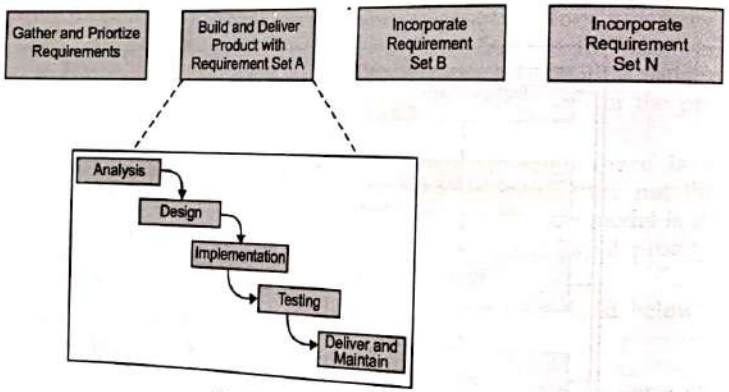


Fig. 1.10 The Incremental Model

In the first release only the functionality A of the product is offered to the customer. The functionality A here consists of core requirements which are critical to the success of the project. In the second release functionality A plus functionality B is offered and finally in release 3 functionality A, B as well as C is offered. Therefore, with each release in addition to incorporating new functionality in the system, functionality of earlier releases may also be enhanced. For example if a text editor is to be built, first version of the

product will have basic editing facilities. The next version of the product can be released with enhanced editing facilities like formatting along with new features like spell checking. The incremental model is used when requirements are defined at the beginning of the project but at the same time they are expected to evolve over time. It can also be used for projects with development schedules more than one year or if deliveries are to be made at regular intervals. The advantages and disadvantages of incremental model are listed below:

Advantages

- As product is to be delivered in parts, total cost of project is distributed.
- Limited number of persons can be put on project because work is to be delivered in parts.
- As development activities for next release and use of early version of product is done simultaneously, if found errors can be corrected.
- Customers or end users get the chance to see the useful functionality early in the software development life cycle.
- As a result of end user's feedback requirements for successive releases become more clear.
- As functionality is incremented in steps, testing also becomes easy.
- Risk of failure of a product is decreased as users start using the product early.

Disadvantages

- As product is delivered in parts, total development cost is higher.
- Well defined interfaces are required to connect modules developed with each phase.
- The model requires well defined project planning schedule to distribute the work properly.
- Testing of modules also results into overhead and increased cost.

1.7.6 The Spiral Model

The Spiral model is also one of the popular evolutionary process model used by the industry. The Spiral model was proposed by Boehm in 1988 and is a popular model used for large size projects. The model focuses on minimizing the risk through the use of prototype. One can view the Spiral model as a Waterfall model with each stage preceeded by the risk analysis stage. A simplified view of Spiral model is shown in Fig. 1.11.

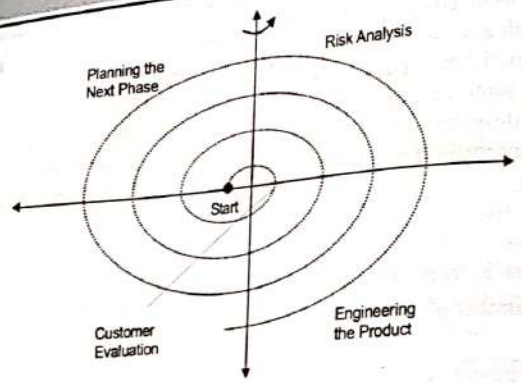


Fig. 1.11 The Spiral Model

The radial coordinate in the diagram represents the total costs incurred till date. Each loop of the spiral represents one phase of the development. The model is divided into four quadrants, each with a specific purpose. Each spiral represents the progress made in the project. In the first quadrant, objectives, alternative means to develop product and constraints imposed on the product are identified. The next quadrant (right upper) deals with identification of risks and strategies to resolve the risks. The third bottom right quadrant represents the Waterfall model consisting of activities like design, detailed design, coding and test. With each phase after customer evaluates the product, requirements are further refined and so is the product. It is to be noted that number of loops through the quadrants are not fixed and vary from project to project.

The steps followed while using Spiral model can be described as follows. The process starts with identification and prioritization of risks. A series of prototypes are developed for the risks identified (highest risk is considered first). For each development cycle of the Spiral model shown in Fig. 1.11 Waterfall model is used. If a risk is resolved successfully planning for the next cycle is done. If at some stage risk cannot be resolved project is terminated.

Spiral model is also termed as *process model generator* or *meta model*. For example if any project requirements are not clear models like Prototyping or Incremental can be derived from the spiral model. The advantages and disadvantages of spiral model are listed below:

Advantages

- The model tries to resolve all possible risks involved in the project starting with the highest risk.
- End users get a chance to see the product early in life cycle.
- With each phase as product is refined after customer feedback, the model ensures a good quality product.

- The model makes use of techniques like reuse, prototyping and component based design.

Disadvantages

- The model requires expertise in risk management and excellent management skills.
- The model is not suitable for small projects as cost of risk analysis may exceed the actual cost of the project.
- Different persons involved in the project may find it complex to use.

Spiral model is generally used for large projects with medium to high risk because in small projects it is possible that cost of risk analysis may exceed the actual cost of project. In other words, Spiral model is a practical approach for solving large scale software development related problems.

This can also be used if requirements of the project are very complex or if the company is planning to introduce new technologies. Some areas where Spiral model is successfully used are decision support system, defense, aerospace, and large business projects.

1.7.7 The Rapid Application Development (RAD) Model

The Rapid Application Development (RAD) model was proposed by IBM in 1980s and later on was introduced to software community by James Martin through his book Rapid Application development. The important feature of RAD model is increased involvement of the user/customer at all stages of life cycle through the use of powerful development tools.

If the requirements of the software to be developed can be modularized in such a way that each of them can be completed by different teams in a fixed time then the software is a candidate for RAD. The independent modules can be integrated to build the final product. The important feature of the RAD model is quick turnaround time from requirement analysis to the final delivered system. The time frame for each delivery is normally 60 to 90 days called *time box* which is achieved by using powerful developer tools like Visual C++, JAVA, Visual BASIC, XML, .NET etc. Block diagram of RAD model is shown in Fig. 1.12.

The RAD model consists of following four phases:

- Requirements Planning – focuses on collecting requirements using elicitation techniques like brainstorming,
- User Description – Requirements are detailed by taking users feedback by building prototype using development tools.
- Construction – The prototype is refined to build the product and released to the customer.
- Cutover – involves acceptance testing by the user and their training.

The process therefore starts with building a rapid prototype (a working model which is functionally equivalent to a subset of final product) and is delivered to customer for use and his feedback. Once the user/customer validates the rapid prototype after using it, Requirement Specification Document is derived and design is done to give final shape to the product. After the product is installed, maintenance of the product is continued by refining the requirements, specification, design or coding phase.

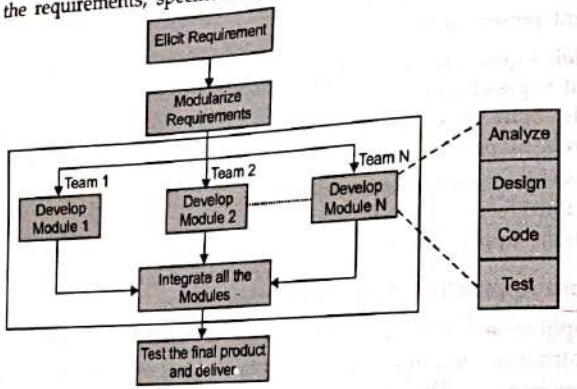


Fig. 1.12 The RAD Model

The advantages and disadvantages of RAD model are discussed below:

Advantages

- As customer is involved at all stages of development, it leads to a product achieving customer satisfaction.
- Usage of powerful development tools results into reduced software development cycle time.
- Feed back from the customer/user is available at the initial stages.
- Makes use of reusable components, to decrease the cycle time.
- Results into reduced costs as less developers are required.

Disadvantages

- The model makes use of efficient tools, to develop the prototype quickly, which calls for hiring skilled professional.
- Team leader must work closely with developers and customers/users to close the project in time.
- Absence of reusable components can lead to failure of the project.

The RAD model should be used for system with known requirements and requiring

short development time. Also It is appropriate to use RAD model for the system that can be modularized and also reusable components are available for development. The model can also be used when already existing system components can be reused in developing a new system with minimum changes.

Comparison of different process models in tabular form is shown in Table 1.1.

Table 1.1 Comparison of Software Process Models

	Waterfall	*V Model*	*Incremental*	*Spiral*	*Prototype*	*RAD*
1. Well defined requirements	Yes	Yes	No	No	No	Yes
2. Domain knowledge of the team members	Adequate	Adequate	Adequate	Very less	Very less	Adequate
3. Expertise of users in problem domain	Very less	Very less	Adequate	Very less	Adequate	Adequate
4. Availability of reusable components	No	No	No	Yes	Yes	Yes
5. Users involvement in all phases of SDLC	No	No	No	No	Yes	Yes
6. Complexity of system	Simple	Simple	Complex	Complex	Complex	Medium

1.8 TYPES OF SOFTWARE

The software is being used in almost all the spheres of human life e.g., hospitals, banks, defense, finance, predicting stock rates, making pictures, running other software and so on. In fact the list is endless. Though it is somewhat difficult to categorize the software in different types but ***based on their applications*** the software can be categorized into following areas:

- System software
- Scientific software
- Networking and web applications software
- Embedded software
- Business software
- Utilities software
- Artificial intelligence software

System Software: Systems software is necessary to manage the computer resources and support the execution of application programs. Software like operating systems, compilers,

editors and drivers etc., come under this category. Operating systems are needed to link the machine-dependent needs of a program with the capabilities of the machine on which it runs. Compilers translate programs from high-level languages into machine languages. Without the presence of system software, a computer cannot function.

Scientific Software: Scientific and engineering software satisfies the needs of a scientific or engineering user to perform enterprise-specific tasks. Such software are written for specific applications using the principles, techniques and formulae specific to that field. Examples are software like MATLAB, AUTOCAD, PSPICE, ORCAD etc.

Networking and Web Applications Software: Networking software provides the required support necessary for computers to interact with each other, and with data storage facilities, in a situation where multiple computers are necessary to perform a task. The networking software is also used when software is running on a network of computers (such as the Internet or the World Wide Web). This category of software include all network management software, server software, security and encryption software and software to develop Web based applications like HTML, PHP, XML etc.

Embedded Software: This type of the software is embedded into the hardware normally in the Read Only memory(ROM) as a part of large system and is used to support certain functionality under the control conditions. Examples are software used in instrumentation and control applications, washing machines, satellites etc.

Business Software: This category of software is used to support the business applications and is the most widely used category of software. Examples are software for inventory management, accounts, banking, hospital, schools, stock markets etc., The software written for Enterprise Resource planning(ERP), project management , workflow management etc., also come under this category.

Utilities Software: The programs coming under this category perform specific tasks and are different from other software in terms of size cost and complexity. Examples are anti-virus software, voice recognition software, compression programs etc.

Artificial Intelligence Software: Software like expert systems, decision support systems, pattern recognition software, artificial neural networks etc., come in this category of software. Such type of software solve complex problems which are not affected by complex computations using non numerical algorithms.

In terms of *copyright*, there are four broad types of software:

- Commercial
- Shareware
- Freeware
- Public domain

Commercial software represents the majority of software which we purchase from software companies, commercial computer stores, etc. In this case when we buy the software, we actually acquire a license to use it, but not own it. Users are not allowed

to make the copies of the software. The copyright of the program is owned by the company. *Shareware* software is also covered by copyright, but the copyright holders for shareware allow purchasers to make and distribute copies of the software, but demand that if, after testing the software, purchaser adopts it for use, then he must pay for it. In both the types of software changes to the software is not allowed and derivative works is also not allowed without the permission of the copyright holder.

Freeware software is also covered by copyright subject to the conditions defined by the holder of the copyright. In general, according to freeware software licenses, copies of the software can be made for both archival and distribution purposes but in this case distribution cannot be for making profit. Derivative works and modifications to the software are allowed and encouraged . Decompiling of the program code is also allowed without the explicit permission of the copyright holder. In the case of *Public domain* software the original copyright holder explicitly relinquishes all rights to the software. Hence software copies can be made for both archival and distribution purposes with no restrictions as to distribution. Similarly modifications to the software are allowed and reverse engineering of the program code is allowed. Development of new works built upon the package is allowed without conditions on the distribution.

In addition to above there are software with *academic* licenses under which the owner of the software allows the universities, colleges and schools to use the software for education and research purpose at a very nominal price. But the users in this case cannot use the software commercially. The copyright of the software is also owned by the company.

SUMMARY

- Software worth billions and trillions of dollars have gone waste in the past due to lack of proper techniques used for developing software resulting into software crisis.
- Software engineering is application of engineering principles to develop quality software.
- The duration of time that begins with conceptualization of software being developed and ends after system is discarded after its usage is called Software Development Life Cycle (SDLC).
- A number of software process models are proposed to organize software engineering activities into phases.
- Main phases of software development life cycle are—requirements analysis, system design, coding, testing, delivery and maintenance.
- Popular process models are waterfall model, V model, incremental model, Spiral model, Prototyping model etc.
- Several process maturity frameworks and quality standards are proposed to improve software quality e.g., CMM, SPICE, Bootstrap, ISO9001 etc.

REVIEW PROBLEMS

1. Define software engineering.
2. List the impact of software engineering on developing software.
3. What are the advantages of software process models?
4. What is software crisis? Discuss the reasons which resulted into software crisis.
5. Draw a schematic diagram to represent the incremental software development life cycle. List the advantages and disadvantages of Incremental process model.
6. What is a prototype? What are the advantages of prototyping software development life cycle model?
7. What is the role of risk management activity in Spiral model?
8. Give examples of software for which spiral model is not useful.
9. Define a prototype. Under what conditions prototyping model is preferred over other models.
10. Why feasibility study is important in a project.
11. Explain the following terms – (i) Technical feasibility, (ii) Economic feasibility, (iii) Legal and contractual feasibility.
12. Spiral model is often termed as meta-model. Justify this statement.
13. Can Spiral model be used for small sized projects? Justify your answer.
14. What do the different cycles in the Spiral model indicate?
15. What process model you will follow for developing
    (a) Editor
    (b) CAD software
    (c) Radiation therapy machine software
    (d) A game
    (e) A hospital management system
    (f) A compiler for new language
    Justify your answer.
16. Classify the software based on the copyright. Make a list of software you are using in the laboratory or home and label them based on the copyright classification.
17. What are the different phases of waterfall model? Explain, which phase of the waterfall model requires maximum effort?

ooo

Chapter 2

SOFTWARE PROJECT MANAGEMENT

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- Project and its attributes
- What is Project Management?
- Important Project Management Skills
- The role of a Project Manager in an organization
- Estimating size, effort and cost of a project using different models like COCOMO, Function Point Analysis (FPA) etc.
- Scheduling a project using techniques like Work Breakdown Structure, Gantt Chart, Network Diagram, Critical Path Method and PERT
- Risk Management

2.1 INTRODUCTION

Project management has become an area of keen interest in the recent years. In order to be successful and competitive, organizations whether government or non-government must use the modern project management techniques and employees of the organization must develop the necessary project management skills. An efficient use of project management techniques result into

- High quality and reliable products
- Managing the resources efficiently

- More satisfied customers
- Higher profit margins
- Completing the projects within time and budget
- High productivity

This chapter introduces the concept of a project and project management techniques and other relevant and supportive information.

2.2 WHAT IS A PROJECT?

Several definitions of a project can be found in the literature. Some of them are given below.

As per Oxford dictionary a project can be defined as "an enterprise carefully planned to achieve a particular aim."
According to the Project Management Institute Inc.(PMBOK® Guide 2004) A project is a temporary endeavor undertaken to create a unique product, service or result.

The objective of any project is to build a good quality product well within the budget and schedule. The final product must also satisfy the requirements as demanded by the end user or the customer. In order to do so a project may use people, hardware, software and other resources. Some of the examples of project are:

- Government undertakes a project to educate all the children who are below 15 years.
- A car manufacturing company takes up a project to develop a new model of racing car by the year 2010.
- University introduces the online sale of its admission forms.
- An organization takes up a project to double its sales in the next financial year.
- A software development company adds a new set of features to its existing products.

Every project is characterized by some *attributes* as given below (kathy07), (Robert02):

1. A project has a unique and well defined objective.
2. A project is of fixed duration. It has a definite beginning and definite end point.
3. In order to meet its objectives, a project uses different types of resources from different areas.
4. A project must have a sponsor or customer to fund its activities.
5. A project involves uncertainty.
6. A project is developed using progressive elaboration.
7. Every project in some way or other is constrained by its scope, time and cost goals.
8. A project is one time effort and is not repeatable.

2.3 WHAT IS PROJECT MANAGEMENT?

The main objective of any project is to deliver a product, service or result within the constraints of its scope, time and budget. Project management is therefore defined as:

Management of procedures, techniques, resources, know-how, technology etc., required for successful management of the project.
According to (Kathy07) project management is the application of knowledge, skills, tools and techniques to project activities to meet the project requirements.
According to PMI, project management is defined as a set of proven principles, methods and techniques for the effective planning, scheduling, controlling and tracking of deliverable oriented work(results) that help to establish a sound historical basis for future planning of projects.

If the final product of the project is software, it is called *Software Project Management*. The important **Project Management Skills** are mentioned below:

- Defining the scope of the project—This skill requires defining all the steps or activities and their relationship in order to complete the project successfully using Work Breakdown Structure(WBS)
- Developing project management plans and other documents.
- Project Cost estimation.
- Project Effort estimation
- Developing an acceptable project schedule to ensure timely completion of the project.
- Identifying and resolving the risks.
- Selecting the appropriate project management tools.
- Selecting the appropriate metrics and tracking the progress of the project by using these metrics.
- Procurement management.

Some important project management definitions are given in Table 2.1.

Table 2.1 : Important Project Management Definitions

Program :	A collection of related projects managed in a systematic way.
System :	A collection of organized components to accomplish a specific function.
Task :	Lowest order of work element in a project that can be managed.
Activity :	An element of work done during project having an expected duration, cost and resource requirements. An activity may consist of set of tasks to achieve a purpose. Sometimes it also refers to a *Phase*.
Stake Holders:	Persons involved in or affected by project activities.

2.4 SOFTWARE PROJECTS VERSUS OTHER PROJECTS

Software projects are different from the other types of projects in following many (Hughes03):

- Software products are more complex as compared to other engineering produc
- Progress made in a software project is not immediately visible.
- Software as compared to other products can be changed with much ease wh is a plus point of software projects.

However there are also problems while working on software projects. Some of t major problems are:

- Difficulty in understanding and documenting the requirements.
- Inaccurate cost and schedule estimate due to lack of relevant historical data
- Changing requirements.
- Poor planning.
- Selecting appropriate tools and methodologies for analysis and design.
- Selecting appropriate process model for a project.
- Lack of quality standards.
- Handling resource constraints.
- Meeting deadlines.
- Lack of communication between end users and developers.
- Lack of training.

2.5 THE ROLE OF A PROJECT MANAGER

The project manager is the person who works closely with the project team member sponsor(s) of the project and other stakeholders to ensure successful completion of th project within the cost and time constraints. The project manager in order to do so mus be able to understand the organization behavior as well as the individual behavior. H is the one who is also responsible for guiding the project team members and monitorin the progress made by the team members. He should also be able to develop a wor culture among the team members. Some of the qualifications a project manager mus have are:

- Ability to work with other team members
- Sound knowledge of project management skills
- Good communication skills
- Excellent decision making capabilities
- Knowledge of project management tools
- Profit oriented outlook
- Assigning jobs to appropriate people
- Ability to retain good people

Some of the primary duties performed by the project manager are:

- Document the project management plan and the quality plan.
- Ensure compliance with the processes and standards being used.
- Ensure the proper and timely completion of project.
- Ensure that all required resources are assigned to the project and clearly tasked.
- Manage day-to-day issues.
- Monitor actual progress made in the project against the project plan.
- Provide input to the project review group as necessary.
- Report project status and performance (with respect to schedule, cost, quality and risk) to the management.
- Incorporate changes in the project plan as and when required.
- To coordinate with the third party vendors.

A template for software project management plan is given towards the end of this chapter.

2.6 PROJECT MANAGEMENT PROCESS

Project management process mainly consists of following activities

1. Feasibility study
2. Project planning
3. Project execution
4. Project termination

2.6.1 Feasibility Study

This phase of the project management process does necessary investigation to ensure that the project is worth considering and starting. In case of large sized complex projects, doing the feasibility study is itself a complete project. It is ensured that project is technically and economically feasible. Additionally all the risks and their potential effects on the projects are also evaluated before a decision to start the project is taken.

2.6.2 Project Planning

To achieve success in any sphere of life, a good planning is necessary and hence it applies to software development also. Planning involves making a detailed plan to achieve the objectives. According to (Kelkar03) plan is defined in following ways:

- It is the outcome of the planning activity.
- It depicts the best possible approach for executing a project.

2.4 SOFTWARE PROJECTS VERSUS OTHER PROJECTS

Software projects are different from the other types of projects in following manne (Hughes03):

- Software products are more complex as compared to other engineering product
- Progress made in a software project is not immediately visible.
- Software as compared to other products can be changed with much ease whic is a plus point of software projects.

However there are also problems while working on software projects. Some of th major problems are:

- Difficulty in understanding and documenting the requirements.
- Inaccurate cost and schedule estimate due to lack of relevant historical data.
- Changing requirements.
- Poor planning.
- Selecting appropriate tools and methodologies for analysis and design.
- Selecting appropriate process model for a project.
- Lack of quality standards.
- Handling resource constraints.
- Meeting deadlines.
- Lack of communication between end users and developers.
- Lack of training.

2.5 THE ROLE OF A PROJECT MANAGER

The project manager is the person who works closely with the project team member sponsor(s) of the project and other stakeholders to ensure successful completion of th project within the cost and time constraints. The project manager in order to do so mus be able to understand the organization behavior as well as the individual behavior. H is the one who is also responsible for guiding the project team members and monitorin the progress made by the team members. He should also be able to develop a wor culture among the team members. Some of the qualifications a project manager mus have are:

- Ability to work with other team members
- Sound knowledge of project management skills
- Good communication skills
- Excellent decision making capabilities
- Knowledge of project management tools
- Profit oriented outlook
- Assigning jobs to appropriate people
- Ability to retain good people

Some of the primary duties performed by the project manager are:

- Document the project management plan and the quality plan.
- Ensure compliance with the processes and standards being used.
- Ensure the proper and timely completion of project.
- Ensure that all required resources are assigned to the project and clearly tasked.
- Manage day-to-day issues.
- Monitor actual progress made in the project against the project plan.
- Provide input to the project review group as necessary.
- Report project status and performance (with respect to schedule, cost, quality and risk) to the management.
- Incorporate changes in the project plan as and when required.
- To coordinate with the third party vendors.

A template for software project management plan is given towards the end of this chapter.

2.6 PROJECT MANAGEMENT PROCESS

Project management process mainly consists of following activities

1. Feasibility study
2. Project planning
3. Project execution
4. Project termination

2.6.1 Feasibility Study

This phase of the project management process does necessary investigation to ensure that the project is worth considering and starting. In case of large sized complex projects, doing the feasibility study is itself a complete project. It is ensured that project is technically and economically feasible. Additionally all the risks and their potential effects on the projects are also evaluated before a decision to start the project is taken.

2.6.2 Project Planning

To achieve success in any sphere of life, a good planning is necessary and hence it applies to software development also. Planning involves making a detailed plan to achieve the objectives. According to (Kelkar03) plan is defined in following ways:

- It is the outcome of the planning activity.
- It depicts the best possible approach for executing a project.

- It states the assumptions to best of project managers ability.
- It includes the list of items which will form deliverables.

Deliverables are the items that customer would like to be demonstrated or delivered as part of contract during the development of software and include requirements specification document, design document, manuals, test plans, demonstration of various components etc., (Hughes03) has proposed a framework of project planning steps called Step Wise Method or Step Wise Planning Process. The main steps of this process are.

- Select project
- Identify project scope and objectives
- Identify project infrastructure
- Analyze project characteristics
- Identify project products and activities
- Estimate effort for each activity
- Identify activity risks
- Allocate resources
- Review/publicize plan
- Execute plan

Each of these steps are further supported by number of activities. For details students can refer to the book Software Project Management by Bob Hughes and Mike Cotterell which gives in depth description of the various practices followed during the execution of a software project. According to (Thayer94) planning consists of following activities

- Set objectives or goals
- Develop strategies
- Develop project policies
- Determine courses of action
- Make planning decisions
- Set procedures and rules for the project
- Develop a software project plan
- Prepare budget
- Conduct risk assessment
- Document software project plans

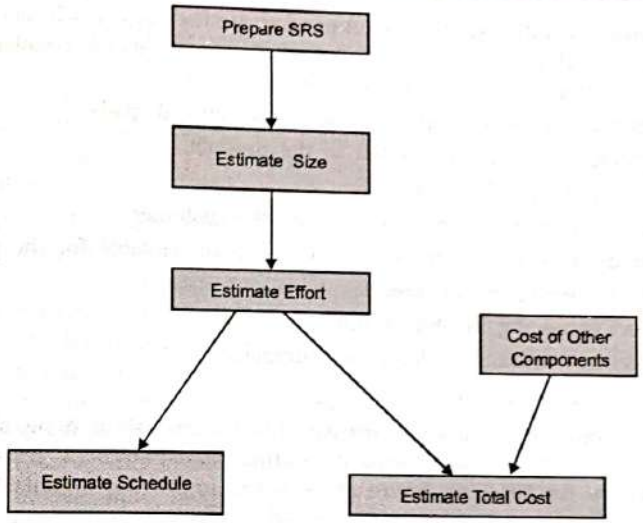


Fig. 2.1 Sequence for Estimating Size, Effort and Cost

Broadly project planning focuses on understanding the projects scope, constructing Work Breakdown Structure (WBS), estimating resource requirements for different project activities, estimating size and schedule, identifying risks and setting up project plan for successful completion of the project. The size of the software to be developed is an important parameter to be estimated which further helps in estimating the effort, cost and schedule of the project. Effort estimation is the primary parameter for estimating the cost and along with the schedule estimation decides the team size of the project. Estimating the cost of the project also involves the cost of manpower, hardware, software, travel, training etc. The order in which these techniques are used is shown in Fig. 2.1. A number of estimation methods are proposed in the literature and are being used in the industry successfully. Some of these will be discussed in section 2.7 and 2.8.

2.6.3 Project Execution

Once the proper project planning is done, project can be executed by selecting appropriate Software Development Lifecycle(SDLC) models (discussed in chapter 1). Though these models are having different characteristics but all of them do follow the following activities of classical Waterfall model i.e. requirements analysis and specification, design, implementation or coding, testing, delivery and maintenance.

2.6.4 Project Termination

Just as project starts due to various reasons, it can also close down or terminate for several reasons. Though we would like the project to terminate successfully, projects can

terminate unsuccessfully also. Successful projects are the ones which meet their objectives well within the budget and schedule. Some of the reasons which result into unsuccessful termination or closedown of the projects are:

- Inconsistency between project and organizational goals.
- Customer no longer interested in the product.
- Fast changing technology.
- Too much change in requirements by the customer.
- Current state of the art technology no longer suitable for the project.
- Project running out of time.
- Project exceeding budget or funds.
- Project not meeting customer requirements.
- Organizational politics.

Once the project is terminated, post-performance analysis is done, staff is reassigned and a final report is also published consisting of experiences, lessons learned and recommendations for handling future projects and improving organization processes.

Describing each and every activity of software project management is outside the scope of this book. However we would be discussing some important project management activities in the next sections of this chapter.

2.7 SIZE ESTIMATION

Estimating the size of the software to be developed takes place during the initial stages of project planning and is crucial to the success of a software project. *Sizing is defined as prediction of coding needed to fulfill requirements.* The size thus estimated is used with other environmental factors to estimate effort, schedule and the project cost.

In order to estimate the size, requirements of the software must be well defined in terms of functional requirements, non-functional requirements, interfaces to be used and the constraints in which the software will work. A number of metrics are proposed in the literature to estimate size. Some of them are:

- Lines of Code (LOC)
- Function points
- Feature points
- Object points
- Number of entities in ER diagram'
- Number of processes in a leveled data flow diagram

Next we will discuss the two popular size metrics being used in the industry i.e. Lines of Code and Function Points.

2.7.1 Lines of Code (LOC)

The basic and simplest metric for estimating the size of a software is Lines of Code (LOC) often quoted in 1000's (KLOC). In a real sense this metric measures the length of a program which is a logical characteristic but not size which is physical characteristic of the program. Estimating the lines of the code of a software before it is written or designed is a tedious job. Still it is one of the most popular metric used in the industry for estimating the size.

While using this technique to estimate the size, the project manager uses the experience of his team members or his/her own experience. Requirements from the WBS or SRS can be converted into the main modules to be developed. At this stage now the experts who have developed similar kind of systems are consulted to get an idea about size of the component on the lower level of WBS. When the size of all the components is added, total size can be found. This is called bottom-up size estimation. Another approach to estimate the size of undeveloped software is by analogy. This is done by comparing the functionality of new system with the existing one and estimating the size. This technique however has some drawbacks because of following reasons:

- Different programming languages used
- Use of different algorithms and data structures
- Different operating environments

While counting number of lines, normally researchers are of the view that comments and blank lines should not be counted. Comments used in the program make the program understandable and easier to maintain but effort required for writing the comments is not as much as writing the code. According to some researchers LOC may be computed by counting the new-line characters in the program. The LOC metric can also be used in other indirect measures as well, such as to find the productivity of the organization. Productivity is defined as:

Productivity = LOC/Effort

Other similar measures used are KDSI (1000's of delivered source instruction) and NLOC, (Non-comment lines of Code).

Advantages of using LOC as size metric are:

- Simple to use
- Universally accepted
- Estimates size from developers point of view

Disadvantages are:

- It is difficult to estimate LOC accurately early in the SDLC.
- Different programming languages may result in different values of the LOC
- No industry standards are proposed to compute LOC.

2.7.2 Function Points as a Unit of Size

Function point analysis (FPA) is a popular method which uses complexity and the number of functions supported by the software to compute the size of the software in terms of Function Point Count (FPC). The method was initially proposed by Allan J. Albrecht of IBM in late 1970s and uses top down approach. Currently International Function Point User's Group (IFPUG) is actively working to evolve this method further.

The underlying concept is that the size of the software being developed is directly proportional to the number of functions it will support. Each of these functions are also characterized by the inputs taken, outputs given, interfaces and the number of files used. Hence these factors will also contribute in calculating the total function points. The steps used for computing Function Points are shown in Fig. 2.2. and are described below briefly:

1. Identify and count following data types function:

- External Inputs (EI)
- External Outputs (EO)
- External Inquiries (EIQ)
- Internal Files (IF)
- External Interface Files(EIF)

External inputs are the events taking place in the system which result into change of data in the system. In other words, they are user or control data entering the system.

External outputs are user and control data coming out of the system e.g., a report, display of data or error message etc.

Inquiries do not change the system data. Instead in this case, inputs from user cause immediate response or generation of output data. Therefore they represent input-output combination.

Internal files are files maintained and understood by customers. Log or index files, do not come under this category.

External interface files are files which are shared by the system and other programs. These files are not maintained by the software.

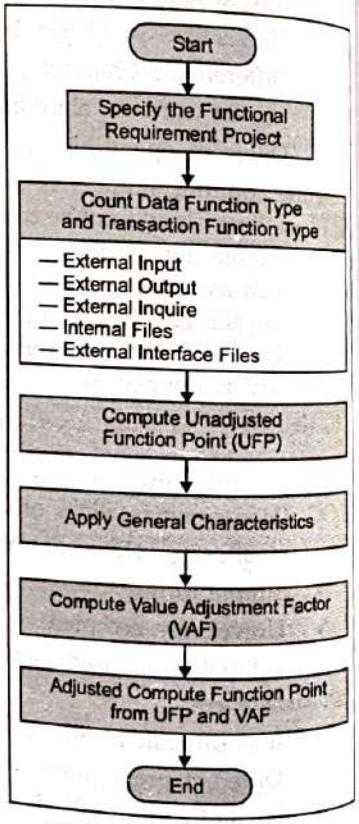


Fig. 2.2 Steps for Computing Function Points

2. Calculate unadjusted function points: In this step first the complexity level of each of data function type and transaction function type as discussed in step 1 is identified. The complexity of EI, EO and EIQ depends on the number of data element types and file types referenced(FTR). For example when a transaction is triggered due to some event taking place in the environment, some information will be updated. In a library management system for example when a person applies for membership of the library, program will update the membership file and cards file by adding new records. Hence in this case number of files referenced are two. Similarly the complexity of internal files and external interface files depends upon the number of data element types and number of record element types in a file. Once the unadjusted function points are computed, based on the complexity level they are multiplied by the weighting factors as given in table 2.2.

Table 2.2: Unadjusted Function Points for Data Type Functions

	Weighting Factors (Multipliers)		
	Simple	*Average*	*Complex*
External Inputs(EI)	3	4	6
External Outputs(EO)	4	5	7
External Inquiries(EIQ)	3	4	6
Internal logical Files(IF)	7	10	15
External Interface Files(EIF)	5	7	10

For example if complexity level of all the five entities is average, total Unadjusted Function Point (UFP) is given by:

$$UFP = 4\,(EI) + 5\,(EO) + 4\,(EQ) + 10\,(IF) + 7\,(EIF)$$

3. Apply 14 general characteristics: FPA method further proposed 14 general characteristics with a rating at 0-5 scale which influence the Function Point Count (FPC). Sum of these 14 general characteristics rating can be between 0-70 and is called Total Degree of Influence (TDI). It means that TDI will be high for system with complex non-functional requirements.

The 14 characteristics as considered by IFPUG are: data communications, performance, transaction rate, processing, reusability, end user efficiency, online update, configuration, multiple sites, operations, change requirements, distributed processing, online data entry and installation.

For example if the volume of the transaction is very high or unpredictable, it will be given a rating equal to 5. But if the transaction is extremely low then its value is equal to 0.

4. Compute the Value Adjustment Factor (VAF) using formula:

$$VAF = (TDI * 0.01) + 0.65$$

5. **Calculate Function Point Count using formula:**

$$FPC = UFP * VAF$$

Finally these function points can be directly mapped to lines of code, depending upon the type of language used.

The main advantages of function point technique is that as function points are based on functionality of the software, they can be computed early in the project and are directly derived from SRS. These function points are also independent of the technology i.e. programming languages, database etc., used to develop the software. Another advantage of using function points is that it can be used as a technique to measure the productivity of projects written in different languages. However as most of the cost and effort estimation models are based on LOC, function points must be mapped into LOC. Also the system may have more than 14 characteristics that have not been considered. The function points are also not suitable for algorithmically intensive systems such as embedded systems and real time systems. In such type of systems function points called *feature points* are used for estimating the size.

Example: 2.1 Consider a project with following data:

Number of External inputs with low complexity = 10
Number of External inputs with high complexity = 10
Number of External outputs with average complexity = 15
Number of External inquiries with average complexity = 13
Number of internal logical files with high complexity = 2
Number of internal logical files with low complexity = 2
Number of External Interface files with average complexity = 7
The system has a very high transaction rate and supports several multiple communication protocols. Calculate the unadjusted as well as adjusted function points.

Solution:

Total unadjusted function points(UFP) after considering the weighting factors are given by:

$$UFP = 3 * 10 + 6 * 10 + 5 * 15 + 4 * 13 + 15 * 2 + 7 * 2 + 7 * 7$$
$$= 30 + 60 + 75 + 52 + 30 + 14 + 49$$
$$= 310$$

The software also supports high transaction rate and multiple communication protocols. Hence these two system characteristics can be assigned a rating of 5. Total degree of influence(TDI) is therefore given by:

$$TDI = 5 + 5 = 10$$

Value adjustment factor(VAF) $= (TDI * 0.01 + 0.65)$
$= 10 * 0.01 + 0.65 = 0.75$

Adjusted Function Points $= UFP * VAF$
$= 310 * 0.75 = 232.5$

2.8 EFFORT AND SCHEDULE ESTIMATION

Developing an error free product i.e. software within the customer's budget and schedule is the topmost priority of any organization. This in turn calls for reasonably good estimates of effort, time, cost and of course quality which effects the schedule and effort.

According to (Kelkar03) estimation is defined as process of reliably predicting the various parameters associated with making a project i.e., size, effort, cost, time, and quality.

Estimation of size of the software to be developed is very important as it forms the basis for effort and schedule estimation. A number of estimation methods to measure size in terms of number of lines of code, function points, object points etc., are proposed and are discussed in section 2.7.

In a software project effort can be defined as the amount of human effort or labor that will be required to perform a task. The units used to specify effort are person-months, person-days or person-hours. Normally one type of unit is used throughout the project.

In order to estimate the effort accurately, the project uses number of inputs like tasks to be performed, programming languages and tools to be used, experience of the team members, estimated size of the software and historical data on similar type of the projects.

Software effort estimation is important because of following reasons:

- Organizations can have proper control over project and they can plan systematically.
- There is a clear understanding of the product.
- Estimation also determines the project feasibility in terms of budget and time constraints.
- It helps in identification of resources to be used during the project.
- Estimation also helps management in taking decision for current as well as future projects.
- Estimation helps in quantifying the impact of risks and guides the project manager to take suitable decision.

In case of improper estimates, the organization may run into legal and financial problems. There are many reasons for poor estimates. Some of them are:

- Requirements not clearly understood.
- Requirements changing frequently.
- All other costs i.e., cost of hardware, software, traveling etc., not taken into consideration while doing cost estimation.

- Unavailability of reliable data from past projects to estimators.
- Lack of expertise.
- Lack of training and knowledge.

Broadly three approaches to effort and cost estimation are used. They are *algorithm approach, analogy based approach and bottom up approach*. The algorithmic approach uses the mathematical formulas to express the relationship between size, effort and schedule. Examples of models based on algorithmic approach are COCOMO model and Putnam estimation model. Analogy based approach uses data from the past projects of the similar kind. This technique is used when the organization in the past have completed the projects of the similar nature. For example, an organization for last few years has developed expertise in the banking domain which can be used in future projects. In this technique therefore comparisons are made with the past project's estimation data and estimates are proposed. Finally in the bottom up approach, project is decomposed into set of tasks and activities. Efforts are then calculated for lower level tasks and are added to give phase and project estimates. Effort thus estimated is used for scheduling the project and for estimating the cost of the project.

We have seen in section 2.7.1 that productivity is expressed as:

Productivity = LOC/Effort

It has been found that as the size of the software increases, complexity of the software also increases. As a result of this effort required to develop the software also increases in a non-linear fashion.

Schedule of the project is described in terms of number of months and can be found by dividing the effort by the team size.

Schedule(in months) = Effort(in person months)/Team size(in persons)

One must remember that men and months are not interchangeable. In order to build a high quality software, rigorous process must be followed and the software must be thoroughly tested. Hence in order to produce good quality software, extra effort and longer schedule are required. Next we will discuss some of the popular effort estimation techniques.

2.8.1 COCOMO: A Regression Model

A number of cost estimation models based on LOC are described in the literature. The technique has some inherent problems e.g., it is not possible to tell in the beginning the project the number of lines of code which in turn is influenced by number of factors e.g., language type, software development environment used, skills of the developer team, use of libraries and efficient reuse of code, number of comments used in the program and many more. In this book we will briefly explain ***Constructive Cost estimation Model (COCOMO)*** based on LOC.

COCOMO is one of the best documented cost-estimation model based on regression model. A regression model is derived by interpreting historical data statistically in order to describe typical relationship between variables. This model takes LOC as input. The model was proposed by Barry Boehm in 1970 and is based on study of 63 projects.

The formula for effort estimation for basic COCOMO model is given by

$$E = a(KLOC)^b$$

where E is effort in programmer months.

Since COCOMO can be applied to wide variety of projects, the values of a and b depend on type of projects used as classified by Boehm. Boehm has identified three categories of project. They are:

(i) Organic
(ii) Embedded
(iii) Semidetached.

The basic features of these three types of project are shown in Table 2.3.

Table 2.3 Important Characteristics of Different Types of Project

	Organic	*Embedded*	*Semi Detached*
Project size	Small	Any size	Large
Type of project	Independent	Product embedded in environment with severe constraints	Independent
Development Experience	Extensive	Moderate	Moderate
Environment knowledge	Extensive	Normal	Considerable
Constant values	$a = 2.4,\ b = 1.05$	$a = 3.6,\ b = 1.20$	$a = 3.0,\ b = 1.12$

Hence effort formulas for three types of projects can be written as:

Organic : $E = 2.4 * (KLOC)^{1.05}$

Embedded : $E = 3.6 * (KLOC)^{1.20}$

Semidetached : $E = 3.0 * (KLOC)^{1.120}$

In addition to basic COCOMO, Boehm also proposed two advanced models called **intermediate COCOMO** and **detailed COCOMO** in which he proposed to make use of 15 cost drivers. The only difference between the two models is that in detailed COCOMO, cost drivers are distributed over different phases of software development life cycle meaning there by that not all cost drivers effect all the phases of SDLC. The cost drivers are classified into four categories i.e., product attributes—concerning the product itself, personnel attributes concerning the people working on the project, computer attributes

concerning the hardware platform used and project attributes relating to project practices and tools.

The cost drivers proposed by Boehm are given in table 2.4. Each of these cost drivers for a given project is rated at a scale: very low, nominal, high, very high and extra high. The scale describes the proportion in which cost driver applies to a project. Values corresponding to each of the six ratings are also defined. The multiplying factors for all cost drivers specific to a project are multiplied to compute Effort Adjustment Factor (EAF). The equation for intermediate COCOMO therefore becomes

$$E = a(KLOC)^b * EAF$$

Table 2.4 List of Cost Drivers

Product Attributes	Software reliability
	Size of application database
	Complexity
Personnel Attributes	Analyst Capability
	Software Engineering Capability
	Applications Experience
	Virtual Machine Experience
	Programming Language Expertise
Computer Attributes	Performance Requirements
	Memory Constraints
	Volatility of Virtual Machine
	Environment
	Turn Around Time
Project Attributes	Use of Software Tools
	Application of Software Engineering Methods
	Required Development Schedule

Boehm also proposed standard development time (T) based on the project type as project size in months. For different types of projects, T is computed using formula given below:

1. **Organic** : $T = 2.5 * (E)^{0.38}$
2. **Semidetached** : $T = 2.5 * (E)^{0.35}$
3. **Embedded** : $T = 2.5 * (E)^{0.32}$

***Example 2.2** The values of size in KLOC and different cost drivers for a project are given below:*

Size = 200 KLOC

Cost driver:

Software reliability	*= 1.15*
Use of software tools	*= 0.91*
Product complexity	*= 0.85*
Execution time constraint	*= 1.00*

Calculate the effort for three types of projects viz. organic, semidetached and embedded, using COCOMO model.

***Solution.** Effort Adjustment Factor (EAF) is*

$$EAF = 1.15 * 0.91 * 0.85 * 1.00 = 0.8895$$

Effort for three types of projects is:

Organic Project

$$E = 3.2 * (200)^{1.05} * 0.8895 = 742 \text{ person months.}$$

Semi-Detached

$$E = 3.0 * (200)^{1.12} * 0.8895 = 1012 \text{ person months.}$$

Embedded

$$E = 2.8 * (200)^{1.2} * 0.8895 = 1437 \text{ person months.}$$

2.8.2 Delphi Method

Delphi approach for estimation is based on obtaining estimates about the project from several experts and generating reliable estimates from them. The whole process is carried out under the control of a coordinator. The steps followed in the approach are:

1. Coordinator explains the tasks and supplies the project specifications along with other relevant information to all the experts.
2. Experts respond by making estimates about the development effort.
3. Coordinator compiles the estimated values received from different experts and circulates the summary along with individual estimates to all the experts.
4. Each expert in response to this, submits his/her views, stating reasons as well.
5. Meetings are also called by the coordinator to discuss the estimates.
6. The process continues till an agreement is reached.

2.8.3 Putnam Estimation Model

In 1960's a study made by Peter V Norden of IBM concluded that projects show a well defined pattern. Later on Putnam verified Norden's finding in software projects also

and said that software development process follows Rayleigh shaped rate curve as shown in Fig. 2.3 and hence fits the mathematical formula for Rayleigh distribution.

Putnam further stated that the curve can be applied not only to the complete life cycle but also the individual phases i.e., coding, testing, assigning people to a project etc. Area under the curve can be used to compute number of valid lines of code, errors found and corrected etc. According to Putnam relationship between size, schedule and effort matched the Norden/Rayleigh function which is given by following mathematical function:

$$m(t) = 2Kat\,(\exp^{-at^2})$$

where $m(t)$ = number of persons at any time t (in years)

K = Total project effort in staff years

a = Acceleration factor (which determines sharpness of curve)

Acceleration factor is given by:

$$a = 1/2t_d^2$$

Here t_d = development time.

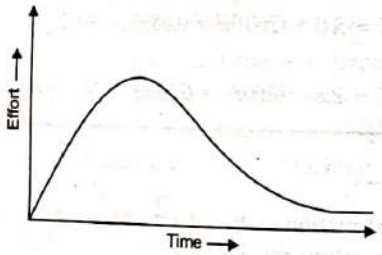


Fig. 2.3 Rayleigh Curve

Norden/Rayleigh function helped in understanding several other important characteristics/metrices of the projects. One such metric is Difficulty metric D which shows that when either manpower demand is very high or development time is short project is difficult to develop. This value is computed by differentiating equation (1) and calculating value at t = 0. Difficulty metric is expressed as:

$$D = K/t_d^2$$

Similarly peak manpower is given by:

$$m_0 = K/t_d\, e^{1/2}$$

From equation (2) and (3), Difficulty metric can be related to peak manpower as:

$$D = K/t_d^2 = \frac{m_0 e^{1/2}}{t_d}$$

Another metric of interest is manpower buildup which can be found from equation (4) by taking the derivative w.r.t. t_d and K respectively.

Manpower buildup D_0 can be expressed as:

$$D_0 = K/t_d^3 \text{ Person/year}^2 \quad \text{...(5)}$$

It was found that value of D_0 is almost fixed for different type of projects and can have values equal to 8, 15 or 27 depending upon whether the project is a new software with many interfaces, a new stand alone software or a software rebuilt from existing software. Depending upon the organization's expertise the value of D_0 can vary slightly from one company to another. The larger the value of D_0, steeper will be the manpower distribution or in other words, manpower buildup will be faster.

Putnam using data of various projects and results of Rayleigh shaped profile of software development, also derived a mathematical non-linear relationship between size of project, effort and time needed to complete it. It was called software equation. This equation makes use of a parameter which is measure of productivity and is expressed as:

$$\text{Productivity measure} = \frac{\text{Size}}{(\text{Effort}/B)^{1/3} * (\text{Time})^{4/3}} \quad \text{...(6)}$$

where effort is expressed in man years, time is in years, size is effective source lines of code and B is special skills factor related to size. The values of B vary with size and its values are tabulated for different system sizes (Putnam 97). Equation (6) can be rearranged to find effort as follows:

$$E = \frac{B * (\text{Size})^3}{(\text{Time})^4 * (\text{Productivity measure})^3} \quad \text{...(7)}$$

It is clear that effort required depends on size, time and productivity measures. The values of productivity measures are represented as Productivity Index (PI). Productivity Index can take values between 1 to 40 and corresponding productivity measure values are also tabulated. Low values of PI are used for highly complex projects or low productivity environments. Similarly high values of PI indicate high productivity and well understood projects. By taking necessary measures, organizations can improve their PI as it makes lots of difference in effort, cost and schedule of project.

2.9 COST ESTIMATION

From customer point of view, a customer is more interested in knowing the cost and time taken for completion of a project. Similarly a project manager must take the activity of cost estimation seriously if he/she wants to complete the project within the approved budget. Following costs must be considered while estimating the total cost.

Cost of effort/manpower: While computing the cost of project, one of the major component is the cost of effort applied by skilled software professionals in terms of person hours, person days or person-months which in turn also decides the team size. A number of techniques are discussed in section 2.8 to estimate effort. Cost of manpower therefore includes the salaries and other incentives given to the employees. Total cost must be calculated by identifying the roles of different persons working on the project and the man months spend by each one of them.

Equipment cost: This includes the cost of hardware, software, printers and other equipments required for doing the project. As these equipments can be used for developing other software also, therefore proportionally the cost of the equipment can be added to the total cost.

Traveling and communication cost: This includes the money spend on traveling, hotel stays, fax, telephone, internet etc., during the software development. The traveling can be done by the team member to visit the customer site, to attend a conference or to attend a course to upgrade his knowledge.

Administration and marketing cost: The company must have an administrative division to manage the overall working of the organization. Similarly every organization has a marketing division to promote the organization and to get the business. Hence some percentage of these costs must be added to the total cost of the project.

Subcontracting cost: In some cases a third part is hired by the company to develop a subset of the product requirements. Hence in such type of cases cost of outsourcing must be added to the total cost.

In order to avoid problems due to wrong estimation proper training in estimation techniques is must and also organizations must follow processes based on past experiences to do estimation.

2.10 SCHEDULING A SOFTWARE PROJECT

Some software projects do not mature because of scope, time and cost constraints. Whenever the customer approaches a software development agency he only wants to ensure that the project requirements are well understood and everything is within budget and delivered on time. This in turn requires a well defined project schedule with minimum conflicts. Conflicts can arise due to individual attitude and work style, changes made to the project and also due to the cultural differences. One of the most important job of the project manager is therefore to ensure the timely completion of the project which is not an easy job. In order to do so he also uses software to analyze schedule related information. One such popular project management software is **MS-PROJECT** which can be used to draw Network diagrams, Gantt charts, to find the critical path and to generate reports. The main steps for scheduling the project are:

1. Identify all the activities of a project and the milestones to be produced
2. Estimate the time span of each of these activities

3. Identify and establish dependencies between these activities.
4. Represent the dependencies among the activities using a dependency diagram say activity graph.
5. Estimate the resources a project team should use to complete the project.
6. Identify the critical path(s).
7. Identify the slack times of non-critical paths.
8. Control and manage the changes to the project schedule.

Next we discuss some of the popular techniques used for scheduling project activities.

2.10.1 Work Breakdown Structure (WBS)

Work breakdown structure is used to represent the total scope of the project in terms of the phases. Each of these phase can be divided into tasks which can be further divided into number of activities as shown in Fig. 2.4. In short WBS describes the project as discrete pieces of work. This document is very important as it forms the basis for managing resources, planning and changing project schedules.

As shown in the Fig. 2.4, WBS is a tree structure with no loops or cycles. Also each of the activity described in WBS takes some fixed time for completion and results in some output or milestone achieved to ensure its completion e.g., in case of requirement analysis milestone achieved is a deliverable requirement specification document.

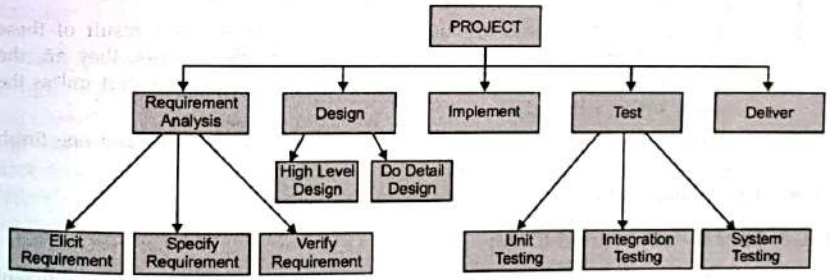


Fig. 2.4 A Work Breakdown Structure

Work breakdown structure does not give any information about ordering of activities and their interdependence. Also it is not clear from WBS which activities can take place concurrently. Hence to overcome these problems, dependency diagram is used.

2.10.2 Dependency Diagram: Network Diagram

Network diagram is a type of dependency diagram used to show the interdependence of different activities of a project and relates to sequencing of project activities. It is also called ***activity graph.*** The diagram consists of three elements:

- *Event or milestone*—It is a significant occurrence in the life of a project.
- *Activity*—It is the work required to move from one event to another event. In other words milestones are produced as a result of these activities.
- *Span time*—It is the actual calendar time required to complete an activity also called the duration of the activity.

A sample network diagram is shown in Fig. 2.5. In this graph milestones are represented by nodes of the graph and activities are represented by the links connecting these nodes.

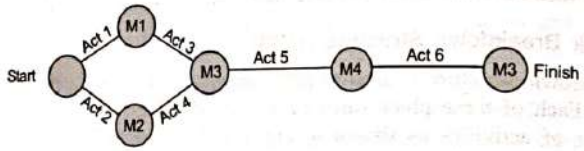


Fig. 2.5 An Activity Graph/Network Diagram

In Fig. 2.5, Act 1, Act 2 ... are activities and M1, M2 ... are mile-stones. The activity graph or the network diagram becomes all the more meaningful if time taken to complete the activity is also shown on the lines connecting the nodes. The main rules for constructing network diagram are therefore

1. Activities are represented by links and take time to finish.
2. As described earlier nodes are the milestones produced as a result of these activities and are instantaneous points in time. In other words, they are the events of the project. It means that in Fig. 2.5, activity 3 can not start unless the milestone M1 is achieved.
3. An activity graph representing a project can have only one start and one finish node.
4. Graph should also not contain any loops.

2.10.3 Critical Path Method (CPM)

Critical Path Method shows the analysis of paths in an activity graph among the different milestones of project and is used to predict the total duration of project. It highlights those activities which are critical for successful completion of project on time. In other words this method is used to find the ***critical path that describes the sequence of activities that take the longest time to complete***. The length of the critical path(s) define the time a project will take to complete. Though the critical path represents the longest path, it represents the shortest time required for the completion of a project. Activities lying on the critical path are called **critical activities** because delay in finishing these activities will result in overall delay of the project. A **Non-critical path** on the other hand consists of sequence of those activities that one can delay and can still finish the project in the shortest time possible.

2.10.3.1 Calculating the Critical Path

Following steps are followed to calculate the critical path.

1. Draw an activity graph or a network diagram.
2. Estimate the duration of each activity
3. Identify all the paths in the network diagram.
4. For each path add the durations of all activities lying on that path.
5. The path with the longest duration is the critical path.

This can be explained with the help of network diagram shown in Fig.2.7. The diagram has 6 paths. They are:

Path1 : B-C-D-E-F-G	*Total duration = 15+11+10+10+5+7 = 58 days*
Path2 : B-C-D-H-I-J	*Total duration = 15+11+10+13+10+4 = 63 days*
Path3 : B-C-D-H-I-K-L	*Total duration = 15+11+10+13+10+5+5= 69 days*
Path4 : A-C-D-E-F-G	*Total duration = 4+11+10+10+5+7 = 47 days*
Path5 : A-C-D-H-I-J	*Total duration = 4+11+10+13+10+4 = 52 days*
Path6 : A-C-D-H-I-K-L	*Total duration = 15+11+10+13+10+5+5= 58 days*

Since the path ***B-C-D-H-I-K-L*** has the longest duration of 69 days it is the critical path of the project. The critical path is shown in the diagram using thick lines. It is quite possible that a project can have more than one critical path and during the project execution it can even change.

2.10.3.2 Making the Schedule Trade-offs

In order to make the schedule trade offs, we need to define real time, available time and slack time for an activity. **Real time or actual time** is the estimated time an activity will take to complete. **Available time** is the time available to complete the activity and **slack time** is defined as the difference between available time and real time. It can be defined as the maximum amount of time by which you can delay an activity and still finish your project in shortest time possible. Another way to express slack time is

Slack time = Latest Start time (LST) – Earliest Start Time (EST)

Where earliest start time is the earliest time an activity can begin and latest start time is the latest time an activity can begin without delaying the complete project. This can be illustrated with the help of an example. Consider the part of activity graph shown in Fig. 2.6.

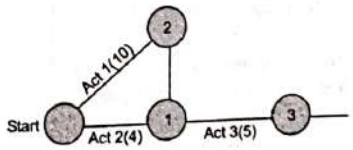


Fig. 2.6

In this graph activity 1 takes 10 days to complete and activity 2 takes 4 days to complete and also they can start immediately. Hence LST and EST are equal to 0 for activity 1. Also activities 1 and 2 are done concurrently and activity 3 can not start until activities 1 and 2 are completed successfully. Hence EST is 0 and LST is 6 for activity 2 because if started at the most on 6th day it will finish within 4 days and activity 3 can be started on 10th day which is the earliest start time (EST) for activity 3. Therefore slack time for activity 2 is

Slack time = LST – EST = 6 – 0 = 6 days

In this manner slack time for all the activities of the activity graph can be computed starting from the beginning. Activities with slack time zero constitute the critical path. EST and slack time can be computed by moving forward starting from beginning and LST can be computed by moving backwards from the finish node. Activities lying on the critical path are called critical activities and special attention must be given to these activities to avoid delays. Similarly in order to reduce the overall duration of the project duration of critical path must be reduced or in other words duration of critical activities must be shortened.

Once the critical path and slack time of activities are found, this information can be used for schedule trade-offs or to shorten a project schedule. Two popular techniques used for shortening the project schedule are **Crashing** and **Fast tracking.** Crashing is a technique used for shortening the project time by focusing on those activities lying on the critical path that can be done in shorter duration at no extra cost. Fast tracking on the other hand focuses on doing the activities in parallel.

Example 2.3 For the activity graph shown in Fig. 2.7. Compute Earliest Start Time (EST), Latest Start Time (LST), slack time and identify the critical path.

Solution

Step 1. Calculating the Earliest Start Time (EST) for each activity

Calculating the Earliest Start Time for each activity is done by moving forward in the activity graph starting from the start node. It is done by computing the date on which each milestone is achieved.

- Activities A and B can start immediately. Therefore EST for activities A and B is day 0 and earliest date for event 0 is also 0.

Contd...

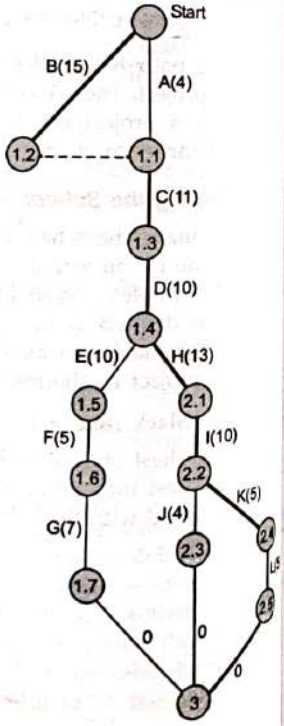


Fig. 2.7

- Time taken to finish activities A and B is 4 and 15 days respectively and activity C cannot start unless both A and B are completed. Hence earliest start time for activity C is 15th day (considering later of 4 and 15).
- Activity C will take 11 days to finish, hence EST for D will be 26th day and it will take 10 days to finish.
- After activity D is finished and milestone 1.4 is achieved, two parallel paths of activities will start and both the activities E and H will start earliest on 36th day.
- Continuing in the same fashion EST for activities F and G are 46 and 51 respectively. Similarly EST for activities I, J, K and L are 49, 59, 59 and 64 respectively.
- Earliest Finish Time for Activities G, J and L will be 58th, 63rd and 69th day respectively. Unless all these activities are completed successfully project is not finished. Hence the earliest date for completion of project is day 69.

Step 2. Calculating the Latest Start Time (LST) for each activity

Latest start time of an activity is the latest time by which an activity can start without introducing delay in the project and is calculated by moving backward from the finish node. While doing this we assume that latest finish date is same as earliest finish date for a project which is 69th day in this example.

- Latest finish time for achieving milestone 3 is 69th day. Activity L takes 5 days to finish. Hence latest start time for activity L is 64th day (69 – 5), for activity G is 62nd day (69 – 7) and for activity J is 65th day (69 – 4).
- LST for activity K is day 59.
- Latest start time for activity I is the date by which both the activities J and K can be started simultaneously. LST for activity J is 65th day and for K is 59th day. Activity I takes 10 days to finish. Hence considering shorter of two, LST for activity I is 49th day (59 – 10).
- The complete table to show latest start time and earliest start time is shown in Table 2.5.

Step 3. Finding the Critical Path

Critical Path is identified by finding the activities with slack time 0. Slack time for each activity is computed using the formula

Slack time = LST – EST

We see from table 2.5 that activities B, C, D, H, I, K and L are the activities with slack 0 and hence these activities constitute the critical path.

Table 2.5

Activities	Duration (days)	Earliest Start Time (EST)	Earliest Finish Time	Latest Start Time (LST)	Slack Time (LST – EST)
A	4	0	4	11	11
B	15	0	15	0	0
C	11	15	26	15	0
D	10	26	36	26	0
E	10	36	46	47	11
F	5	46	51	57	11
G	7	51	58	62	11
H	13	36	49	36	0
I	10	49	59	49	0
J	4	59	63	65	6
K	5	59	64	59	0
L	5	64	69	64	0

A sample network diagram is shown in Fig. 2.8. The path BCDHIKL shows the critical path.

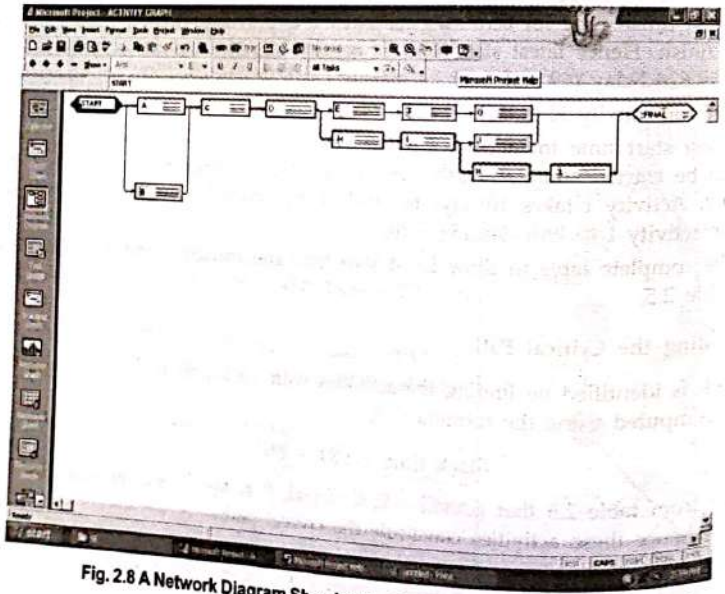


Fig. 2.8 A Network Diagram Showing the Critical Path Using MS-PROJECT

2.10.4 Gantt Chart

The Gantt chart is used to represent project plans graphically where each task is represented by a horizontal bar. The length of the bar is proportional to the completion time of the activity and also specifies the start and the end date. Different types of activities can be represented through different colors, shapes or shades. The chart also contains milestones and dependencies among different tasks. Gantt charts are generally used for simple projects or describing project plans for some part of large projects. A special type of Gantt chart called **Tracking Gantt Chart** is used to compare the planned and the actual project schedule information. A sample Gantt chart using MS-PROJECT is shown in Fig. 2.9.

2.10.5 Project Evaluation Review Technique (PERT)

A PERT chart is also used to show different project task activities and their relationship with each other. An important feature of PERT is that this chart also takes into account the uncertainty associated with activity duration estimates and is useful in high risk projects.

Though the practitioners use the term CPM and PERT interchangeably, the PERT is different from CPM in estimating the duration of each activity. PERT while estimating the duration uses three values as described below

1. Time taken by an activity to complete under normal circumstances say n.
2. Shortest time to complete the activity say s.
3. Maximum time taken by activity to finish considering all constraints say m.

Expected time required to finish the activity is computed using the formula

$$t_p = \frac{m + 4n + s}{6}$$

This technique does not indicate the earliest finish date of the project but only the expected date by taking into account the uncertainty or risks of the real world. All the calculations are now done using the estimated duration t_p.

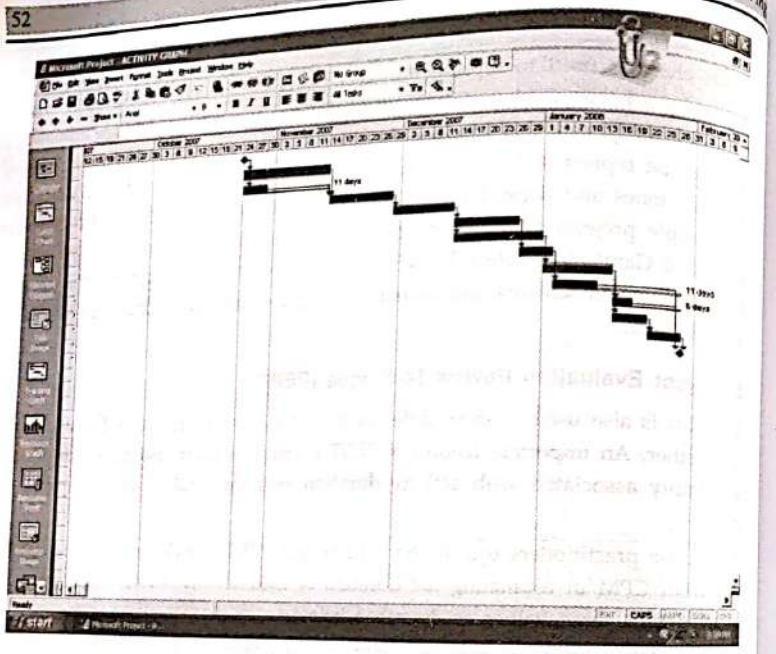


Fig. 2.9 A Sample Gantt Chart

2.11 RISK MANAGEMENT

Risk management can be defined as identifying and understanding the risks that may cause project delay or even failure in some cases. It also involves planning so as to minimize their effects on the project performance.

According to (Robert02) risk management is about understanding the internal and external project influences that can cause project failure.

Some of the issues that risk management attempts to answer are (Charette89):

- Identifying the risks confronting the current situation.
- Probability of occurring losses as a result of these risks.
- Cost of losses due to risks.
- Losses in case of worst happenings.
- Reducing or resolving the risks and in turn the potential losses.
- Risks resulting as a result of alternatives used.

A number of definitions of risk also exist in the literature. Some of them are given below.

Risk is uncertainty or lack of complete knowledge of the set of all possible future events (Robert02).

Risks are factors or aspects which are likely to have a negative impact on project's performance (Kelkar03).

Risk is a probability that some adverse circumstances will actually occur (Sommerville02).

In simple language risks are those unknown events which if occur can even result into project failure. Some of the possible risks are:

- Experienced staff members leaving the company before the completion of project.
- Change in technology.
- Change in product requirement.
- Change in government policy.
- Competition from other organizations.
- Financial problems.
- Underestimating cost and effort.

After completing the project plan, risk analysis should be carried out and its results along with the consequences of risks (if they occur) must be documented in the project plan.

A number of proposals have been given by the researchers to categorize risks. Broadly risks can be placed in three categories. They are:

- Project risks : affecting project schedule or resources and occur mainly due to resource constraints, fund problems, problems with vendors etc.
- Product risks : affecting software quality, occur mainly due to incomplete SRS, wrong design specifications etc.
- Process risks : risks related to managerial and technical procedures followed to build the product.

Risks can also be classified as internal or external risks (Robert02). Internal risks are the risks which are known and are under the control of project manager. External risks on the other hand are unknown risks and are not under control of project manager. Risk management is identifying and controlling these risks.

2.11.1 Risk Management Process

The process of risk management consists of following stages:

1. Risk identification
2. Risk analysis and quantification

3. Risk planning
4. Risk monitoring and control

Next we discuss each of these stages briefly.

2.11.1.1 Risk Identification

Risk identification is the most crucial stage of risk management process. It focuses on identification of different types of risks related to technology, tools, people, estimation, politics, market etc., and documenting the characteristics of each. Risk identification in software engineering, broadly identifies three types of risks (Charette97). They are process risks, product risks and organizational risks. Process risks are concerned with the processes used along with automation support during software development. Product risks concern the actual product related risks i.e., architecture, design, implementation, hardware, reliability, human factors etc. Organizational risks focus on the risks concerning the organization who is developing the software i.e. its infrastructure, expertise etc. Several techniques like brainstorming, interviewing and Delphi technique can be used for identification of risks. Risk identification is also done by using techniques like checklist, problem decomposition and decision-driver analysis. The risks identified and their related information is maintained in a document called ***risk register***. Information like risk number, name, description, frequency of the occurrence, impact to the project, type of risk etc. can be maintained in the risk register.

2.11.1.2 Risk Analysis and Quantification

During risk analysis stage, probability of occurrence of risks and their impact on the project is studied. This is done for all the risks identified in the risk management stage. A number of techniques can be used at this stage. Some of the techniques are Brainstorming, Delphi method, Sensitivity analysis, Probability analysis, Monte Carlo simulation and Decision tree analysis. In most of the cases past experience and judgement of project manager plays a very important role. The probability of occurrence of risks can be assessed at a scale of 1 (low) to 10 (very high) whereas effects can be assessed as insignificant, serious, moderate, catastrophic. Therefore a numeric value is assigned to each risk in order to compare it with other risks. Quantification also involves calculating the risk exposure (RE) factor which is computed using formula:

$$RE = P * L$$

where RE = Risk Exposure
P = Risk Probability
L = Loss

All the risks must be prioritized in order to focus on critical risks which affect the project performance most. For each high priority risk, RE is computed. The results of this stage must be properly tabulated.

2.11.1.3 Risk Planning

The risk planning stage of risk management process is concerned with identifying strategies for managing risk. This can be achieved through risk avoidance, risk acceptance, risk transfer, risk mitigation and developing contingency plans (in case worst happens). The technique of risk avoidance focuses on restructuring of project so as to avoid that risk. Risk transfer usually solves the problem of risk impact by shifting the consequence of risk to third party e.g., buying insurance. Risk mitigation is reducing the impact of the risk by reducing its probability of occurrence. The risk planning is also done for all the risks identified in the risk identification stage.

2.11.1.4 Risk Monitoring

The risk monitoring activity of risk management is a continuous process which involves regular assessment of identified risks in terms of their probability of occurrence and their impact on the project. Techniques like top-ten risk tracking, milestone tracking and corrective actions can be used here to monitor the risks.

2.11.2 Risk Decision Tree

Risk decision tree is produced by using decision analysis techniques. These techniques map all possible alternatives in the form of a decision tree. Initial decision is modeled as the root node and from there branches representing number of choices or alternatives emerge. Each branch has a probability of occurrence associated with it. For a node all branches probabilities sum must be equal to one. Each branch also has consequence associated with it called leaf. As a result risk exposure for each of the choice or branch can be computed. Total risk exposure of a choice is sum of risk exposures for all leafs under that choice. This value is in turn used by project manager to take appropriate decision. A sample risk decision tree is shown in Fig. 2.10. We see from the decision tree in Fig. 2.10 that for alternative 1 total gain is Rs. 1000 where as for alternative 2 total gain is only Rs. 650.

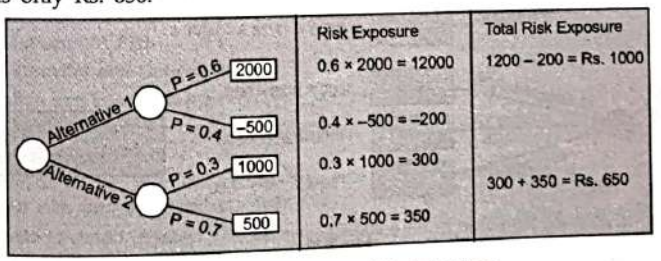


Fig. 2.10 A Sample Risk Decision Tree

Example 2.4 In a casino, there are two options to play a game. In option A if you roll two dices and get multiple of 5, you win Rs. 10800. If you get multiple of 3, you win Rs. 7200 and in other cases you have to pay Rs. 7200 to the casino. In second option if you

get multiple of 4 you win Rs. 3600. If you get 2 or 12, you win Rs. 14400. In other cases you pay Rs. 720 to the casino. Which game you should play?

Solution. We will first compute the total risk exposure for option A and option B respectively as given below.

For Option A

Risk (e)	Risk Probability (RP)	Risk Impact (RI)	Risk Exposure (RE = RP * RI)
1. Multiple of 5	7/36	Rs. 10800	Rs. 2100
2. Multiple of 3	12/36	Rs. 7200	Rs. 2400
3. Others	17/36	– Rs. 7200	– Rs. 3400

Total Risk exposure = Rs. (2100 + 2400 – 3400) = Rs. 1100

Option B

Risk (e)	Risk Probability (RP)	Risk Impact (RI)	Risk Exposure (RE = RP * RI)
1. Multiple of 4	9/36	Rs. 3600	Rs. 900
2. 2 or 12	2/36	Rs. 14400	Rs. 800
3. Others	25/36	– Rs. 720	– Rs. 500

Total Risk exposure = Rs. (900 + 800 – 500) = Rs. 1200

As average gain is more in case of option B, one should play option B. Same can be represented using a risk decision tree as shown in Fig. 2.11.

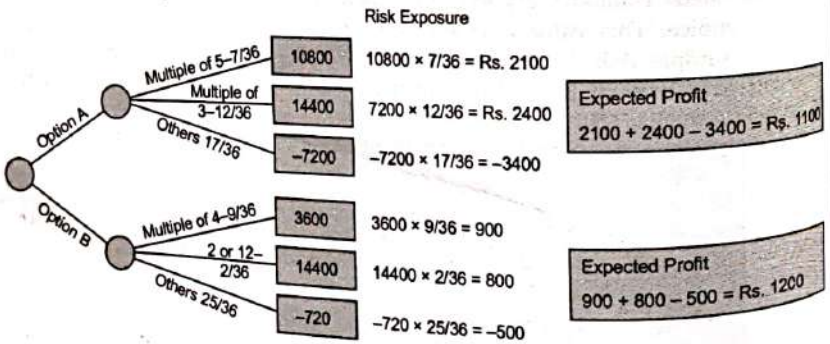


Fig. 2.11

2.12 PROJECT MANAGEMENT PLAN

A *Project Management Plan* is a document used to integrate and coordinate the

information from all the project management knowledge areas in order to guide a project's execution and control. Depending upon the size of the project, the details are given in the project management plan. A sample template for the software project management plan is given below:

Name of the Project

1. **Overview of the Project**
   a. Document Version Number
   b. Date
   c. Document Author(s):
   d. Project Sponsor
   e. Scope of the Project: provides the overview and scope of the project along with the constraints.
   f. Project Deliverables: lists and describes the parts that will be produced as a part of the project.
2. **Team Structure:** This section describes the name of the project manager and other team members according to their roles in the project and their specific responsibilities. The roles played by different team members are technical writer, requirements engineer, programmer, tester, chief architect, quality assurance expert etc.
3. **Estimation and Resource Details:** This section describes the effort, cost, resource and duration estimates along with the effort estimation techniques used.
4. **Project Schedule and Staffing Details:** Chart like Gantt charts, PERT etc., can be used here to represent the project schedules.
5. **JAD Session Results:** This part documents the JAD session results in terms of critical success factors, high level prioritized goals, benefits of the product to the sponsor, risks involved etc.
6. **Configuration Management Plan:** This section lists the names of the entities that will be managed via this configuration management plan, such as Requirements Document, privacy policy, design document, prototype, security policy etc. This section also describes policies regarding the management of change and versioning within the software system.
7. **Software Quality Assurance Plan:** This section discusses the project reviews conducted by the SQA group and other software team members.
8. **Risk Management Plan**
9. **Project Tracking and Control Plan**
10. **Verification and Validation Plan**
11. **Technical Process Plan:** documents information about process models, tools and techniques used in the project.
12. **Glossary of Terms:** lists any terms, particularly domain-specific terms, that are used in the document with an understandable definition.
13. **Document Revision History:** This section includes a list of significant changes that have been made to this document after 1.0 version has been submitted for assessment.

2.13 PROJECT MANAGEMENT TOOLS

Project management tools are the automated tools used for increasing the efficiency and

productivity of project manager and other team members during project manage... activities. Use of these tools helps the manager to track the progress made during ... project, explore various approaches for estimation and to make project plans with minim... risks.

These tools help the project manager in developing work breakdown struct... computing critical path, PERT etc., and representing these using multiple views. T... also provide project visibility and better communication between team members. So... tools also have built-in data of different types of past projects and help the proj... manager in computing cost, effort and size estimates accurately and in less period... time. Several types of reports for a project can also be generated using these tools i... short period of time. Some popular commercial tools supporting project manage... activities are listed below:

1. MS-Project : This tool developed by Microsoft helps in tracking ... project progress and project scheduling activities.
2. Estimate Professional : This tool is developed by Software Productivity Ce... (SPC), Canada and is used for Project estimation (ba... on COCOMO and Putnam) and planning.
3. SLIM : This tool is suite of project estimation and control to... and has built-in data base of 5000 projects. It ... developed by Quantitative Software Management.
4. Function Point Manager : Tool for Function Point Analysis, developed by A... Corporation.
5. COSTAR : An estimation tool based on the Constructive C... Model, developed by Softstar. The tool supports m... of the versions of COCOMO model.

SUMMARY

- Project is defined as an enterprise which is planned carefully to achieve ... particular goal.
- Project management is management of procedures, tools, technology, techniq... etc., required for managing the project successfully.
- Project management consists of four activities: feasibility study, project planning... project execution and project termination.
- Successful projects are the projects which meet their objectives within budg... and schedule.
- Project manager is responsible for successful completion of the project.
- For project scheduling, techniques like Work Breakdown Structure (WBS), CP... PERT, Gantt chart etc., are used.
- Critical activities if delayed can delay the overall project.

- A number of techniques like COCOMO, FPA, Putnam Estimation Model etc., are used for effort estimation.
- Risk management is one of the important activity of project management process and involves identifying and resolving the project risks.
- A number of project management tools are available in the market which improve the productivity and efficiency of project manager and other team members.

REVIEW PROBLEMS

1. List at least seven deliverables in a software project.
2. Define a project. List the attributes of a project.
3. What do you mean by project management? Why it is important?
4. What are the important project management skills?
5. What is the role of the project manager in an organization?
6. Define work breakdown structure. Draw a work breakdown structure for constructing a house.
7. What is the difference between a process model and a work breakdown structure.
8. Identify the critical activities and critical path for the problem given in Fig. 2.12 . Also calculate the slack time for the non-critical activities.

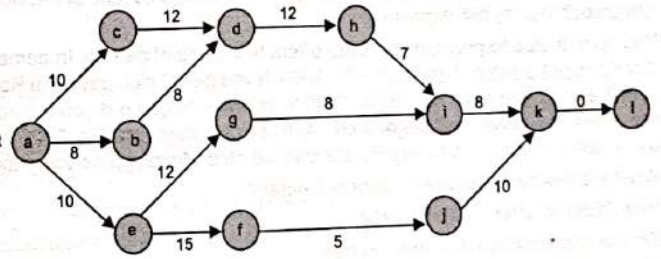


Fig. 2.12

9. What are the advantages and disadvantages of estimating cost? Explain briefly the main cost components that must be considered while estimating the cost of the project.
10. Explain briefly different techniques of software cost estimation as proposed by Boehm.
11. The values of size of program in KLOC and different cost drivers are given below:
    Size — 300 KLOC, Complexity — 0.95, Analyst Capability — 1.05,
    Application of Software Engineering Method — 0.8, Performance Requirement—0.75.
    Calculate the effort for three types of projects i.e., organic, semi detected and embedded using COCOMO model.
12. List the steps used for computing function points using function point analysis technique.

13. Consider the system shown below:

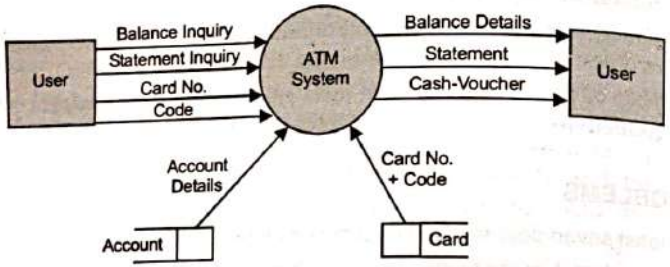


Fig. 2.13

For this system calculate the unadjusted function points. Assume the average complex for each type of item.

14. Identify at least ten risks that can result into project failure or delay.
15. Describe briefly different stages of risk management process.
16. What is the importance of risk management?
17. Define risk exposure.
18. Rolling the three dices and getting multiple of 3 results into loss of Rs. 5000. Calculate (a) risk probability, (b) risk exposure.
19. You go to a club to play games. Club offers two types of games. In game A if you roll the dices and get 8, the club pays you Rs. 1000. If you get 12 club pays you Rs. 5000. If you get 3, 5, 7,9 or 11 you have to pay Rs. 1000 to the club. In game B you are again asked to roll two dices. If you get multiple of 4 i.e., 4, 8, 12 club pays you Rs. 5000 but if you get any other number, you have to pay Rs. 2000 to the club. Which game you should play?
20. What are the main reasons for poor estimates?
21. How PERT is different from CPM?
22. Write a short note on risk decision tree.
23. Describe Putnam estimation model.
24. Under what situations while working on a project you will prefer PERT over a Gantt chart
25. While working on a Embedded type of project using COCOMO model, size of product estimated is 200000 LOC. Calculate the effort and the development time.
26. What are the main items of the project management plan? Explain briefly.
27. List the advantages and disadvantages of using LOC as size metric.
28. MarkII Function Point Analysis is another popular method used for size estimation. List the salient features of this method after studying the literature.
29. What is the importance of slack time?
30. State True or False and justify your answer.

(a) LOC gives the physical size of the program.
(b) Cost of a project consists of cost of manpower only.
(c) PERT is used when high amount of uncertainty is associated with the project.
(d) Schedule estimation is done before estimating the size of the project.
(e) Adjusted function points are computed by multiplying the unadjusted function points with total degree of influence.
(f) Slack time is the time available to complete the project.
(g) Risk mitigation usually solves the problem of risk impact by shifting the consequence of risk to third party.

31. The activities along with the time required to finish the activities in a project are as follows:

Activity Number	*Duration(in weeks)*	*Immediate Predecessor*
a	10	—
b	5	a
c	8	b
d	10	a
e	10	c,d
f	6	a
g	9	f
h	13	e,g

Draw the network diagram and Gantt chart representation of project.

32. Identity the type of risk
(a) Incremental model is selected instead of prototype model.
(b) Estimation of cost is wrong by 25%.
(c) User Identity are not as per customer satisfaction.
(d) Project manager leaving the organization in the middle of project.
(e) Subcontractor not completing the work on time.

○○○

Chapter 3

REQUIREMENT ENGINEERING

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- What is a Requirement?
- Different Types of Requirements
- Different phases of Requirements Engineering Process i.e., Requirements Elicitation, Requirements Specification, Requirements Validation and Requirements Management
- Requirement Elicitation Techniques
- Challenges in Eliciting Requirements
- Software Requirement Specification(SRS) and its Characteristics
- Organization of SRS
- Role of a System Analyst

3.1 INTRODUCTION

The two critical dimensions of software development are 'What to build' and 'How to build'. The hardest part of building a software system is deciding and defining precisely what to build (Aggarwal05) and more importantly specifying and documenting it for further reference. Given the fact that requirement errors are numerous as well as the time consuming and costly, not much attention is paid to the requirements engineering activities. According to a study done by Boehm (Boehm81) it was found that 10% of the

errors were due to incorrect requirements. However the more recent surveys sugg
that between 44%[UKH95] and 80%(TTN97) of all defects were introduced in
requirements phase.

It was also found that cost of the errors that are introduced during the requirem
phase and fixed later in the software development life cycle increases exponen
According to (Boehm84) it costs 5 to 10 times more to correct errors during coding
during the requirements phase and it costs 100 to 200 times during the maintena
phase. (CON90) and (DOD91) also reported the similar results thereby showing
there is tremendous potential to save cost and time by improving requirement prac
Understanding requirements of any information system is most crucial to that pr
and can be really rewarding for the development of the information system.

The basic goal of requirements phase in the SDLC is to produce the Requirem
Specification Document (RSD) or **Software Requirement Specification** (SRS) wh
describes the complete external behavior of the proposed software system(Davis93).
process of developing this document is called *Requirements Engineering*. A numbe
definitions of requirements engineering can be found in literature as given below.

*According to (Loucopoulos95) requirements engineering can be defined as – "t
systematic process of documenting requirements through an interactive co-operati
process of analyzing the problem, documenting the resulting observations in
variety of representation format and checking the accuracy of the understandi
gained."*

*According to (Davis93) requirements engineering is "the systematic use of proce
principles, techniques, tools and languages for the cost effective analysis
documentation and ongoing evolution of user needs and the specification of th
external behavior of the system in order to satisfy these needs."*

Inputs to this activity of requirements engineering are requirements which are inform
and fuzzy and output is clear, well defined, complete and consistent requirements writ
in formal notation called RSD or SRS as shown in Fig. 3.1.

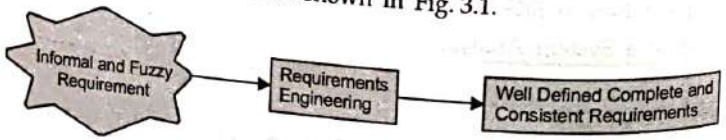


Fig. 3.1 Requirement Engineering Process

The requirements engineering is the most crucial and most important activity in
software development life cycle because the errors introduced at this stage are the m
expensive errors requiring lot of rework if detected late in the software life cycle. Al

the requirement specification document produced as an output of this activity is used by successive stages of life cycle i.e. for generating design document, for coding, for generating test cases in software testing and also plays a lead role in acceptance testing. It also forms the basis of a contract between the developing organization and the end user or customer.

It is to be noted that wide variety of methodologies have been proposed by researchers for expressing requirements specification e.g. natural language descriptions, structured analysis techniques, object oriented requirements analysis techniques, Petri net etc. Additionally number of standards and practices like IEEE- Std 830, DOD-STD-2167A etc. are also proposed to document software requirements.

3.2 WHAT IS A SOFTWARE REQUIREMENT?

In simple language, a requirement can be defined as a feature which end user needs in the existing or the new system.

According to IEEE Standard 729 [IEE83] a requirement is defined as
(i) a condition or capability needed by a user to solve a problem or to achieve an objective
(ii) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document;
(iii) a documented representation of a condition or capability as in (i) and (ii).

Broadly requirements can be classified as

- Functional requirements
- Non-functional requirements
- Domain requirements

In the next sections we will briefly discuss each of these type of requirements.

3.2.1 Functional Requirements

As the name implies, functional requirements focus on the functionality of the software components that build the system. As these components would be doing some kind of transformations on the input, the high level functional requirements are expressed in terms of

- inputs
- outputs
- processing input data in some way

In simple words functional requirements are the services which the end users expect

the final product to provide. For example in a hospital management system a doctor should be able to retrieve the information about his patients. Each high level function requirement may involve several interactions or dialogue between the system and outside world. Depending upon the user input each high level functional requirement takes different course of action leading to different scenarios. These scenarios can be called *sub requirements* of the high level functional requirement. In order to accurately describe the functional requirements all possible scenarios must be enumerated.

There are many ways of expressing functional requirements e.g., natural language, structured or formatted language with no rigorous syntax and formal specification language with proper syntax. In addition number of Function oriented methods (Stevens74), (DeMarco78), (Myers78), (Gane78), (Ross85), (Yourdan89) and Object Oriented Requirement analysis methods like Jacobson method(Jacobson92), Rumbaugh method(Rumbaugh91), Booch method(Booch94), Coad and Yourdon Method(Coad91) and UML(UML03) are also proposed to document the functional requirements of an information system.

3.2.1.1 Documenting Functional Requirements

In order to document the functional requirements follow the following steps:

(a) Identify the set of services supported by the system.

(b) Specify each of the services or functionality by identifying the input data, output data and processing done on input in order to get the output.

(c) Identify all the sub requirements for a high level requirement corresponding to the different user interactions.

A sample requirement "Retrieve Patient Record" in a hospital management system will be documented as illustrated in example 3.1

3.2.2 Non-functional Requirements

Non-functional Requirements (NFRs) are the constraints imposed on the system and deal with issues like maintainability, security, performance, reliability, portability, scalability and many more. The other terms used to specify NFRs are:

- Quality attributes (Boehm78)
- Constraints (Roman85)
- Goals (Mostow85)
- Non-behavioral requirements (Davis93) etc.

A number of researchers have proposed techniques for classifying the NFRs. According to (Roman85), NFRs can be classified as:

- Interface constraints
- Performance constraints: response time, security, reliability, storage space etc.
- Operating constraints

Example 3.1 Retrieve patient record

R1: Retrieve Patient Record

Description: *Once the doctor selects the retrieve patient record function, he would be asked to enter the patient code. The system would search the patient in the database and if found would display the past history of the patient. The details to be displayed include : Patient name, address, contact number, date of birth, brief description of the diagnosis, details of past visit dates and the treatment prescribed on those visits. It also displays the results of the tests done of the patient on different dates*

R1.1 : Select retrieve patient record option

State: *User has logged in using valid user/password and the system has displayed the main menu.*

Input: *"Retrieve Patient Record" option*

Output: *Prompt user to enter the patient code*

R1.2 : Select period

Input: *Patient code*

Output: *Prompt user to enter the start and end date*

R1.3 : Search and display

Input: *start and end date*

Output: ***(i)*** *Details of the patient whose patient code matches the patient code entered by the user is displayed. The details to be displayed include : Patient name, address, contact number, date of birth, brief description of the diagnosis, details of past visit dates starting from the start date and the treatment prescribed on these dates. It also displays the results of the tests done of the patient on different dates starting from the start date.*

(ii) *In case no match is found for the patient code entered by the user an error message "Invalid Patient Code" is displayed on the screen.*

Processing: *Search the Patient details based on patient code.*

- Life cycle constraints: maintainability, reusability, portability, flexibility etc.
- Economic constraints

Yet another scheme for classification is proposed by (Sommerville92) where NFRs are divided into three groups- process requirements, product requirements and external requirements.

Non-functional requirements play a very important role in deciding the fate of the software project. If not dealt properly they can result into:

- Inconsistent software
- Cost overruns
- Unsatisfied clients/users
- Unsatisfied developers

Further, interpretation of NFRs for different system/projects by different people can be different. Hence they are subjective and can be relative. They are normally difficult to test, and hence evaluated subjectively (Thayer90), (Chung00). It is to be noted that there is no formal list of NFRs and universal classification scheme supporting all type of application domains. A number of standards and frameworks are also proposed to represent NFRs. A list of NFRs consisting of 161 NFRs is given in (Chung00). NFRs are still an open area of research.

Not many methods are proposed in the literature to capture and formally represent non-functional requirements of the information system. The process of specifying non-functional requirements requires the knowledge of the functionality of the system as well as the knowledge about the context within which the system will operate.

Each of the functional and non-functional type of requirement can be a known or unknown requirement. Unknown requirements are the requirements about which end users don't have any idea.

3.2.3 Domain Requirements

Domain requirements are the requirements that are specific to an application domain e.g. banking, hospital etc. These requirements are therefore identified from that domain model and are not user specific. For a satisfactory working of the system these are mandatory requirements to be implemented or satisfied. For example in an engineering college there will always be requirements concerning faculty, students, staff and the departments.

3.3 REQUIREMENT ENGINEERING PROCESS

Requirement engineering process consists of following main activities:

- Requirements elicitation
- Requirements specification
- Requirements verification and validation
- Requirements management

In all these four activities users contribute significantly.

3.3.1 REQUIREMENTS ELICITATION

The activity of requirements elicitation is concerned with the understanding of problem domain at the beginning of the project because requirements are fuzzy and are not clearly understood by the analyst. Hence best solution is to acquire knowledge about a specific problem domain and become expert in that.

The process of acquiring information/knowledge about a specific problem domain through various techniques to build the requirements model is called requirements elicitation.

This activity of requirement elicitation helps the analyst to gain knowledge about the problem domain which in turn is used to produce formal specification of software to be developed to meet the customer needs. The various sources of the domain knowledge can be users, business manuals, existing software of same type, standards and other stake holders of the project.

It is to be noted that elicitation does not produce the formal models of the requirements understood. Instead it widens the knowledge domain of the analyst and thus helps in providing input to specification stage for producing formal models. Inputs to requirement elicitation methods include high level business objectives, statement of scope and feasibility statement. Proper elicitation of requirements of the product helps in:

- Defining the product
- Controlling the requirements in effective manner
- Identification of risks and also resolving them
- Prioritizing requirements
- Identification of appropriate tools and methods to build the product
- Effort and cost estimation

(Kishore01) in their book Software Requirements and Estimation further divide the requirements elicitation activity into following tasks

- ***Fact Finding*** – studies the organization in which the final system will be installed and defines the scope of the proposed system.
- ***Collecting Requirements*** – captures information from different users viewpoint.
- ***Evaluation*** – focuses on identifying inconsistencies in the requirements collected
- ***Prioritization*** – prioritizes the requirements depending upon cost, user needs, ease of development etc.
- ***Consolidation*** – Consolidates the requirements from the information gained in a way that can be analyzed further and ensures that they are consistent with the goals of the organization

Requirements elicitation techniques are discussed in detail in section 3.4.

3.3.2 Requirement Specification

The purpose of requirements specification activity is to produce formal software requirements models. These models specify the functional and non-functional properties of the system along with the constraints imposed on the system. These models are used in successive stages of SDLC and are used as an agreement between the end users and the developers.

Input to the requirement specification stage is the knowledge acquired in raw format during the requirement elicitation stage. During specification more knowledge about the problem may be required which can again trigger the elicitation process. Similarly if the

user is not satisfied with the requirement validation phase, elicitation and specifica phase is again repeated. A number of structured analysis and object oriented analy techniques have been proposed to develop these models. They will be discussed in de in chapter 5 and 6 respectively.

3.3.3 Requirement Verification and Validation

Requirement validation is defined as a process to ensure the consistency of requireme model with respect to customer's needs. The validation is done not only for the fi model but also for all intermediate models generated. If requirements are not validat errors in the requirements definition would propagate to the successive stages result into lot of modification and rework. This in turn will result into legal and financ implications. Following steps should be followed while validating requirements:

- Ensure that requirements are consistent. They are not conflicting with oth requirements.
- Ensure that requirements are complete in all respect.
- Ensure that requirements are realistic and realizable.

It is seen the reviews, prototyping, test cases generation are effective ways to valid requirements. During reviews end users and developers both study and consider t requirements. In prototyping an executable model of the product with minimal set requirements is built and is demonstrated to the customers. Customers in turn use t prototype to ensure that it meets their needs. In test case generation, it is made su that it is possible to design test cases for each and every requirement documented oth wise there is a need to reconsider the requirement. The output of this stage is requirement model which is consistent and meets the users expectations.

3.3.4 Requirement Management

An important characteristic of the software requirements is that requirements ke changing. In other words they are dynamic in nature. Therefore we must be able incorporate these changes in a systematic and controlled manner. This is ensured throug requirements management part of the requirements engineering process through a w defined change management process. It is ensured that SRS is made as modifiable possible so as to incorporate new requirements or the changes in the existing requirement

3.4 REQUIREMENTS ELICITATION TECHNIQUES

A number of techniques have been proposed in the literature to elicit the knowled about the domain. There is no single approach that can be used in all the cases. T success of an elicitation technique used depends on maturity of analyst, developers, us and the customer involved. Analyst in order to elicit the requirements successfully m use the best subset of the elicitation techniques. Next we will present some of the popul techniques used for eliciting requirements.

3.4.1 Interviews

Interview is one of the most popular technique for understanding the problem domain and is used by almost all the analysts at some point during the SDLC. It may take the form of questionnaire, open ended interviews and focus groups (Joseph93). Analyst can simply ask the users about their expectations from the system but the main problem is that users can bypass their limitations. So analyst can probe the user through a set of questions thus avoiding the above mentioned problem. This type of interview is called *structured interview*. The focus group is a kind of group interview where groups are brought together to discuss some topic of research to the researcher and allow more natural interactions than open ended interviews.

(Robert02) has suggested the following steps in interviewing: create the questions, select the interviewers, plan contacts, conduct the interview, close the meeting and determine where to go next. Questions must be created carefully so as to extract maximum knowledge about true requirements. Since it is not possible to interview every stake-holder, one must choose the representation carefully. They can be entry level personnel, mid-level stake holders, CEO's and VP's of Company, academicians (in order to have a different perspective) and of course users of the system. After doing some research on the persons you wish to contact, interviews can be conducted on phone, personally or even over the internet. Once enough knowledge is extracted, the person being interviewed is motivated because his/her services could be required in future.

3.4.2 Brainstorming

Brainstorming used in many business applications, is a group technique to promote creative thinking and can be used during requirements elicitation process to generate new ideas and solve problems. In this technique, the session consists of a group of people who are free to say what ever comes to their mind irrespective of their relevance. They are also not criticized for the ideas. The more requirements that can be identified in the beginning, the better. It is always easier to select from a long list of ideas or to create a new idea by combining lots of existing ideas.

The brainstorming session is attended by a group of 5 to 10 people. There are three types of participants with different roles. They are:

- Leader
- Scribe
- Participants

The role of the leader is to lead the brainstorming session. This is done by encouraging the participants to come up with new ideas and helping the scribe to capture these ideas. He/she can also help the participants to combine the ideas. The role of scribe is to write down each idea briefly in such a way that it is visible to all the participants present in the session. In order to do so, the scribe can also use tools like flip charts, overhead projectors and white boards. The primary role of participants is to produce new ideas.

After the session is over, leader edits the ideas with the help of one or two persons who are familiar with the domain. The incomplete ideas are discarded by the leader and kept for later reconsideration. Rest are documented to be used as input to Software Requirement Specification document. The leader of the session therefore should try to extract maximum ideas from the participants. Brainstorming sessions normally last for two to three hours.

3.4.3 Task Analysis

Task analysis is a technique of decomposing the problem domain into a hierarchy of tasks and subtasks performed by the users. For example the task 'Donate Blood' may consist of following subtasks.

1 Donate Blood

- 1.1 Donor approaches the blood bank
- 1.2 Staff takes the sample of the blood
- 1.3 Staff checks for any infection in the blood and the blood group
- 1.4 If found OK, he takes the blood and stores it.
- 1.5 Staff updates the quantity of the blood group in the inventory.
- 1.6 Blood bank issues a card to the donor for later use

All other techniques used for eliciting requirements can also be used to identify tasks/subtasks for providing valuable inputs for organizing knowledge about an existing system. It is an effective technique for eliciting requirements concerned with Human-Computer Interaction (HCI).

3.4.4 Form Analysis

Forms plan an important role in any organization and contain lot of meaningful information about data objects of the domain as shown in the faculty registration form in Fig. 3.2. Forms are important source of information for modeling the static view or data aspect of the system. For example in the form shown in Fig. 3.2, a number of concepts like 'Employee', 'Name', 'Salary', 'Address', 'Department' etc., can be found and can be used to model entities, relationships or attributes.

Faculty Registration Form

Name :
Address :
Position :
Department
Date of Joining :
Salary :

Fig. 3.2 A Sample Form

Normally forms are used as an input to the process of constructing Entity Relationship(ER) model(Chen76). Instead of manually constructing data model from forms automated approaches for form analysis(Choobine88) can also be used and are more productive. A part of ER model derived from the form in Fig. 3.2 is shown in Fig. 3.3.,

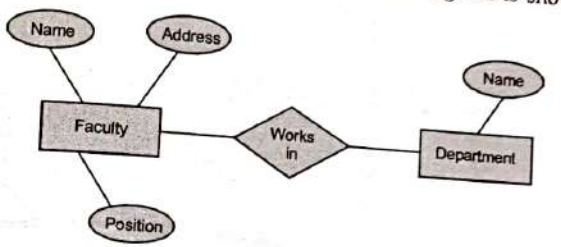


Fig. 3.3 ER Diagram Based on Form of Fig. 3.2

3.4.5 User Scenario and Use Case Based Requirements Elicitation

This is a technique for capturing requirements from users point of view. This technique is based on notion of scenario. According to (Loucopoulos95) a scenario is a story that illustrates how a perceived system will satisfy a user's needs. Use Case scenario, Use Case and Use Case diagram are often taken as one but in fact they are different. Use Case scenarios are unstructured descriptions of the user requirements. On the other hand Use Cases are structured descriptions of the user requirements. It is a narrative text which describes the sequence of events from user's perspective. Use Case diagrams are graphical representation to show the system at different levels of abstraction.

An actor is a person, machine or some other information system that is outside the boundary of the system and interacts with the use cases. In this technique first actors interacting with the system and Use Cases are identified and this information is used to draw Use Case diagrams. These diagrams are supported by activity diagrams to show workflow view of activities. Use case models thus developed play a major role in understanding the requirements of business and have following benefits:

- Easily understandable by managers, developers, users and other stake holders of the project.
- Supports validation and implementation testing.
- Supports traceability.
- Easily converted into object models.
- Describes the existing system under development accurately.

A sample use case diagram is shown in Fig. 3.4. Use case model would be discussed in detail in chapter 6 while discussing Object Oriented Requirements Analysis techniques.

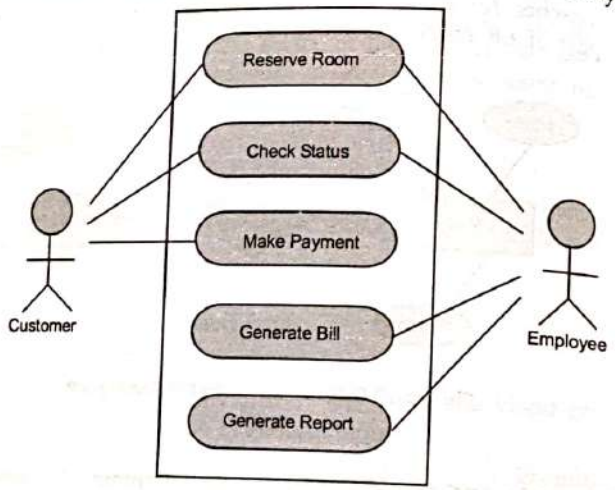


Fig. 3.4 A Sample Use Case Diagram of Library

3.4.6 Delphi Technique

In a Delphi technique session, participants are made to write the requirements. These requirements are then exchanged among participants who give their comments to get a revised set of requirements. This process is repeated till a final consensus is reached.

3.4.7 Domain Analysis

This technique focuses on reuse of requirements from similar domain. Several definitions of domain analysis exist in the literature and are given below.

According to (Prieto91) domain analysis is the process of identification, acquisition and evolution of reusable information on a problem domain.

According to (Kishore01) domain analysis is the process of identifying, collecting, organizing and representing the relevant information in a domain.

In this technique therefore within a domain, existing systems are studied and a generic domain model which is an abstraction of main features of applications of the domain is produced. This is just like representing the requirements at another higher level of abstraction called meta level. These requirements at meta level can then be modified to fit the problem domain as shown in Fig. 3.5.

Thus in the domain analysis paradigm, the requirement elicitation process consists of sequence of following two steps:

(i) Selection of reusable requirements

(ii) Adaptation of requirements selected in (i) for incorporating in the new model.

It is to be noted that this technique will not give good results for immature or evolving domains.

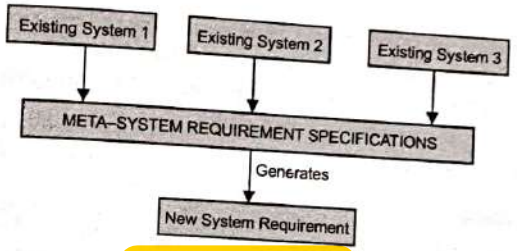


Fig. 3.5 Domain Analysis

3.4.8 Joint Application Design (JAD)

Joint application Design(JAD) is a registered trademark of IBM and was developed by IBM in 1970's. It is also a type of extremely effective, structured and disciplined session because it is represented by persons who are well informed and knowledgeable. They are users, customers, functional experts, system experts and other stakeholders of the project. As compared to brainstorming sessions which normally last for two to three hours JAD sessions can last up to three days and can even produce high level software models like data flow diagrams, use case diagrams etc. instead of documenting only ideas.

The JAD sessions are attended by sponsor, facilitator, developers, scribe and the users. The sponsor represents the top management and is always available to resolve any issue which has come up during the sessions. He may not attend all the sessions. The facilitator ensures that sessions are carried out smoothly and all participants attend the session with an open mind. The role of scribe is to capture and record the ideas properly. Users contribute significantly in the JAD meetings by explaining their requirements. Developers of the system inform the participants in the meeting about the latest tools and technology that can be used to meet the requirements.

3.4.9 Facilitated Application Specification Technique (FAST)

FAST was developed specifically for collecting requirements and is similar to brainstorming and JAD. As in the case of brainstorming meeting is conducted at some neutral site which is attended both by developers and customers/end users. The meeting is controlled

by a facilitator and an informal agenda is published. Before starting of session, l objects (interacting with the system, produced by the system and used by the sys functions and constraints is made. This list is presented in the session for discus Participants before starting the session also agree not to debate. After discussion s of the entries from the list is eliminated and new entries are also added to the list process is continued till a consensus is reached. Additionally following activities take place:

- Identification of external data objects.
- Detailed description of software functionality.
- Understanding the behavior of software.
- Establishing the interface characteristics.
- Describing the design constraints.

3.4.10 Prototyping

In addition to use prototyping as a SDLC approach, it can also be used as an effe tool for requirements elicitation. Some users/customers are not clear about requirements and some are not able to visualize the working of the final product though they have the SRS document with them. At this point prototyping becomes effective means to convey to users how the product will work and to get their feedba Based on their feedback received user interfaces can be modified, new requirements be added or some existing can be dropped. This technique can also be used to ens that a requirement as asked by user is technically feasible.

The use of prototyping as requirement elicitation technique helps in more involvement and more satisfied users but at the same time has problems like -

- Customers may consider the prototype as the final product and are not re to wait for the final product with full functionality.
- Total cost of project is increased as the cost of developing the prototyp added to it.
- Developers in a hurry to deliver the product may not like to document SR

3.5 CHALLENGES IN ELICITING REQUIREMENTS

The most important activity of software life cycle is the extraction and refinement of product requirements. However there are many challenges also. Some of them are

- Understanding large/complex system requirements.
- Undefined system boundaries.
- Users not clear about their needs.
- Representing requirements in suitable form.
- Conflicting requirements.

- Requirements changing quite often.
- Resolving TBDs (To be determined).
- Partitioning the system suitably in order to reduce complexity.
- Validating and tracing requirements.
- Proper documentation of the requirements.
- Meeting the time and budget constraints of the customer.
- Identifying the critical requirements.

3.6 SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

The output of requirement engineering process is Requirement Specification Document (RSD) also called Software Requirement Specification (SRS).

A SRS is a document which contains the complete description of software without talking much about implementation details. It is a set of precisely stated properties and constraints which the final product must satisfy.

In other words this document establishes boundaries on the solution space of the problem domain. SRS aims at bridging the communication gap among the different stake holders of the project and also forms a contract between the customer and the developing organization.

Depending upon the type of project and the person who is writing it, SRS may be detailed or general. The document is used as a reference tool for all the successive stages of software life cycle and should specify the external system behavior, constraints and responses to undesirable events in detail. It should be easy to modify and serve as an authentic document for the system testers and maintainers. SRS can be represented using written textual document, a set of models, requirement specification languages, use cases, mathematical models or prototype.

SRS must address the following issues:

- Functional requirements in terms of inputs required, outputs generated and the processing details
- Non-functional requirements e.g., accuracy, efficiency, maintainability etc.
- Static requirements
- Execution constraints e.g., response times, throughput times
- Standards to be followed
- Security requirements
- Company policies
- Interface with other external agents i.e., persons, software or hardware

3.7 CHARACTERISTICS OF A SRS DOCUMENT

Quality characteristics of a good SRS are as given below:

1. Correctness
2. Completeness
3. Consistency
4. Unambiguousness
5. Ranking for importance and/or stability
6. Modifiability
7. Verifiability
8. Traceability
9. Design independent
10. Testability
11. Understandable by customer
12. Right level of abstraction

Correctness

An SRS is said to be correct if every requirement stated in the document is correct what is expected from the system. All the users may be asked to review the docum to ensure that the requirements stated in the SRS are as per their needs.

Completeness

An SRS is said to be complete if

(a) All the significant functional and non-functional requirements to be satisfied the software are included in the SRS.

(b) Response to all valid and invalid class of data is included.

(c) All figures, tables, and pages are numbered. They are also properly named a referenced.

(d) Sections in the SRS should avoid the label "To Be Determined" (TBD) as far possible. If at all it can not be avoided it must be accompanied by informati like who is going to resolve it and when it will be resolved.

Consistency

As SRS is said to be consistent if there is no conflict between the requirements in t documents. Different types of conflict may exist between the requirements. Some them are:

- Conflicts between characteristics of real-world entities e.g., in a proje management system a requirement can state that a project manager can supervi only one project where as another requirement can state that a project manag can supervise maximum 5 projects.

- Temporal or logical conflicts between two items e.g., one requirement may state that reports should be generated once in a month where as another requirement may state that reports should be generated once in a week.
- Conflicts can also arise if different terminology is used to describe same real world object or event e.g., the terms customer and client which represent the same entity.

Unambiguousness

Unambiguousness in the SRS document is supported only if each and every requirement stated in it has only one interpretation. Unambiguousness in the SRS document can be best achieved by using requirements specification languages, requirement analysis and modeling techniques (i.e., ER diagram, data flow diagram etc.) and reviewing the document written in natural language by a third person.

Ranking for Importance and/or Stability

Requirements may be classified as essential or desirable. Some criteria must be used to distinguish the two. This can be done by associating an identifier with each requirement to indicate the stability or the importance of that requirement.

Modifiability

Requirements keep changing and changes must be reflected properly in the SRS. *An SRS is said to be modifiable according to IEEE-standard 830-1998 if changes can be made to the requirements easily, completely and consistently while retaining its structure and style.* It means that requirements must be properly indexed and cross referenced.

Verifiability

As SRS document is said to be verifiable if for each requirement written in SRS, there exists some cost effective process using which a person or tool can ensure that software meets the requirements. Verifiability depends on the way a requirement is stated in the document e.g., a requirement "user interface of the software should be user friendly" is ambiguous and will have multiple interpretations by multiple users. Hence it will not be possible to verify this requirement. Hence one must document the requirements carefully and should avoid using ambiguous statements or using non measurable quantities.

Traceability

Traceability is an important quality feature of the SRS document and is supported by SRS document if origin of the each requirement is clearly documented. All the requirements must be traceable while designing or testing the product. Therefore all the paragraphs must be numbered hierarchically and all requirements in the paragraph must also be numbered properly. One should be able to trace a requirement to a design component and then to the code segment in the program. Similarly one should be able to trace the requirement to the corresponding test cases.

Design Independent

An SRS is said to be design independent if it does not include any implementation details. In fact there must be provision for multiple design alternatives to support requirement.

Testability

An SRS must be written in such a manner that it is easy to generate the test plans and test cases from the document.

Understandable by Customer

An SRS should be easily understandable by the end user/customer. One must avoid using formal notations to represent requirements. This is because the end users may be expert in their problem domain but they need not be computer science experts.

Right Level of Abstraction

All the details in the SRS document must be represented at the right level of abstraction as per the objective of the specification phase. In case SRS is written for developing the software, requirements must be specified in as much detail as possible. But if the SRS is written for say feasibility study then SRS can document the requirements at higher level of abstraction.

3.8 ORGANIZATION OF SRS

There are various ways to organize the SRS for proper understanding. Different organizations use different guidelines to document SRS as per their needs. A popular standard published by Institute of Electrical and Electronics Engineers (IEEE) to organize Requirements Specification Document(RSD) or SRS is IEEE standard 830. The guidelines proposed in this standard can be customized by the organizations to meet their needs.

A simple template for organizing the SRS is shown in Fig 3.6. A brief description of various components is as follows:

Title Page—This section of the SRS consists of name of the project, author(s) and the version control information.

1. **Introduction.** This section of the SRS provides the overview of complete SRS in terms of its purpose and the audience being targeted. Scope of the project briefly describes its functions, benefits, objectives and goals of the software being specified. This section in addition to these also describes the various terms and acronyms used in the SRS and the references made to all other documents while writing the specification. In the end this section describes the outline of the rest of the document.

   Overall Description. This section provides a background to understand the requirements specified in section 3 of the document in terms of

   2.1 ***Product Perspective***: This part of SRS establishes the connection between the product and other projects i.e., whether the product is independent or part of larger system.

   2.2 ***Product Functions***: Provides a summary of functions that will be performed by the software.

   2.3 ***Users Characteristics***: Describes the user characteristics in terms of their expertise, knowledge etc.

   2.4 ***Constraints :*** This sections describes the details of various constraints related to hardware, high level languages, safety considerations, protocols, etc.

   2.5 ***Assumptions and Dependencies:*** This section describes the dependencies which influence the SRS or trigger the changes to be made in the SRS for example availability of a specific operating system or software to be used for developing the system.

3. **Specific Requirements.** This section is the most important section out of all the sections. This section describes all the relevant details which are necessary for the system designers to design a fool proof system.

   3.1 ***External Interface Requirements:*** This section describes the details of user Interfaces (in terms of screen formats, reports layout, function keys etc.) software required(Operating Systems, DBMS or any other tool to develop the product), hardware Interfaces by specifying the interfaces between the hardware components and the product and communication interfaces.

   3.2 ***Functional Requirements:*** This section describes all the functional requirements of the system in detail in terms of inputs, outputs and processing of the system. If required models like data flow diagrams, UML etc. can be used to describe the details pictorially.

   3.3 ***Performance Requirements:*** This section describes the speed requirements in terms of number of transactions processed per second. It can also specify the requirements like number of users that can be supported simultaneously by the software.

   3.4 ***Design Constraints:*** Describes the constraints imposed on the system by the standards and the hardware used.

   3.5 ***Non Functional Requirements:*** This section describes the quality attributes or the constraints imposed on the system. Some of the quality characteristics which are described in this section are: safety, efficiency, security, reusability, testability, portability, maintainability, interoperability, expandability, usability, correctness and reliability. Depending upon the product being developed some of these can be selected and described in this section.

4. Other Requirements

4.1 *Apportioning of Requirements:* This section gives the details of those requirements which are given to third party for development.

4.2 *Database:* This sections describes the details of the database which will be developed as a part of the project.

4.3 *Schedules and Budgets:* This section describes the detailed project plan and schedule and budget estimate. This section is optional.

A sample SRS based on the techniques discussed in chapter 5 and 6 is given in appendix A.

A SAMPLE FORMAT FOR WRITING SRS

A title page

1. Introduction
- **1.1** Purpose
- **1.2** Scope
- **1.3** Terminology used
- **1.4** Reference to other documents
- **1.5** Overview

2. Overall Description
- **2.1** Product perspective
- **2.2** Product functions
- **2.3** User characteristics
- **2.4** Constraints
- **2.5** Assumptions and dependencies

3. Specific Requirements
- **3.1** External Interface requirements
  - **3.1.1** User Interface requirements
  - **3.1.2** Software Interface requirements
  - **3.1.3** Hardware Interface requirements
  - **3.1.4** Communication Interface requirements
- **3.2** Functional requirements
  - **3.2.1** Functional requirement -1
    - **3.2.1.1** Brief description (Using text or models)
    - **3.2.1.2** Inputs
    - **3.2.1.3** Outputs
  - **3.2.2** Functional requirement -2
    - **3.2.2.1** Brief description (Using text or models)
    - **3.2.2.2** Inputs
    - **3.2.2.3** Outputs
  - **3.2.3** Functional requirement - 3
  - **3.2.4** Functional requirement - 4
- **3.3** Performance requirements
- **3.4** Design constraints
- **3.5** Non-functional requirements

4. Other requirements
- **4.1** Apportioning of requirements
- **4.2** Database
- **4.3** Schedule and budgets

Appendices

Fig. 3.6 A Sample Template for Organizing SRS

3.9 ROLE OF A SYSTEM ANALYST

A system analyst is a person with unique skills and experience who undertakes the job of system analysis. He/she is the who is responsible for identifying the business goals and activities and the strategies for meeting these objectives. Jobs carried by the analyst are:

- Identifying and documenting the business goals.
- Identifying and satisfying different stakeholders of the project.
- To consult the managers, data processing professionals and users in order to define the problem statement clearly.
- To propose a plan to solve the problem and to meet the management objectives.
- To coordinate the design and testing activities.
- Resolve conflicts and suggest solutions.

Some important skills required by the analyst are:

- Ability to communicate
- Excellent analytical skills
- Knowledge of latest technology
- Knowledge of business environment
- Creative and imaginative mind
- Well educated
- Ability to coordinate different team member's activities

SUMMARY

- Requirement engineering is a process of producing requirement specification document and consists of four main activities-requirements elicitation, requirements specification, requirements validation and requirements management.
- Requirement engineering is the most important activity of the software development life cycle and should not be underestimated.
- A number of requirements elicitation techniques like interviews, task analysis, brainstorming, use case diagrams, form analysis etc., can be used to elicit the requirements.
- Requirement specification document or SRS must be implementation independent.
- Analysis must ensure that SRS being developed is complete, consistent, unambiguous, traceable, modifiable, verifiable, understandable etc.
- A number of standards like IEEE standard 830, DOD-STD-2167A are proposed to organize SRS.

REVIEW PROBLEMS

1. What is a requirement? Explain briefly.
2. List at least twenty non-functional requirements you can think of.
3. What are the different stages of requirements engineering process? Explain briefly.
4. Write down using natural language, functional requirements of a Library management system. Explain which requirement elicitation techniques you used while documenting the requirements.
5. Why is software requirement specification important?
6. What are the characteristics of SRS? Explain?
7. Develop an SRS document as per [IEEE-Std 830] guidelines for a shopping mall.
8. List few challenges faced during requirements elicitation.
9. Identify which of the following requirements are functional requirements and which are non-functional requirements:
   (a) The inventory management system must enable the user to add new items to the stock.
   (b) The inventory management system must be implemented using ORACLE and Visual BASIC.
   (c) The response time of system must be less than 5 seconds.
   (d) The inventory management system must enable the user to generate different types of reports.
   (e) The system must be easy to understand and use.

○○○

Chapter 4

SOFTWARE DESIGN, CODING AND DOCUMENTATION

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- Why design is important?
- The concept of modularity.
- Different types of coupling and cohesion
- Level oriented design
- Function oriented design
- Object oriented design
- Structure chart
- Coding guidelines
- Different types of documentation

4.1 INTRODUCTION

Software design is an iterative process during which the software requirements specified in SRS are analyzed and converted into description of the internal structure and organization of the system. The document thus produced is called the ***software design specification document*** and serves as the basis for construction of the software. There are two main activities

- **Software architectural design or high level design** – In this the system is decomposed and organized into high level components or modules and the interfaces between these components are also described.

- **Software implementation or detailed design** – Here each component is described in sufficient detail so that it can be easily coded by the programmers.

In other words the process of design takes as input a specification (a description of what is required) and transforms it into a blueprint for construction (a description of what is to be built). The design stage also acts as a bridge between SRS and the final solution. In the requirements analysis stage the emphasis is on understanding the problem domain but the system design stage focuses on solution domain and on optimization of final system. As per IEEE design can be defined as:

> ***Design is both the process of defining the architecture, components, interfaces, and other characteristics of a system or component and the result of that process.***

The design process involves creativity and thus it is difficult to summarize it in a simple design formula. In a software design problem, a number of solutions can exist. The designer must use his past experience and existing methodologies to propose a rational solution to the problem. The designer must plan and execute the design strategy taking into account certain established design practices. According to (kelkar07) design involves both diversification and then convergence. *Diversification* refers to different technologies, repertoires, knowledge etc. *Convergence* deals with choosing the best possible alternative from amongst available alternatives. ***IEEE Standard 1016*** discusses the guidelines for the description of design.

4.2 WHAT IS A GOOD DESIGN?

Design phase plays a very important role in SDLC as it affects the quality of the proposed system. A properly designed system results into reduced maintenance costs. A design is said to be good if it is

(*i*) **Complete** i.e., it will build everything as required by the end user.
(*ii*) **Economical** i.e., it will not build what is not required.
(*iii*) **Structured** i.e., it can accommodate changes due to changed requirements, error etc.
(*iv*) **Constructive** i.e., it says how to build the product.
(*v*) **Traceable** i.e., it can be traceable to analysis model
(*vi*) **Uniform** i.e., it uses the same building techniques throughout.
(*vii*) **Testable** i.e., it can be shown to work.

4.3 MODULARITY

The concept of module is not new in writing software. It is seen that software is best designed and maintained as a set of small easily handled components called *modules*.

> ***A module can be defined as a sequence of lexically contiguous program statements bounded by boundary elements i.e., statements like begin and end. Also each module has a name by which it can be called or invoked by other modules.***

In broader sense a module represents a subroutine. In different programming languages different names like *function, subprogram, procedure etc.*, are used to represent modules.

While partitioning the system two objectives must be met. They are:

(*i*) In case error occurs in some module, it should not effect the working of other modules. It means faults must be localized or specific to a module.
(*ii*) Similarly when a module is modified, there should be minimal need to change other modules.

The main aim of the design phase is therefore to partition the system through the process of partitioning and propose a model of inter related modules that can be developed, modified and tested independently. It means that as far as possible inner working of the modules must be hidden from each other and they must act as black boxes to each other. This principle is called **information hiding.** This module independence depends on two properties called ***coupling*** and ***cohesion.*** While coupling is the degree of interdependence between two modules, cohesion is the measure of strength of elements of an individual module and is property of single module. While designing focus must be to minimize the coupling between the modules and to increase the cohesion within a module. Next we discuss different types of coupling and cohesion.

4.3.1 Cohesion

Cohesion is the property of a single module. It can be described as a glue that keeps the data elements within a single module together. Designer must try to achieve high cohesion within a single module. Constantine and Yourdan has identified seven levels of cohesion as described below:

1. **Coincidental Cohesion.** In this, the data elements or instructions within module have no relation to each other. It is the worst kind of cohesion.
2. **Logical Cohesion.** In a module, supporting this type of module, components are logically related to each other *e.g.*, a module which contains routines to support different types of inputs. Within the module these routines do not communicate with each other by exchanging information. Hence required routine to be executed are chosen from outside the module by providing necessary control information. Logical cohesion is difficult to maintain.
3. **Temporal Cohesion.** In such type of cohesion, module consists of activities which are temporary related to each other. All the instructions occur at the same point in time. Example is an error module which is invoked whenever failure takes place. The module

- Cancels all tasks
- Closes all windows
- Saves all data files
- Shuts down the machine

These are tasks which are not logically or functionally related but take pla when failure takes place.

4. **Procedural Cohesion.** In a module supporting procedural cohesion elements a involved in unrelated activities in which control not data flows from one activit to another and activities are executed in some given order as shown in Fig. 4.

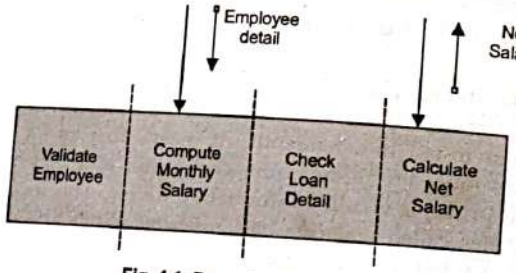


Fig. 4.1 Procedural Cohesion

In this module all the activities separated by dashed lines are sequential in nature

5. **Communicational Cohesion.** A module with communicational type of cohesion is the one whose elements contribute to activities that use the same input or output data. Sequence is not important in this case as shown in Fig. 4.2. In this module different activities performed are to retrieve the personal information of an employee, to calculate the salary details of the employee, to retrieve the details of different projects on which the employee has worked etc. In all these the input is the employee code. A module supporting the communicational cohesion can be split into independent functionally cohesive modules.

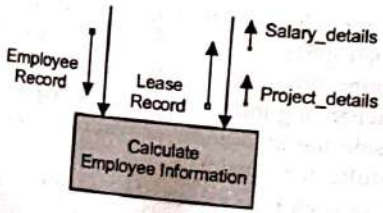


Fig. 4.2 Communicational Cohesion

6. **Sequential Cohesion.** If in a module output elements of one activity is used as input to another activity, module is said to support sequential cohesion as shown in Fig. 4.3.

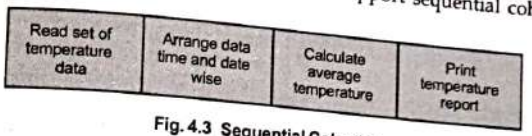


Fig. 4.3 Sequential Cohesion

In this module first set of temperature data is read and next it is arranged date and time wise. After that day wise average of temperature is calculated and then the results are printed.

7. **Functional Cohesion.** This is the most desirable kind of cohesion with in a module. In such type of module all elements contribute to execution of one single function. This feature is better supported in object oriented languages. Modules with single purpose are easiest and cheapest to maintain.

4.3.2 Coupling

Coupling is the property of modules which shows the dependence between the modules. As we want modules to behave as black boxes to each other, designer should aim for minimum coupling between the modules. Different type of coupling (from worst to best) is as given below:

1. **Content Coupling.** This is the least desired kind of coupling and should be totally avoided. This type of coupling occurs if one module refers to the inside of another module either by changing data or passing control. For example in pascal one can branch from one procedure to a label in another procedure.
2. **Common Coupling.** Two modules are said to be common coupled if they share same global data area. The common coupling has following disadvantages:
   (a) Modules become inflexible as they have to refer to same global data and reverse is restricted.
   (b) If by mistake global data is corrupted by any module, all other modules using it are affected.
   (c) Programs and modules become difficult to maintain.
3. **Control Coupling.** In such type of coupling between modules, one module controls the execution of second module by passing a control information. It means that inner details of second module are known to the first module and black box concept no longer exists. In this type of coupling at least one control flag exists between the two modules as shown in Fig. 4.4.

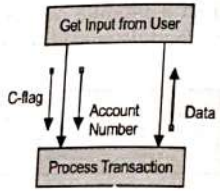


Fig. 4.4 Control Coupling

In this flag depending upon the value of C-flag either cash is withdra deposited or balance is shown to the user.

4. **Stamp Coupling.** Two modules are said to be stamp coupled if a compo data structure or record is passed from one module to another which makes modules more dependent on each other. For example if data is changed adding extra field to it, another module which is using it is also changed. Stam coupling is not bad but not as good as data coupling.

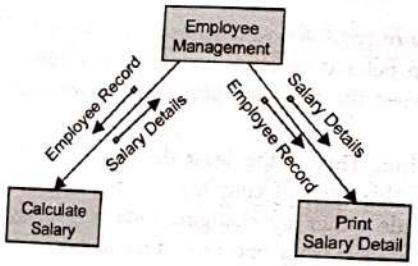


Fig. 4.5 Stamp Coupling

5. **Data Coupling.** This is the coupling between modules that cannot be avoide In this type of coupling minimum and essential data is exchanged betwe different modules.

The designer must aim for *low coupling* among the modules. Some guidelin for reducing coupling between modules are:

- Use simple interfaces.
- Use minimum number of interfaces.
- Reduce the complexity and the number of parameters passed among differe modules.
- Reduce the sharing of global data by different modules.
- Avoid referring to the inside of another module by a module either changing data or passing control.

4.4 SOFTWARE DESIGN APPROACHES

A number of software design approaches are proposed in the literature. Details of software design approaches can vary greatly. Some consist of a set of guidelines, while others include set of rules and a set of coordinated diagrammatic representation. Many of these approaches are really full-fledged software design methods.

Different approaches have been used to develop software solutions for different problems. However, in some problems, different approaches have been integrated or combined in a logical manner to derive a software solution.

4.4.1 Level Oriented Design

In the level-oriented design approach, two different strategies can be used. They are:

- Stepwise refinement
- Design by composition

The first strategy starts with a general definition of a solution to the problem then through a step-by-step process produces a detailed solution (this is called Stepwise Refinement). This is basically dependent on the system requirements and is a top-down process. The other strategy is to start with a basic solution to the problem and through a process of modeling the problem, build up or extend the solution by adding additional features (this is called design by composition).

The ***stepwise refinement*** *is a top-down process which starts at the top level and by functional decomposition, breaks down the system into smaller functional modules.* Smaller modules are more readily analyzed, easier to design and code. Functional decomposition is an iterative "break down" process called stepwise refinement, where each level is decomposed to a more detailed lower level. For example the function of admitting students may be decomposed into following sub functions:

(*i*) Advertise in the paper
(*ii*) Invite applications
(*iii*) Shortlist candidates
(*iv*) Conduct examination
(*v*) Admit the students based on merit list

Here the function shortlist candidates and Conduct Examination can be further decomposed into sub functions. The advantages of this approach are:

- The software is easy to understand and maintain.
- Supports parallel development of different modules by different team members.

The main problems with this strategy are:

- The designer using this strategy must have a complete understanding of the problem or system at hand. Otherwise, it could lead to extensive redesign later on.

- The decisions made at the early stages effect the design structure of the softwa
- There are no inherent procedures or guidelines to determine whether fur decomposition is needed or not.

The top-down process is often used in the initial phase of the design process decompose the system into main modules which can later worked upon.

The *design by composition strategy* starts with the initial solution and through iterative composition process expands the solution to include additional modules. T approach is similar to the bottom-up design, where the lowest level solution is develop first and gradually builds up to the highest level. This strategy is useful when requirements are not very clear or iterative life cycle models are used. This strategy helps in identifying some common modules that can be shared by higher modules.

In practice the combination of both top down and bottom approaches is used partitioning the system.

4.4.2 Data Flow-Oriented Design

Salient features of this approach are:

- Data flow-oriented design approach is often called **Structured Design.**
- This approach uses the flow of information among different processes to der the program structure called **Structure Chart.**
- The flow of information can be shown with the help of data flow diagram(DF discussed in detail in chapter 5.
- Using a set of guidelines a DFD can be transformed into a structure chart
- A DFD can be mapped into the design structure by two means - transfo analysis or transaction analysis.
- Transform analysis is applied when the data flow in the input-output stream h clear boundaries. Transaction analysis is applied when a single information ite causes flow to branch along one of many paths.
- A few examples of structured design or data flow-oriented design methodolog are Structured Analysis and Design Technique (SADT), Systematic Activ Modeling Method (SAMM) and Structured Design (SD).

The notation for drawing structure chart is explained in section 4.5. The structu design technique is explained in detail in chapter 5 by taking a suitable example.

4.4.3 Object Oriented Design

The object oriented paradigm consists of two activities –*object oriented analysis* (OO and *object oriented design*(**OOD**). In the object oriented analysis phase focus is understanding the problem domain and identifying the objects and classes, their attribu and methods. Interest is also to understand the association between different class OOA consists of Use Case models, business models, activity diagrams, collaborat diagrams, sequence diagrams and class diagrams to identify the initial set of classes a their relationships.

An object consists of a private data structure and related operations that may transform the data structure. It also has an interface through which operations can be performed on the objects by passing messages. A set of objects showing similar behavior is represented by a class. In object oriented design now we concentrate on classes identified in OOA and design them in more technical detail. The process of OOD consists of following steps:

(*i*) Design the classes identified in OOA. Check the existing class library in order to reuse the classes. If the existing classes are not suitable then only design the new classes. Most object oriented languages like C++, Java, smalltalk etc., have rich built-in libraries.

(*ii*) Design the attributes in detail by naming it correctly, specifying the data types used, visibility etc. The attributes must indicate the state of the class at some point of time. Drop the attributes which are not relevant.

(*iii*) Design the methods by specifying the input parameters being passed, output parameter to be returned, algorithm of the method and the visibility. Visibility defines how a method will be accessed by other classes.

(*iv*) Design the user interface classes in addition to the main classes.

(*v*) Refine the classes.

The ultimate aim of OOD is to design classes which are simple and easy to maintain. In order to achieve this:

- Design classes which are as far as possible independent of other classes.
- Make maximum use of class libraries.
- Organize the classes into a hierarchical structure using the concept of inheritance so as to reduce the programming effort. The use of inheritance also results into cohesive classes with single purpose.
- Keep minimum information contents in a class.
- Make use of ***Design Patterns.*** Design patterns represent solutions to problems that arise when developing software within a particular context. Patterns support reuse of software architecture and design. For example there can be a pattern to admit student in a school, to place an order etc.

Object oriented analysis and design is discussed in detail in chapter 6 of this book.

4.5 STRUCTURE CHART

The purpose of system design stage is to produce an overall hierarchical implementation model of the software system in terms of modules and to show exchange of data between these modules. Structure chart is an important tool used to show this modular structure. The concept of black box as we know is not new in the field of engineering. In a black box, user knows the function but not the inner details. Structure chart, taking advantage of this very fact partitions the whole system into black boxes. A sample structure chart is shown in Fig. 4.6.

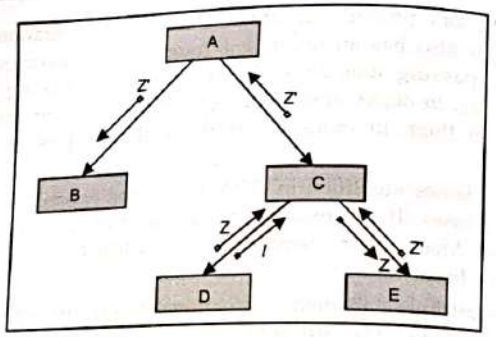


Fig. 4.6 A Sample Structure Chart

As shown in the diagram, a structure chart is composed of modules. Each mod can be characterized by input, output and a function. In the Fig. 4.6 rectangular boxes *B, C, D* and *E* are modules connected by long arrows. Each of this module has a spec function to perform. There is a single coordinating module at the root level. As i clear from diagram, module *A* can call modules *B* and *C* and module *C* can call modu *D* and *E*. The calling module is called ***superordinate*** module and module being calle ***subordinate*** module. Modules at the lowest level do not call any other modules.

Modules can be further classified into following four types:

(*a*) **Efferent Module** – This module receives data from superordinate module a after processing passes it to subordinate module. In other words this modul responsible for outputting the transformed data.

(*b*) **Transform Module** – This module receives data from superordinate module a after processing sends it back to superordinate module.

(*c*) **Coordinate Module** – This module coordinates the working of two or m modules. It receives data from one subordinate module and passes it to oth subordinate module. Middle level modules coordinate the lower level modu and also at the same time perform some processing as well.

(*d*) **Afferent Module** – This module receives data from subordinate module a after processing passes it to superordinate module. This module is there responsible for inputting the data.

Modules in a structure chart communicate with each other through exchange information shown by small arrows. Two types of information exchange takes pl First is the data couple which shows the actual data exchanged between modules and totally unavoidable. They are shown by arrows with dot at the end. It can be a sin data element or a group of data elements or even a record. Second type of informa is control information shown by arrows with a blob at the end.

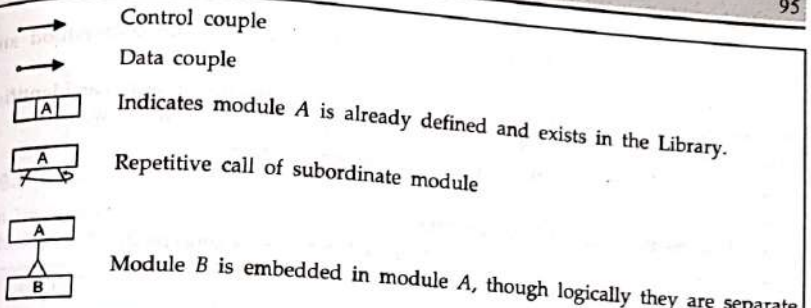


Fig. 4.7 Structure Chart Notation

A module A is said to be **visible** to module B if it directly or indirectly calls module B. Superordinate modules should not be visible to subordinate modules. Number of modules that can invoke a given module is called **Fan-in** of the module. High fan-in results into high reuse. Similarly the number of modules that can be invoked by a given module is called the **Fan-out** of that module. High fan-out indicates poor design. The notation for drawing structure chart is shown in Fig. 4.7.

4.6 CODING

The process of writing an algorithm using the correct syntax of a programming language like C, C++, JAVA etc., is called coding.

The coding starts after the detailed design document is ready and is only one small facet of the software development lifecycle. In chapter 5 several techniques like structured language, decision table, Nassi Shiederman diagram etc., are discussed to write the algorithm specifications. During coding these design specifications are converted to executable code/program. While writing algorithm using some notation following criteria must be followed:

- An algorithm must be brief and concise.
- The algorithm description must be unambiguous.
- The algorithm specification must be easily translated into executable code.

The steps used for developing a program are therefore:

(*a*) Understand the requirements. If it is a large size project, develop SRS.

(*b*) Produce the design document from SRS.

(*c*) For different modules identified in the design document. Write down the process specifications.

(*d*) Transform the process specifications into program code using suitable programming language.

(e) Compile and test the program.

(f) Document the program properly so that it can be easily understood and maintained.

Yourdan in his book "Techniques of Program Structure and Design" has identified Several qualities of a good program. Some of these qualities are listed below.

- A good program works according to the specifications.
- It is developed in time.
- It is flexible and easily modifiable.
- It is efficient as far as execution time and memory is concerned.
- It has minimum testing costs.
- It is easy to maintain.

In order to achieve above mentioned qualities coding standards or guidelines are followed by the software development organizations. These guidelines help the programmers to commit minimum mistakes and write a program which is easily understood, tested and maintained.

4.6.1 Coding Guidelines

Some important coding guidelines are:

1. Define the variable names as per programming language rules.
2. Use meaningful names for variables and the functions. They must indicate their role in the program.
3. In case programming language supports both explicit as well as implicit data definition, the programmer must go for explicit definition.
4. Avoid using go to statements.
5. Specify pre conditions and post conditions for every program and function.
6. Document the program properly. Use meaningful comments.
7. In order to make the program easy to read, use spaces, blank lines and indentation properly. If available use automated tools like prettyprinter to improve the appearance of the program.
8. Try to achieve the minimum coupling between different modules/functions using the concept of information hiding.
9. Aim for modules with high cohesion.
10. Try to write the code which can be reused.
11. Each module must document the following information:
    - Module name
    - Author name
    - Date of creation

- Function performed by the module
- Changes made to the module along with the date on which changes were made
- References to other modules if any
- Inputs required by the module
- Output produced by the module

4.6.2 Structured Programming

An unstructured program consists of lot of go to statements and results in a program which is difficult to understand, debug and maintain.

Structured programming is a technique which focuses on replacing go to statements with other structured branching and control statements.

The structured programming technique also makes use of top down program design. The statement in a program written using structured programming technique can be either a simple computational statement or a control statement. The control statement can be one of the following:

- Simple or nested If-then-else statement
- Subroutine/function call
- Looping constructs e.g., while loop, for loop etc.
- Case constructs

The benefits of using structured programming are:

- Programs are easy to understand and debug.
- Testing becomes easier which results in increased programmer productivity.
- Global optimization becomes much easier.

4.7 SOFTWARE DOCUMENTATION

Documentation is the most important characteristic of any software. It is a written text that accompanies computer software. It either explains how the software operates or how the users can use it and may mean different things to people in different roles. A well documented software is easy to understand and maintain. Documentation can be of following types:

4.7.1 Architecture or Design Documentation

This documentation gives the overview of software and its relation to external environment. The document talks very little about the code and also does not describe how to program a particular routine. Instead it focuses on different aspects of the system i.e., architecture, user interface, code, design etc., and suggests alternate approaches.

4.7.2 Technical Documentation

Technical documentation consists of documentation of code, algorithms, interfaces a APIs etc. as required by the programmers. When creating software if only code is writ it is difficult to understand and maintain it. Hence there must be some text along w it in the form of comments to describe various aspects of its intended operation. It also possible to auto-generate the documentation from the source code. Comments a be extracted from the source code and reference manuals can be created. Some popul tools to auto-generate the documentation are ***Twin Text, javadoc,*** ROBODoc a ***Doxygen.***

4.7.3 End User Documentation

End user documentation consist of manuals describing how to use the software and meant for the end-users, system administrators and the support staff. The us documentation provides the following information:

- Different features of the software
- Explains how to use the software
- Provides trouble shooting assistance

The user documentation can be organized using either a ***tutorial approach*** (illustrate stepwise description of each feature), ***thematic approach*** (different chapters concentrati on different areas of interest) or ***list approach*** (different commands listed and explain alphabetically via cross referenced indices). The tutorial approach is useful for new use whereas list approach is of great use to advanced users. Now a days independent author publish books in tutorial form to assist the software developers who are new to th software.

4.7.4 Marketing Documentation

Whenever a new software is introduced in the market it must be promoted. Marketi documentation therefore talks about brief description of the product and other promotion material. This type of documentation creates an interest in the user about the product b informing what exactly the software does and how it meets their expectations. Th documentation also compares this product with other products of similar kind in the marke

One must not forget that in the past projects failed resulting into huge financial los due to lack of proper documentation. Now a days organizations are also employi ***technical writers*** to write different types of documentation.

SUMMARY

- *Software design* is an iterative process during which the software requirement specified in SRS are analyzed and converted into description of the internal structu and organization of the system called the ***software design specification docume***

- Design document is a very important document used by programmers, testers and maintenance people of the project.
- A good design reduces the maintenance cost of the software.
- Software is best designed and maintained as a set of small easily handled components called ***modules.***
- The module independence depends on two properties called ***coupling*** and ***cohesion.*** The coupling is the degree of interdependence between two modules, cohesion is the measure of strength of elements of an individual module and is property of single module. While designing focus must be to minimize the coupling between the modules and to increase the cohesion within a module.
- Different design approaches are level oriented design, data flow oriented design and object oriented design.
- A popular notation used for representing the design is structure chart.
- The process of writing an algorithm using the correct syntax of a programming language like C, C++, JAVA etc. is called coding.
- Documentation is written text that accompanies computer software. Different types of documentation are design documentation, technical documentation, user documentation and marketing documentation.

REVIEW PROBLEMS

1. What do you mean by software design? What are the characteristics of a good design?
2. What is the difference between coupling and cohesion? Explain with suitable example.
3. Explain different types of coupling and cohesion by taking suitable example.
4. What is a structure chart? How it is different from a flow chart?
5. Compare the data flow oriented design with object oriented design.
6. What will happen if you increase the cohesion of different modules in your design?
7. Explain different approaches to software design.
8. How will you improve the software design with high coupling?
9. Why documentation is important? Explain different types of documentation.
10. Go through the literature and identify some popular design patterns used in OOD.
11. What are the important features of bad design? What principles are used to convert a bad design into a good design.
12. Write down the important components of IEEE standard 1016 used for writing design description.
13. Write the design documentation for your class room project on which you are working using the standard IEEE 1016.

○○○

Chapter 5

STRUCTURED ANALYSIS AND DESIGN

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- Why design is important?
- Different views of modeling
- Structured analysis techniques
- Static view using ER diagram
- Functional view using Data Flow Diagram
- Dynamic view using State Transition Diagram
- Structured design: transforming a DFD into a structure chart
- Writing the process specifications and data dictionary

5.1 INTRODUCTION

In order to document the requirements of the problem domain requirements must be elicited, analyzed, specified and verified as discussed in chapter 3. During each of these activities models may be constructed. Several methodologies have been proposed in the literature to represent the requirement of application domain. The models developed using these methodologies help the different team members of the project to understand different aspects of the software under development and hence often are called the ***software blueprints***. The advantages of developing these models or blueprints are:

- Better understanding of the problem domain.

- Smooth transition to the working software.
- Provide guidance in size and effort estimation.
- Help in understanding the performance issue.
- Better maintenance of the final product.

A model of a system describes a specific aspect of the system under consideration and is also called a ***conceptual model***. The activity of developing this conceptual model is called conceptual modeling.

According to (Mylopoulos92) the process of conceptual modeling is the activity of formally defining aspects of the physical and social would around us for the purposes of understanding and communication.

Main principles used in developing models are abstraction mechanisms and decomposition. Abstraction is the process of identifying and representing some particular details of the modeled domain. Some important abstraction mechanisms used during conceptual modeling are:

- Classification
- Aggregation
- Generalization
- Association

Classification is used to relate instances to the types and results into grouping of objects which play same role e.g. the entity FACULTY refers to all persons that share the role of teaching in departments. ***Associations*** are used to establish links between entity classes e.g., faculty **teaches** students. ***Aggregation*** is an abstraction mechanism which allows to build new objects out of existing objects through part-whole or is composed of relationship e.g., a car is composed of wheels, engine, chassis etc. It is to be noted that aggregation is transitive and and symmetric.

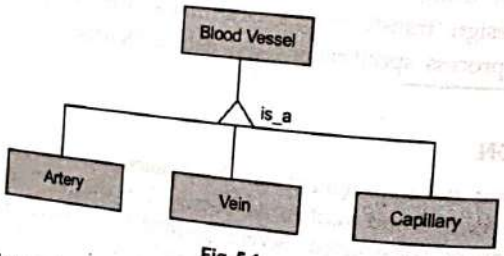


Fig. 5.1

Generalization is used to identify a generic object out of existing one or more objects by capturing the similarities between these objects. The object at generic level is super type and at lower level is subtype. The subtype objects are related to supertype object through is_a relationship as shown in Fig. 5.1. Decomposition allows decomposing a complex system into multiple sub-systems which are easy to handle.

5.2 DIFFERENT VIEWS OF MODELING

Whenever we are working on a problem, in order to understand the problem properly, we must look at the problem from three different viewpoints and develop the models to represent these views. The different viewpoints are:

- Static or structural viewpoint
- Functional viewpoint
- Dynamic viewpoint

Static view of the problem domain focuses on the main information structures and the relationship between these structures. This view therefore describes the 'what' part of the problem domain. The models used to develop the static view are often termed as semantic data models. These models emerged as a result of two properties data independence and abstraction. Data independence says that model should be independent of implementation details whereas abstraction provides techniques for representing the system details in a organized manner. A popular semantic data model used for modeling the static view is Entity Relationship (ER) model (Chen76). ER model is discussed in detail in section 5.2.

Functional view describes the functionality of the system in terms of the main processes of the business and the flow of data among these processes. This view therefore describes the 'how' part of the problem domain. ***Dynamic view*** shows the time dependent behavior of the system i.e., how the system responds to the external environment. Functional view and dynamic view together also called the ***behavioral view*** of the problem domain i.e., they focus on the how part of the application domain *i.e.*, activities operating on the information structures (identified using static view) and events that trigger these activities. Events may be external to the system or may be generated internally by the system *e.g.*, placing of order by the customer is an example of event taking place in the system.

Several models exist in the literature to support these view points. A popular technique is ***structured analysis*** technique to model these views of the problem domain. Modeling of complex objects in areas like CAD/CAM and Computer Aided Software Engineering (CASE) has given rise to ***object oriented paradigm***. In object oriented paradigm main unit of design is object which encapsulates the structural and behavioral aspect in one cohesive unit i.e., the object itself. In the following sections we will discuss the structured analysis and design techniques in details. Object oriented analysis techniques will be discussed in detail in chapter 6.

5.3 STRUCTURED ANALYSIS

Structured analysis techniques result in graphical representation of the software system under development. These techniques are used to model the static, functional as well as dynamic view of the problem domain. The main artifacts produced as a result of using these techniques are:

- Entity relationship diagram
- Function decomposition diagram
- Data flow diagram
- State transition diagram
- Data dictionary
- Process specifications
- Context diagram
- Decision trees

Next we will discuss each of these in detail.

5.3.1 Entity Relationship Model

Entity Relationship (ER) model is one of the most popular semantic data model propo by Chen in seventies (Chen76). It is still the most important data model used in relatio data base design. During requirements analysis stage of software life-cycle, it is used model the static view of the application domain i.e., it models the important informati structures and the relationship between these structures. For example in an applicat related to hospital domain we will be interested to store information about doct patient, ward, nurse, tests etc., and the relationship between these. In other words data model models all the data in an application that will be input, output, transform and stored. Data models also aid in physical storage of data in different formats. T most popular format is relational databases. Concepts used in ER model are:

- Entity
- Relationship
- Attribute
- Primary key
- Role
- Cardinality
- Functionality of the attribute
- Degree of a relationship
- Generalization.

Now we discuss each of these briefly.

Entity

Entity is something about which the organization wants to store data. It can be a place, organizat person, object, event or even a concept. Each entity is unique in the sense that it can b distinguished from the other entity. While reading the requirements textually, entiti are identified as nouns. Examples of valid entities are Customer, Hospital, Student, Cours Account, Sale, Machine etc. Sometimes terms like entity instances and entity types a also used in literature. Entity type can be defined as a collection of entities having s

properties. An entity instance on the other hand is a single instance of entity type. Corresponding to one entity type, thousands of entity instances are stored in data base. In an ER diagram an entity type is represented by a rectangle with the name of the entity written inside the rectangle as shown in Fig. 5.2.

Fig. 5.2 Entity Symbol

Relationship

Relationships show, how instances of entity types are connected to each other. While going through the textual requirements of the application, they are represented by verbs. For example, in the requirement: "Student enrolls for a course" enrolls is the relationship which associates the entities student and course. Graphically it is represented by a diamond sign with name of relationship written inside and connected by lines to the entities as shown in Fig. 5.3.

Fig. 5.3 A Relationship

There can exist multiple relationships between two entities.

Cardinality

Number of instances of entities participating in the relationship is called cardinality of the relationship. Cardinality can be shown by various ways in the ER diagram. One way to represent cardinality is shown in Fig. 5.4.

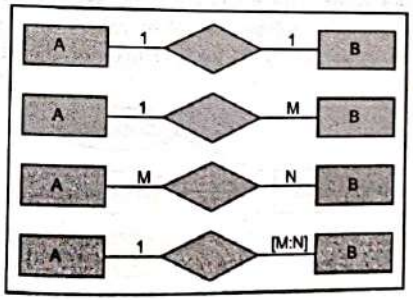


Fig. 5.4 Cardinality Notation

Another notation for representing cardinality is shown in Fig. 5.5.

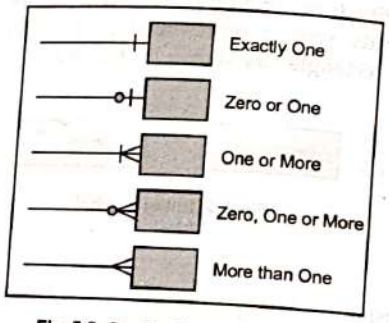


Fig. 5.5 Cardinality Notation

Consider the ER diagram shown in Fig. 5.6. It is read (left to right) as a department can hire minimum one and maximum *N* faculty members. If read right to left, it means that a faculty can be hired by at most one department.

Fig. 5.6 An Example to Illustrate Cardinality

Attribute

Attributes are the qualifying properties or characteristics of the entities. For example, entity department can have attributes like department-number, department-name, location etc. In the ER diagram attribute is represented by placing its name in an ellipse with a line connecting it to corresponding entity as shown in Fig. 5.7.

As attributes represent some data, values can be assigned to an attribute from a domain. A domain of the attribute defines the set of values or range of values which an attribute can take. Default value also must be specified for the attributes if the value is not specified by the user.

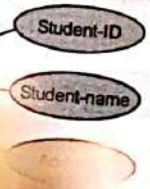


Fig. 5.7

Attributes can be classified as atomic and complex attributes. Complex attributes are ones which are composed of other simple or complex attributes. For example attribute student-name can be composed of attributes firstname, middle name and last name. They are represented graphically as shown in Fig. 5.8.

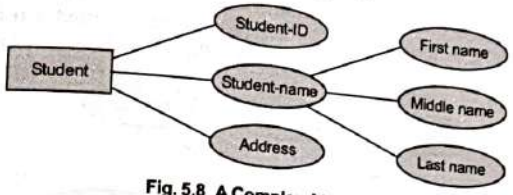


Fig. 5.8 A Complex Attribute

Atomic attributes on the other hand are the ones which cannot be decomposed *e.g.*, age. Just like entities, relationships can also have attributes *e.g.*, many students enrolling in several courses get grades in these courses which can be represented as attribute as shown in Fig. 5.8*a*.

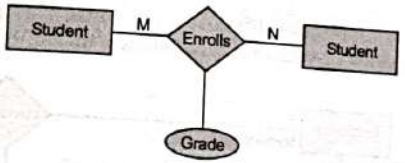


Fig. 5.8a Attributes of Relationship

Primary Key

Primary key is an attribute or set of attributes, which uniquely identifies different instances of an entity type. For example, attribute student-id can be used to identify different instances of entity type student and thus is the primary key of entity type employee. Primary key is represented by underlining the attribute as shown in Fig. 5.9. It is mandatory to assign some value to primary key. The value of a primary key can not be NULL.

A primary key if used in another entity to identify instances of a relationship is called a foreign key and matches the primary key. A foreign key results into referential integrity constraints.

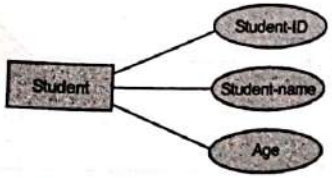


Fig. 5.9 A Primary Key

Functionality of the Attribute

Functionality of an attribute specifies whether the attribute is single-valued or multivalued. Single-valued attributes can have only one value whereas multivalued attributes can have multiple values. For example, attribute degree of student entity type can have value B.E., M.E. MBA and is therefore a multivalued attribute. On the other hand the attribute age can take only one value say 30 or 40 and is a type of single valued attribute. Graphically multivalued attributes are shown by double-lined ellipse as shown in Fig. 5.10.

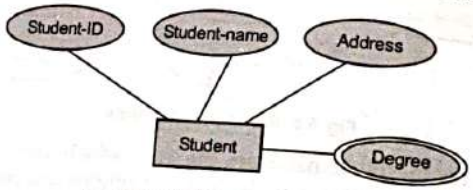


Fig. 5.10 Multi Valued Attribute

Role

In the relationship, entity type plays different roles. Roles are specified by writing the role names on different sides of relationship as shown in Fig. 5.11.

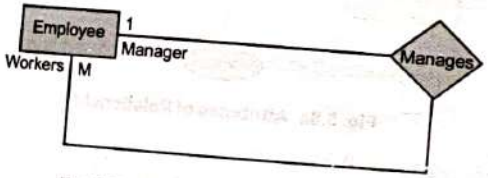


Fig. 5.11 Roles Played by Entities

In this case we see that employee plays the role of manager and worker in the relationship manages.

Degree of Relationship

Number of entity types participating in a relationship is called degree of that relationship. Based on degree, relationship can be classified as:

(a) *Unary Relationship.* In this instances of only one entity type participate in the relationship as shown in Fig. 5.12.

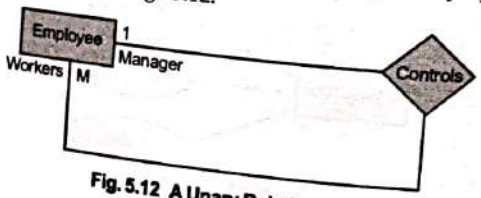


Fig. 5.12 A Unary Relationship

Here one instance of employee playing the role of manager is associated to M instances of entity employee playing the role of worker. It is also called recursive relationship.

(b) *Binary Relationship.* It is the most common type of relationship and associates instances of two entity types as shown in Fig. 5.13.

Fig. 5.13 Binary Relationship

If read left to right, it says that department hires M number of employees. In opposite direction, it can be read as: each employee is hired by at the most one department.

(c) *Ternary Relationship.* A ternary relationship relates instances of three entity types as shown in Fig. 5.14.

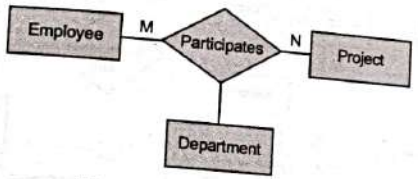


Fig. 5.14 Ternary Relationship

The relationship shown in Fig. 5.14 tracks the participation of a particular employee of a department in a project.

Generalization

Generalization is a technique which helps in identification of common features between entities and grouping these features into a new entity called **supertype.** The entity which inherits characteristics from supertype entity is called subtype. Additionally a subtype entity has certain attributes which are specific or unique to that subtype. As an example consider a generalization hierarchy as shown in Fig. 5.15.

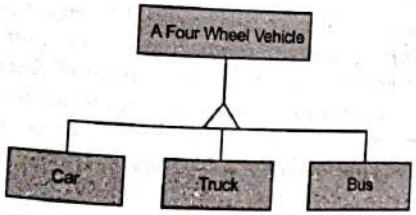


Fig. 5.15 A Generalization Hierarchy

In this generalization hierarchy the entities car, truck and bus are four wheel vehicles so they share some common attributes. Additionally they have some attributes (e.g. attribute route-number in entity bus) which are unique to them.

A sample ER diagram is as shown in Fig. 5.16.

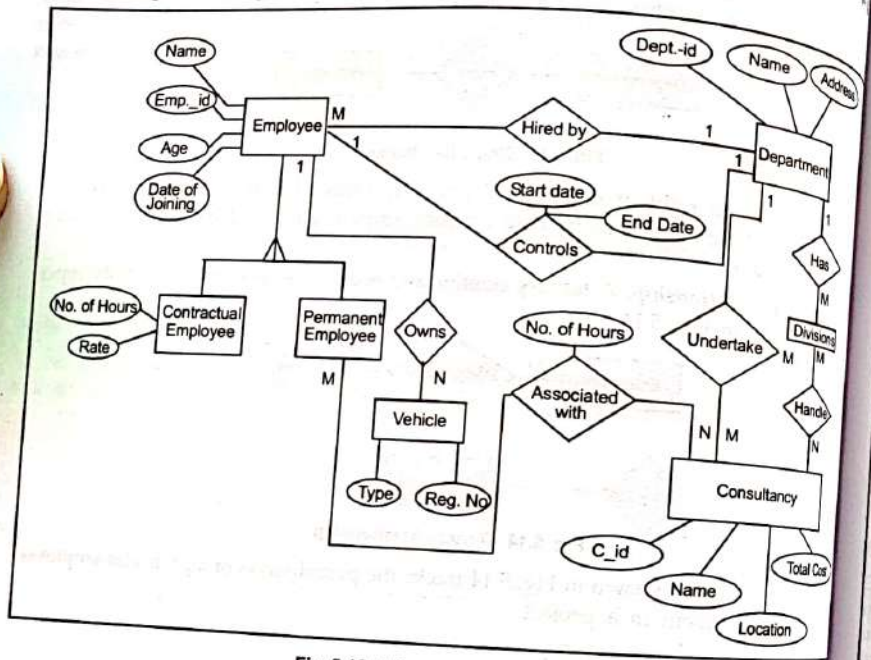


Fig. 5.16 A Sample ER Diagram

Example 5.1 *Art galleries all over India keep information about various artists enrolled with the galleries. i.e., name, age, place of birth, art style. Each artist is assigned a unique id. Galleries also keep information for each art piece i.e., name of artist, the year it was made, title (which is unique), type of art and price. From time to time exhibitions are also organized by the galleries to display work of a artist. Additionally different art pieces are also classified into groups of various kinds e.g. portraits, spiritual work, modern art etc. A given art piece may belong to more than one art group. Every year number of customers visit the gallery. Galleries also record information about customers i.e., name, address, liking for artist and groups of art and total money spend by him. Customers may be members or non-members of gallery. Members are also offered discounts by the gallery. Develop the ER diagram for the same.*

Solution. *In this problem if we go through the text, the entities of interest are– Gallery, Exhibition, Discount, Artist, Customer, Art piece and Art group. Attributes which highlight the properties of entities are artist-id, name, age, place of birth, art style, year-made, title etc. Relationships are the verbs e.g., enroll, visit etc. Readers are advised to go through the problem, and identify the entities, attributes and relationships and understand the solution given in Fig. 5.17.*

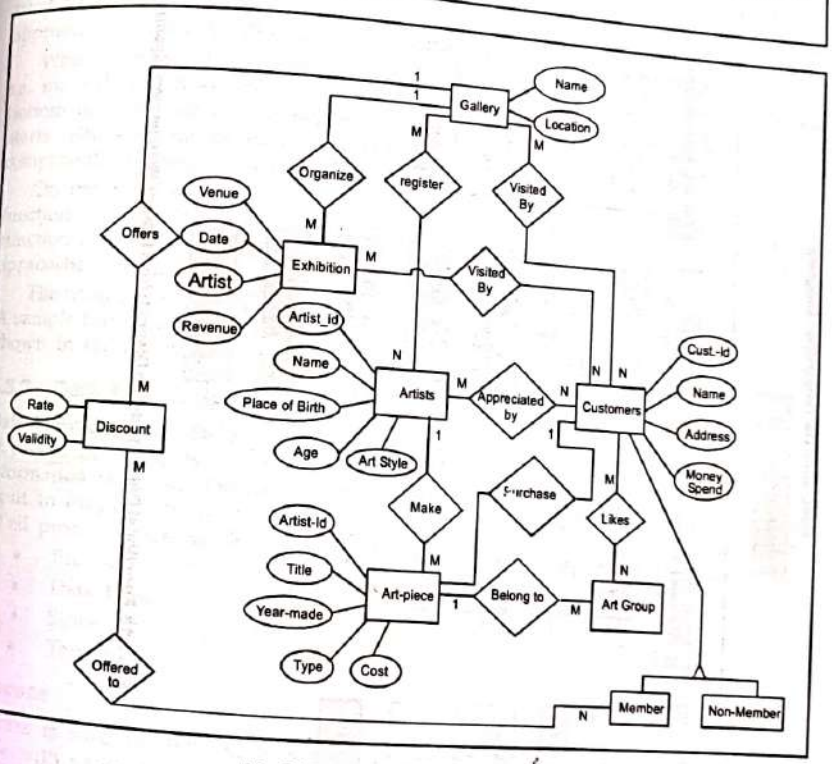


Fig. 5.17 ER Diagram for Example 5.1

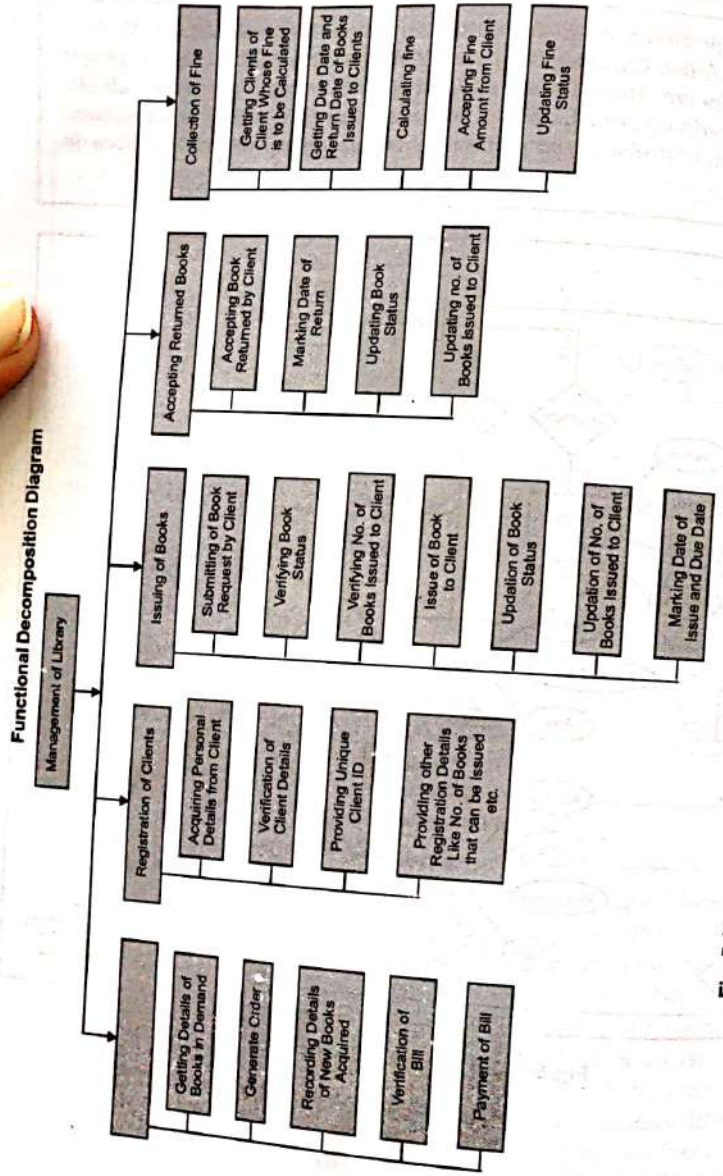


Fig. 5.18 Functional Decomposition Diagram for a Library Management System

5.3.2 Function Decomposition Diagram (FDD)

Function decomposition diagram is an important diagram for modeling the functionality or functional view of the system. It is the simplest and a useful technique for producing a hierarchy of functions where each function is described by a simple unambiguous sentence starting with a verb. In other words it is similar to a family tree. Each parent function is explained in detail by one or more child functions/activities. The function at root level corresponds to the entire system. The function decomposition diagram can be used for identification of processes while drawing data flow diagram using classical approach (as would be discussed in section 5.4).

While drawing FDD analyst must take into account all the activities of the business *i.e.*, manual as well as automated activities. Two approaches *i.e.*, top-down approach and bottom up approach can be followed while drawing FDD. In top down approach one starts with a single sentence representing functioning of complete business and then components of these (each one a business function) is produced at different levels.

On the other hand in the bottom up approach analyst compiles a list of business functions from various sources *e.g.*, manuals, forms, interviews etc., and tries to fit each function at a convenient place in the topdown hierarchy. In actual scenario both the approaches are used.

The main purpose of *FDD* is to gain proper understanding of functioning of a business. A sample function decomposition diagram for a library is shown in Fig. 5.18. The diagram shown in the figure is not complete. The readers are advised to complete the diagram.

5.3.3 Data Flow Diagram

Data Flow Diagram (DFD) is used to model the functional view of the system in terms of processes and flow of data between these processes. In other words it shows the information flow and the transformations applied to the information as it flows from input to output. The technique for modeling flow of data between processes is also called process modeling. Important concepts used in drawing data flow diagrams are:

- Process
- Data Flow
- Store
- Terminator

Process

Process is used to show some kind of transformation on data. It is represented by a circle with name of process and number as shown in Fig. 5.18*a*.

Fig. 5.18*a* A Process Icon

Process names should be meaningful verb object phrases not nouns. For example we use the process name salary instead of compute salary in Fig. 5.18*a*, it will not show any kind of transformation or processing.

Data Flow

Data flows show data in motion between different processes, process and store or external agent or process. They are represented by arrows which are labeled as shown in Fig. 5.19. Data flow labels are normally entity or attributes in ER diagram i.e., the static view.

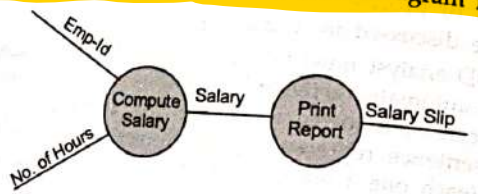


Fig. 5.19 Data Flows

A data flow therefore represents

- data input to a process
- output from a process
- insertion of new data in the store or retrieval of existing data
- updating existing data in the store
- deletion of existing data from the store

A data flow can be further classified as convergent and divergent data flows. A convergent data flow is formed by merging multiple data flows into a single data flow as shown in Fig. 5.20*a*.

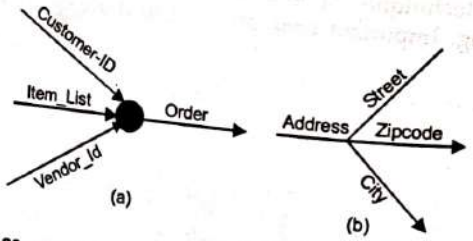


Fig. 5.20 (a) A Converging Data Flow (b) A Diverging Data Flow

A diverging data flow is one which breaks up into multiple data flows as shown in Fig. 5.20(*b*). In case of converging data flows, different data flows from various sources must join to form a single unit of data for further processing.

Store

Store represents data at rest and can be shown graphically by the symbols as shown in Fig. 5.21. In simple language a store represents those things whose information, the business wants to store e.g., employees, departments, order etc. Normally stores correspond to entities in the ER diagram. They are named as plural of corresponding data model entities. It means that if in ER diagram an entity student exits, there must be a store called students in the *DFD* to ensure that both *ER* diagram and *DFD* are consistent. At the time of coding stores are implemented as database or files.

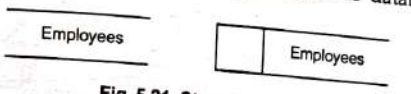


Fig. 5.21 Store Symbols

External Agent

External agents are also called terminators and represent people, organization or other systems external to the system being developed. The system will interact with external agents through exchange of information. They are represented graphically by a rectangular box as shown in Fig. 5.22. External agents provide input to the system and also receive output from the system.

Fig. 5.22 Symbol for Terminator or External Agent

A sample data flow diagram is as shown in the Fig. 5.23. As it is seen from the diagram, data flow diagram shows the flow of data among processes but it does not show the order in which different data arrive at a process.

5.3.3.1 Context Diagram

A context diagram is also called a level 0 data flow diagram. It is a diagram in which working of the whole organization is represented by a single process and its interaction with external agents is shown through exchange of data. These external agents can be some people, organization, some other system or even some other software. In a way this diagram defines the boundary of the system being developed and shows how a system interacts with the external world.

A sample context diagram for examination system is shown in Fig. 5.24.

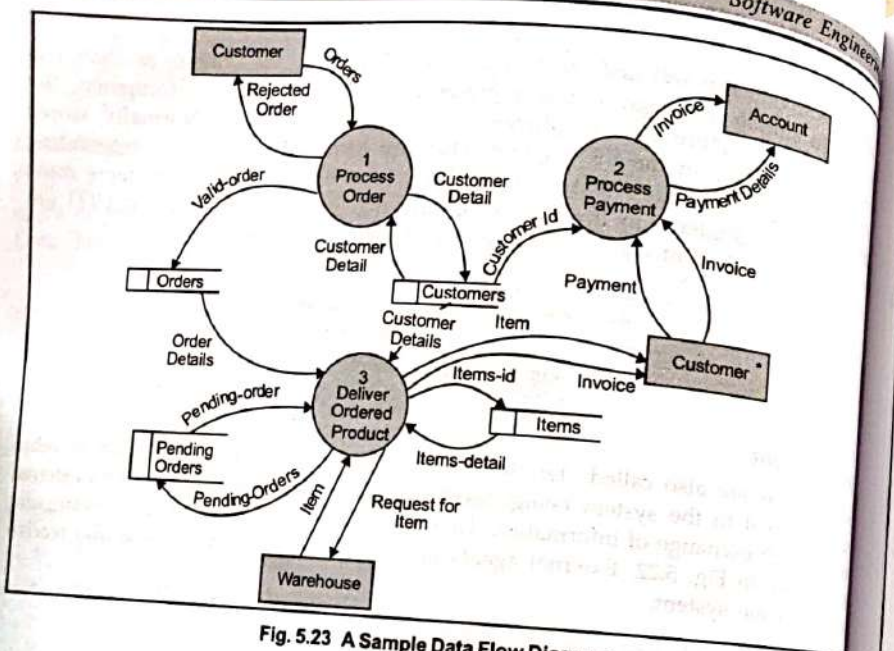


Fig. 5.23 A Sample Data Flow Diagram

5.3.3.2 Leveled Data Flow Diagrams

If we are working on a larger system, it is seen that number of processes may be as high as one hundred. It is not possible to represent these 100 processes on single sheet of paper. Hence instead of drawing one DFD, a leveled DFD is drawn. In this, we start with level 0 DFD in which a single process represents the working of complete system. Level 0 DFD or context diagram is decomposed into level 1 DFD consisting of main processes required to support the working of system being developed. At the next level, *i.e.*, level 2 each of these processes can be further decomposed to show more details as illustrated in Fig. 5.25.

Here main process library at level 0 is decomposed into level one DFD consisting of three processes of *P*1, *P*2 and *P*3. Similarly *P*1 is decomposed into three processes *P*11, *P*12 and *P*13 at level 2. These processes are numbered as 1.1, 1.2 and 1.3 indicating that parent process is process number 1 at level one. In the similar manner processes *P*2 and *P*3 can also be decomposed. This process continues till more decomposition is possible. According to heuristics, one should go maximum up to 4 or 5 levels. Also while going from one level to another, one should see that inputs and output of a process should match. There should be no extra or less inputs or outputs. The main advantage of using leveled DFD is that functionality of the system is represented in a systematic way which is much easier to understand and read.

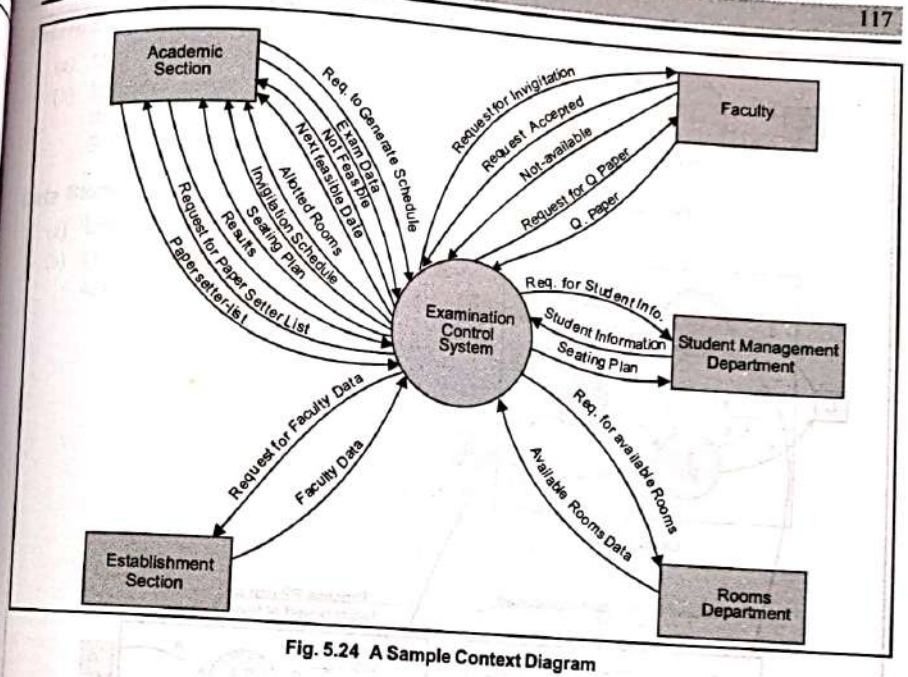


Fig. 5.24 A Sample Context Diagram

5.3.3.3 Guidelines for Drawing DFD

A system analyst or developer should follow the following guidelines while drawing the DFD:

Process

(*a*) All the processes must be numbered.
(*b*) Use meaningful verb-object phrases for process names *e.g.*, compute salary, generate reports. Avoid nouns as process names.
(*c*) Avoid processes with only inputs. They are called sinks.
(*d*) Avoid processes with only outputs.
(*e*) If necessary redraw the DFD as many times as required.

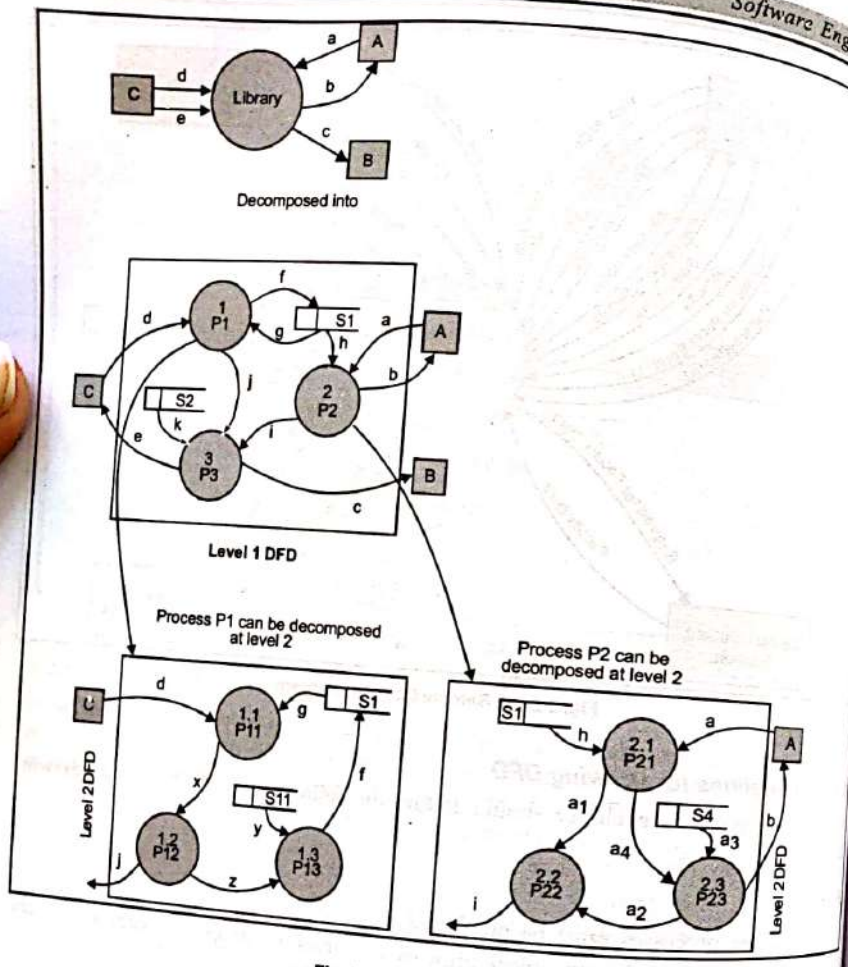


Fig. 5.25 A Leveled DFD

Data Flow

(*a*) Make sure that all data flows are labeled.

(*b*) Data flows cannot move from a store to another store or to an external agent. It must move through a process.

(*c*) A data flow does not move back to the same process it leaves.

External Agent

(*a*) Use noun phrases to label external agents.

(*b*) External agents cannot interact directly with a store or any other external agent through data flows. Data must be moved through a process as shown in Fig. 5.26 and 5.27.

Data Store

(*a*) Use meaningful noun phrases to label data stores.

(*b*) Data cannot move directly from data stores to other data stores or external agents. It must move through a process.

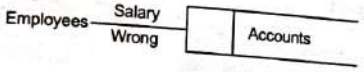


Fig. 5.26

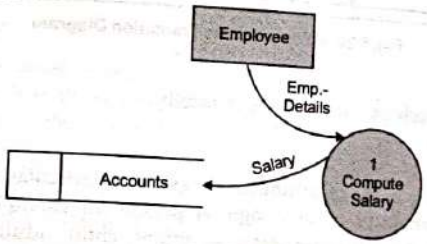


Fig. 5.27

5.3.4 State Transition Diagram (STD)

State Transition Diagram (STD) models the dynamic view *i.e.*, time dependent behavior of the system. In the past it was used only for special category of systems *i.e.*, real time systems, process control systems, space shuttle programs etc. But in today's scenario this diagram is also used for modeling some part of business systems *e.g.*, user interface. Major components of the diagram are:

- State
- Action
- Arrow
- Condition.

A sample STD for an ATM machine would be as shown in Fig. 5.28.

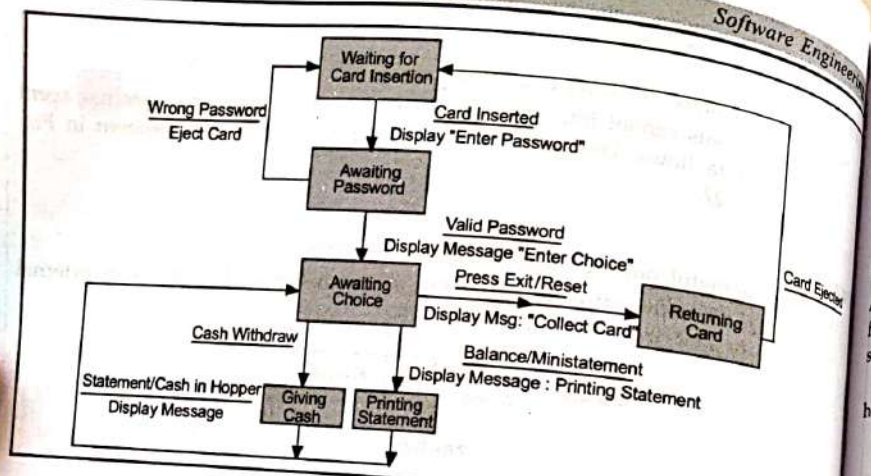


Fig. 5.28 A Sample State Transition Diagram

Next we discuss each of these concepts briefly.

State

A state is described by a set of attribute values at a particular instant of time and is represented by a rectangle or an oval sign. A person for example depending upon his age can be in one of the following states — infant, child, adult, middle-age man, old man.

Fig. 5.29 Symbols for State

State can be either initial/start state, end/final state or in between state. In general a system can have only one initial state and single or multiple final states.

Action

Whenever system changes states in response to a condition, it performs one or more actions. It is also possible that it may not perform any action. Some of the actions can be displaying a message, running the electric motor, generating some output etc.

Arrows

Arrows connect two or more states indicating that state S1 changes to state S2 as a result of some condition C being satisfied.

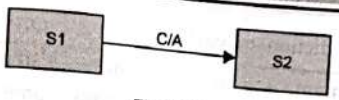


Fig. 5.30

Condition

A condition is some event in the external environment which causes the system to change from say state S1 to S2. This event can be anything, say an interrupt, signal or arrival of some data.

In the diagram shown in Fig. 5.31 *e.g.*, man changes state from child to adult when he attains age of 18 years.

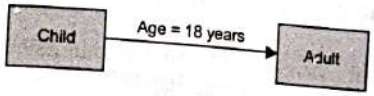


Fig. 5.31

5.3.5 Data Dictionary

Data dictionary is the most important part of structured analysis model. As far as definition is concerned, it is the organized listing of all data elements of the system with their precise and unambiguous definitions. Data dictionary contains information about

- Meaning of aggregate item with comments
- Units of elementary items
- Definition of data/control flows
- Definition of data stores
- Definition of entities, relationship, attributes, external agents
- Definition of external control/data flows
- Local data elements used in writing process specifications and many more.

A number of schemes are proposed to represent the details in data dictionary. One such popular scheme is given in Fig 5.32.

=	is defined as/composed of
+	and
{ }	iteration (0 or more occurrences)
min{ }max	iteration with min & max values specified
( )	optional data elements
[ ]	selection of one data from several choices separated by |
@	Store identifier
**	Comment.

Fig. 5.32 Notation for Writing Data Dictionary

Some examples of data dictionary entry are:

(a) **Order = Company_name + address + 1 {ordered_item} 10**

It is read as: Order is composed of Company_name, address and minimum one and maximum ten ordered_item.

(b) **Student = [Undergraduate / Postgraduate]**

It is read as: Student is an Undergraduate or Postgraduate student.

(c) **Student_name = first_name + (middle_name) + last_name**

It is read as student_name is composed of first_name, middle_name (which is optional) and last_name.

Additionally details about elementary data items can be described as shown below:

height = * student's height at the time of admission to defense academy
units : centimetres; range 160–190*

Building the data-dictionary for any project is a time-consuming job but it is important as it results into a consistent and precise system.

5.3.6 Process Specifications

In a DFD processes are shown as black boxes. Nothing is clear about the detailed working of these processes *i.e.*, how they work and what algorithm etc., is used to achieve the functionality.

Logic modeling or process specification is used to give the description of logic contained in a process at the lowest level of decomposition. Higher level processes which are decomposed into detailed DFD's, do not have process specification. Process specification can be written using

- Structured Language
- Pre/Post Conditions
- Decision Tables
- Decision Trees
- Flow Charts
- Nassi-Sheiderman Diagrams etc.

5.3.6.1 Structured Language

As name suggests it is language with a structure. In other words, it is a subset of English language that supports language constructs to represent sequence, repetition and conditional statements in order to specify the process details. It can therefore be called structured English. Alternate names for structured English are ***Program Design Language*** (PDL) and ***Program Statement/Specification Language*** (PSL).

A sentence in the structured English can be a simple verb-object phrase (*e.g.*, Add 7 to total-items) or it can take the form of an algebraic equation as shown below

Z = (X * Y)/(10 + no._of hours)

Verbs are normally chosen from a small set of action oriented verbs and vary from organization to organization. Some of the examples are ADD, DELETE, GET, READ, UPDATE, SORT, VALIDATE etc.

Objects in these verb-object phrases are the data elements already defined in the data-dictionary. Additionally they can be data elements local to a particular process used for holding intermediate results.

Some of the important constructs are:

(a)
```
IF     condition
       sentence-1
ELSE
       sentence-2
ENDIF
```

(b)
```
DO WHILE   condition
       sentence
END DO
```

(c)
```
REPEAT
       sentence
UNTIL condition
```

(d)
```
DO CASE
     case variable = value1
                sentence-1
     case variable = value2
                sentence2

     case variable = value n
                sentence n
          DEFAULT
                sentence_n+1
END CASE
```

A sample process specification using structured English is shown in Fig. 5.33.

```
Gen-Category = 0
DO WHILE there are more applications to process
    READ next application from APPLICATIONS
    if stu-category = 'General'
        Gen-Category = Gen-Category + 1
END DO
DISPLAY Gen-Category
```

Fig. 5.33 Structured English Example

5.3.6.2 Decision Table

When the process logic is very complicated, involving multiple conditions, it is not possible to represent the process logic efficiently with structured English. In such type of scenario, decision tables are used to represent the logic.

	Rule 1	Rule 2	Rule 3	Rule *n*
Condition 1	✓	✓	✓	✓
Condition 2	✓			
Condition 3		✓	✓	
Condition *n*			✓	✓
Action 1	✗			
Action 2		✗		
			✗	✗
Action *n*		✗		

Fig. 5.34 Decision Table

Main parts of a Decision table are:

1. Condition Stubs
2. Action Stubs
3. Rules

Condition stubs list all the conditions relevant to the decision. Action stubs part of table lists all the possible actions that will take place for a valid set of conditions. Rules part of the table specifies which set of conditions will trigger which action. Structure of a decision table is shown in Fig. 5.34. Steps followed to construct decision tables are:

1. List all conditions and the values conditions can have.

2. List all possible actions.
3. List all possible rules in terms of combination of conditions.
4. For each rule define the action.

Next we illustrate it with the help of an example.

Example 5.3 An airlines offers attractive discounts to its customers based on the flights taken by them per year. Customers/Passengers are broadly classified into economy class, business class and executive class passengers. For each of these class, normal air fares for that class are charged. For each flight passenger also earns some points. In case a passenger takes flights thrice a year, airlines offers 10% discount on the original air-fare on the purchase of tickets in rest of the year. In case a business class passenger has earned 400 points, he is offered 30% discount on the tickets in rest of the year. In case passenger belonging to any of three category earns 1000 points, he is offered a free ticket for any destination in the world. For executive class passengers airlines offers special schemes. In case executive class passenger makes 1500 points airlines offers free holiday package for two abroad. Draw the decision table.

Solution. In this problem, the first step is identification of conditions and actions.

The conditions are

C_1 — Passenger from any class
C_2 — Flights taken > 3 per year
C_3 — Business class passenger
C_4 — Executive class passenger
C_5 — Flights taken a year ≤ 3
C_6 — Points earned ≥ 400
C_7 — Points earned ≥ 1000
C_8 — Points earned ≥ 1500

The actions are

A_1 — Charge standard fares
A_2 — Offer 10% discount
A_3 — Offer 30% discount
A_4 — Offer a free ticket
A_5 — Offer a free holiday package for two

The decision table is shown in the Fig. 5.35.

Conditions/Actions	Rules				
	1	2	3	4	5
C_1 — Passenger from any class	✓	✓			
C_2 — Flights taken > 3 per year		✓		✓	
C_3 — Business class passenger					
C_4 — Executive class passenger			✓		
C_5 — Flights taken per year ≤ 3	✓				
C_6 — Points earned ≥ 400					✓
C_7 — Points earned ≥ 1000				✓	
C_8 — Points earned ≥ 1500					
A_1 — Charge standard fares	✗	✗	✗		
A_2 — Offer 10% discount		✗			✓
A_3 — Offer 30% discount					
A_4 — Offer a free ticket			✗		
A_5 — Offer a free holiday package				✗	

Fig. 5.35 Decision Table

5.3.6.3 Decision Tree

A decision tree serves the same purpose as a decision table. But for some people it is much easier to read. A decision tree for example 5.3 is shown in Fig. 5.36.

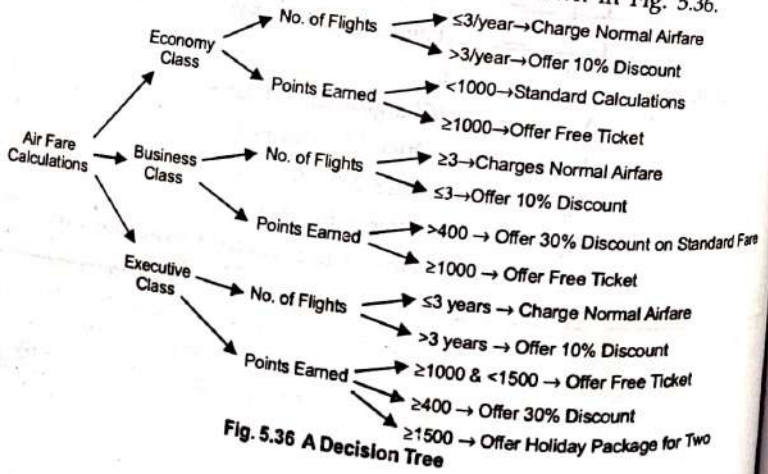


Fig. 5.36 A Decision Tree

5.3.6.4 Nassi-Scheidermann Charts

These charts were developed by Nassi and Scheidermann to overcome the problems of logic flow charting techniques. The symbols used to draw the charts are shown in Fig. 5.37.

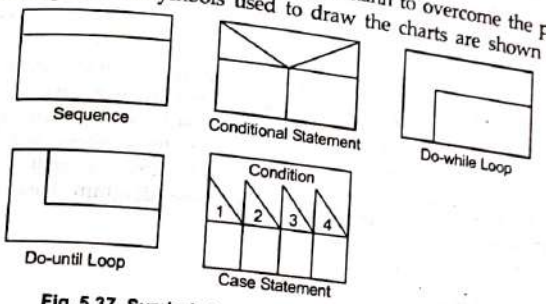


Fig. 5.37 Symbols to draw Nassi Scheidermann Chart

Rules for using the Nassi-Scheidermann charts are:

(a) Charts always start from the top and are rectangular in form.

(b) Movement in the chart while reading is always from top to bottom. Only at the end of the loop if exit condition is not satisfied, we again begin from starting of the same loop.

A sample of Nassi-Scheidermann chart is shown in Fig. 5.38.

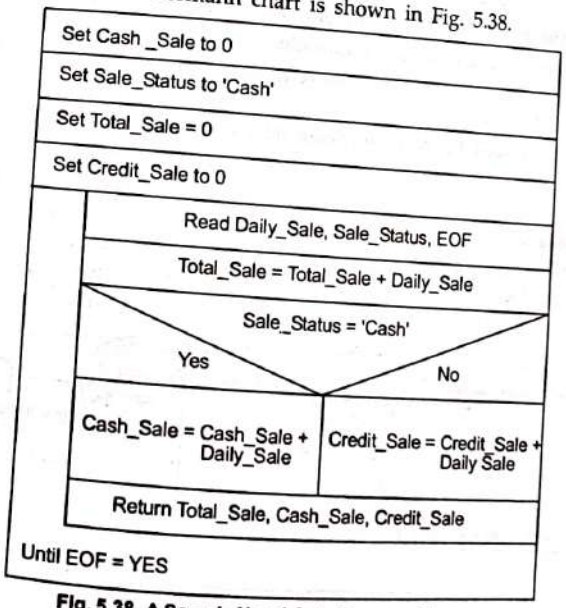


Fig. 5.38 A Sample Nassi-Scheidermann Diagram

5.4 STRUCTURED DESIGN: DERIVING STRUCTURE CHART FROM STRUCTURED ANALYSIS MODELS

The conversion of structured analysis model (Data flow diagram, Control flow diagram) into structure chart can be done using two techniques called "transform analysis" and "transaction analysis". The names of the two techniques are basically taken from the two types of information systems *viz.*, transaction centre systems and transform centre systems. According to (*Jeffrey00*) a transaction centered system focuses on dispatch of data to their appropriate locations for processing whereas transform centered system focuses on derivation of new information from existing data. It is these properties of the two types of system which are used in transaction analysis and transform analysis respectively. For both of these techniques, the starting point is data flow diagram. Next we describe each of these techniques.

5.5.1 Transform Analysis

Transform analysis is a structured process which converts a transform centered DFD into a structure chart. In transform centered DFD, there is an area of transformation or computation (consisting of one or more processes) where one or multiple data flows converge. The data after transformation goes along one or more paths. One can start with level-1 DFD or detailed DFD (if more details are required). Once DFD is selected following steps are followed:

(*a*) Identify the central transform.

(*b*) Identify the top level input (afferent) and output (efferent) module.

(*c*) Identify the main coordinating module.

(*d*) If required elaborate afferent, efferent and transform modules further *i.e.*, do the detailed design.

In order to understand the steps in detail let us consider the DFD given in Fig. 5.39 and produce the structure chart for the same in a systematic way.

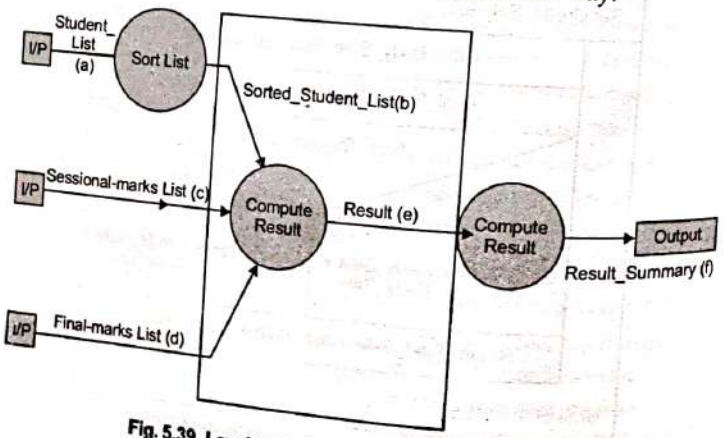


Fig. 5.39 Level 1 DFD Showing Central Transform

(*a*) **Identify the Central Transform** – Start tracing each of the afferent flows (inputs to the system) in the forward direction till they can be identified as inputs to the system. This can be achieved by asking questions that if they are still inputs to the system. Similarly start tracing each of efferent flows (output from the system) in the backward direction till the point they can be identified as outputs of the systems and disappear. This can again be done by asking questions that if they are still outputs of the system. While doing this we must keep in mind that transformations like sorting, matching, selecting, checking, formatting etc., just manipulate the data input or output and don't do any kind of transformation of data. The area (consisting of one or more processes) therefore where afferent and efferent flows disappear constitute the central transform. Each of the process in the central transform is mapped to a high level transform module in the structure chart. If we consider the DFD in Fig. 5.39, we find that Compute_Result process is the central transform as afferent inputs sorted student list, sessional marks and final_marks converge at this point and the efferent flow *i.e.*, result emerges out of here. The process summarize_result is not a central transform because it merely rearranges the result of the students year wise. Central transform is shown inside the boundary drawn with dashed line in Fig. 5.39.

It is to be noted that process involving one input and one output or a process which is merely an assembly point for the data generated by the system should not be identified as central transform as far as possible.

(*b*) **Identification of Top Level Input (Afferent) and Output (Efferent) Modules** – For all afferent data flows entering the central transforms and all efferent data flows leaving central transforms identify the corresponding top level afferent and efferent modules respectively. As shown in Fig. 5.39, we find that afferent flows disappearing into central transform are sorted_student list, sessional_marks_list and final_marks_list. Therefore corresponding to these top level afferent modules are Get_student_list, Get_sessional_marks and Get_Final_marks. Similarly efferent flow emerging out of control transform is result. So a top level efferent module, say output_result can be shown in the structure chart.

(*c*) **Identification of Coordinating Module** – Generally there is no one to one correspondence between the coordinating modules in the structure chart and DFD. It must be created and is normally the main module of the system. In other words if viewed by the user, it is the software. At this stage the top level structure chart corresponding to DFD of Fig. 5.39 would be as shown in Fig. 5.40.

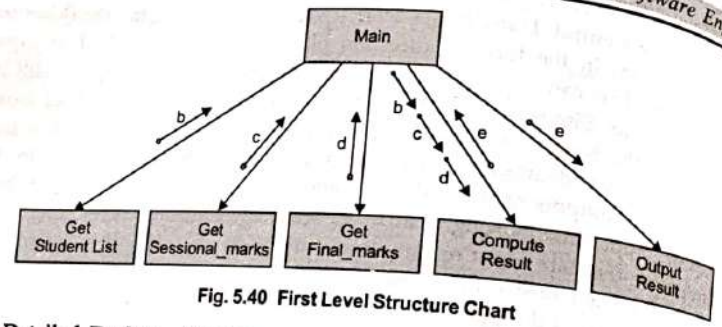


Fig. 5.40 First Level Structure Chart

(*d*) **Detailed Design** – Once we come up with the top level design, we expand the structure chart by considering each of afferent and efferent flows in turn from left to right along with the DFD. For example, if we consider the afferent flow sorted_student_list, we find that by moving backwards in the DFD, input student_list must be sorted before the input sorted_input_list is generated. Therefore, we must add two modules as subordinate modules to the Get student_list module as shown in Fig. 5.41.

Repeating in this manner for each of the afferent and efferent flows detailed structure chart is drawn as shown in Fig. 5.42.

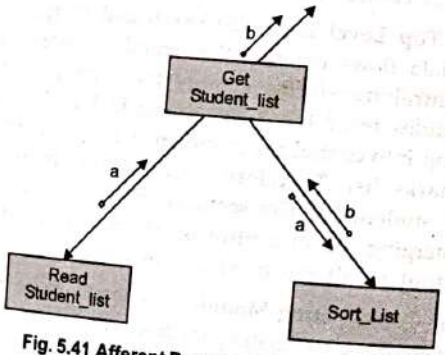


Fig. 5.41 Afferent Portion of Structure Chart

In case of leveled DFD we know that a process at level n may correspond to a DFD at level $n + 1$. Hence to show more details in the structure chart, DFD at level $n + 1$ corresponds to another level of modules where process at level n acts as coordinating module and hence structure chart is refined.

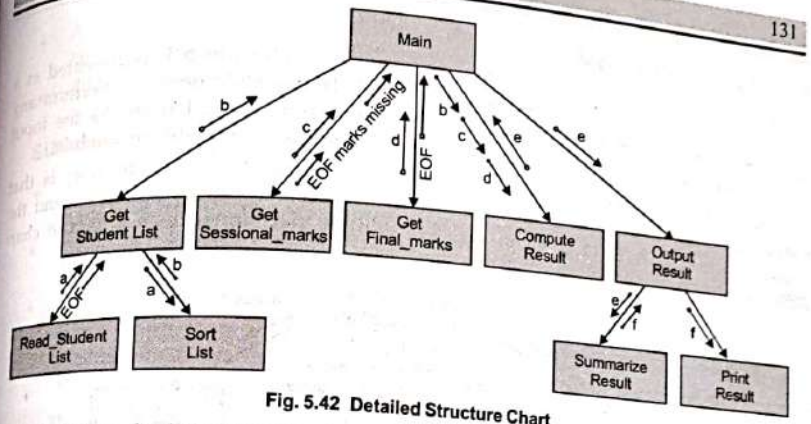


Fig. 5.42 Detailed Structure Chart

Last step in finalizing the structure chart is to add the control flags at appropriate points. Whenever a superordinate module calls subordinate module it must know about successful completion of job or error so that it can take appropriate action. It is therefore shown with the control parameters. In Fig. 5.42 we have shown end of file (EOF), marks_missing as control parameters.

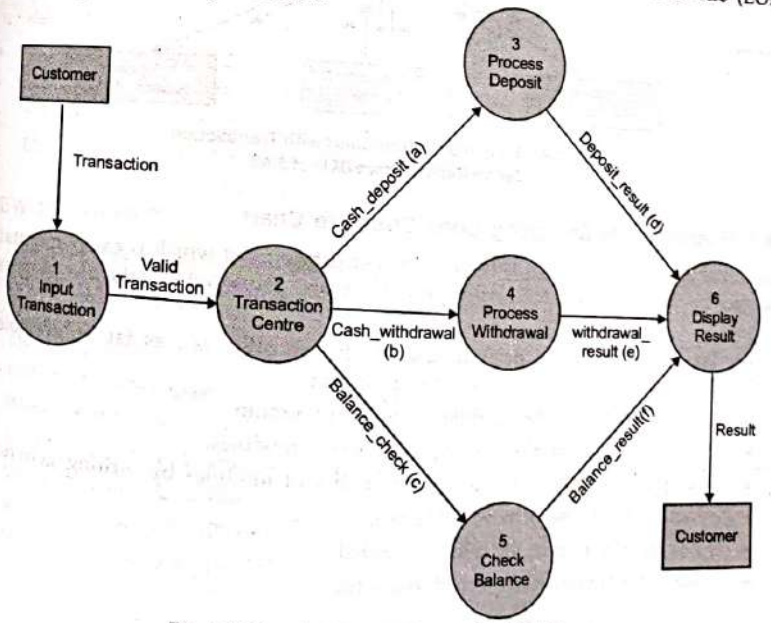


Fig. 5.43 Sample DFD with Transaction Center

5.5.2 Transaction Analysis

In this approach focus is on identifying the transaction centre which is represented as a high level module with a diamond shaped symbol. This module does not perform any transformation on data as in the case of central transform. Instead it checks the input data for type of transaction and routes it to the appropriate subordinate module.

The main difference between transaction analysis and transform analysis is that transaction analysis focuses on identification of modules that can be placed around the transaction centre. A DFD given in Fig. 5.43 will be transformed to a structure chart shown in Fig. 5.44.

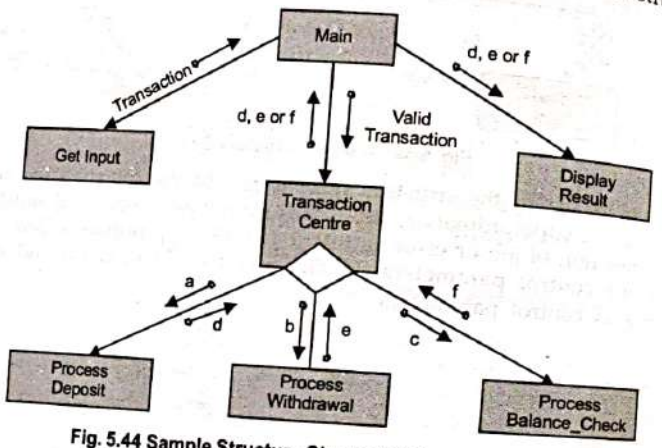


Fig. 5.44 Sample Structure Chart with Transaction Centre Derived from DFD of 5.43

5.5.3 Guidelines for Designing Good Structure Chart

The aim of good system design is to produce a structure chart which is easily maintainable and can be coded easily. Following guidelines are proposed by the developers and analysts.

- The modules should be independent of each other *i.e.*, as far as possible they should behave as black boxes to each other.
- Coupling between the modules must be minimum.
- The designer must aim for highly cohesive modules.
- The designer must also aim for reusability of modules by writing generic code.
- Module should be of reasonable size.
- Error handling modules must be added.
- Factor the central transform if required.

- Afferent and efferent modules must be factored and reorganized keeping the system balanced.
- Add all flags that are necessary on a structure chart.

SUMMARY

- Models help the different persons involved in a project to understand different aspects of the software under development and are also called software blueprints.
- A model of a system describes a specific aspect of the system under consideration at a higher level of abstraction and is called conceptual model.
- The activity of developing conceptual model is called conceptual modeling.
- Some important abstraction mechanisms are classification, aggregation, generalization and association.
- All the three views i.e., static view, functional view and dynamic view must be developed to represent the problem the information structures of application domain and the relationship between them.
- Functional view focuses on functionality of the system.
- Dynamic view depicts the time dependent behavior of the application.
- A number of structured and object oriented methodologies are proposed to develop static, functional and dynamic view.
- ER model is used for developing static view.
- DFD is used for developing the functional view.
- State transition diagram is used for developing the dynamic view.
- Output of system design stage is a document called structure chart.

REVIEW PROBLEMS

1. What are the different views of modeling? Describe briefly.
2. Define (*i*) Entity (*ii*) Relationship (*iii*) Attribute (*iv*) Cardinality (*v*) Primary Key (*vi*) Foreign Key.
3. Define (*i*) Process (*ii*) Store (*iii*) Data flow (*iv*) Terminator.
4. List other names used for data flow diagram?
5. Write a short note on leveled data flow diagram.
6. What is the use of drawing context diagram?
7. List the guidelines for drawing data flow diagrams?
8. What is a data-dictionary? Explain briefly the notation used for describing data-dictionary.
9. Identify the errors in the DFD given in Fig. 5.45.

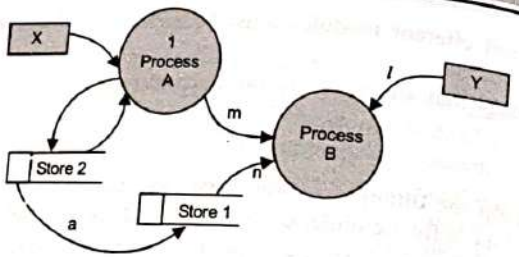


Fig. 5.45

10. What is function decomposition diagram ? What is the advantage of drawing function decomposition diagram?
11. Draw a function decomposition diagram to represent the functionality of
    (a) A blood bank
    (b) Library
12. What is a state transition diagram? Explain briefly.
13. Define an event? Illustrate different types of events by giving at least two examples of each type.
14. Draw a state transition diagram to model the working of
    (a) a washing machine
    (b) a lift
    (c) a microwave oven
    (d) GUI of any software.
15. Define each of the following terms:
    (a) Structure chart (b) Efferent module
    (c) Afferent module (d) Transaction centred design
    (e) Central transform (f) Coordinating module
    (g) Data couple
16. What is the difference between Generalization and Aggregation? Illustrate with the help of suitable example.
17. What is the transform analysis? Describe briefly.
18. Write down the steps to transform a data flow diagram into structure chart.
19. List the guidelines for good design.
20. Explain in detail different techniques used for representing the processing logic of each module.
21. A module of an employee management system computes salary of the employees. Each employee can have status of worker, manager and instructor. Each of these employees are paid a fixed salary per week. However, if the employees work more than

40 hours per week, they are paid fixed salary plus extra money per hour as per following rate.

Worker — Rs. 100
Manager — Rs. 200
Instructor — Rs. 150

In case an employee works for more than 65 hours/week, an additional allowance of Rs. 2000 is also given. Represent the processing logic of this module using structured English and Nassi-Scheidermann chart.

22. What is a decision table? What is the difference between decision table and decision-tree?
23. Represent the processing logic of module described in Q.21 using decision table and decision tree.
25. Using the data-dictionary notation described in the chapter write the data dictionary entry for
    (a) Students course registration form
    (b) Credit card information
    (c) A sales order
25. Consider the Level 1 DFD drawn in Fig. 5.46. Using transform analysis transform this DFD into a structure chart.

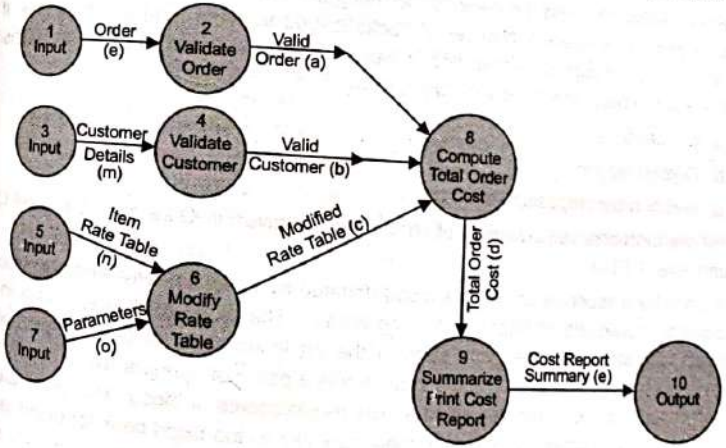


Fig. 5.46

26. Functional requirements of a Library Management System are given below:
    The main functions of a library are acquiring of new books, cataloging of books, issuing books, accepting returned books, registering clients and collection of fines from the members.

Library has a list of publishers or vendors to whom orders are placed. It first checks books in demand by asking faculty members and places an order for them. Each order is given order number and is stored along with other details like title, author, edition, publisher and copies of each book ordered. After fulfillment of order, payment is made and corresponding bill number and total amount paid is stored. Each book is given a unique book_ID and a location number of type *x.y* (*x* is shelf no. and *y* is row no.) Book details like book _ID, title, author, edition, publisher, subject etc., is stored. Library maintains a catalogue through which any client can search. For title or author etc. details along with location and status of books that match search criteria are the output of search. If book is not available, it also displays the date by which book may be available along with the address, name and contact number of the member to whom book is currently issued.

Members of the library can be students or faculty members. For registration, each member is given his/her client_ID and number of books that can be issued to him/her.

For getting a book issued, member submits a request indicating book_ID and his client_ID. Member record is then checked to ensure that the total number of books issued to the member do not exceed the specified limit. Book is then issued to the member and corresponding client-ID, date of issue, due date are stored. Book status is also marked as 'issued'. Number of books issued to member is increased by one.

At the time of returning the book by the member, return date is stored and book status is changed to 'available'. Number of books issued to client is reduced by one. If due date is before return date then fine is calculated and member is asked to deposit the fine. For the requirements given above draw

(*a*) ER-diagram

(*b*) Context diagram

(*c*) Function decomposition diagram

27. For the functional requirements of library system given in Q.26 draw the level 0 DFD and level 1 DFD.

28. A blood bank receives and stores blood donated by various people and also issues the blood to individuals or hospitals/nursing homes. The blood bank also has a in-house latest laboratory equipped with state of the art instruments to test samples of blood received from the donors. The blood bank has a panel of vendors which supply various non-consumable/consumable items as per requirements of blood. The blood bank also has a panel of doctors. Whenever a donor approaches the blood bank for blood donation, his blood sample is taken and tested for various tests. If approved, few units of blood are taken and stored in the bank. The details of donor (name, address, blood type, blood group, telephone number) is stored in the blood bank database for future use. The donor is issued a card which is valid for a period of one year. The donor during this period by showing the card can get the free unit of blood in case of requirement. In case his blood sample is not approved, his application for blood donation is rejected. In case individuals or hospitals approach the blood bank for blood, they are asked to replace the same number of units of blood. From time to time blood bank also organizes blood donation camps. For this date, venue and time is decided and is advertised in the newspapers. Two doctors from the panel are also associated with the camp to handle emergency situations. Regular donors are informed about the camp.

For the functional requirements stated above draw

(*a*) ER diagram

(*b*) Event list

(*c*) Level 1 DFD

29. For the level 1 DFD drawn for Q.28 draw the structure chart using transform analysis.

30. Netaji Subhas Institute of Technology along with M/s Infotek Corporation is planning to organize an international conference on software engineering. An executive committee consisting of academicians from leading institutes and persons from industry is formed to manage the conference. Information is sent to all the leading institutes and industries inviting papers for the conference. Deadline for receiving the papers is also fixed. A panel of experts is constituted for reviewing the papers. Once papers are received, each paper is sent to three experts for review. Results after the review are compiled and authors are informed about the acceptance/rejection of papers. In case of acceptance, the authors are supposed to send the final copy of the paper on format as specified by the organizer before a fixed date along with the undertaking to attend the conference. It is mandatory for one of authors of the paper to attend and present the paper. The organizing committee has also made arrangements with three leading hotels for the participants stay. Participants at the time of sending the registration form along with fees can also book the hotel by paying a nominal advance. The complete information about the conference is also available at the website. Participants can also do the registration on-line through the website. The organizing committee has also made arrangements with a traveling agency for local sight-seeing of the participants.

For requirements specified above model the

(*a*) Static view

(*b*) Context diagram

(*c*) Function decomposition diagram

(*d*) Event list

31. Delhi Vidyut Board computes the monthly bill of its customers as per following details:

If monthly consumption is less than equal to 300 units and load type is residential, calculations are done at the rate of Rs.2.00/unit. In case of commercial type of load, rate of Rs. 3.50/unit is used. If monthly consumptions is greater than 300 units and less than equal to 400 units, rate of Rs. 2.50/unit is used for units above 300 units for residential load. In case of commercial load, for consumption greater than 300 units,

rate of Rs. 5/unit is charged for consumption above 300 units and less than equal to 400 units. For monthly consumption greater than 400 units for residential load, rate of Rs. 3.00/unit is charged for consumption above 400 units. Additionally a surcharge of 2% is also charged on total value. In case of commercial load a surcharge of 3% is charged on total value for consumption greater than 500 units per month.

Draw a decision table and a decision tree for the above processing logic.

○○○

Chapter 6

OBJECT ORIENTED PARADIGM AND UML

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- Difference between the structured analysis and object oriented analysis techniques.
- Basic concepts of object oriented analysis i.e., class, object, inheritance, information hiding and polymorphism
- Unified Modeling Language (UML)
- Class diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- State chart diagram
- Component diagram
- Deployment diagram
- Unified process

Object Oriented Requirement Analysis (OORA) is a technique for understanding the problem domain and representing the requirements of a software system in terms of problem domain objects and interactions between these objects. It is based on the best concepts from object oriented programming languages, information modeling and

knowledge based systems in order to manage complexity. The object is the fundamental principle in Object technology. At the simplest level every thing which we see, touch, feel or sense can be called an object. Examples of real life objects are student, account, book and many more.

6.1 THE TRADITIONAL PARADIGM VERSUS THE OBJECT ORIENTED PARADIGM

As information systems started becoming more and more complex, representation of requirements became an important task. As a result number of structured methodologies were proposed by the practitioners and the researchers. Use of these methodologies in developing software introduced discipline in the team and the organization. As a result quality of information systems also increased. But as information systems grew more and more complex, practitioners realized that traditional structured methodologies were becoming less effective. According to a study conducted by a research firm – The Standish Group on 280,000 development projects completed in 2000, only 28 per cent projects were successfully completed and 23 percent were never completed or canceled. The rest 49 per cent projects though they were delivered but had problems like less functionality, over budget, and late delivery.

In function decomposition method analyst maps the problem domain into functions and sub functions which are difficult to construct and are highly volatile. Also the problem domain understanding can not be verified for accuracy. In the second approach *i.e.* structured analysis approach analyst maps the problem domain into processes and flow of data between these processes. Two techniques *i.e.*, classical structured analysis and Modern Structured Analysis by Yourdan predominate in structured analysis. The main problem here is to decide on processes and decomposing a process into a new DFD while going from level *n* to *n*+1. Event partitioning approach tried to solve the problem to a certain extent but problem remained if number of events were too high as in the case of complex information systems.

It was found that conventional process oriented requirements analysis methods such as structured analysis and function decomposition did not fit properly into object oriented design and required a great deal of reorganization. Transition from analysis to design was not a straight forward task and caused lot of problems and frustration. Same problem came when analyst tried to use results of structured analysis for object oriented design. Hence a need was felt to propose an analysis technique that was compatible with Object-Oriented (OO) design. OORA technique decomposes the problem domain in terms of interacting objects and the resulting document is easily compatible with *OO* design. According to (Coad91) the motivations for proposing OORA techniques were:

1. To handle more complex and challenging problem domains.
2. To improve the analyst and problem domain expert interaction.
3. To improve the internal consistency of analysis results.

4. To support reuse of analysis results.
5. To represent commonality explicitly.
6. To build stable specifications.
7. To support transition from analysis to design phase in a systematic manner.

OORA therefore differs from the traditional process oriented method in two ways:

1. The way system is partitioned into components called objects and
2. Interaction among these components.

In process oriented approach as discussed in chapter 5, a system is described as a set of interacting processes whereas in object oriented approach one describes the system in terms of interacting objects. In traditional approach systems and designers while working on software project used to think in terms computer based entities such as databases, programs etc., but in the object oriented paradigm they now think in terms of real world objects thereby reducing the semantic gap remarkably. This also makes the understanding of requirements easier and software can be also changed easily. ***Identification of the objects is therefore the primary task of OORA.***

6.2 BASIC CONCEPTS OF OORA

The basic concepts of OORA are described below and are also those of object oriented design and object oriented programming languages. They are:

6.2.1 Object

Conceptually an object can be described as a thing with which we can interact through a well defined interface. Messages can be passed to objects to which they respond depending upon internal state of the object. A number of definitions of object exist in literature. According to (Sidney97) *an object can be described as a data structure together with a collection of functions or operations that act on or refer to that structure.* The functions can change object's current state, retrieve current values making up the object's current state or compute new values from object's current state. The motivation for identifying objects is to match the system as close as possible to the real world. As each object encapsulates a set of methods and a state to which the methods have access, it makes an object a cohesive unit of data and functions as shown in figure 6.1.

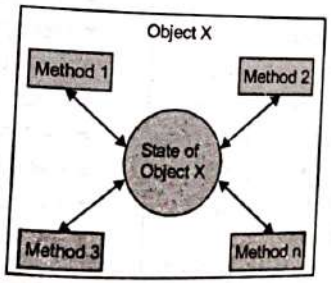


Fig. 6.1 An Object

Objects interact by sending messages to each other through a well defined interface as shown in figure 6.2. An object may send a message to itself also. In response to the messages received, objects perform services for the sending object. An interface describes what kind of operations an object will perform or what form of messages an object accepts. As an example let us consider an object Patient. This object can have attributes name, address, registration number, date of birth, telephone, doctor name and disease name. Now as we need other objects to be able to retrieve and manipulate the details of patient object we need to provide a set of methods which the patient object can use as interface with the external world. Figure 6.3 lists same of the attributes and methods of the patient object.

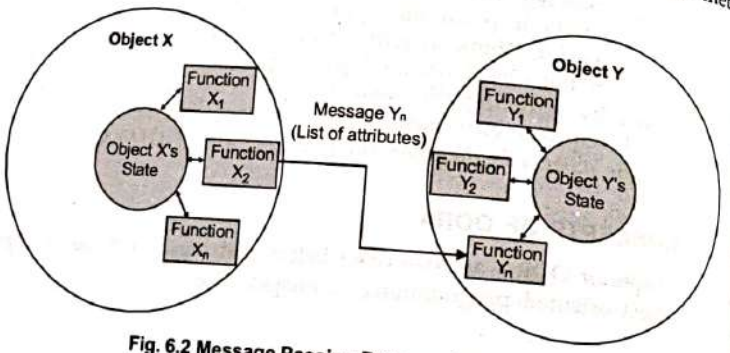


Fig. 6.2 Message Passing Between Objects

If we club all the operations provided by all the objects in a problem domain, it will give us the functional requirements of the problem domain. Hence the primary goal of OORA is identification of objects, operations performed by these objects and to determine how operations are encapsulated by different objects.

Patient Object	
Attributes	*Operations*
• Name	• Set name
• Address	• Get address
• Telephone	• Get history
• Disease-history	• Admit patient
• Registration code	• Discharge patient
• Bed-number	• Set telephone
• Admission-date	• Get telephone
• Discharge-date	• Get name
• Doctor	• Set address

Fig. 6.3 An Example of Patient Object

6.2.2 Classes

Objects can be further organized into classes based on their similarities and dissimilarities. It is to be noted that classes are identified for multiple objects with similar behavior but not for a single object. Behavior of an object is the way object responds in terms of state change and message passing. Also objects from the same class have the same interfaces. Object belonging to a class is also sometimes called **instance** of a class. For example a class Vehicle can have car, truck, bicycle, motorcycle and bus objects as its instances as show in Fig. 6.4

To make it more clear, let us consider an object ramesh_account which understands the messages debit_account(2000.00) and credit_account(10000.00). As each object belongs to a class, therefore the object ramesh_account may belong to class Account with a public interface to provide operations debit-account(money) and credit_account(money). Hence the way an object responds to a message is determined by the class to which the object belongs.

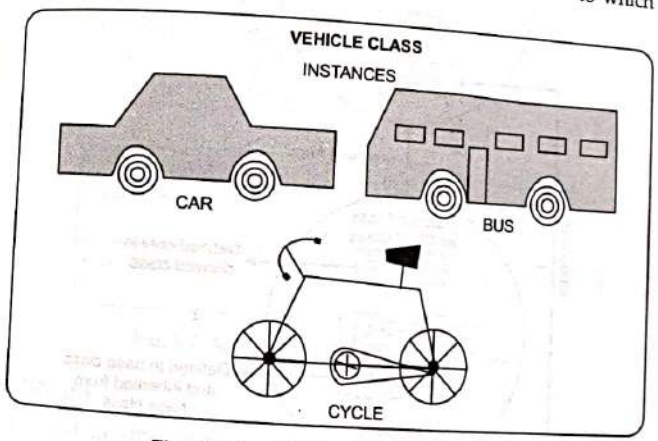


Fig. 6.4 Class Vehicle and its Objects

Some standard functions normally a class has are:

- Constructor functions to create instances of new objects
- Destructor functions to discard existing instances of classes
- Set and get functions to update and retrieve the current state of the object.
- Additional functions relevant to the problem domain which may or may not have restricted access.

6.2.3 Inheritance

Inheritance is the most important feature of OORA and permits *reusability*. Classes can be organized into hierarchy of classes and sub classes through "is_a" predicate. In OORA

the "is_a" relation is realized through inheritance which is the process of creating new classes from the existing base class. New classes which are created are called derived classes or subclasses. Each subclass inherits all the capabilities of base class. Additionally it can have some additional features of its own as shown in Fig. 6.5.

A method provided by parent class or base class may be left undefined and is replaced in the derived class (specific to the derived class) with a specific version called specialization as shown in Fig. 6.5

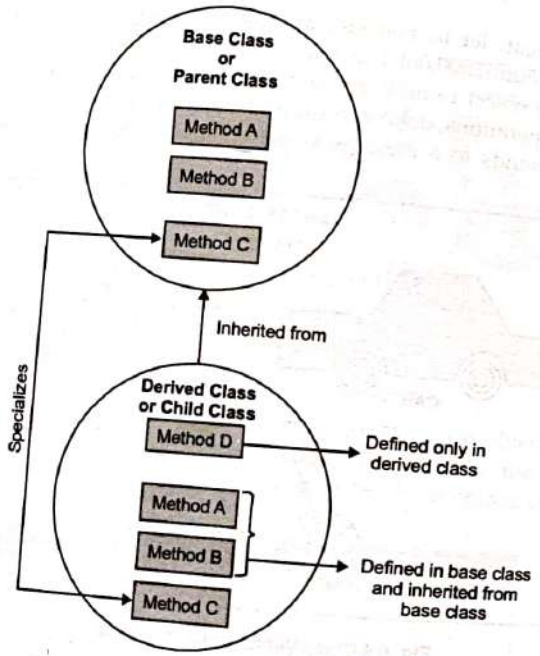


Fig. 6.5 Inheritance in Classes

A subclass can also have multiple base classes in which it inherits all the features of base classes. This is called multiple inheritance as shown in Fig. 6.6.

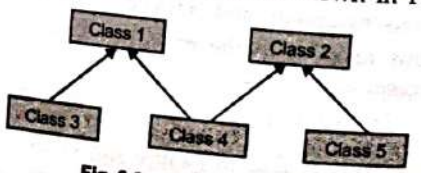


Fig. 6.6 Multiple Inheritance

An example illustrating inheritance among classes is shown in Fig. 6.7. In this example, there are three classes ***account***, ***saving_account*** and ***current_account***. Class account is parent class where as the other two classes are child classes or derived classes. The attributes and operations of class ***account*** are inherited by saving_account and current_account classes as well. For example the operations Open_account, Close_Account, withdraw_cash, Deposit_cash() etc., will be used by both current_account and saving_account class. In other words we can resuse the programs written for superclass account. Additionally current_account class has additional attributes firm_name and overdraft_limit and one additional operation check_limit(). Saving_account has one additional attributes Locker_No and operation Print_FD_details().

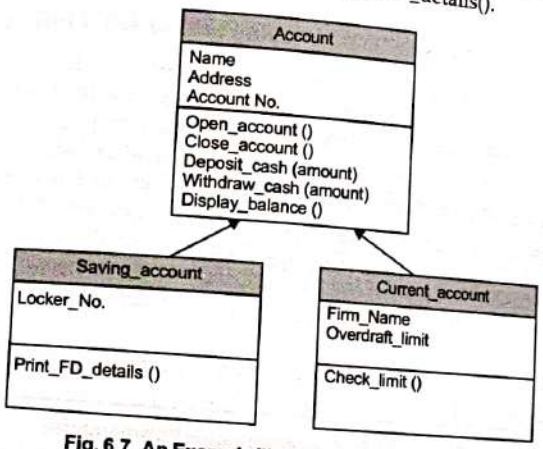


Fig. 6.7 An Example Illustrating Inheritance

6.2.4 Polymorphism

Polymorphism is yet another powerful concept of OORA. The word polymorphism is derived from Greek and means having many shapes. At OORA level therefore, this property allows a single function at conceptual level to take different forms depending on the object type on which it is acting *i.e.*, different classes support identically named methods with different arguments. For example a class text_object can have method to print text whereas another class graphics_object can have identically named method print to handle graphics data.

The decision to activate a polymorphic function call is done by a process called **dynamic binding**, which dynamically binds the message sent to the appropriate code at runtime. To make it more clear, a sample C++ program is given in Fig. 6.8. In this program the function add is used to add two or three integers. Depending upon the number of arguments being passed, appropriate add function is invoked at runtime.

```
Class addnumbers
{
    public
    void add (int a, int b)
    {
        cout << "Sum is" << a+b;
    }
    void add (int a, int b, int c)
    {
        cout << "Sum is" << a+b+c;
    }
}
main ()
{
    addnumbers test object;
    test object.add (5, 4);
    test object.add (10, 20, 30);
}
output
    Sum is 9
    Sum is 60
```

Fig. 6.8 A C++ Program Showing Polymorphism

6.2.5 Information Hiding

The property of information hiding is supported through the concept of black box. Objects in a class through a well defined interface provide services to the user and hide the implementation details. According to [Schach03] information hiding is a way of enhancing maintainability by making implementation details invisible outside an object.

For example consider a class **Employee** and suppose that total_Salary is an attribute of this class. Now this attribute can be implemented either as an integer or a real number. Furthur this attribute can be declared **private** which is possible in case of Object Oriented languages. It means that now this attribute becomes invisible outside the object and can be accessed only via an operation say *compute_Salary*. The only way to change the value of this attribute is by sending a message to operation *compute_Salary*.

Now the question which immediately comes to our mind is why this route through an operation is taken to access the total_Salary attribute instead of accessing it directly. The advantage is that in case we change the implementation details of total_Salary attribute

it will not affect any other part of information system thereby reducing the regression faults in the code. Similarly when a message is passed to an object it becomes irrelevant for the module who sends the message that how the request is carried out.

A number of OORA techniques are proposed in the literature. Some of the popular techniques are Object Oriented Analysis by Coad & Yourdan, Object Modelling Technique (OMT) by Rambaugh, Object Oriented Analysis and Design (OOAD) by Booch and many more. Latest technique which has almost become an industry standard is Unified Modelling Language (UML03) proposed by Jacobson, Booch and Rumbaugh. In the next section, we would discuss Unified Modelling Language.

6.3 UNIFIED MODELLING LANGUAGE (UML)

UML was created and proposed more out of need to represent and develop software systems in a methodical way using object oriented approach. Most of work in this area is done by three most prominent methodologists, *i.e.*, Grady Booch, James Rumbaugh and Ivar Jacobson, also known as three amigo's in object oriented circle. In fact, UML was the fusion of most of their ideas proposed in their independent OORA techniques *i.e.*, Booch method, Rumbaugh's OMT and Jacobson's OOSE method and its development was an open process, where number of companies like IBM, HP etc., also contributed significantly. Today the scenario is that industry has almost recognized UML as a standard language for developing software blueprints. In this section we will briefly discuss salient features of UML.

In short UML supports specifications for 9 diagrams which capture and document different perspectives of problem domain starting from inception to installation and maintenance too. These 9 diagrams are:

- Class Diagram
- Object Diagram
- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- State Chart Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram

In chapter 5, we have discussed about different views of modeling. These different views are modeled efficiently using above mentioned diagrams of UML. In context with UML, we would use the term view to organize UML diagrams which describes similar aspect of Universe of Discourse. UML supports following different views of modeling:

Static View

As explained in chapter 5, static view focuses on the what part of the system and does not specify how the system elements will behave. It is just like a simple blueprint. Static view is modeled using class diagram and object diagram. Class diagram is the main diagram which identifies the main classes (defines rules about objects) and the relationship among these classes. This diagram is very important as it is used as a source for code generation or design stage. Object diagram on the other hand is a special type of class diagram showing instances of classes *i.e.*, objects and can be used to test or understand a class diagram.

Functional View

The functional view focuses on the how part of the system and in UML it is supported through Use Case diagram and Activity diagram. Use Case diagram describes the system from user's perspective in terms of the features that a user expects the system to provide. It identifies the actors (usually other systems or human elements of system) which trigger these functions or receive events from the system. Use Cases are the high level goals or functions which the system should support.

Logic of each of the Use Case is modeled with the help of Activity diagram which describes various processes including conditional logic, sequential task and concurrency in detail.

Dynamic View

The dynamic view consists of all the diagrams which show how different objects interact with each other. Three diagrams are used to capture the dynamic view. They are:

- Sequence diagram
- Collaboration diagram
- State chart diagram

Sequence and Collaboration diagrams show object interaction whereas Statechart diagram shows an object's reaction to external stimuli and its possible states during an object's lifespan.

Architectural View

In addition to static, functional and dynamic view, UML also talks about architectural view. Deployment diagrams and Component diagrams are used to show the architectural view of the problem domain. Deployment diagrams are used to show how software will be implemented on the hardware whereas Component diagrams model the runtime software elements and are drawn in conjunction with Deployment diagrams.

In the next sections, we will discuss some of these diagrams briefly. For detailed study of UML notation, readers should refer to "The Unified Modeling Language User Guide" by Booch, Rumbaugh and Jacobson Published by Adison-Wesley under Object Technology Series (Booch02).

6.3.1 Use Case Diagram (UCD)

A Use Case Diagram describes the system from the users point of view. In other words it shows the relationship between the system and the external world. This diagram is also used to show the boundary of the system. Use Case diagrams are used in the initial stages of software development i.e., requirements analysis stage and are very useful for

- Understanding the requirements of the problem domain
- Generating test cases
- Contributing to users understanding of the system
- Validating design
- Creating project schedule

There are six simple concepts which are used to draw the Use Case Diagram. They are:

- System
- Actor
- Use Case
- Associations
- Dependencies
- Generalization

In simple language a ***Use Case Diagram is a collection of Actors, Use Cases and their Communications.*** Next, we briefly describe each of these concepts used for drawing UCD.

System

A system icon shows the boundary of the system to be built and is represented by a rectangle as shown in Fig. 6.9.

Fig. 6.9 System Icon

Actor

Actor is a role played by a person, organization, other system or device which interacts with the system by using the use cases. In other words, an actor is one who initiates some events which in turn trigger the activities of the organization. Fig. 6.10 shows some actor icons used in UML.

It is to be kept in mind that actors are always outside the boundary of the system. So as said earlier, actors can be people, an organization, some other software, hardware, networks etc. Each of these can be represented by one or more actors. For example a person working in the bank can have an account in the same bank. Therefore in this case he plays the role of two actors *i.e.*, employee and the customer.

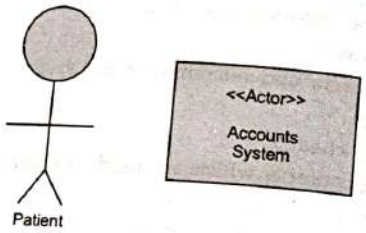


Fig. 6.10 Symbols for Actor

Use Cases

A Use Case shows the required functionality or goals of the system without emphasizing how it will be achieved. In other words, it defines the requirements of the system. In simple language, Use Cases are those features of the software which actors would like the system to support e.g., querying the admission status, printing the inventory report. If Use Case diagram is too complicated and big, it is better to partition the Use Case diagram in to several Use Case diagrams, where each diagram represents some major functionality of the problem domain. Use Cases are written using verb-object phrases e.g., withdraw cash, show balance etc. Notation for representing Use Cases is shown in Fig. 6.11.

Fig. 6.11 Use Case Notation

Associations

Associations show the communication of actors with Use Cases and are represented by straight lines connecting actors and Use Cases. Associations can be unidirectional or bidirectional. Unidirectional associations can be shown by putting arrow on one side where as bidirectional association are represented by putting arrows on both sides or no arrows at all. In case of unidirectional association there is only one way communication where as in the bidirectional association the communication is both way (e.g., a Use Case supporting some kind of query).

Dependencies

Dependencies are used to show relationship between Use Cases and is shown through dashed arrows between two Use Cases. There can be two types of relationships between Use Cases **<<include>>** and **<<extend>>**. Stereotype notation *i.e.*, word in guillemets << >> is used to show relationship. The main advantage is that through this feature UML supports reuse in the software being developed. <<include>> dependency stereotype says that execution of one Use Case includes the working of other Use Case always whereas <<extend>> stereotype says that one Use Case may or may not use the other Use Case. Fig. 6.12 shows the associations and dependencies in UCD.

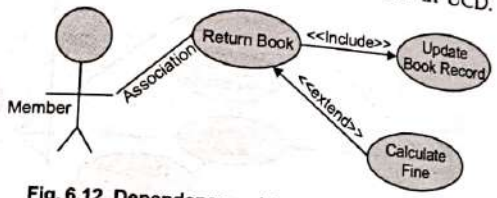


Fig. 6.12 Dependency and Association Notation

It is to be noted that direction of arrow in <<extend>> dependency is from the extension Use Case to the one which is calling it. On the other hand in the <<include>> dependency the arrow direction is from the main Use Case to the one which is being required by it. In Use Case diagram shown in Fig. 6.12, Return-book Use Case will always call update book_record Use Case but it may or may not call calculate-fine Use Case.

Generalization

Finally is_a relationship *i.e.*, concept of inheritance among Use Cases and actor is shown through generalization. A solid line with hollow triangle is used to show generalization. A sample Use Case diagram to show generalization is shown in Fig. 6.13.

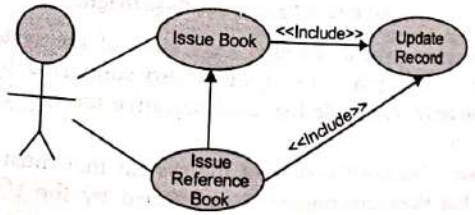


Fig. 6.13 Generalization Among Use Cases

Guidelines For Drawing Use Case Diagrams

The guidelines used for drawing Use Case diagram are:

1. Set the boundary of the system.

2. Identify actors.
3. Identify Use Cases.
4. Establish associations between actors and Use Cases.
5. Identify <<include>> and <<extend>> relationships between the Use Cases.
6. Identify generalization between actors and Use Cases.
7. Refine the diagram by reviewing it more than once.

A sample Use Case Diagram for a part of Library is shown in Fig. 6.14.

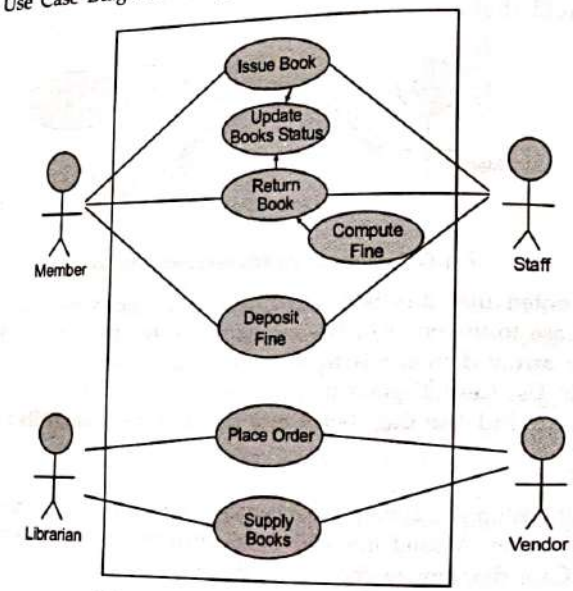


Fig. 6.14 A Sample Use Case Diagram

A Use Case Diagram gives us an high level view of the system. Therefore in order to understand it in more depth, a Use Case is also supported by a textual description called **Use Case narrative**. A sample Use Case narrative for Use Case Register Complaint is shown in Fig. 6.15.

Here ***Assumptions*** describe the state of the system that must be true before we can use the Use Case but these conditions are not tested by the Use Case. ***Pre-conditions*** also describe the state of system that must be true before we use the Use Case but these conditions are tested by the Use Case before proceeding further. For example in the Use Case narrative shown in Fig. 6.15 unless a proper patient code in proper format is entered, Use Case will not start. Similarly ***post-conditions*** describe the state of the system that must be true when the Use Case ends. Though there is only one way to start the Use Case, there are many ways to end it. These are documented in Use Case termination part of the Use Case narrative.

Name — Register Complaint
Author — S. Sabharwal
Number — 22
Last update — 14/01/2004
Assumption — User has permission to use this feature
Preconditions — Provide a valid patient registration number

Flow of Events/Dialog:

1. Use Case begins when the patient selects register complaint option.
2. The patient enters his/her registration number.
3. The system verifies the registration number. If incorrect the system will prompt the patient to reenter the registration number or exit.
4. If registration number is correct, the system opens a window for the patient to record his/her complaint.
5. After writing, the patient exits by pressing the exit button.

Post conditions: In case of normal termination complaint is saved in complaints database for management use.

Use Case termination:
1. The user presses exit after he is done.
2. The Use Case may time out.
3. The user may exit anytime during the execution of use case by pressing the Cancel Key.

Fig. 6.15 The Register Complaint Use Case Description

6.3.2 Classes and Class Diagram

Class diagram is the most important diagram which shows the static view of the system by showing the classes and their relationships. A class integrates the statics (data) and dynamics (behavior) into one cohesive unit. UML notation to represent a **class** is a rectangle consisting of three parts - class name, class attributes and operations as shown in Fig. 6.16.

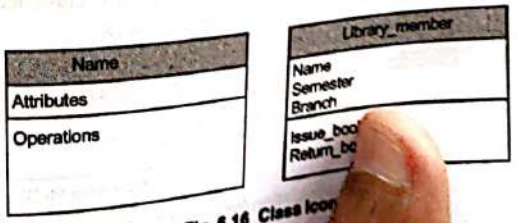


Fig. 6.16 Class Icon

Additionally, it also specifies the visibility of attributes and operations *i.e.*, which other objects can see the attributes and operations. This typically includes public (+), private (-), protected (#) and package (~) as visibility values.

While writing operation specification, it must specify all arguments and their data types (separated by colon), return data type, its visibility and the constraints within { }. For example Compute_Cost operation is written as

+ Compute_cost(Order: Order): Rupees {The total cost is sum of all items cost (unit price * quantity) mentioned in the order less the trade discount}

A detailed class specification for patient class is as shown in Fig. 6.17.

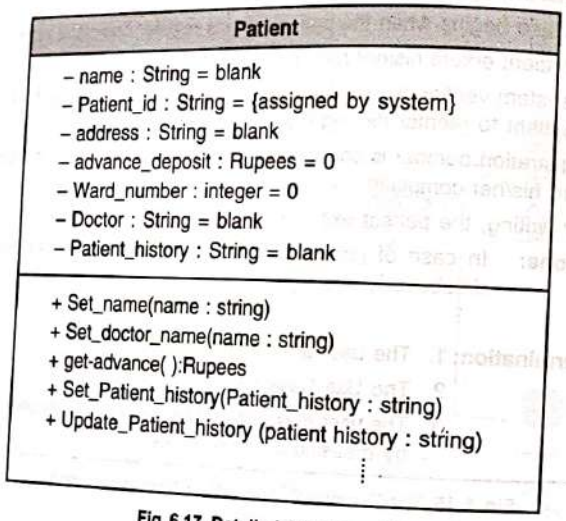


Fig. 6.17 Detailed Class Specification

Three types of relations *viz.*, **association, aggregation** and **generalization** can exist between classes in the class diagram. **Association** relates a class *A* to class *B* and is shown as a link between classes. While reading textual description of requirements they correspond to verbs and classes correspond to nouns.

Multiplicity (number of participating objects), role and constraints are also shown in the association. Constraints are shown between { } below the class icons.

An association between two classes is shown in Fig. 6.18.

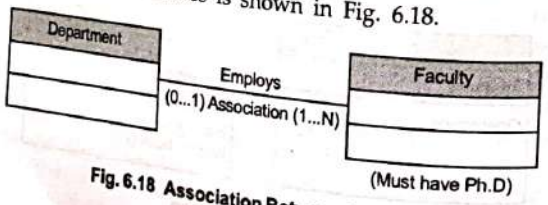


Fig. 6.18 Association Between Classes

Is_a relationship between classes is shown through **generalization.** The classes between which is_a relationship is identified are called superclass and subclass. Subclass inherits the features of superclass in addition to features which are specific to subclass. A generalization is represented as shown in Fig. 6.19. Here Department is the superclass and classes Computer Engineering and Mechanical Engineering are sub classes.

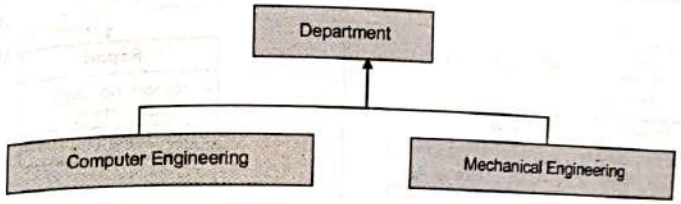


Fig. 6.19 Generalization in Classes

Aggregation is a special type of association which supports building complex objects out of existing objects. For example different components can be assembled to make a car as shown in Fig. 6.20.

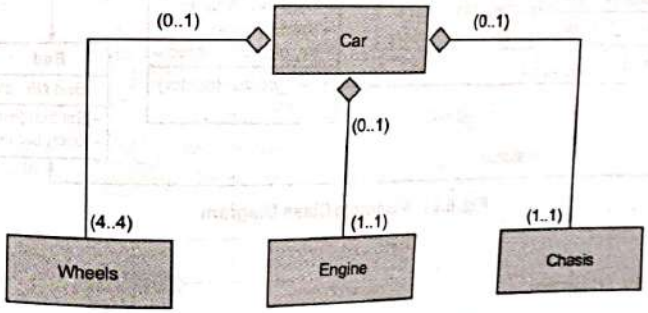


Fig. 6.20 Aggregation Relationship

To represent aggregation a diamond symbol is attached to the aggregate class. A sample class diagram is shown in Fig. 6.21.

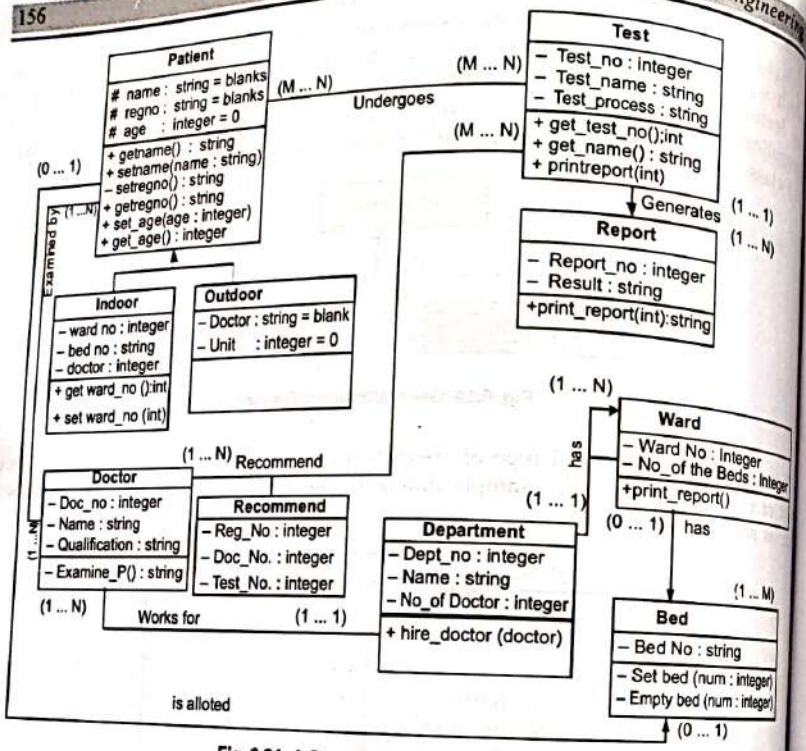


Fig. 6.21 A Sample Class Diagram

6.3.3 Activity Diagram

Activity diagram is used to model the functional view and focuses on flow of activities for a process. It resembles a flow chart. In simple language, an activity diagram is a series of interconnected activities which are linked by transitions.

Each of the activities in the activity diagram is a step (computation of data, query, verification of data, report generation etc.) in the process. Transition lakes place when activity is over and/or the guard condition is satisfied. From decision point various paths exit for each condition. Similarly at a merge point two or more paths meet and continue as one. Each activity diagram has a starting point and single/multiple ending points. Activity diagrams can also be divided into object ***swimlanes*** to indicate which object is responsible for which activity. Transitions can also fork into multiple parallel activities.

Notation used to draw activity diagram is given in Fig. 6.22.

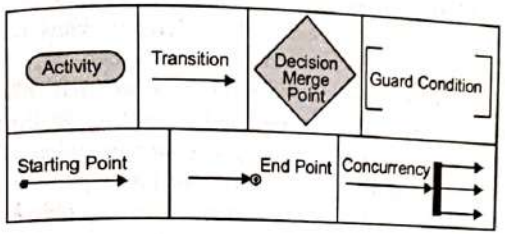


Fig. 6.22 Notation for Drawing Activity Diagram

A sample Activity diagram to withdraw cash from ATM is shown in Fig. 6.23

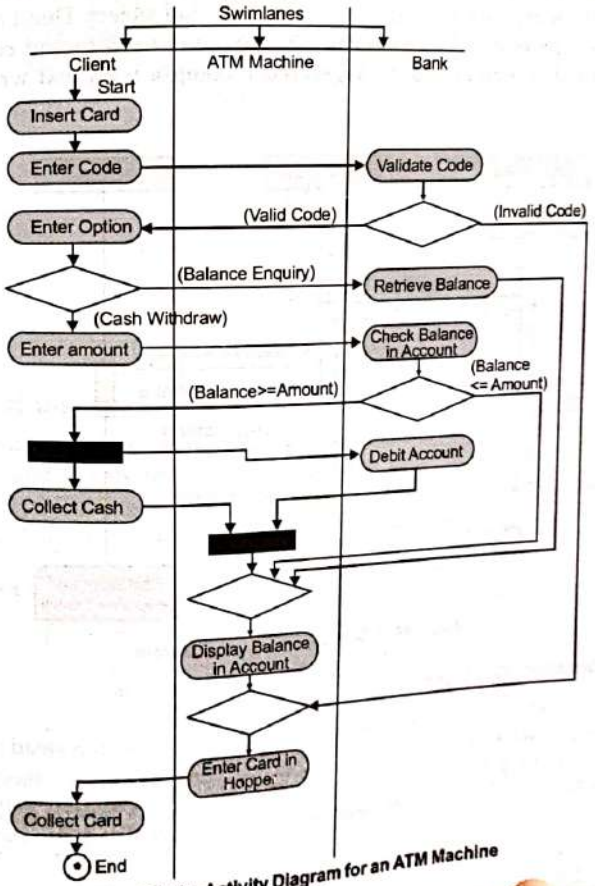


Fig. 6.23 An Activity Diagram for an ATM Machine

6.3.4 The Sequence Diagram

Class diagram and Object diagram show the static view *i.e.*, important objects of problem domain and how they are related to each other. Dynamic view on the other hand shows the interactions among the objects i.e., how the objects behave. The Sequence diagram is an interaction diagram that shows how objects talk to each other. In other words the diagram shows how operations are carried out according to time and how objects are created and manipulated. The Sequence diagrams are modeled at objects level to allow for scenarios. Multiple objects from same class can participate in the Sequence diagram. The Sequence diagram uses three main concepts objects, object lifeline and messages. Messages can be synchronous as well as synchronous.

Fig. 6.24 shows a sample Sequence diagram to create bill at a diagnostics lab. In this diagram each **vertical dotted line** is **lifeline** which represents the time for which an object exists. **Arrow** represents a message call to the other object. Duration for which an object is active is represented by **activation bar. Note** is used to add comments to the UML diagram and is represented by **dog-eared rectangle** with text written inside the rectangle.

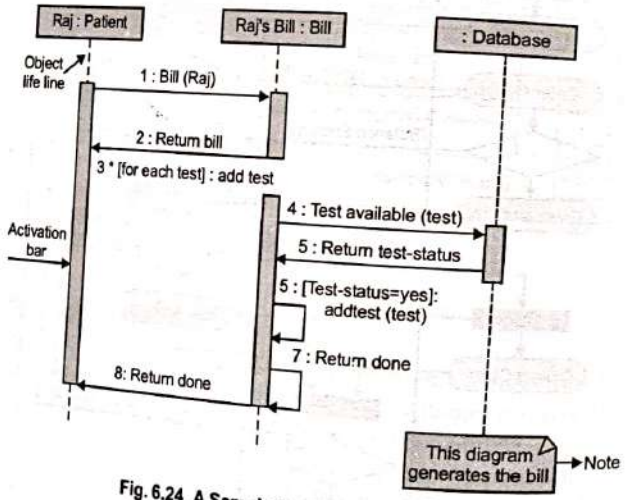


Fig. 6.24 A Sample Sequence Diagram

6.3.5 Collaboration Diagram

Collaboration diagram also models the object interactions but instead of focusing on time the messages are sent, it focuses on interactions with respect to objects relationships. Hence objects can be placed anywhere in the diagram. This diagram helps in validation of Class diagram by checking that each association is required for passing messages among the classes. This helps in identification of new operations and thus refines the Class diagram. A sample Collaboration diagram is shown in Fig. 6.25.

As shown in Fig. 6.25, object roles rectangles can be labeled with object names or class or both. Further all the messages are numbered to show the order of execution of messages. The format for specifying the message is

message_sequence no. : [condition] operation or return.

Same format as clear from Fig. 6.24 is also used in Sequence diagram.

Collaboration diagram does not give any information about when object is active or inactive which is clear in Sequence diagram.

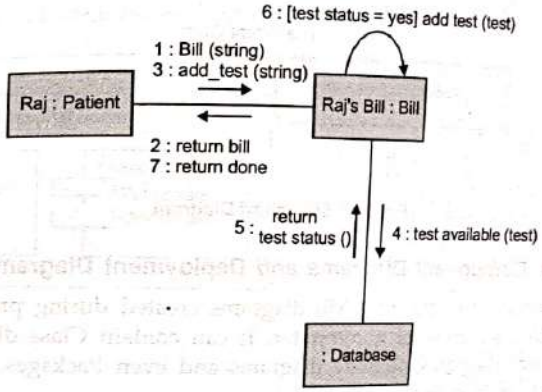


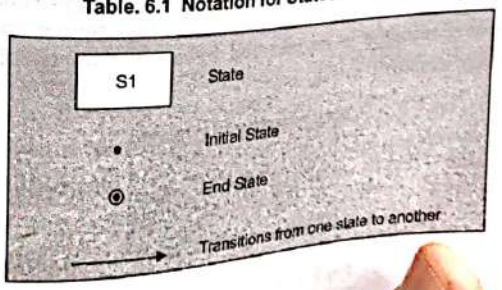
Fig. 6.25 A Sample Collaboration Diagram

6.3.6 Statechart Diagram

Statechart diagram also models the dynamic view. It shows all the possible states in an object's lifetime and events that trigger the change in states.

It is similar to state transition diagram discussed in chapter 5. The notation used to draw Statechart diagram is given in Table 6.1

Table. 6.1 Notation for Statechart Diagram



Events that cause change of state and corresponding actions are labeled on arrows. A sample Statechart diagram is shown in Fig. 6.26.

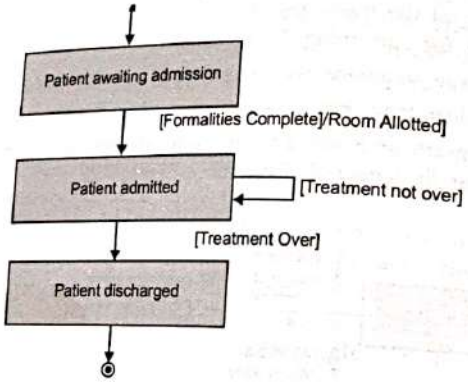


Fig. 6.26 Statechart Diagram

6.3.7 Packages, Component Diagrams and Deployment Diagram

Packages are used to organize the UML diagrams created during project logically and work like directory structure of a computer. It can contain Class diagrams, Use Case diagrams, Sequence diagrams, Activity diagrams and even Packages. The symbol used to represent Package is shown in Fig. 6.27.

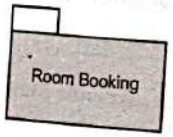


Fig. 6.27 A Packaging Icon

Physical Implementation of the software is shown using Component diagram where components represents software modules. Physical architecture of the hardware is modeled with Deployment diagram. It also shows the relationship among various software and hardware. A processing resource is represented by a node on which software resides.

A component icon is shown in Fig. 6.28.

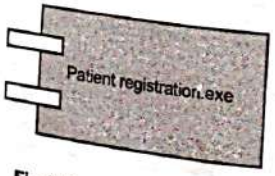


Fig. 6.28 A Component Icon

A node icon is a 3D box as shown in Fig 6.29.

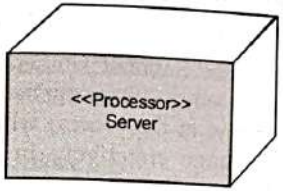


Fig. 6.29 A Node Icon

A sample Deployment diagram is shown in Fig. 6.30.

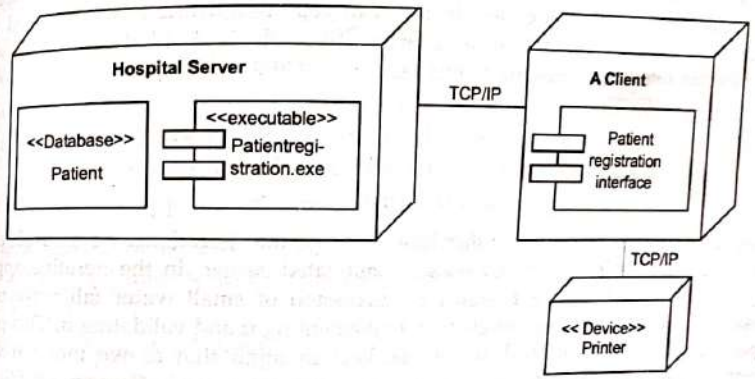


Fig. 6.30 A Sample Deployment Diagram

6.4 RATIONAL UNIFIED PROCESS

Rational Unified Process is a software engineering process which aims at producing high quality software using UML that meets the needs of its end users within a predictable schedule and budget [Jacobson00].

It is a process product as well as a process framework that can be adapted by an organization to suit its needs. The Rational Unified Process is built on following six commercial practices to deliver a well-defined process.

- Develop Software Iteratively
- Manage Requirements
- Use component – based Architectures
- Visually Model Software

- Continuously verify software quality
- Control changes to software

The detailed discussion about the Rational Unified Process is beyond the scope of this book. Readers can refer to [Jacobson00] for further reading. In this book however we will briefly describe the overview of the process which is in fact a guide for how to use the Unified Modeling Language effectively in order to clearly communicate requirements, architectures and designs.

6.4.1 Process Overview

The Process has a two dimensional view where the horizontal axis (Time dimension) shows the dynamic aspect in terms of cycles, phases, iterations and milestones and the vertical axis represents the static view in terms of activities, artifices, workers and work flows. In the Rational Unified Process software life cycle is divided into cycles where each cycles has following consecutive phases.

- Inception Phase
- Elaboration Phase
- Constructions Phase
- Transition Phase

Each of these phases can be further broken down into iterations. As a result overall quality of product is improved and risks are mitigated earlier. In the iterative approach life cycle of a large project is broken into succession of small water fall projects each addressing some requirements, designing it, implementing it and validating it. This process is repeated till you are finished. It must be kept in mind that as we move from one iteration to next or from one phase to the next emphasis on different activities also changes. The outcome of different phases is shown in table 6.2.

Table 6.2: Different Phases and Outcome of Phases in Rational Unified Process

S. No.	Phase	Outcome
1.	Inception Phase	• In initial use case model (upto 20% complete) • A business model if necessary • A project plan and project glossary • A vision document showing projects key requirements and constraints • An initial risk assessment • One or more prototypes
2.	Elaboration Phase	• At least 80% complete use case model • A software Architecture details • Non-functional requirements or other requirements not associated with a specific use case • A development plan for the overall project

Contd..

3.	Construction Phase	• Integrated product on the adequate platforms • The user manual • A description of current release.
4.	Transition Phase	• Validation of new system using Beta testing • Parallel operation with a legacy system that it is replacing • User training • Marketing and distribution of the product.

The Static Aspect of the Process

The static aspect of the Rational Unified process is described using

Workers : (Analyst, Designer, Architect, Use case author etc.)
Activity : (A unit of work performed by a worker)
Artifact : (A piece of information used, modified or produced by process e.g., Use Case model. Class Diagram, source code etc.)
Workflow : (A sequence of activities that produce a result of observable value)

The Rational Unified Process talks about nine core process Work flows [Jacobson00] which are divided into six core Engineering work flows and three core supporting work flows. They are:

Core engineering work flows

1. Business modeling work flow
2. Requirements work flow
3. Analysis and Design work flow
4. Implementation work flow
5. Test work flow
6. Deployment work flow

Core supporting work flows

7. Project Management work flow
8. Configuration and Change Management work flow
9. Environment work flow

The purpose of each of there work flows is documented in table 6.3 and table 6.4.

Table 6.3

S.No.	Core Engineering work flow	Purpose
1.	Business modeling work flow	• To develop a business object model using business use cases.
2.	Requirements work flow	• To develop a vision document describing users need and high level features of system using a use case model .
3.	Analysis and Design work flow	• To transform requirements into a design model consisting of classes which are collaborating with each other. • Aggregate classes into packages and subsystem.
4.	Implementation workflow	• To implement classes and objects in terms of source files, binaries, executables etc. • To perform unit testing of developed components. • To integrate components into an executable system.
5.	Test workflow	• To verify the interaction among components and also their proper integration. • To ensure all defects are corrected before the deployment of software. • To ensue that all requirements as asked by user are implemented.
6.	Deployment work flow	• To package, install and distribute software. • To conduct beta testing. • To produce external releases of software.

Table 6.4

S.No.	Core supporting work flow	Purpose
1.	Project Management work flow	• To provide a frame work for resolving risks. • To provide guide lines for planning, executing and monitoring project.
2.	Configuration & change Management work flows	• To manage the numerous artifacts produced by different people working on the common project. • To provide guidelines for version control, parallel development etc.
3.	Environment work flow	• To provide the support to development organization for tool selection & acquisition, process customization etc.

6.5 PROBLEM SOLVED USING UML: A CASE STUDY

In this section we will follow UML to analyze and design a software. The problem description is as follows:

A clinic is in the business of providing special dental services to the patients. The clinic has a panel of doctors which visit the clinic on specific days and time. Doctors can register themselves with the clinic by applying to the clinic. If registered they are issued a registration number. The clinic tries to provide the latest techniques to its patients. In order to do so, whenever a new technology or technique for treatment is announced, the management adds it to the clinic. The patients can take appointment for a doctor either telephonically or personally. The clinic staff checks the availability of doctor on requested date and time and if available gives the appointment to the patient. Everyday in the evening, schedule of appointments of different doctors is printed and mailed to them. Similarly appointment schedule of different patients for next day is printed and patients are reminded about their appointment telephonically. A patient can also cancel his appointment. In that case the concerned doctor is informed about the revised schedule.

The patient on scheduled date and time visits the clinic and doctor performs the necessary services on him/her. If the patient is coming to the clinic for the first time, clinic registers information about the patient and allocates a unique registration code for future reference. The doctors are also allowed to access the system for retrieving the patients records in order to check the treatment given to them on their past visits. After giving the required treatment/services to the patient, the doctor records the same and medicines prescribed etc. in a format given by the clinic and gives back the same to the clinic. This is required by the clinic to update patients visit records for future reference. In case patient is asked by the doctor to come for checkup, next appointment is given to the patient and is also recorded in the visit details. Patient also deposits the consultation fee and charges for other services at the reception and staff member issues a receipt to the patient. Whenever a new technique or technology is introduced at the clinic, details are added to the system and patients are also informed by post. Every 15 days a report is printed about the payment to be made to the doctors. Once the payment is made to the doctor details are also added to the system. Develop a class diagram using UML to implement these requirements.

Solution

(a) First step to solve the problem is develop an initial business model using use case diagram. This is done in requirement work flow part of RUP. If we go through the requirements, we can see that the clinic has following main business activities

- Register the personal details of a doctor
- Register the personal details of a patient
- Schedule an appointment

- Cancel the appointment
- Update patient visit records
- Generate reports
- Compute doctor payment
- Update patient payment records
- Retrieve patient information
- Add new services/techniques

The person in this case who is going to use the system is an employee of the clinic or the doctor. Additionally the doctors and patients initiate the use cases and play the roles of actors. They are the ones who provide useful data which is entered into the system by the clinic employees. Therefore the initial business model using the use case diagram is shown in figure 6.31.

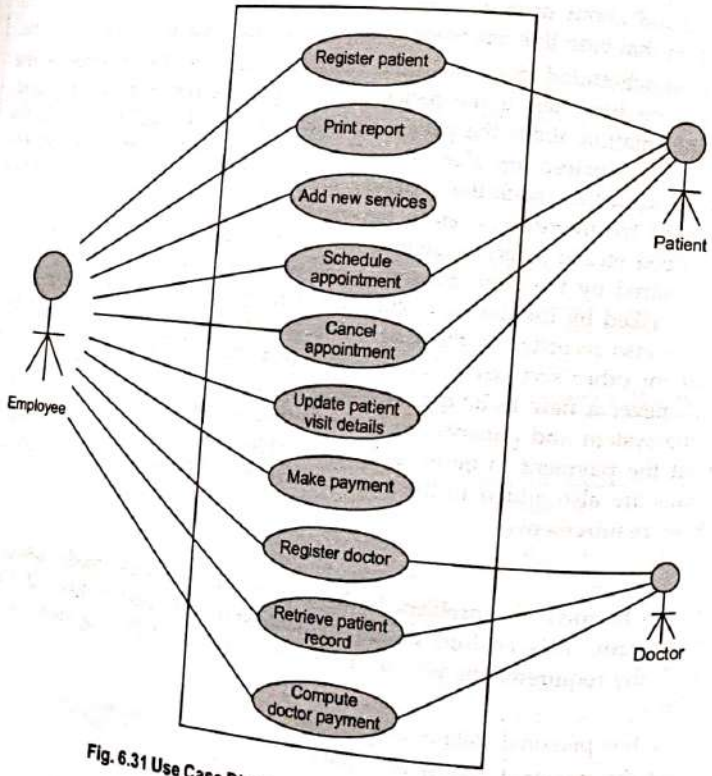


Fig. 6.31 Use Case Diagram Showing the Initial Business Model

Initial use case descriptions are shown in Fig 6.32 to Fig. 6.41

Name	:	Schedule appointment
Purpose	:	Enables the clinic to give appointment to patient.
Description	:	The system inputs the date and time of appointment as asked by the patient and confirms the appointment after checking doctor's schedule.

Fig. 6.32

Name	:	Cancel appointment
Purpose	:	Enables the clinic to cancel the appointment of the patient.
Description	:	The system cancels the appointment of the patient and updates appointment schedule.

Fig. 6.33

Name	:	Add new services
Purpose	:	Enables the clinic to update information about new services added to clinic for treating patients.
Description	:	The system adds the name and details of new services offered by the clinic.

Fig. 6.34

Name	:	Update patients visit record
Purpose	:	Enables the clinic to update the services performed by the doctor on patient on a visit.
Description	:	The system inputs the services performed by doctor on the patient on a visit and updates patients record.

Fig. 6.35

Name	:	Make payment
Purpose	:	Enables the clinic to update payment record.
Description	:	The system inputs the details of payment made by the patient & updates payment record.

Fig. 6.36

Name	:	Print report
Purpose	:	Enables the system to print different reports.
Description	:	The system enters the details and outputs the corresponding reports as desired by the user.

Fig. 6.37

Name	:	Retrieve patient record
Purpose	:	Enables the doctor to retrieve the patient's record about his past visits.
Description	:	The system enters the patient name or registration code and displays the past visit records on the screen.

Fig. 6.38

Name	:	Register patient
Purpose	:	Enables the system to input the personal details of the patient.
Description	:	The system enters the personal details of the patient and generates a unique registration code.

Fig. 6.39

Name	:	Register a doctor
Purpose	:	Enables the system to input the personal details of the doctor.
Description	:	The system enters the personal details of the doctor and generates a unique registration code.

Fig. 6.40

Name	:	Compute doctor payment
Purpose	:	Enables the system to compute the payment to be made to the doctor.
Description	:	The system enters the registration number of the doctor and the time period and calculates the payment to be made for the specified period to the doctor.

Fig. 6.41

(b) Next step is to analyze the requirements which we have captured in step (a). This can be done by refining and structuring there requirements in order to have a precise understanding of the requirements. This also helps the analyst to structure the architecture of the system also. If we look at Rational Unified Process, these activities are part of Analysis and Design workflow. In the next iteration therefore use cases can be described in detail to reflect these detailed specifications as shown below in Fig. 6.42 to Fig. 6.51.

Name	:	**Register a doctor**
Author	:	S. Sabharwal
Purpose	:	Enables the system to input the personal details of the doctor.
Assumption	:	Use case starts on demand.
Description	:	1. User enters the details of doctor to be registered. They are: • First name of doctor • Last name of doctor • Address • Contract number • Mobile • Email • Specialization 1 • Specialization 2 • Specialization 3 • Visit days • Time slot 2. System responds by storing the details and allocating a registration code to the doctor.
Post conditions:		1. On successful completion doctor database is updated and a registration code is displayed. 2. In case of error, message is displayed.
Use case termination:		1. Any time user can exit the use case by pressing the cancel key 2. Use case may time out.

Fig. 6.42

Name	:	**Schedule appointment**
Author	:	S. Sabharwal
Purpose	:	Enables clinic staff to give appointment to patient.
Assumption	:	Use case starts on demand.
Description	:	

a. User enters the doctor's name for appointment.
b. The system displays the appointment schedule on screen.
c. User informs about the available dates to the patient.
d. Patient confirms the date for appointment.
e. User fixes the appointment and updates the appointment schedule.
f. In case patient is not clear about doctor, he provides the name of services he wants to avail.
g. User enters the services details and in response the system provides the name of the doctors & their visit days.
h. Patient selects one of the doctors and confirms the appointment date.
i. User updates the appointment schedule.

Post conditions: 1. On successful completion appointment schedule is updated.
2. In case of error message is displayed.

Use case termination: 1. Any time user can exit the use case by pressing the cancel key
2. Use case may time out

Fig. 6.43

Name	:	**Cancel appointment**
Author	:	S. Sabharwal
Purpose	:	Enables clinic staff to cancel the appointment given to the patient.
Assumption	:	Use case starts on demand.
Description	:	

1. User enters the patient name or registration code and date.
2. System responds by canceling the appointment and updating the appointment schedule if date is correct.
3. In case wrong information is given system prints error message.

Post conditions: 1. On successful completion appointment schedule is updated.
2. In case of error message is displayed.

Use case termination: 1. Any time user can exit the use case by pressing the cancel key
2. Use case may time out.
3. User presses the exit key.

Fig. 6.44

Name	:	**Register a patient**
Author	:	S. Sabharwal
Purpose	:	Enables clinic staff to input the personal details of a new patient
Assumption	:	Use case starts on demand.
Description	:	

1. If patient has come for the first time user provides following details
   - Name of patient
   - Date of birth
   - Address
   - Doctor name
   - Telephone number
   - Disease description
   - Mobile
   - e-mail
2. System checks the patient records.
3. It not found it stores the details and generates a unique patient registration number
4. It found system displays the massage "Entry with given name and Telephone no. exists" and exits.

Post conditions: 1 On successful completion patient data is updated and patient registration code is displayed.
2. In case of error message is displayed.
3. In case record already exists system displays the massage "Entry with given name and Telephone no. exists".

Use case termination: 1. Any time user can exit the use case by pressing the cancel key.
2. Use case may time out.

Fig. 6.45

Name	:	*Retrieve patient record*
Author	:	S. Sabharwal
Purpose	:	Enables doctor to retrieve patient record.
Assumption	:	Use case starts on demand.
Pre-condition	:	Provide a valid patient code.
Description	:	

1. Doctor enters the patient code.
2. The system displays the details of services provided to the patient in his last visits.

Post conditions:
1. On successful completion patient information is displayed on the screen.
2. In case of error message is displayed.

Use case termination:
1. Any time user can exit the use case by pressing the cancel key.
2. Use case may time out.
3. User presses the exit key.

Fig. 6.46

Name	:	**Add new services**
Author	:	S. Sabharwal
Purpose	:	Updates the information about new services being added to the clinic.
Assumption	:	Use case starts on demand
Description	:	

1. User enters the following details about new services
   - Name of service
   - Brief description
   - Charges
   - Number of sittings required
2. The system updates the services database and confirms the same.

Post conditions:
1. On successful completion services database is updated.
2. In case of any error, message is displayed

Use case termination:
1. Any time user can exit the use case by pressing the cancel key.
2. Use case may time out.
3. User presses the exit key.

Fig. 6.47

	:	**Calculate payment to be made to a doctor.**
Name	:	S. Sabharwal
Author	:	Calculates the payment to be made to a doctor for a specific period.
Purpose	:	Use case starts on demand.
Assumption	:	
Description	:	

(*i*) The user enters the starting date and end date.
(*ii*) The system retrieves the payment details for the doctor.
(*iii*) The system calculates the payment to be made to doctor.
(*iv*) The system displays and prints the payment details date wise and also the total payment to made to the doctor for the period entered.

Post conditions :
1. On successful completion doctor database and payment details are updated.
2. In case of any error message is displayed.

Use case termination:
1. Any time user can exit the use case by pressing the cancel key
2. Use case may time out.
3. User presses the exit key.

Fig. 6.48

Name	:	**Update payment**
Author	:	S. Sabharwal
Purpose	:	Enables clinic to update the payment made by the patient
Assumption	:	Use case starts on demand
Description	:	

1. The user enters the following details
   - Patient code
   - Doctor name
   - Payment
   - Date
   - Type of service
2. The system updates the payment details and acknowledges the same by issuing a receipt to the patient.

Post conditions:
1. On successful completion payment database is updated and a receipt is printed.
2. In case of any error message is displayed

Use case termination:
1. Any time user can exit the use case by pressing the cancel key
2. Use case may time out.
3. User presses the exit key.

Fig. 6.49

Name	:	Update patient record
Author	:	S. Sabharwal
Purpose	:	Enables the clinic to update the patient visit record for the services performed.
Pre condition	:	Provide a valid patient code.
Assumption	:	Use case starts on demand
Description	:	

(i) The user enters patient code to the system.

(ii) The system retrieves the information about patient.

(iii) The user enters following information:

- Date of visit
- Doctor name
- Services performed
- Details of test results
- Medicines prescribed
- Next date of visit if required
- Services to be performed on next visit

(iv) The system updates the information and acknowledge the same

Post conditions:
1. On successful completion patient database is updated.
2. In case of any error message is displayed

Use case termination:
1. Any time user can exit the use case by pressing the cancel key
2. Use case may time out.
3. User presses the exit key.

Fig. 6.50

Name	:	Print report
Author	:	S. Sabharwal
Purpose	:	Enables clinic to print different reports.
Assumption	:	Use case starts on demand
Description	:	Following reports are generated on demand:

Doctor appointment schedule

1. The user enters the date for which report is required.
2. The system displays the report for different doctors, who will be visiting the clinic on that date.

Contd...

3. The report has information about time, doctors and patients name. Also if some specific treatment is to be given that is also mentioned.

Patient appointment schedule

1. The user enters the date for which report is required
2. The system displays the reports having following details Patient name, Doctor name and time of appointment.

Doctor payment details

1. The user enter the starting date and end date.
2. The system retrieves the payment details for each doctor.
3. The system calculates the payment to be made to the doctor.
4. The system displays and prints the payment details date wise and also the total payment for each doctor.

Post conditions:
1. On successful completion required reports are displayed on screen and printed.
2. In case of any error message is displayed.

Use case termination:
1. Any time user can exit the use case by pressing the cancel key.
2. Use case may time out.
3. User presses the exit key.

Fig. 6.51

Similarly a number of other reports can be printed. Readers are advised to identify few more reports and write the use case details.

The Rational Unified Process describes the process of identifying requirements as a iterative process, therefore if we go through the requirements we will find that we have over looked few use cases.

Consider the sentence in the use case description of "Print Doctor payment detail" "The system calculates the payment to be made to doctor." It means that information system must have a use case "Compute doctor payment" connected to use case "Print doctor payment details" through <<include>> dependency.

Similarly the sentence in use case Schedule Appointment "The system checks the appointment schedule and updates the appointment schedule" uses the use case "Check Appointment schedule" and "update Appointment Schedule".

In this manner we can go through the requirements and identity the new use cases. The modified use case diagram is shown in Fig. 6.52.

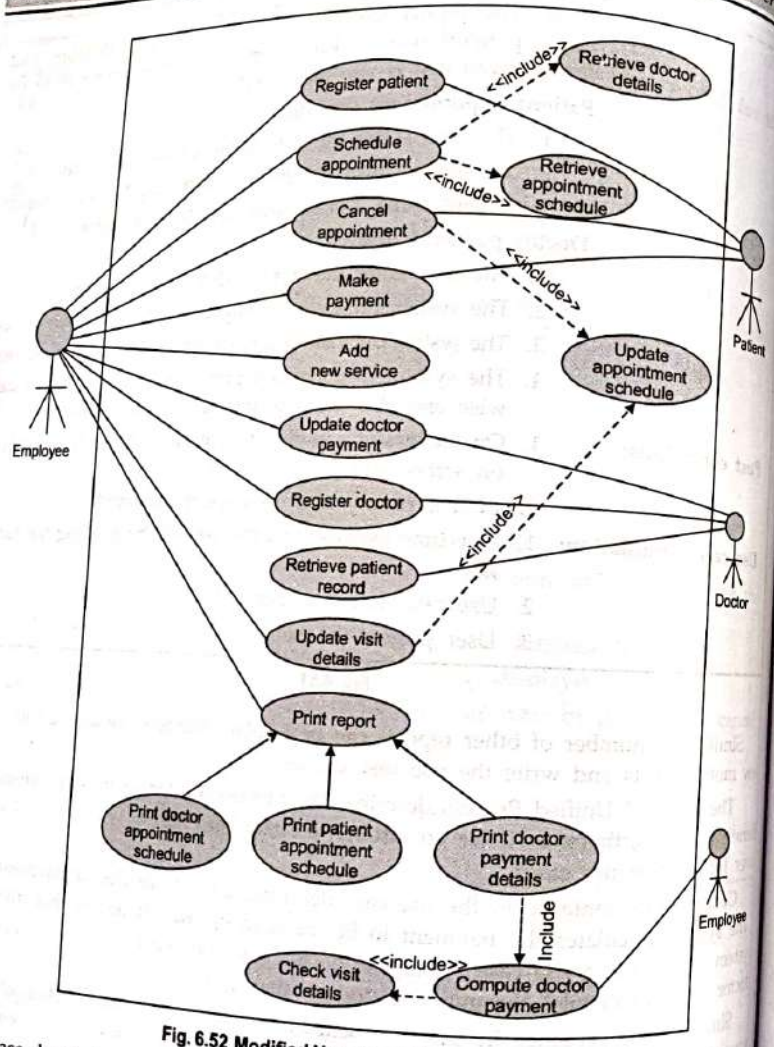


Fig. 6.52 Modified Use Case Diagram

Use case description part illustrates in detail, the way user expects to interact with the system when user invokes the use case. Each instance of the use case is called a scenario. Scenarios give detailed examination of every possible outcome of use case. It can also be defined as a single logical path through the use case, showing one possible outcome. In case of complex use cases, some times we need some way to check the

completeness of the use case. Identification of scenario here play very important role and also help in design of test cases at early stages of software development life cycle.

Activity diagrams help to represent use case descriptions visually and are very helpful in the case of complex use cases. Once the actively diagrams are drawn, the identification of scenarios also becomes easier. The activity diagrams for the use cases cancel Appointment and Schedule Appointment is shown in Fig. 6.53 and 6.54. In the similar manner activity diagrams for other use cases can be drawn. One possible scenario in the activity diagram of "Cancel Appointment" use case is marked in Fig. 6.53.

By the time, requirements workflow is complete user's requirements are refined and are quite clear. Next step is extraction of classes from use case descriptions. As proposed by Unified process this step is also performed during analysis workflow.

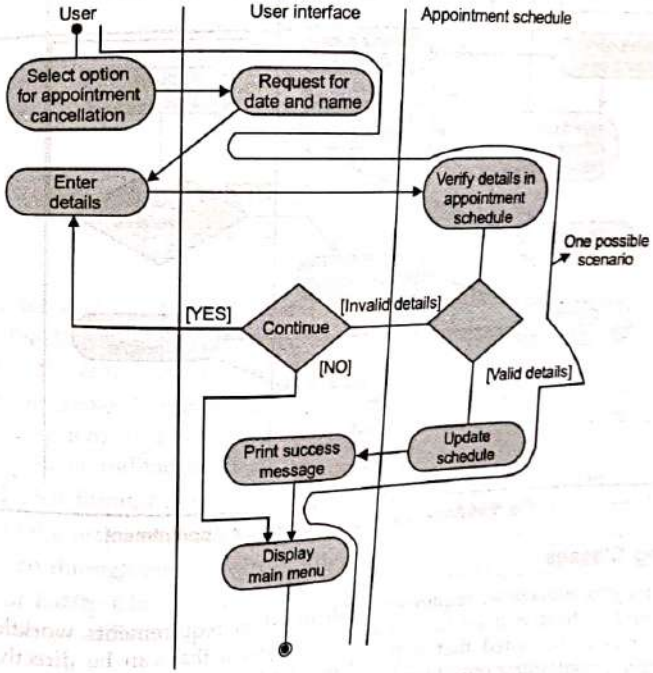


Fig. 6.53 Activity Diagram for Cancel Appointment

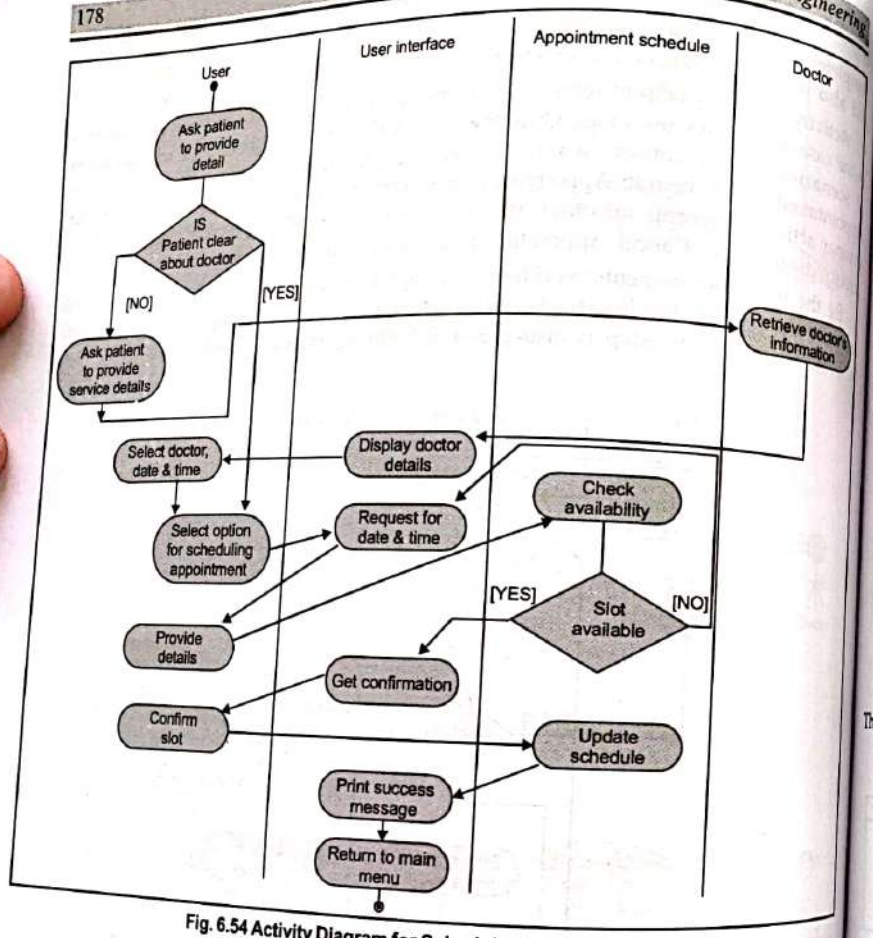


Fig. 6.54 Activity Diagram for Schedule Appointment

Extracting Classes

During analysis workflow, requirements identified in requirements workflow are used as input and output is a set of classes and subsystem that can be directly mapped to software. It is to be noted that at this stage, classes are represented at very high level of abstraction, containing only high level attributes and relationships between the classes. A class can be of three types

1. Entity class
2. Boundary class
3. Control class.

Entity Class: Entity classes are used to model real world objects that are long lived. In this problem patient class and doctor class are examples of entity class because information about patients and doctors has to be stored in the information system.

Boundary Class: Boundary classes are used to model the interaction between the users of the system i.e., actors and the information system itself. These classes are used to support input and output in the system. Examples are user interface class, classes to support interfaces with devices like printer, window etc. Hence in the example of clinic system we can have boundary classes to take input and to print different reports.

Control Class: Control classes are used to model the dynamics of the system and are used to model complex computations, sequence of events, transactions and controlling other classes. In the clinic system we can identify a control class "Compute Payment" to calculate the payment to be made to the doctor.

The notations for these three types of classes is given in Fig. 6.55

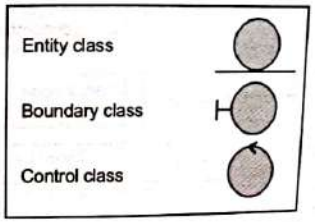


Fig. 6.55

The Procedure for extracting initial classes from Use cases is as following:

1. Read the Use case descriptions. Noun's are normally taken as entity classes. Identity the entity classes and establish relationship among them.
2. Draw the dynamic modal (state chant diagram, interaction diagrams) for each class or system as a whole to identity the operations that can be performed to the system or individual class
3. Model each input/output screen, printed report as a boundary class.
4. Identify the major computations to be done and represent them as control classes.

Now let us go through the Use case descriptions and identify different types of classes.

Identification of Entity Classes: After going through the Use case descriptions following entity classes are identified:

- Patient
- Doctor
- Service
- Visit
- Appointment
- Payment

The initial set of classes and their relationships are shown in Fig. 6.56. In this diagram the attributes are also mentioned.

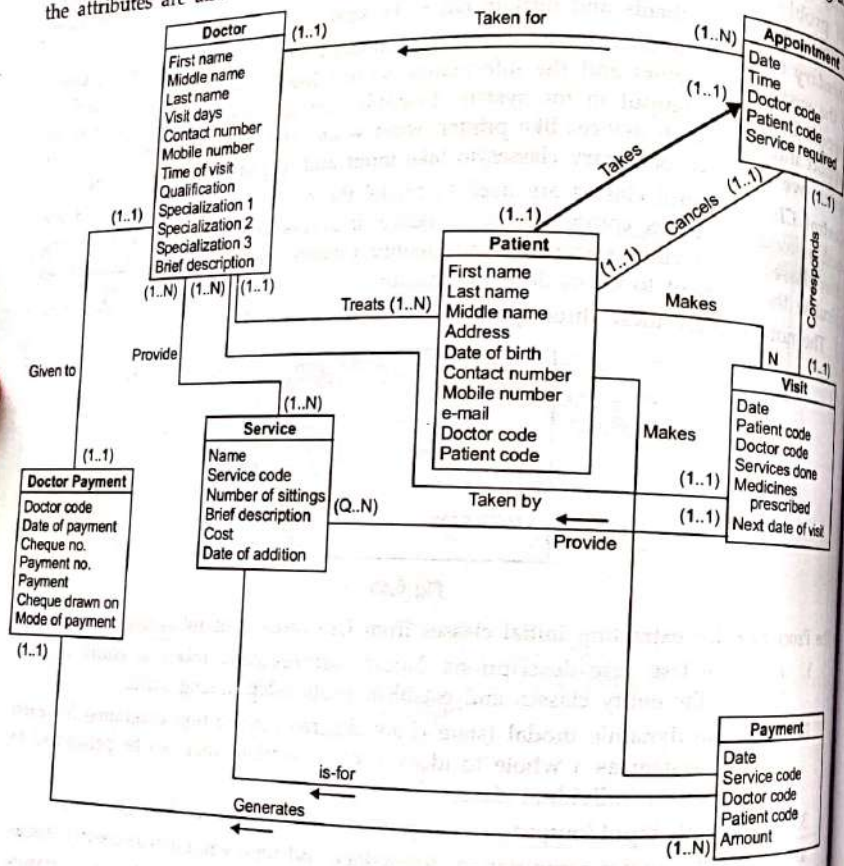


Fig. 6.56 Class Diagram

Identification of Boundary Classes – As boundary classes support interaction between users and the information system following can be modeled as boundary classes

- User Interface class for input/output screen
- Date outpolling class

In this case he can broadly think of two screens – one for using all Use cases i.e. different features of system and other for generating reports. Hence our user interface screen will some what look like as shown in Fig 6.57.

EYE CLINIC

MAIN MENU

1. Register Doctor
2. Register Patient
3. Schedule Appointment
4. Cancel Appointment
5. Deposit Payment
6. Retrieve Patient Record
7. Retrieve Doctor details
8. Update Visit Record
9. Print Report
10. Update Doctor Payment Record

Please Enter your choice (0 to Exit)_ _ _

Fig. 6.57

Second Screen is for generating different Reports and is called Report generation interface. The initial version of Screen will be as shown in Fig. 6.58.

EYE CLINIC

MAIN MENU

1. Print Doctor Appointment Schedule
2. Print Patient Appointment Schedule
3. Print Doctor Payment details
4. Print daily Appointment details
5. Main Menu

Enter Your Choice.....

Fig. 6.58

Hence we see that the following classes are finally identified.

Entity Classes – Doctor, Appointment, Patient, Visit, Service, Payment, Doctor_payment

Boundary Class – User Interface Class, Report interface class, Print Patient appointment report class, Print Doctor appointment report class, Print payment report class.

Control class – Compute_payment class, Schedule_appointment class.

In addition to these classes there will be a class which will monitor the overall working of the program.

Next step is to identify the operations that are supported by each of these classes or to define the interface of each of these classes. This can be done by drawing the sequence and the collaboration diagrams for each use case scenario. Messages passed to an object are part of the object interface which responds to these messages. In addition to these there are operations to create and delete instances of classes, to retrieve, set and modify the attributes of a class. The sequence diagrams and the collaboration diagrams for different use case scenarios are shown in Fig. 6.59 to Fig. 6.74. The visit class with attributes and operations (as identified from dynamic view) is shown in Fig. 6.76. The readers are advised to complete the description of rest of the classes.

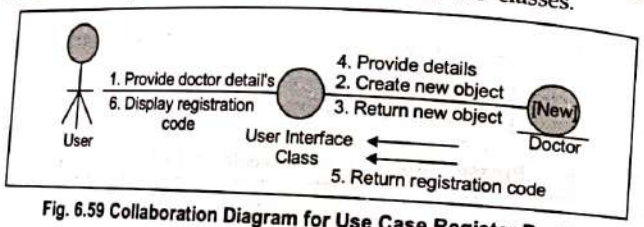


Fig. 6.59 Collaboration Diagram for Use Case Register Doctor

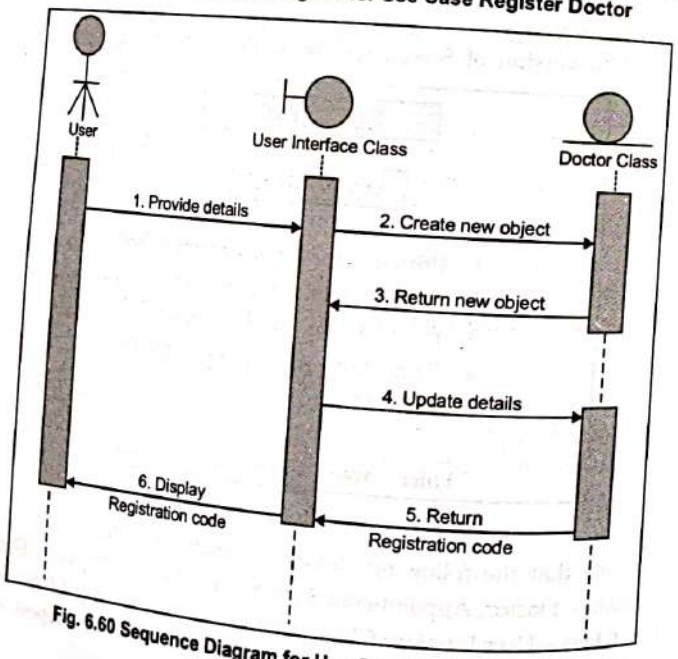


Fig. 6.60 Sequence Diagram for Use Case Register Doctor

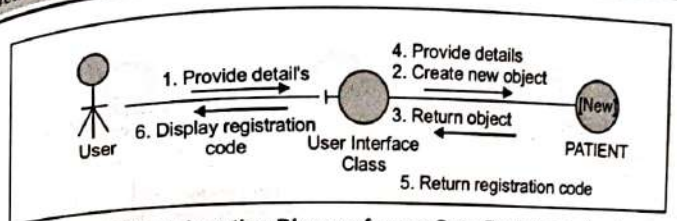


Fig. 6.61 Collaboration Diagram for use Case Register Patient

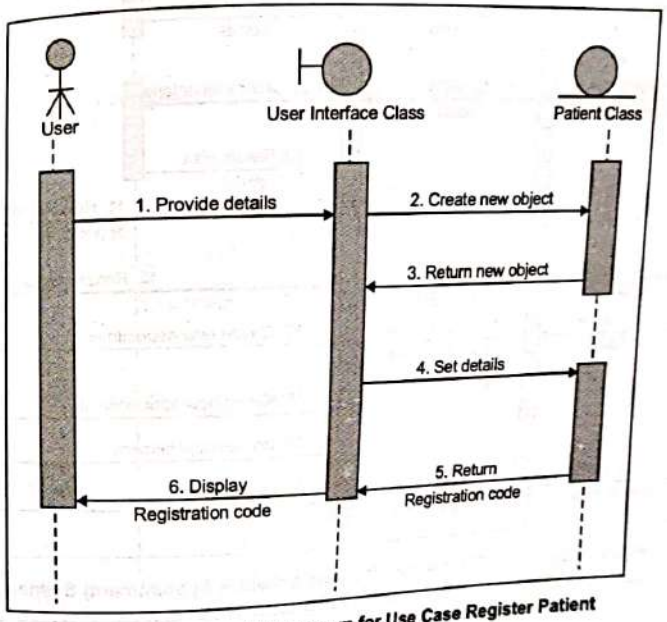


Fig. 6.62 Collaboration Diagram for Use Case Register Patient

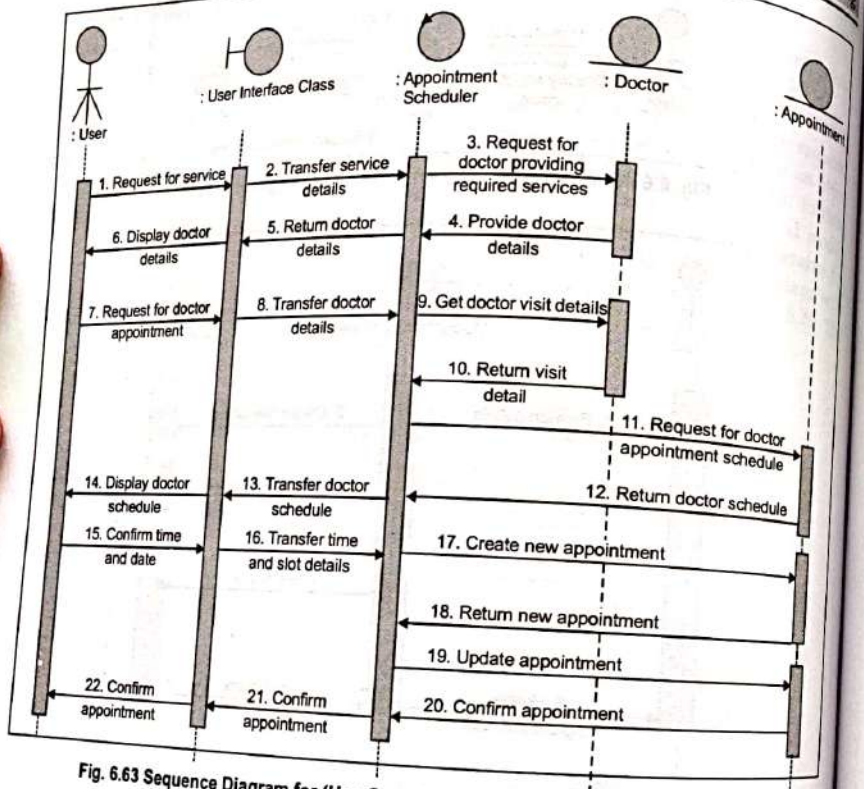


Fig. 6.63 Sequence Diagram for (Use Case Schedule Appointment) Scenario 1

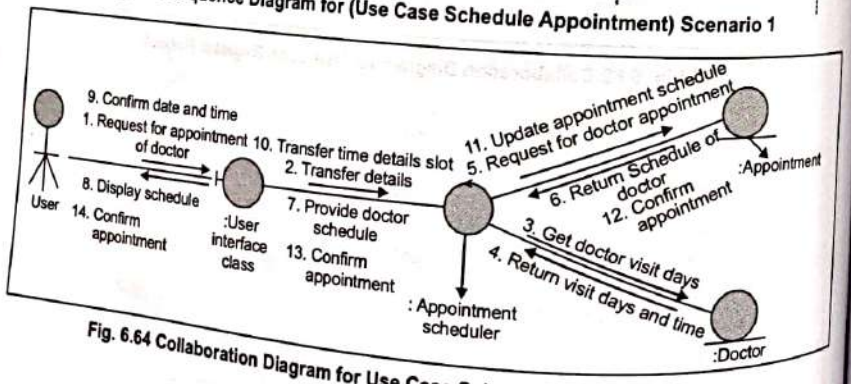


Fig. 6.64 Collaboration Diagram for Use Case Schedule Appointment Scenario-2

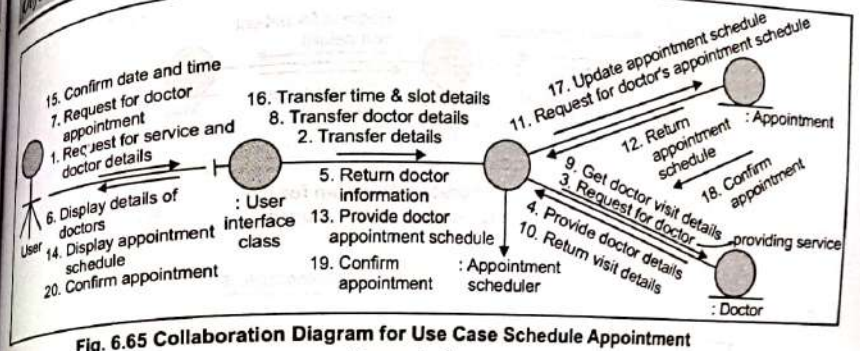


Fig. 6.65 Collaboration Diagram for Use Case Schedule Appointment Scenario-1

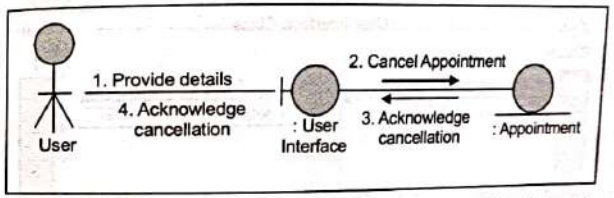


Fig. 6.66 Collaboration Diagram for Use Case Cancel Appointment

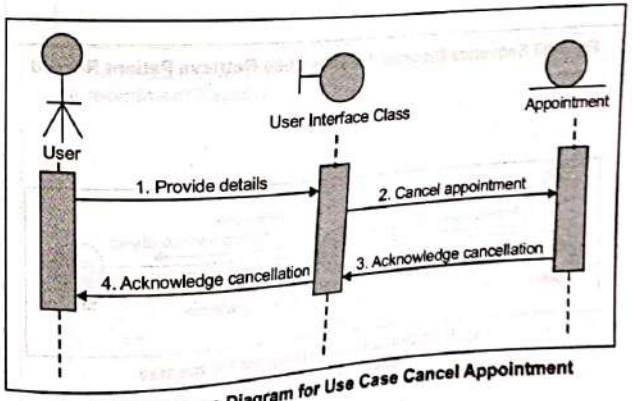


Fig. 6.67 Sequence Diagram for Use Case Cancel Appointment

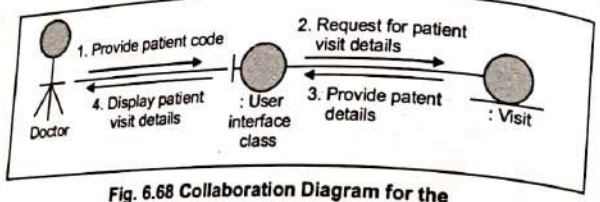


Fig. 6.68 Collaboration Diagram for the Use Case Retrieve Patient Record

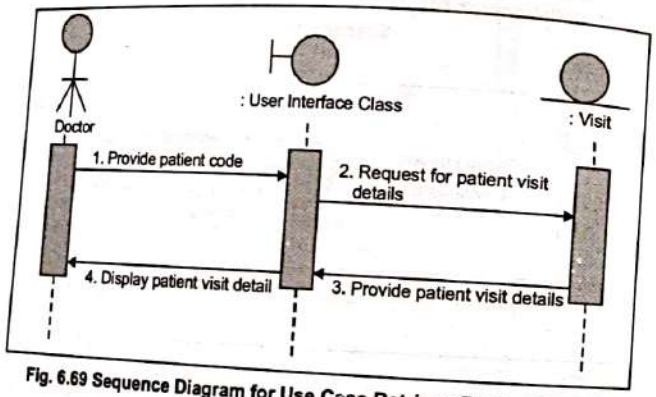


Fig. 6.69 Sequence Diagram for Use Case Retrieve Patient Record

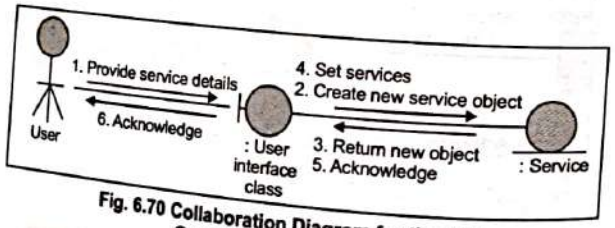


Fig. 6.70 Collaboration Diagram for the Use Case Add New Services

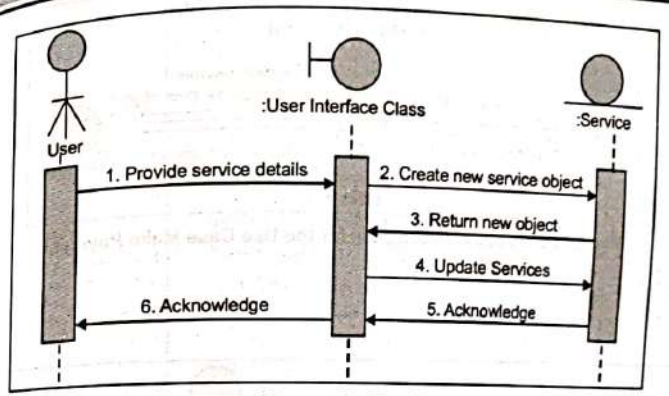


Fig. 6.71 Sequence Diagram for Use Case Add New Services

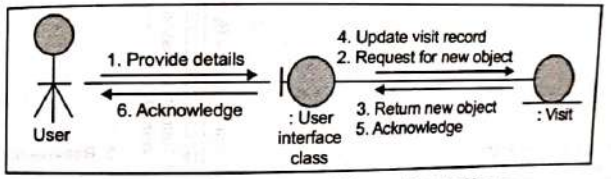


Fig. 6.72 Collaboration Diagram for the Use Case Update Patient Visit Record

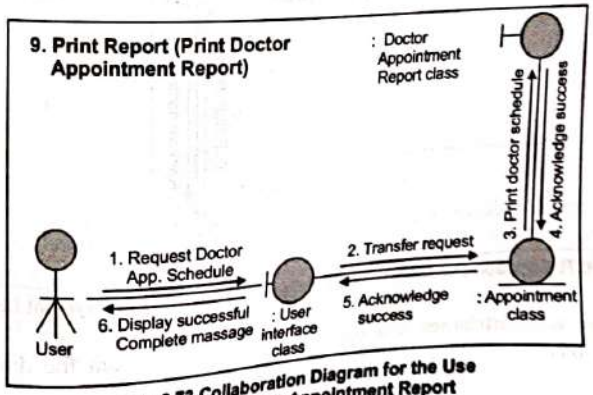


Fig. 6.73 Collaboration Diagram for the Use Case Print Doctor Appointment Report

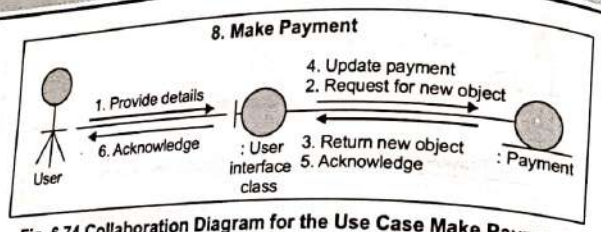


Fig. 6.74 Collaboration Diagram for the Use Case Make Payment

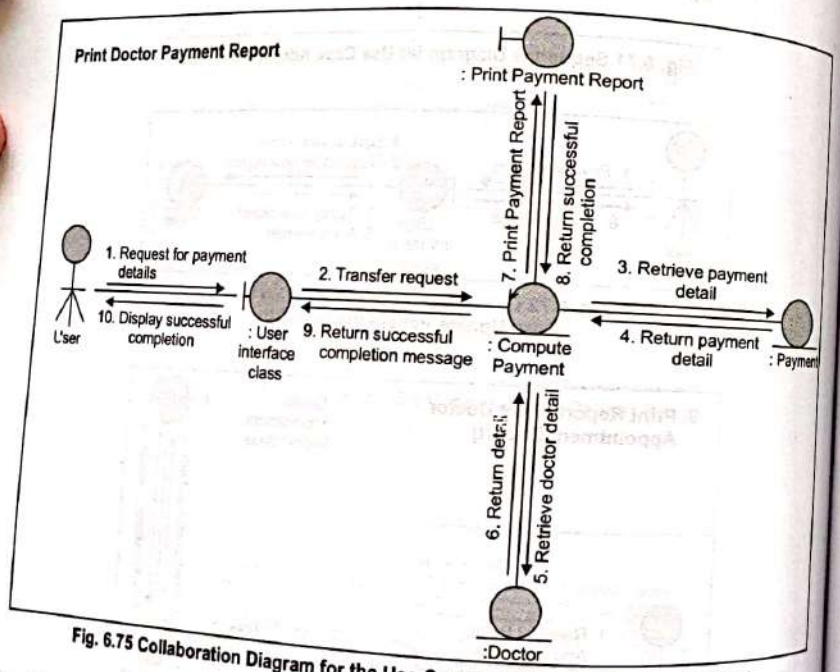


Fig. 6.75 Collaboration Diagram for the Use Case Print Doctor Payment Report

The Visit Class with attributes and operations as identified from the dynamic view is shown in Fig. 6.76.

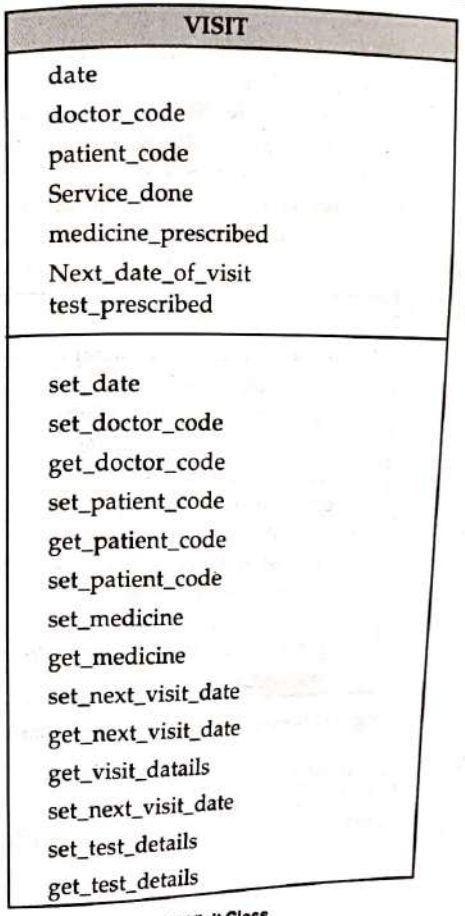


Fig. 6.76 Visit Class

Readers are adviced to identify and draw other classes also in the similar manner. In the design workflow, more details are added to the class in terms of data structures to be used with class attributis and the operations are also described in detail. A detailed description of Visit class is given in Fig. 6.77.

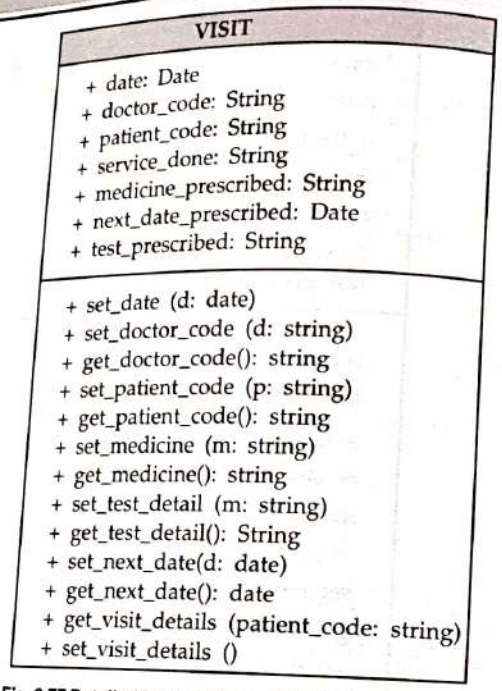


Fig. 6.77 Detailed Descriptions of Visit Class

In the similar fashion, readers can drow the detalied description of other classes. Once done, then set of classes can be transformed into C++/JAVA or any other object oriented programming language code to get a working system.

SUMMARY

- OORA is a technique for modeling problem domain in terms of objects and interactions among objects.
- A number of object oriented analysis and design methodologies like OOA, OMT, UML etc., are proposed but UML has almost become industry standard for developing software blueprints.
- UML makes use of 9 diagrams to model static, functional and dynamic view of the application domain.
- Class diagram is the most important diagram used for generating code and for reverse engineering.
- Other diagrams of UML i.e., Use Case diagram, Activity diagram, Collaboration diagrams, Sequence diagram and State chart diagram help in understanding the

system and hence help in refining the class diagram.

- The operations of all the classes if combined, represent the functional requirements of the system.
- Transition from analysis to design using object oriented programming languages is straight forward as far as object oriented requirement analysis techniques are concerned.
- The main concepts of interest in OORA are classes, objects, inheritance, polymorphism and information hiding.
- Rational Unified Process is a software engineering process which aims at producing high quality software using UML that meets the needs of its end uses within a predictable schedule and budget.
- The Rational Unified Process talks about nine core process Work flows which are divided into six core Engineering work flows and three core supporting work flows.

REVIEW PROBLEMS

1. List the similarities and dissimilarities between structured analysis and object oriented requirements analysis techniques.
2. How is a class different from an entity?
3. What is the relation between an object and a class?
4. Explain in detail basic concepts of OORA.
5. List 9 diagrams used in UML. Also specify for modeling which view each of these diagrams is used.
6. What is the purpose of drawing Use Case diagram?
7. What are the main concepts used for drawing Use Case diagram?
8. Explain the different types of classes proposed by UML.
9. What is the purpose of an Activity diagram? For the example discussed in the chapter draw an Activity diagram for the Use Case Print Report.
10. What is the difference between a Sequence diagram and a Collaboration diagram? Illustrate using suitable example.
11. What is the use of Statechart diagram?
12. Write a note on Class diagram. For a library system draw a class diagram.
13. Differentiate between a Package, Component diagram and a deployment diagram.
14. For the requirements given in Q.26, Q. 28 and Q. 30 of chapter 5 draw
    (a) Use Case diagram
    (b) Class diagram
    (c) Activity diagram
    (d) Sequence diagram
    (e) Collaboration diagram

15. Study the object modelling technique proposed by Rumbaugh and describe how it is different from UML.

16. A hotel has several rooms. Each room is identified by its unique room number. The room has its other attributes such as type and status. A set of rooms is under a hospitality manager who manages through a set of room service employees. The hospitality manager is in charge of the room service tasks to be performed. The charges of room services are added to the final bill of the room. There are 5 types of rooms

- Single Bed
- Double Bed
- Double Bed Deluxe
- Suite
- Deluxe-Suite

For each customer a unique id is assigned along with the room number, customer's check-in, check-out date and time. Every room has status as empty, occupied or advance booking. For occupied status, room is assigned the customer_id and for advance booking the name of person, the expected check-in date and time, date and time of advance booking and the advance amount deposited is also recorded.

When a customer is allocated a room, the check-in date and time is recorded for that room. For each customer, attributes such as name, address, Social-Security Number (SSN) and telephone number are also to be recorded. For each room service task, room number, date and time, service number, bill number, contents and description are recorded. At the time of check-out, the check-out date and time is recorded to generate the bill. The bill amount for each customer is recorded along with the mode of payment.

Draw the Use Case diagrams, Class diagram, Sequence diagrams and Collaboration diagrams for the above specifications and identify the final set of classes. Make suitable assumptions if required in the requirements.

17. (a) Starting with the minimum set of requirements draw the initial business model for following problems

(i) Training and placement division of an institute

(ii) On line ordering system for buying clothes

(iii) Inventory management system.

(iv) Art Gallery.

(b) From the initial bushiness model, following the unified process approach, draw the detailed use case diagram, sequence diagrams and collaboration diagrams

(c) Derive the final class diagram.

Chapter 7

SYSTEMS ENGINEERING

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- What is a System
- Characteristics of a System
- Systems Development Life Cycle
- Types of System

7.1 WHAT IS A SYSTEM?

Often people use the words "System" and "Systems approach" but it's meaning is not always clear. In simple English a system is a collection of interrelated elements or components working together for achieving some common objective as shown in Fig. 7.1.

An excellent example to illustrate the system concept is human body consisting of number of systems e.g., digestive system, skeletal system, nervous system, respiratory system etc. Each of these combined together result into proper working of human body. We observe that each of these has well defined fixed boundary and well defined functions to perform. A system further consists of several components. For example digestive system consists of mouth, food pipe, stomach, liver etc., which interact with each other and work together to digest the food. Same is the case while developing systems for some organization and all its aspects i.e., hardware, software, purpose and the people involved must be suitably tackled. But in most of the cases project teams are divided into hardware and software teams and over a period of time very little interaction takes place between the two thus resulting into number of problems and errors.

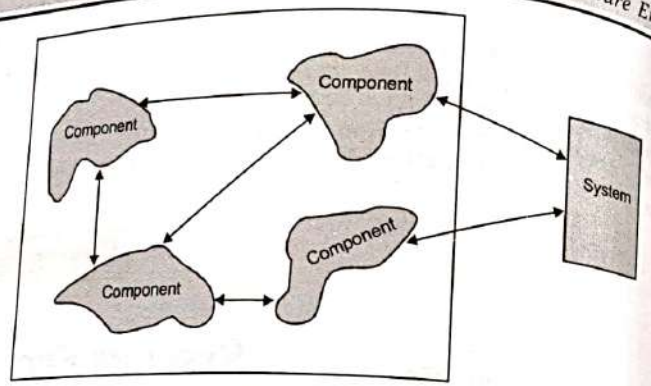


Fig. 7.1 A System

7.2 CHARACTERISTICS OF A SYSTEM

Before discussing further it is important to discuss the characteristics of a syste (Pirbhai94) has described important characteristics of the system as follows:

- Every system has some purpose or set of purposes to achieve.
- Every system has several interrelated components which together fulfill syste intended purpose.
- Every system interacts with the environment through exchange of informatio
- Every system must fit into its operating environment.
- All systems exhibit predictable behavior.
- All systems are processors of information, materials and/or energy.
- All systems come in hierarchies. Therefore system exists to maintain its high level system.
- There are only three types of processors used in building system i.e. hardwa software and people. All of them may or may not be present in a system
- All systems at any one level of system hierarchy form a network while existi as independent peers.

The best way to define the system is to define it from user's perspective whi consists of automated as well as manual components of the system (inclusive of hardwa and software).

7.3 A SYSTEM'S APPROACH

System's Approach through the process of modeling (i.e., building models of differ parts of systems) help the developers/users to understand and develop the system

This approach of developing systems is also called *"Systems Engineering"* by some researchers. According to (Eisner 88) the systems engineering is defined as "an iterative process of top-down synthesis, development and operation of a real world system that satisfies in a near optimal manner the full range of requirements for the system." Absence of systems approach/systems engineering in a project often results into following problems

- Cost overrun
- Missed schedule
- Requirements not met
- Improper integration of subsystems
- Improper working of delivered system
- Maintenance problems
- Unmanageable systems

The very purpose of system engineering is to ensure an optimum solution which meets all the requirements and provides a balanced development schedule, cost and performance. To achieve this research community has also proposed to develop system requirements model and system architecture model for the entire system, subsystems, and finally allocated the system requirements to hardware, software and people respectively.

7.4 SYSTEM DEVELOPMENT LIFE CYCLE

A typical system development life cycle illustrates the different phases of a system development. The main phases of a system development life cycle are

1. System Analysis
2. System Architecture Design
3. Subsystem Analysis/Design
   - Hardware analysis & design
   - Software analysis and design
4. System Implementation
5. System Integration
6. System Testing
7. System Installation
8. System Maintenance

(a) **System Analysis** – System analysis is the most important activity of system engineering and concerns with identification of goals/objectives of the system. From the goals, requirements and constraints are derived and documented. Classical system engineering also proposes to develop system requirements model and enhanced requirements model of the system to be developed using the techniques discussed in chapter 4. All these requirements are documented in

system specification document. Feasibility analysis of the project/system to be developed is also done to avoid losses. Formally feasibility can be defined as the *ensure* of how beneficial or practical the development of an system will be to an organization (Jeffrey00) and feasibility analysis is the process used to measure the feasibility. Feasibility can be broadly classified as

- Technical feasibility
- Operational feasibility
- Economic feasibility
- Schedule feasibility
- Legal feasibility

Technical Feasibility – Technical feasibility focuses on technology related issues, practicality of available technical solution, risks involved and resources (including human resources and expertise) available. It also accesses that whether the available technology is mature enough to meet the system needs.

Operational Feasibility – Operational feasibility is people oriented and focuses on evaluating whether a system will work properly in the organization as well as the feedback of end users about the problem. All the issues like performance, efficiency, providing information and services to the users, security etc., of the system are covered by operational feasibility. Usability analysis to test system interface is also performed on the proposed system's prototype.

Economic Feasibility – Economic feasibility of the system is the measure of cost effectiveness of the system and includes cost benefit analysis, cost involved, income generated etc. While doing cost benefit analysis, fixed costs (cost of developing the system) and costs for operating the system are also taken into account. Major factors contributing to cost calculation are salaries, training, cost of hardware and software, lease payments, cost of other equipments like floppy disks, CDs, printer paper, overhead costs (e.g., maintenance, telephone/Fax services) etc.

Schedule Feasibility – Schedule feasibility analysis concerns building the proper development schedule of the system based on the deadline given by the user. Schedules can be represented using Gantt Chart or a PERT Chart.

Legal Feasibility – Legal feasibility focuses on dealing with legal issues like infringement, contracts etc.

(b) **System Architecture Design** – This stage of system development life cycle is concerned with designing the architecture of the complete system which is a top-down, high level design of the system. One or more architectures can be proposed at this stage. Architecture templates as shown in Fig 7.2 can also be used at this stage.

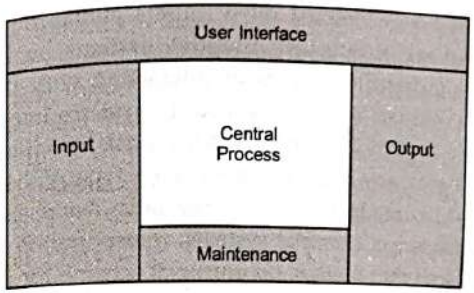


Fig. 7.2 Architecture Template

Depending upon the requirements, modules/subsystems are also segregated into hardware and software modules/subsystems.

(c) **Subsystem Analysis/Design** – At this stage now the focus is on each of the subsystems (related to hardware/software). The requirements of each of these subsystems (hardware or software) are rebuild and enhanced. Design documents for each of them is produced so that they can be passed to the developers. Software engineering therefore is only a part of the system development life cycle.

(d) **System Implementation** – At this stage different components for the system are build as per design specification

(e) **System Integration and Testing** – System integration involves combining smaller parts into larger parts systematically and occurs both for hardware as well as software. This ultimately results into a full system in which hardware, software and people are integrated to meet the system requirements. Each integration activity is also followed by testing so as to verify the characteristics of integrated component. Proper testing plans are also generated to support the testing activity.

(f) **System Installation and Maintenance** – Final System is installed at its operation location and tests are conducted to ensure its proper working. Thereafter maintenance of the system which is a continuous and life long activity is done. A small team of hardware and software engineers are trained to handle the maintenance activity.

7.5 TYPES OF SYSTEMS

There are may ways to classify the systems. According to (Pirbhai94) systems can be classified into

- Mechanistic Systems
- Adaptive Systems

Mechanistic systems take a defined set of inputs and through a transform proce... generate a well defined set of outputs. Adaptive systems on the other hand are ... systems which by monitoring the outputs or interacting with the environment chang... the transform process. (Yourdan 89) has classified the systems into two categories: Natura... Systems and man-made systems. Natural Systems exist in nature and serve their own... purpose. Man does not play any role in creating such type of systems e.g. Solar System... Molecular System etc. Man made systems on the other hand are constructed, operated... and maintained by humans e.g. financial systems, transportation systems, manufacturing... systems etc. It is to be seen that there is no universal scheme for classification of systems...

Computers are an integral part of most of these man-made systems and are totally... unavoidable in today's scenario. These system are therefore also called automated systems... Next we will define some of the important Information Systems being used in the... organizations.

(a) **Office Automation Systems** – These type of systems support day to day... processing tasks of a business house and consists of tools like spread sheets... word processors, presentation systems etc. These systems help in increasing the... productivity of individual and business house as a whole by providing improved... work flow and communications among employees of the organizations.

(b) **On-Line Systems** – In these type of systems, inputs to the system are given... directly from where they are created and outputs are also directly returned to... the concerned persons. Terminals through which users interact may be located... at a large distance (hundreds of kms) from the main computer (Yourdan89)... These type of systems directly interact with the people doing transactions/queries...

(c) **Decision Support Systems** – These type of systems are used by the management... to take decisions related to business. These systems help the managers, analysts... etc. by providing information, models, data analysis tools etc. in order to take... suitable decisions. These systems are therefore are not used in day to day... working of the organization but used on adhoc basis. For example such type of... systems can be used to analyze the market sales and frame the sale policies... accordingly. Techniques like data-mining are used by these systems to identify... useful patterns in large databases storing information about last so many years...

(d) **Expert Systems** – Expert Systems are the systems which using the techniques of... artificial Intelligence built/capture into them the expertise and knowledge so as... to imitate or simulate the logic or thinking of the experts. In addition to data... and information, rules are built into the system in order to simulate the reasoning... of experts. Special programming languages like LISP, PROLOG etc., are used to... build such type of systems.

(e) **Real Time Systems** – Real Time Systems are the systems which control the environment by quickly generating and returning the results to the environment after receiving input from it. In such type of systems time constraints on actions is very important and system must respond within absolute specified interval of time. Examples of such type of systems are process control systems, data acquisition systems, patient monitoring systems etc. It is important to note that real time systems interact both with the environment and the people.

(f) **Transaction Processing Systems** – These are the systems which collect, store and process data about different transactions taking place in the organizations. Any event taking place in the organization that modifies the existing data or creates new data is called a transaction e.g. an order placed by the customer, withdrawal of cash from bank etc. Transactions are processed as per predefined rules and procedures and data is stored in standard format for later use by other people. These type of systems are designed to respond to business transactions as well as for initiating transactions. These are the most common systems to be used. Transaction processing systems can also be on-line systems.

(g) **Communication Systems** – These type of systems help the people to exchange information efficiently. Tools like teleconferencing, e-mail, voice mail, fax, intranets, extranets etc., are used to build such type of systems.

SUMMARY

- A system is a collection of interrelated elements or components working together for achieving some common goal.
- Each system has well defined boundary and function to perform.
- System engineering is the process of developing system through top down synthesis.
- Absence of systems engineering results into problems like improper requirements, cost overruns, improper schedule etc.
- Software engineering is one of the activities of system development life cycle.
- Feasibility study an important activity of system analysis and design is done to avoid losses and involves technical feasibility, economic feasibility, schedule feasibility, operational feasibility and legal feasibility.
- Different types of systems of interest are—online systems, real time systems, transaction processing systems, expert systems, decision support system etc.

REVIEW PROBLEMS

1. Define a system and its characteristics?
2. What is systems engineering?

3. What are the different stages of system development life cycle?
4. What is feasibility analysis? Why it is important?
5. Explain different types of feasibility.
6. How systems can be classified?
7. Explain different types of systems.
8. List at least 3 examples of
   (a) Transactions processing systems
   (b) Natural systems
   (c) Real time systems.

○○○

Chapter 8

SOFTWARE TESTING

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- What is software testing?
- Cost of errors
- Different stages of software testing process
- Verification techniques
- Validation techniques
- Object oriented testing
- Debugging
- Software testing tools

8.1 INTRODUCTION

The prime objective of testing is to find errors. Though terms like bugs, faults, failures, mistakes, defects etc., are also commonly used, in this book we will be using the term error most of the time. Errors can be in any intermediate work product of software life cycle i.e., SRS, design document, user manuals and program. The word test is derived from the Latin work "testum" which means a pottery vessel used to measure weight. Similarly software testing is done to measure the quality of the software. Whereas quality assurance focuses on the processes that prevent defects from being introduced in the product, testing deals with finding the defects. Proper testing of the software ensures that the software is ready for its use. Although crucial to software quality, software

testing is still an art due to limited understanding of the software engineering principles and is still evolving.

Testing involves operation of a system or application under normal as well as abnormal conditions with real or simulated inputs and evaluating the results. Testing if done early in the software development lifecycle reduces errors in the final product. It is seen that almost 56% of errors are introduced in the requirements specification phase, 27% in design phase, 7% in coding and rest 10% errors are introduced due to other reasons. The software tester should therefore aim for finding the errors as early as possible. Every software project small or big must formally test the software at least once using a suitable test plan. The main limitation of testing is that it can only show the presence of errors but cannot prove the absence of all errors.

8.2 COST OF ERRORS

The cost of the errors in a software are logarithmic in nature i.e., they increase ten times as time increases. The errors that are detected late in the SDLC are the most expensive to fix. The errors which cannot be detected in the earlier stages migrate to the successive stages and can lead to even failure of the software. It has been found that cost of fixing the errors in the early stages is far less as compared to fixing them late in SDLC. It is therefore clear that the testing must be done throughout the SDLC.

8.3 TESTING: SOME DEFINITIONS

Several definitions of testing exist in literature. Some of them are given below.

According to (Myers79) testing is the process of executing a program or system with the intent of finding errors or weakness in a work product.

As per IEEE/ANSI, 1990 [Std 610, 12-1990], software testing is the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspect of the system or component.

Testing is also defined as planning, designing, building, maintaining and executing tests and test environments (Hetzel91).

In all the definitions suggested by researchers, the objective is to discover and correct errors. Testers also while finding errors should be clear that their aim is to target the work product not any individual or team of developers who developed the product. Their main job is to add value to the work product, thereby enhancing its quality. This they achieve by executing the program, reviewing the SRS, examining the internals of program and doing many more other things.

8.4 TESTING TERMINOLOGY

Mistake — A mistake is an action performed by a person leading to incorrect result.

Fault — Fault is the outcome of the mistake. It can be a wrong step, definition etc., in the program. A fault ultimately during the lifetime of software may or may not result into failure.

Failure — Failure is the outcome of fault.

Error — Amount of deviation from the correct result is called error.

Tester — A person whose job is to find fault in the product is called tester.

Testware — A work product produced by software test engineers or tester consisting of checklists, test plans, test cases, test reports, test procedures etc.

Testplan — A document prescribing the approach for testing activities.

Test case — A test case is a set of inputs (preconditions plus actual inputs) and expected outputs (post conditions and actual outputs) for a program under test. A successful test case discovers the errors in the program and consists of following information:

- A unique test case ID to distinguish it from other test cases.
- List of test case items to be tested.
- Input specifications in term of list of names and values for the test case.
- Expected outputs when the test case exercises actions including non-functional qualities.
- Dependency on other test cases.
- Supporting infrastructure in terms of hardware, software etc.

Debugging — It is a systematic review of program text in order to fix bugs in the program.

8.5 TESTING GUIDELINES

Some important guidelines for testing the software are given below:

- Testers while testing the product must have a destructive attitude in order to do effective testing.
- Testing must start the moment requirement analysis phase starts in order to avoid defect migration.
- Both functional as well as non-functional requirements of the software product must be tested.
- As far as possible testing must be supported by automated testing tools.
- Full testing i.e., starting from requirement phase till acceptance testing must be used for critical software.

- Testing should also be conducted by a third party independently for effective results.
- Testware must be properly documented using software test standards and controlled using configuration management system.
- Quantitative assessment of tests and their results must be done.
- An efficient testing process must be used by the organization.
- Testing is never 100% complete.

8.6 TESTING LIFECYCLE

In a normal software development lifecycle, the testing starts after the implementation or coding stage. As discussed earlier testing consists of both verification and validation therefore the activity of testing must start the moment the project is started. The Dotted-U model proposed by Software Development Technologies (Edward00) shown in Fig. 8.1 shows in detail the integration of the software development lifecycle and the testing lifecycle. It is clear from the diagram that for every development phase there is corresponding testing phase in the form of verification or validation. In the rest of the chapter we will be discussing some of the important techniques used during verification and validation of the software.

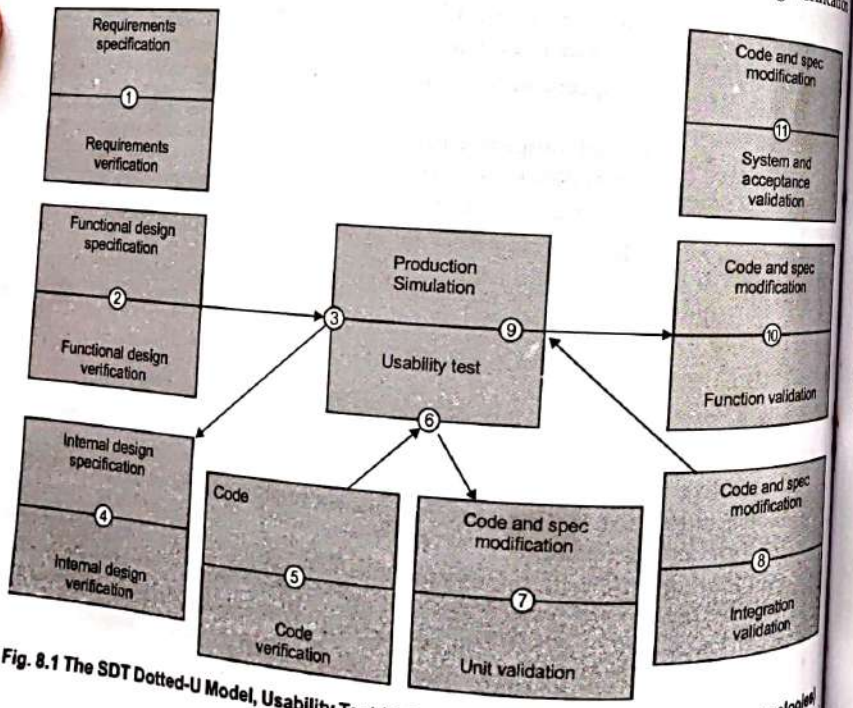


Fig. 8.1 The SDT Dotted-U Model, Usability Test (© 1995, 1996 Software Development Technologies)
[Included with permission]

8.7 TYPES OF TESTING

The aim of testing is not only to test the end product but any other intermediate non-executable work products produced during software development life cycle (SDLC) as well. In order to do so, testing can be broadly divided into two forms:

- Verification Testing
- Validation Testing

Verification Testing according to IEEE/ANSI is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase. Verification testing is used for requirements verification, design verification and code verification.

Validation testing on the other hand involves execution of final end product. It is defined by IEEE/ANSI as the process of evaluating a system or component during or at the end of development process to determine whether it satisfies specified requirements.

Next we discuss the two types of testing in detail.

8.8 BASIC VERIFICATION METHODS

According to (Edward00) verification is a human examination or review of the work product. The activity of review does not involve executing the product. A review according to IEEE Standard Glossary of Software Engineering Terminology is defined as

A review is a process or meeting during which a work product or a set of work products is presented to project personnel, managers, users, customers, or other interested parties for comment or approval.

If conducted properly reviews can find many defects early in the life cycle and can reduce lot of investment later in testing, maintenance and customer support. It is seen that if done properly reviews can find 60-90% of the defects.

Some of the advantages of reviews are:

1. Reviews improve the product quality by finding problems early in life cycle so that they are inexpensive to correct.
2. While testing cannot test all possible paths, with reviews it is possible to follow all the execution paths in the code.
3. As meetings are conducted regularly, people come to know about good points about others work.
4. Reviews help the project managers in maintaining a discipline as they get constructive, meaningful and measurable feedback.

5. Reviews help preventing defects from getting injected into the product under development as a result of feedback received during review of requirement and design documents.
6. Reviews also help in improving the domain expertise.

Any software artifact which is deliverable must be reviewed. Possible candidates for review are SRS, schedule and budget estimates, design document, project management plan, test plan etc. Reviews are normally done in each phase before exiting a phase and completing a project deliverable. While doing the reviews, persons are drawn from all stakeholder categories and the different roles played by them are:

- *Author* : *The person who developed the product in the beginning.*
- *Moderator* : *The Person who leads the meeting.*
- *Presenter* : *The person who presents the facts about the product.*
- *Scribe* : *The person who documents the results found during the review i.e., the defects or errors found, their type etc.*
- *Reviewer* : *The person who finds the defects/errors. This role is played by all the participants.*

Reviews can be classified as ***formal reviews*** or ***informal reviews***. Formal reviews are done using a published process and are conducted after each phase of software development cycle. Results of formal reviews are documented. Informal reviews on the other hand are conducted depending upon the need and their results are also not documented. Purpose is to exchange information.

A number of techniques like inspections, walkthroughs, buddy checks etc., are used to review the work products.

In **inspections** a group of people well prepared in advance about work product attend the meeting to collect information and find defects. The presenter is any person other than the developer.

Walkthroughs are also similar to inspections, but here participants come without preparation to attend the meeting. Presenter in this case is the author of the product. Walkthroughs are mainly used for communication purposes.

In **buddy checks**, the work product is given randomly to any one and is asked to find errors. As it is said that any form of testing is better than none.

8.9 BASIC VALIDATION METHODS

Validation is done by executing the end product. There are two fundamental testing strategies used at this stage. They are:

- Black box testing
- White box testing

Black box testing aims at testing the functionality of the product and is done without the product's internal knowledge. White box testing on the other hand is used to test internals of the program and hence requires the knowledge of internal details of the program.

8.10 BLACK BOX TESTING

Black box testing is done to validate all the functional requirements of the system by checking for incorrect or missing functions, interface errors etc. Hence it is also known as interface or functional testing.

The internal structure of the code is not considered while designing functional tests and hence are called "black box". The term black box indicates that inside details of the box are irrelevant. Test cases for black box testing are designed to test the software systematically and in detail. For each test case input values, expected outputs and actual outputs are recorded. Some of the techniques used for black box testing are:

1. Syntax driven testing
2. Cause effect graphs
3. Equivalence partitioning
4. Boundary value analysis

8.10.1 Syntax Driven Testing

This type of testing can be applied to systems that can be syntactically represented by some grammar. For example in compilers, language can be represented using Context Free Grammar(CFG). Once represented using CFG, test cases can be designed to test all the productions. Other systems are interpreters, assemblers and scripting languages. Test cases can be designed to test each production rule of the grammar. To illustrate the syntax driven parsing consider the Lex program given below. Lex program is used to generate the lexical analyzer for particular specifications.

```
%{
    # define NUMBER 100
    # define COMMENT 101
    # define TEXT 102
    # define VARIABLE 103
%}
%%
[0 - 9] +\[0 - 9]+      {return NUMBER ;}
#*                      {return COMMENT ;}
\" [^\"\n] *\"          {return TEXT ;}
[A - Z] +               {return VARIABLE ;}
\n                      {return '\n' ;}
%%
□
```

Now in this case test cases are designed to test the lex program in such a way that each of the regular expression is tested at least once. Similarly consider the following grammar

```
stmt-list : stmt '\n' | stmt-list stmt '\n'
          ;
stmt      : Name '=' expr
          | expr
          ;
expr      : expr '+' expr
          | 'expr '-' expr
          | Name
          ;
```

In this example also, test cases can be designed so that each production of the grammar is executed at least once.

8.10.2 Equivalence Partitioning

If we examine the working of a system, we find that more than one type of inputs some time result in the similar behavior of the system. For example the transaction, placing an order with bookstore would behave in the similar fashion, only difference being the number of items ordered. It is of this property which is made use of in the equivalence partitioning technique. The purpose of equivalence partitioning is that if tests are distributed over all possibilities, it will result into effective testing.

The idea is to partition the input domain of the system into a finite number of equivalence classes such that each member of the class would behave in similar fashion i.e., if a test case in one class results in some error, other members of class would also result into same error.

This technique increases the efficiency of software testing as number of input states are drastically reduced. The technique involves two steps:

(a) Identification of equivalence classes and

(b) Generating the test cases

(a) **Identification of equivalence classes** – Following guidelines may be used for identifying equivalence classes:

(i) Partition any input domain into minimum two sets: valid values and invalid values.

(ii) If a range of valid values is specified as input- select one valid input within range and two invalid inputs outside at each end of range. For example if input requires the height of student between 160 cm and 170 cm select one valid input for height say 165 cm and two invalid inputs i.e., (height < 160 cm and height > 170 cm).

(iii) If a set of defined values are specified as inputs, define one valid input from within the set and one invalid output outside the set. For example if input to degree requires one of the values as B.E., M.E. or Ph.d, define one valid input as one of these and invalid input as any value other than B.E, M.E or Ph.d say MBA. The same principle will apply for a set of input values.

(iv) If a specific number (N) of valid values or enumeration of values (i.e., 10 15 16 18 20) is specified as input space, select as valid inputs - first, last and nominal value from centre of range i.e., (10, 20, 16 in this case) and one outside each end as invalid inputs say (8 and 22).

(v) If a mandatory value is defined in the input space say input must start with $, define one valid input where first character is $ e.g., $credit and one invalid set of input where first character is not $ say account.

(b) **Generating the test cases** – Following steps are followed:

(i) To each valid and invalid class of input assign a unique identification number.

(ii) Write test cases covering all valid class of inputs.

(iii) Write test cases covering all invalid class of inputs such that no test case contains more than one invalid input so as to ensure that no two invalid inputs mask each other..

Example 8.1 Equivalence Partitioning Example — Paint Store

Let us consider a software module that accepts the name of a paint item, company name and list of pack size of the item(in ml). As per specifications paint name must consist of alphabetic characters (3 to 20 characters). Company name can be Berger, ICI, Nerolac or Asian. Size must be a value in the range 50 to 20000 ml, whole numbers only. The size must be entered starting from smaller size in ascending order. For each paint item maximum 8 sizes i.e., 50 ml, 100 ml, 200 ml, 500 ml, 1000 ml, 5000 ml, 10000 ml and 20000 ml are available. The item details are entered in following order: paint name followed by comma, company name followed by comma then followed by list of sizes separated by;. Ignore any blanks or spaces in the input.

The module produces the following reports showing

(a) Names of paint items, their size and company name

(b) The same as report (a) but sorted alphabetically by company name.

Derive the valid, invalid equivalence classes for the above module.

Solution – *Using the guidelines discussed in equivalence partitioning technique following equivalence classes are derived.*

S. No.	External Input Conditions	Valid Equivalence Classes	Invalid Equivalence Classes
1.	Item name	alphabetic	non-alphabetic
2.	Item name	non-blank	blank
3.	Item name length	3 to 20 characters	<3 characters,> 20 characters
4.	Company name	Berger	Shalimar
5.	Size value	numeric	non-numeric
6.	Size value	whole number	decimal
7.	Size value range	50—20000	<50 >20000
8.	Size values	ascending order	non-ascending order
9.	Size values	non-blank	blank
10.	No. of size values entered	one to eight	> 8
11.	Order of entering details	item name followed by company name followed by list of sizes	item name not first
12.	Company name	non-blank	blank
13.	Separator between each entry in list	comma	any other character other than comma
14.	Separator between list of sizes	semicolon	any other character other than semicolon.

8.10.3 Boundary Value Analysis

It has been observed that boundaries are a very good place for errors to occur. Hence if test cases are designed for the boundary values of input domain, efficiency of testing increases, thereby increasing the probability of detecting errors. Boundary value analysis is refinement of equivalence partitioning approach with two differences. First is that test cases are designed by selecting one or more elements so that each edge of the input class become subject of test. Secondly test cases are also designed by exploring the output conditions. Guidelines for boundary value analysis are:

(a) For given range of input values say 10.0 to 20.0 identify valid inputs as ends of range (10,20) invalid inputs just beyond the range (i.e 9.99 and 20.01) and write test cases for the same.

(b) For a given number of inputs say (e.g 5, 7, 8, 10, 15) identify minimum and maximum values as valid (5 and 15) and one beneath (4) and beyond these input values (16).

(c) Use the guidelines (a) and (b) for each output condition. For example, if a program calculates the yearly bonus to be paid to the employees minimum say Rs. 0 and maximum say Rs. 5000 then write test cases that cause bonus of rupees zero and Rs. 5000 to be added to an employee's salary. Also write test cases that add negative bonus or bonus greater than 5000 rupees.

Using the functional specification for paints store described in example 8.1 students are advised to identify the list of boundary values to be tested. Consider input as well as output conditions.

8.10.4 Cause-Effect Graphing

Cause-effect graphing is another popular black box testing technique that establishes relationships between logical input combinations called causes and corresponding actions called effect. The causes and effects are represented using a boolean graph. The following steps are followed:

(a) For a module identify input conditions (causes) and actions (effect).

(b) Develop a cause-effect graph.

(c) Transform cause-effect graph into a decision table.

(d) Convert decision table rules to test cases. Each column of the decision table represents a test case.

While drawing cause-effect graphs, three types of nodes are used: AND, OR and negation nodes. In AND node, effect results if all inputs are true. In OR node, effect results if one of inputs is true and in case of negation effect occurs if all inputs are false i.e., normal Boolean algebra rules are followed. Now we will illustrate this technique with the help of a suitable example.

Example 8.2: In an income tax processing system if the annual taxable salary of a person is less than equal to Rs. 60000/- and expenses don't exceed Rs 30000, 10% income tax is charged. If the salary is greater than Rs. 60000 and less than equal to Rs. 200000 and expenses don't exceed Rs. 40000, tax of 20% is charged. For salary greater than Rs. 200000,5% additional surcharge is also charged. If expenses are greater than Rs. 40000, surcharge is 8%. Design the test cases for problem using cause effect graph technique.

Solution. **Step 1:** Identification of cause & effects

Causes	Effects
C_1 – Salary < = 60000	E_1 – Compute tax at 10% rate
C_2 – SALARY > 60000 & < = 200000	E_2 – Compute tax at 20% rate
C_3 – Salary > 200000	E_3 – Compute tax at 20% rate plus 5% surcharge
C_4 – Expenses < = 30000	E_4 – Compute tax at 20% rate plus 8% surcharge
C_5 – Expenses < = 40000	
C_6 – Expenses > 40000	

Step 2: Cause effect graph is as shown below in Fig. 8.2.

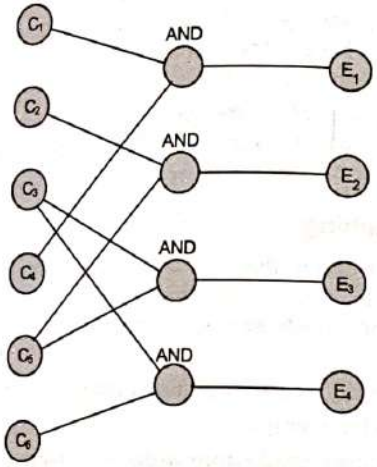


Fig. 8.2 Cause Effect Graph for Example 8.2

Step 3: The cause effect graph of Fig. 8.2 can be transformed into a decision table where each row corresponds to a cause or effect and columns of table correspond to test cases as shown in Fig. 8.3.

		1	2	3	4
Causes	C_1	1	0	0	0
	C_2	0	1	0	0
	C_3	0	0	1	1
	C_4	1	0	0	0
	C_5	0	1	1	0
	C_6	0	0	0	1
Effects	E_1	×	–	–	–
	E_2	–	×	–	–
	E_3	–	–	×	–
	E_4	–	–	–	×

Fig. 8.3 Decision Table Corresponding to Cause Effect Graph of Fig. 8.2

Though 2^6 test cases would be generated for 6 causes but here we see that only 4 test cases would be required to test the system. Set of test cases can be:

1. Salary – 20000, expenses – 2000
2. Salary – 100000, expenses – 10000
3. Salary – 300000, expenses – 20000
4. Salary – 300000, expenses – 50000

It is clear from the example that cause-effect graphing technique builds a Boolean network which expresses the effects as related to combination of causes (eliminating all redundant combination of causes) and finally summarizes these conditions and actions into a decision table. Decision table is used to derive the test cases which can also take into account boundary value conditions.

8.11 WHITE BOX TESTING

White box testing is done to test internals of the program. This is done by examining the program structure and by deriving test cases from the program logic. White box testing is also known as *structural testing, glass box* testing, and *clear box* testing. Test cases are derived to ensure that

(a) All independent paths in the program are executed at least once.
(b) All logical decisions are tested i.e., all possible combinations of true and false are tested.
(c) All loops are tested.
(d) All internal data structures are tested for their validity.

Next we will discuss some popular white box testing techniques.

8.11.1 Statement Coverage

In this type of testing each statement of the code is executed at least once. This is a weak form of testing as it sometimes fails to detect the fault in the code. Consider the program segment given below:

```
If(x > 50 && y < 10)
                Z = x +y;
Printf("%d\n",Z);
x = x+1;
```

In this test case, the values x = 60 and y = 5 are sufficient to execute all the statements.

The main disadvantage of statement coverage is that it does not handle control structures fully. Also it does not report whether loops reach their termination condition or not and is insensitive to the logical operators. Simply going for 100% statement coverage does not necessarily results into a good test case as it does not tell us about the quality of test cases. It only answers that all statements are executed at least once.

8.11.2 Decision Coverage/Branch Coverage

This type of white box testing by designing suitable test cases focuses on executing each branch of each decision (such as if statement and while statement) at least once.

Consider the following program segment for example

```
If((x < 20) AND (Y > 50))
        total = total + x;
else
        total = total + Y;
```

This must be tested using test cases such that (($x < 20$) AND ($Y > 50$)) is evaluated as true at least once and as false also at least once. Test cases with following inputs can be used

```
x = 10,     Y = 55
x = 10,     Y = 10
```

A disadvantage of this approach is that this may ignore branches within a boolean expression. Consider the following code e.g.,

```
If    (X  && (Y || add_digit()))
      Printf("success \n");
else
      Printf("failure\n");
```

Now branch coverage would completely exercise the control structure without calling the function add_digit( ). The expression is true when X and Y are true and false if X is false.

8.11.3 Condition Testing

Condition testing is done to test all logical conditions in a program module. They can be easily identified as:

(*i*) Relational expression of the type (Expr1 op Expr2) where Expr1 and Expr2 are arithmetic expressions.

(*ii*) Simple conditions.

(*iii*) Compound conditions composed of two or more simple conditions.

(*iv*) Boolean expressions.

Test cases are designed so that at least once each condition takes on every possible values. For example let us consider a program segment given below

```
If    ((X) && (Y) && (!Z))
      Printf ("Valid");
else
      Printf("Invalid\n");
```


Now this program segment must be tested with each condition (X,Y,Z) true once and false once.

Hence two valid test cases are:

(*i*)	X = T	Y = T	Z = F
(*ii*)	X = F	Y = F	Z = T

8.11.4 Multiple Condition Coverage

In this type of testing decision/branch coverage is satisfied and test cases are designed for all possible combination of conditions.

Hence for the code given below

```
If    (X && Y && !Z)
      Printf("Valid input");
else
      Printf("Invalid input\n");
```

Test cases will have following values

X = F	Y = F	Z = F
X = F	Y = F	Z = T
X = F	Y = T	Z = F
X = F	Y = T	Z = T
X = T	Y = F	Z = F
X = T	Y = F	Z = T
X = T	Y = T	Z = F
X = T	Y = T	Z = T

Hence 8 test cases in total would be required.

The drawback of this approach is that it can be very tedious to find the minimum number of test cases for complex boolean expressions.

8.11.5 Basis Path Testing Using Flow Graph Analysis

Basis path testing helps the tester to compute logical complexity measure (McCabe76) of the code. This value in turn defines the maximum number of test cases to be designed by identifying basis set of execution paths to ensure that all statements are executed at least once (Pressman97).

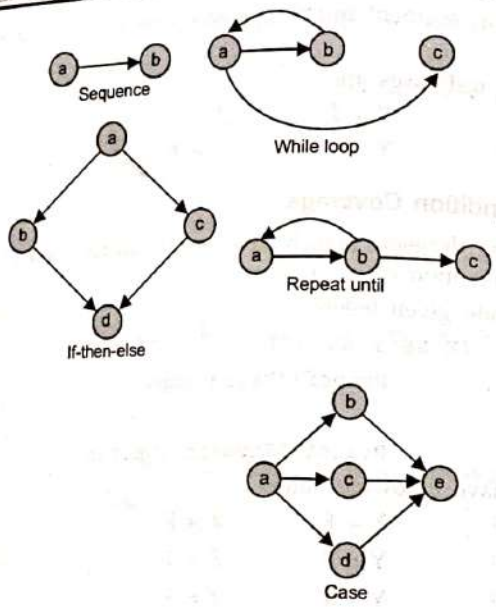


Fig. 8.4 Notation for Drawing Flow Graph

Following steps are followed:

(*i*) Construction of flow graph from source code or flow charts.

(*ii*) Identification of independent paths.

(*iii*) Computation of cyclomatic complexity.

(*iv*) Test case design.

(*i*) **Construction of flow graph.** A flow graph consists of a number of nodes represented as circles connected by directed arcs. Direction indicates the flow of control direction. Notation for drawing flow graphs is shown in Fig. 8.4.

(*a*) **Construction of flow graph from flowchart.** While transforming a flow chart into a flow graph sequence of processing boxes and diamond box can map into a single node. Same can be applied to code. A node that contains a condition is called a predicate node. Let us first consider a simple flow chart as shown in Fig. 8.5.

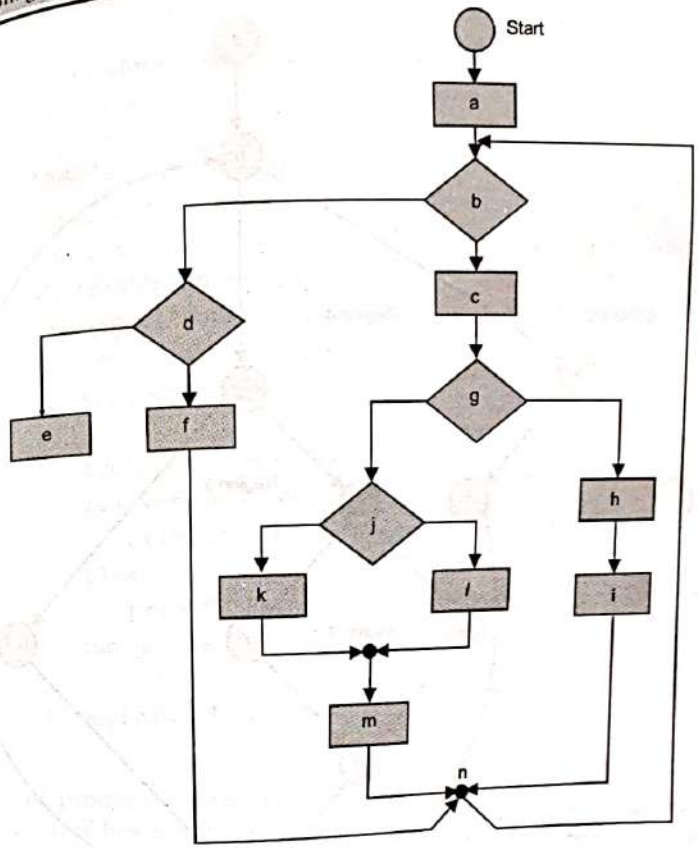


Fig. 8.5 A Sample Flow Chart

Fig. 8.5 can be transformed into a flow graph as shown in Fig. 8.6.

As discussed earlier diamond *g* and processing box *c* can be mapped to a single node. Same is the case for processing boxes *h* and *i*.

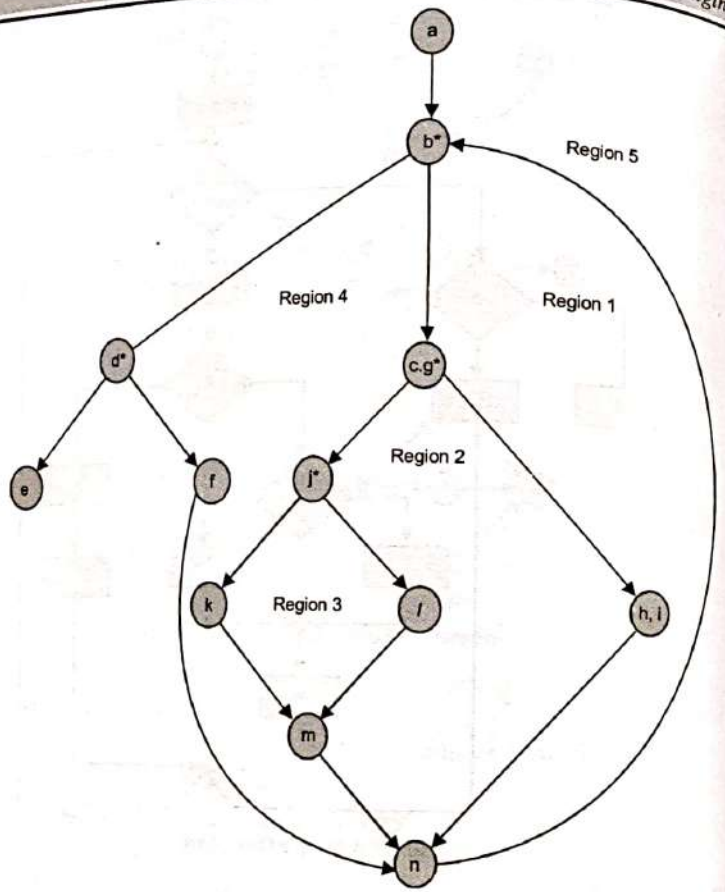


Fig. 8.6 A Flow Graph for Fig. 8.5 Flowchart

(b) Construction of flow graph from source code. Next we consider a small program written in C and illustrate how we can transform this program into flow chart. This program inputs the marks of 5 subjects of 25 students and outputs average marks and the pass/fail message.


```
01 main() {
02     int num_student, marks, subject, total;
03     float average;
04     num_student = 1;
05     while(num_student <=25)    {
06         total = 0;
07         subject = 1;
08         while(subject <= 5){
09             scanf("Enter marks out of 100...%d/n",&marks);
10             total=total + marks;
11             subject++;
12         }
13             average = total/5;
14             if(average >= 50)
15                 prinf("PASS.. Average marks are%f\n", average);
16             else
17                 printf("FAIL.. Average marks are%f\n", average);
18             num_student++;
19     }
20     printf("End of Program\n");
21 }
```

The process of producing flow graph starts with dividing the program into parts where flow of control has single entry and exit point. In the above example there is sequence of three parts. The first part from lines 002 to 004 consists of declaration and initialization of local variables. The second part comprises the while structure between lines 005 and 019 and third part is the single Printf statement on line 20. Hence the flow graph sharing these three regions of code is shown in Fig. 8.8. The three parts where flow of control exit and enter are between nodes *a* and *b*, *b* and *c* and *c* and *d*. Out of these only the region between *b* and *c* requires further refinement.

Fig. 8.7

Now if we study the second part i.e., between lines 005 and 019, it can again be divided into four parts. The first part consists of sequence of statements 006 and 007. Second part comprises the while structure between lines 008 and 012. The third part is single assignment statement at line 13 and fourth part is the if-then-else structure between lines 14–17. Using the flow graph notation of Fig. 8.4 the flow graph of Fig. 8.7 can be refined as shown in Fig. 8.9.

The statements corresponding to various nodes are given in Fig.8.8

Nodes	Statement Numbers
a	2–4
b	5
e	6–7
f	8
z	9–12
g	13–14
h	15
i	17
j	18
c	19
d	2

Fig. 8.8

(ii) **Computation of Cyclomatic Complexity:** In simple language cyclomatic complexity is a metric to measure logical complexity of the program. This values defines the number of independent paths in the program to be executed in order to ensure that all statements in the program are executed at least once. In other words it gives us the value for maximum number of test cases to be designed. The cyclomatic complexity C(G) of a graph G can be computed using one of the following three ways:

(a) C(G) = Number of regions in the flow graph.

Regions are the areas bounded by nodes and edges in a flow graph. While counting regions, area outside the graph is also counted as one region. In Fig. 8.5 there are five regions in all.

(b) C(G) = E – N + 2

Where E = number of edges in the flow graph

N = Number of flow graph nodes

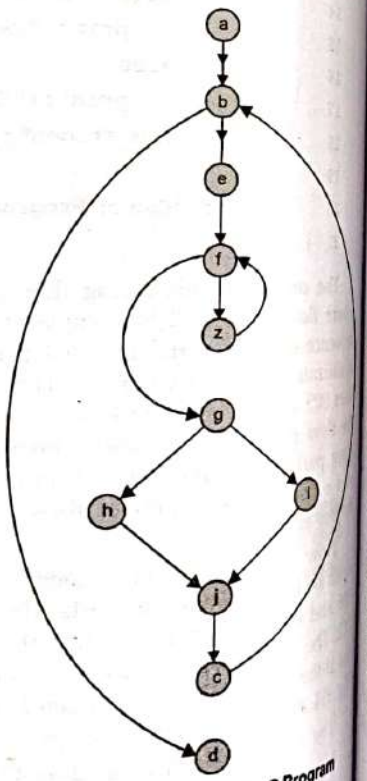


Fig. 8.9 Flow Graph for C Program

For Fig. 8.5, E = 15 and N = 12. Hence cyclomatic complexity is

C(G) = 15 – 12 + 2 = 5

(c) C(G) = Number of predicate nodes + 1

In Fig. 8.6 there are 4 predicate nodes(marked*)

Hence C(G) = 4 + 1 = 5

(iii) **Identification of independent paths** – Independent path in a program is a path consisting of at least one new condition or set of processing statements. In case of a flow graph it must consist of at least one new edge which is not traversed or included in other paths. Number of independent paths are given by value of cyclomatic complexity. For flow graph of Fig. 8.5 there are therefore 5 independent paths and they are:

Path 1 :– a – b – d – e

Path 2 :– a – b – d – f – n – b – d – e

Path 3 :– a – b – c – g – j – k – m – n – b – d – e

Path 4 :– a – b – c – g – j – l – m – n – b – d – e

Path 5 :– a – b – c – g – h – i – n – b – d – e

Each of these path consists of at least one new edge.

It is to be kept in mind that this basis set of paths is not unique.

(iv) **Design of test cases** – Test cases can now be designed for execution of independent paths as identified in step 3. This ensures that all statements are executed at least once.

Automating the derivation of basis set

A software tool using the data structure graph matrix can be developed to automate the process of deriving flow graphs and the set of basic paths. A graph matrix is a simple square matrix where each side has number of nodes equal to flow graph nodes. Connection between flow graph nodes is shown by adding a link weight say 1 between the corresponding nodes. Presence of 0 shows absence of edge between the nodes. Graph theory algorithms can now be applied to these graphs matrices to produce the set of basic paths.

8.11.6 Data Flow Testing

As the name suggests this type of testing is based on use of data structures and flow of data in the program. This is because data structures are important constitutes of any program and hence must be taken into consideration while designing test cases. Terminology used during data flow testing is

Definition or def

A statement in the program where an initial value is assigned to a variable e.g.,

$i = 1$, sum = 0

Basic–block

It is set of consecutive statements that can be executed without branching e.g.,

```
sum = sum + next;
Billvalue = billvalue + sum;
next ++;
```

C–use

It is also called computation use and occurs when variable occurs for computation. A path can be therefore identified starting from the definition and ending at a statement where it is used for computation called dc path. The value of variable is also charged.

P–use

P use is similar to *C*-use except that in the statement the variable appears in the condition. So a path can be identified starting from definition of variable and ending at statement where the variable is appearing in the predicate called dp path.

All–use

In this paths can be identified starting from definition of a variable to its every possible use.

du–use

In this, a path can be identified starting from definition of a variable and ending at a point where it is used but its value is not changed called du path.

As a part of a data flow testing all possible paths as discussed above *i.e.*, all use paths can be identified and anomalies (like variable defined but not used) in the program can be detected.

The Example of Different Paths

For a C program as given in Fig. 8.10, program is divided into basic blocks represented by rectangular boxes. Test cases can be designed for all possible *dp*, *dc* and *du* paths to find out anomalies in the program. In this example we see that variable *d* is defined but never used. Some of the *du*, *dc* and *dp* paths are identified.

Readers are advised to identify rest of the paths in the program.

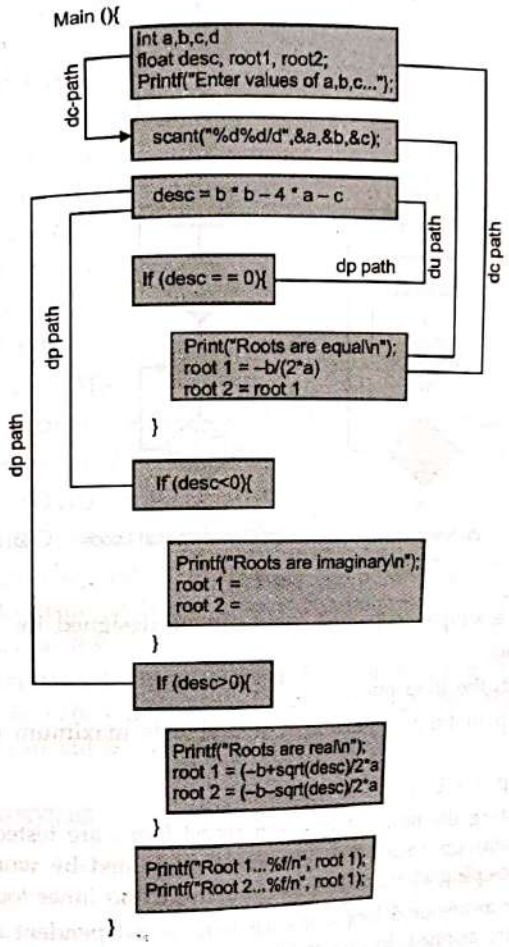


Fig. 8.10

8.11.7 Loop Testing

In majority of the algorithms, the loops are used extensively. Hence proper attention must be given to loop testing while performing white box testing techniques. Loops can be classified as simple loops, nested loops and concatenated loops as shown in Fig. 8.11. Unstructured loops must be redesigned as structured loops in order to design proper test cases. A number of techniques have been proposed for testing loops.

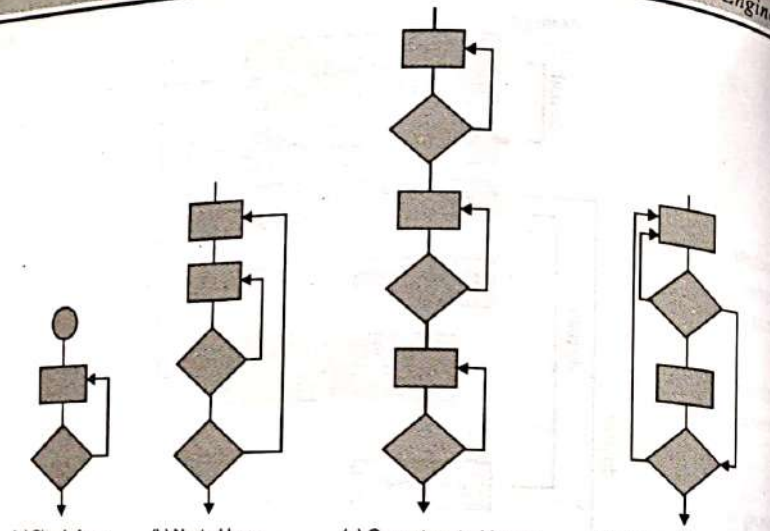


Fig. 8.11

For example for a simple loop, test cases can be designed to:

(*a*) skip the loop.

(*b*) pass through the loop once.

(*c*) execute loop m times where $m < n$ and n is maximum number of allowable passes.

(*d*) execute loop $n - 1$, n or $n + 1$ times.

Similarly for testing the nested loops, innermost loops are tested first while all outer loops are set to minimum value. This process is continued by working outward in the same fashion (i.e., keeping all outer loops at minimum and inner loops at typical values).

In the case of concatenated loops, if each loop is independent of other, simple loop testing techniques are applied. In case they are not independent, nested approach is used.

8.11.8 Mutation Testing

Mutation testing is an error based testing technique which looks for presence of errors. The technique comprises of two steps.

(*a*) A set of program variants called mutants is generated by introducing known bugs representing typical errors. Possible changes that can be done are:

- Scalar variable replacement i.e., each occurrence of variable x is replaced with all other variables in scope.
- Arithmetic operator replacement i.e., each occurrence of an arithmetic operator is replaced with all possibilities.

The program with mutated statement is called mutant. To illustrate this consider the program given below:

```
1. INPUT A
2. INPUT B
3. TOTAL = 0
4. FOR I = 1 TO A
5. IF (B > 0)
6. TOTAL = TOTAL + B
7. INPUT B
8. NEXT I
9. PRINT TOTAL
```

In this program mutants can be generated by changing variable A to B or TOTAL. Similarly statement 6 can be changed by changing arithmetic operator + to

TOTAL = TOTAL – *B*

or TOTAL = TOTAL * *B* etc.

(*b*) Test cases are designed to distinguish the original program from the mutant. The ability to distinguish the mutant from the original program is determined by the quality of test data.

A test case differentiates two programs if different results are produced by the two. A mutant is said to be called if it is detected by the test case. The objective is to find the test cases which can kill the mutants.

8.12 LEVELS OF TESTING

As discussed in chapter 4, during the design phase software is represented in terms of a hierarchy of interconnected modules. Hence instead of testing the software as a whole, testing of software is done at various levels or stages. They are:

(*a*) Unit testing

(*b*) Integration testing

(*c*) System testing

(*d*) Acceptance testing

8.12.1 Unit Testing

Unit testing concerns testing smallest components of the software *i.e.*, modules. Test cases are therefore designed to test

- Program logic

- Functionality
- Interfaces
- Boundary conditions
- Data structures
- All paths in the program

The focus in unit testing is therefore on the performance of single module by finding defects in that module. At this level white box testing techniques are used.

The **drivers** and **stub** modules are also required to test each module in isolation. A driver module is a program which accepts the test case data to be inputted and printed by the module to be tested. Stub modules on the other hand simulate the modules called by module to be tested. For leaf level or terminal modules no stubs are required.

8.12.2 Integration Testing

The process of combining multiple modules systematically for conducting tests in order to find errors in the interface between modules is called integration testing. A study has shown that almost 40% of the errors are due to integration and interface problems. Integration testing is done after successful completion of unit testing. Number of strategies can be followed to do integration testing.

They can be classified as:

(*a*) Incremental strategy

(*b*) Non-incremental strategy

(*c*) Mixed strategy

In the incremental approach, one or more modules are added to the already tested set of modules, there by increasing the set of merged modules incrementally. The non-incremental approach is just the opposite because all the modules are simultaneously merged and tested. The mixed approach is the combination of incremental and non-incremental strategies.

Possible integration errors as provided by (Beizer90) are:

- Protocol design errors
- Incorrect parameter values
- Wrong entry or exit
- Wrong subroutine calls
- Input/output format errors
- Interface errors
- Errors due to use of global variables

Normally black box testing techniques are used at this level. It is to be noted that integration testing can be applied at

- Program level to test modules.
- Subsystem level to test various programs.
- System level to test subsystems.
- Network level to test different systems of network.

Next we will discuss some popular integration testing techniques.

(*a*) **Bottom up integration** – As clear from the title, in this form of testing we start with testing the terminal modules and move up in the hierarchy. That is the module at one level above are tested by calling previously tested modules till all the modules are tested. In this manner any kind of interface problems between modules can be identified to understand more. Bottom up testing uses drivers which are used to supply tests to module under test. A driver reads tests from a file, calls the module under test iteratively and compares the actual outputs with expected outputs. After the module is tested its driver is replaced by actual module to be tested and its driver. Consider the hierarchy of modules shown in Fig. 8.12.

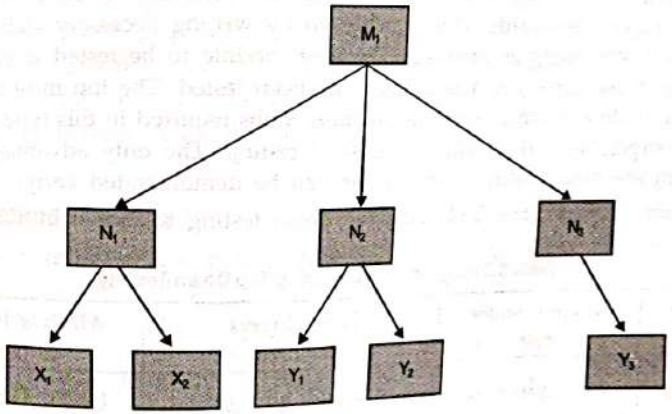


Fig. 8.12

In this structure first module X_1, X_2, Y_1, Y_2 and Y_3 are tested as standalone entities using necessary drivers. This is followed by adding N_1, N_2 and N_3 one by one followed by adding M_1. The only disadvantage of bottom up integration is that the functionality of the complete system is not visible until the root level module (M_1 in this case) is added.

Integration order for bottom-up testing for the modular structure shown in Fig. 8.12 is shown in Table 8.1. In this figure driver (A) denotes driver of module A.

Table 8.1 Integration Order for Bottom Up Testing for Modular Structure of Fig. 8.10

Testing order	Modular under test	Drivers used	Module interaction to be listed
1.	X_1	driver (X_1)	Unit testing of X_1
2.	X_2	driver (X_2)	Unit testing of X_2
3.	Y_1	driver (Y_1)	Unit testing of Y_1
4.	Y_2	driver (Y_2)	Unit testing of Y_2
5.	Y_3	driver (Y_3)	Unit testing of Y_3
6.	X_1, X_2, N_1	driver (N_1)	$X_1 - N_1$, $X_2 - N_2$
7.	Y_1, Y_2, N_2	driver (N_2)	$N_2 - Y_1$, $N_2 - Y_2$
8.	Y_3, N_3	driver (N_3)	$N_3 - Y_3$
9.	N_1, N_2, N_3, M_1	driver (M_1)	$M_1 - N_1$, $M_1 - N_2$, $N_1 - N_3$

(b) **Top down integration** – In this approach instead of starting at the terminal level module, testers begin with the top most module in the module hierarchy and move downwards. This is achieved by writing necessary stubs and drivers at different stages as required. The next module to be tested is the one whose at least one super ordinate module has been tested. The top most module is the only module which is tested in isolation. Stubs required in this type of integration are expensive and increase the cost of testing. The only advantage is that the complete functionality of the system can be demonstrated early.

Integration order for Fig. 8.12 using top down testing is shown in Table 8.2.

Table 8.2 Integration Order Using Top Down Testing

Testing order	Module under test	Stubs/drivers required	Module interaction to be tested
1.	M_1	stub (N_1), stub (N_2), STUB (N_3)	UNIT TESTING M_1
2.	M_1, N_1	stub (N_2), stub (N_3)	M1 – N_1
3.	M_1, N_1, N_2	stub (X_1), stub (X_2) stub (N_3), stub (X_1) stub (X_2), stub (Y_1), stub (Y_2)	$M_1 - N_1$, $M_1 - N_2$

Readers are advised to complete the integration order of Table 8.2.

(c) **Big-Bang testing** – It is the least used and least effective approach of integration testing. In this approach all the modules/components after testing individually are combined together and tested in one go. The biggest problem in this approach is debugging of errors which can be associated with any module.

(d) **Sandwich testing** – This testing involves combination of top down and bottom up testing techniques. Both the techniques are applied simultaneously.

8.12.3 System Testing

System testing focuses on the testing of product as a whole by verifying its working with the original requirements specification document. Several types of testing is used at this stage. Some of them are

(i) **Function testing** – It is done to test functionality of the system using black box testing techniques.

(ii) **Performance testing** – This type of testing deals with quality related issues like security, accuracy, efficiency using stress test, volume test, security test, reliability test etc. Volume testing is done to ensure that software can handle the required volumes of data. Stress testing is used to find out the maximum load conditions at which the software fails to handle the required load. Similarly security testing and reliability testing is done to ensure that the program meets its reliability and security requirements.

System testing is done at the developers end before the product is given to customer for use.

8.12.4 Acceptance Testing

After the system testing is completed successfully by the developers, acceptance testing is done at the customer end. It is the customer or end user who now designs the test cases. In this type of testing emphasis is on usability testing of the product. Acceptance testing is supported through *Alpha* and *Beta* testing. Alpha testing is done when the software is made operational for the first time to be tested by the users at developer's site. Hence it is possible that it will involve making lot of changes to program code.

Beta testing follows alpha testing but now the testing is done by the users at the customer's site who validate the product after using it for few days. At this stage few changes as compared to alpha testing would be made to the product.

8.13 REGRESSION TESTING

Whenever changes are made to the software a set of tests (one or more) are automatically run and the new outputs are compared with the older ones to avoid unwanted changes. This type of testing is called regression testing. It is applied after the software or program is modified. Regression testing can be applied both during development phase as well as during maintenance of the software.

In the development phase regression testing is done after correcting the errors found during testing of software. Similarly during maintenance phase of software life cycle as a result of adaptive, corrective and preventive maintenance (discussed in detail in chapter 10) modifications are made to the software. Adaptive maintenance involves adapting the system to meet new requirements or environment, preventive maintenance focuses on enhanced functionality and corrective maintenance involves fixing the errors. As a result of these lot of program code is changed. Regression testing is therefore, done to ensure that program is performing as per specifications.

8.14 DEBUGGING

In the context of software engineering debugging is the process of fixing a bug in the software.

This is the activity which begins after the software does not execute properly or fails. The activity concludes by addressing or solving the problem and testing the software successfully. A significant amount of time is spend on debugging during maintenance of the software. Debugging is one activity which requires lot of expertise, skills and knowledge. Finding expert debuggers is very difficult.

The activity of debugging is quite different from the testing. In testing the focus is to find errors, bugs etc., whereas the activity of debugging starts after the bug is reported in the software.

The various steps involved in debugging after the defect is identified in the software are as follows:

(a) Identity the problem in the software and create the defect report.
(b) Assign the defect to software engineer to ensure that the effect is genuine.
(c) Analyze the defect by understanding the main cause of the problem. This is done by modeling the system using system documentation, finding and testing the candidate flaw(s) and if required repeating it number of times. Debugging tools can also be used at this stage of debugging for analyzing the defect.
(d) Resolve the defect by making required changes to the software.
(e) Validate the final corrected software.

A number of strategies are used to debug the software in order to find cause of defect at different stages of debugging life cycle. Some of them are:

(a) Studying the buggy system for longer duration in order to gain deeper understanding of the system. It also helps debugger to construct different representations of system to be debugged depending on the need. Study of the system is also done actively to find recent changes made to the software.
(b) Backward analysis or tracking of program which involves tracing the program backwards from the location of failure message in order to identify the region of faulty code. A detailed study of region is then conducted to find the cause of defect.

(c) Forward analysis of the program involves tracking the program forward using break points or print statements at different points in the program and studying the results at these points. The region where the wrong results are displayed is the region which should be given attention to find causes of defect.
(d) Using the past experience of debugging the software with problem of similar nature. However success of this approach would be dependent on the expertise of the debugger.

These strategies of debugging are also supported by number of tools and techniques e.g., interpreters, sophisticated editors, profilers, breakpoint debuggers, code-based tracing; print statements etc. A lot of commercial and public domain software like gnb are available for debugging in the market.

8.15 OBJECT ORIENTED TESTING

As information systems are becoming more complex, Object-Oriented (OO) paradigm is gaining popularity because of its benefits in analysis, design and coding. Conventional testing methods discussed in the chapter can not be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message passing, polymorphism, concurrency etc., (Smith90). According to (David98) the dependencies occurring in conventional system are:

- Data dependencies between variables
- Calling dependencies between modules
- Functional dependencies between a module and the variable it computes
- Definitional dependencies between a variables and its types

But in OO systems there are following additional dependencies (Norman92):

- Class to class dependencies
- Class to method dependencies
- Class to message dependencies
- Class to variable dependencies
- Method to variable dependencies
- Method to message dependencies
- Method to method dependencies

Additional testing techniques are therefore, required to test these dependencies. Another issue of interest is that it is not possible to test the class dynamically, only its instances i.e., objects can be tested. Similarly concept of inheritance opens various issues e.g., if changes are made to a parent class or superclass, in a larger system of class it will be difficult to test subclasses individually and isolate error to one class. Additionally concept of control flow can not be directly used in OO systems as objects are created, their states are changed and finally they are destroyed. Flow of control can also be

thought of as passing messages among objects(Smith90). Therefore there is a need to propose techniques in order to test OO systems systematically. A number of researchers have proposed techniques to test OO systems (Smith90), (Robert94), (Paul 94), (Pei97), (Turner93). In this book which is meant mainly for the readers studying first-course in software engineering, we will not be discussing techniques to test OO systems in detail. However some Simple techniques are:

(i) **Class testing based on method testing:** This approach is the simplest approach to test classes. Each method of the class performs a well defined cohesive function and can therefore be related to unit testing of the traditional testing techniques. Therefore all the methods of a class can be invoked at least once to test the class. Consider the example program written in C++.

```
Class Account
{
  private:
     int      acc_num;
     float balance;

  public:
     Account (int n)
     {
        acc_num = n;
        balance = 0;
     }
     float check_balance()
     {
        return balance;
     }
  private
     increase_balance(float n)
     {
        balance = balance + n;
     }
     decrease_balance(float n)
     {
        balance = balance-n;
     }
};
```

Test cases can be designed to test the class Account by invoking all the methods for creating a new account, depositing cash, with drawing cash and checking the balance at least once.

(ii) **The Class testing based on data bindings:** The approach discussed above tests the methods in isolation and does not give meaningful results. Another approach proposed by (Kim96) is testing a class based on data bindings which measures the interface between the components of system and is used for generating test cases. According to (Kim96) an actual data binding set in a class is defined as an ordered triple $< m_1, d, m_2 >$ where m_1 and $m_2 \in$ Methods, $d \in$ Data, m_1 assigns a value to d and m_2 references d.

Data bindings between different methods of a class provide us with a way to measure interactions in its class as they are shared by different methods of a class. These data bindings are used to generate MM path (a sequence of a pair of methods represented by actual data binding) for designing test cases using flow graph technique.

8.16 SOFTWARE TESTING TOOLS

In order to make testing easier and effective and to improve the quality of product, testing process can be supported by different types of testing tools called Computer Aided Software Testing Tools (CAST). With increased code complexity these tools have become necessity. The advantages of using automated testing tools are:

1. They improve productivity and quality of software development.
2. These tools help in running large volumes of test unattended for 24 hours.
3. Tools help in generating test cases automatically thus making the job of testing easier and faster.
4. Testing tools also help in identifying errors which are difficult to find manually.
5. These tools automate the regression testing by re-running the software automatically whenever changes are made to the software.
6. Testing can be achieved in less time by using automated tools.
7. They also increase the productivity of the testers.

These testing tools can be classified in different ways. The simplest approach to classify these tools is static testing tools and dynamic testing tools. Static testing tools test the software without actually executing it whereas dynamic testing tools test the software by actually executing it. (Edward00) has classified the tools based on the activities performed by the tools. The important activities are:

- Reviews or inspections
- Test planning
- Test design and development
- Test execution and evaluation
- Test support

(Edward94) has categorized tools to cover following areas:

- Software measurement

- Coverage analysis
- Regression testing
- Test planning

Testing tools supporting software measurement are metric based and focus on determining complexity metric of the program. They compare source program to some predefined metric standards and generate results. Coverage analysis tools ensure that the software is thoroughly tested by pointing at which part of software is not tested. Regression testing tools are finally used to automate the re-running of tests. Some important testing tools used during testing life cycle are discussed below.

8.16.1 Testing Tools for Reviews

These tools belong to the category of static testing tools and are used for reviews, inspections and walkthroughs of requirements, design and code. Tools in this category can be the tools to do syntax and semantic analysis of the source code and to find errors which sometimes are missed by the compilers. Complexity metrics can also be used to measure the complexity of the program as complex parts of the program are generally the areas where defects are found. By using metrics like:

- Size metrics to calculate program size in terms of no. of lines, function points etc.
- Halstead's metrics
- Code-complexity metrics to find data flow or control flow complexity of program based on McCabe's cyclomatic complexity metric

Information about probable areas of errors in program, number of test cases required etc., can be found.

8.16.2 Test Case Generators

Test case generators are the automated testing tools which are used to generate test cases from requirement specifications, programs or test design languages. Writing test cases is the most important activity in the testing process as all other testing activities are dependent on running the test case. Therefore writing good quality test cases is key to effective and successful testing of the software. In an organization at CMM level 1, test cases can be written by the developers but in case of mature organization *i.e.*, CMM level 2 and above, testing must be supported by automated tools. Test case generators which are most popular are the specification based test generators in which the requirements written in some formal language such as LOTOS, Z, Specification Description Language (SDL) are input to tool which generates the test cases. The test case generator tools use rules called test design techniques to generate test cases. The rules can be defined to support functional testing, boundary value analysis, equivalent class partitioning, cause effect graphing, event driven testing, product based testing etc.

8.16.3. Capture/Playback and Test Harness Tools

One of the most boring and time-consuming activity during testing life cycle is to re-run manual tests number of times. At this stage capture/playback tools are of great help to the testers. These tools do this by recording and replaying the test input scripts. As a result tests can be replayed without attendant for long hours specially during regression testing. Also these recorded test scripts can be edited as per need i.e. whenever changes are made to the software. These tools can even capture human operations e.g., mouse activity, keystrokes etc.

A capture/playback tool can be either intrusive or non-intrusive. Intrusive capture/playback tools are also called native tools as they along with software under test, reside on the same machine. Non-intrusive capture/playback tools on the other hand reside on the separate machine and is connected to the machine containing software to be tested using special hardware. One advantage of capture/playback tools is to capture the errors users frequently make and which developers cannot reproduce.

Test harness tools are the category of capture/playback tools used for capturing and replaying sequence of tests. These tools enable running large volume of tests unattended and help in generating reports by using comparators. These tools are very important at CMM level 2 and above.

8.16.4 Coverage Analysis Tools

The objective of coverage analysis tools is to ensure that the software is tested thoroughly and to assist the tester in identifying the parts of software not covered. The important levels at which runtime coverage information can be obtained using these tools are statement level, branch level, call pair level and path level.

Branch coverage is important for unit testing where as call-pair coverage is used for system testing to find interface errors. The main advantage of coverage-analysis tools is to improve productivity of the testers. Also the testers can focus on less tested parts of the program and design test cases for the same. For effective testing 100% coverage is preferred.

8.16.5 Test Comparators

A test comparator is a tool which compares the results of software under test with the expected results and generates the discrepancy report. There tools can be used to compare both text and graphical data and are normally an essential part of capture/playback tools. The comparator tools can also be used to compare the results of current time with that of last time.

8.16.6 Memory Testing Tools

These category of tools focus on testing memory related problems such as

- Accessing memory locations which are out of range specially in case of arrays.
- Problem related to allocation of memory but not freed.
- Using non-initialized memory location.

These tools can assist in identifying errors which can cause serious problems during runtime.

8.16.7 Simulators

These testing tools are sometimes also called emulators. They are used to simulate hardware or software with which software under test interacts. These testing tools are normally used when it is too costly or dangerous to test the system directly e.g., telecommunication systems, nuclear power station etc. These simulators are specific for a particular system and need to be tested rigorously before they are used.

8.16.8 Test Database

In some test outputs, complete information about what software did is not available. In addition to display the results on screen it can also manipulate the data stored in a database. A test database is therefore a sample of the database being manipulated when the software is executed and is often special purpose. Comparator tools are also used here to compare the database values before and after the execution of software.

8.17 POPULAR COMMERCIAL TESTING TOOLS

Some of the popular commercial testing tools are listed below in Table 8.3.

Table 8.3 List of Popular Commercial Testing Tools

Name of tool	*Vendor*	*Applications*
1. SQA Manager	Rational Software Corporation	Test management
2. ReviewPro	Software Development Technologies	Review management
3. Visual Quality	McCabe and Associates	Complexity analysis of source program
4. JavaStar	Sun Microsystems Inc.	Capture/playback
5. SQA Robot	Rational Software Corporation	Capture/playback
6. JavaScope	Sun Microsystems Inc.	Coverage analysis
7. Performance Studio	Rational Software Corporation	Simulator/performance analysis
8. BoundsChecker	Compuware	Memory testing
9. CA-Datamacs/II	Computer Associates	Database generators

Nowadays automated tools are also available in the market that support regression testing.

SUMMARY

- Software testing is one of the most crucial activity of software development life cycle.
- Testing of software is done to find errors in the software.
- Testing can show only presence of errors but not their absence.
- Verification plus validation defines software testing.
- Verification testing involves reviews, inspections walkthroughs and buddy checks.
- Testing must be conducted by third party independently for effective results.
- Validation testing is done by actually executing the software and involves black box and white box testing strategies.
- Black box testing is done to check the functionality of the product.
- White box testing is done to test internals of the program.
- Different levels of testing are unit testing, integration testing, system testing and acceptance testing.
- Acceptance testing is done by the end user or customer.
- The activity of debugging starts after the bug is reported in the software.
- Testing activity can be supported by automated testing tools.

REVIEW PROBLEMS

1. Differentiate between mistake, fault, failure and error by taking suitable examples.
2. List at least five guidelines for effective testing.
3. What is the difference between verification and validation testing?
4. List few techniques used for verification testing. Explain briefly.
5. Consider a software module that accepts information about employees of an organization as per following details name, title, age, list of qualification, number of publication. As per specification name must be 6 to 40 characters long and first character must be an alphabet. Title can be either director, professor, or lecturer.
   Age must be between 21 and 60 years. List of qualification can be from B.E, M.Tech, Ph.d or MBA. Minimum one and maximum four qualifications can be included in the list.
   All the fields except number of publication must be non-null.
   The module prints the report of employees in ascending order based on.
   (*i*) Title
   (*ii*) Number of publication
   (a) Derive the valid, invalid equivalance classes for the above module.
   (b) Identify the list of boundary values to be tested. Consider input as well as output conditions.

6. Consider the flow chart shown in Fig. 8.13 and 8.14. For these flow charts compute McCabe's cyclomatic complexity factor.

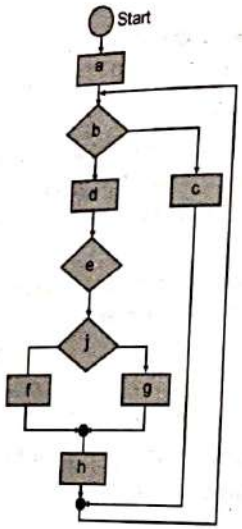


Fig. 8.13

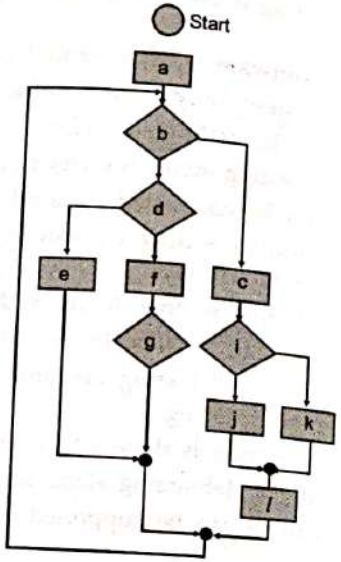


Fig. 8.14

7. Study the C program given below which counts the numbers of characters, blanks, tabs in a line.

```
#include <stdio.h>
main()
{
    int wcount = 0;
    int bcount = 0;
    int tcount = 0;
    int sum;
    while((ch = getch())!='\n'){
        if(ch = = ' ')
            bcount++;
        elseif(ch == '\t')
            tcount++;
        else
            wcount++;
```



```
    }
    sum = wcount + tcount + wcount ;
    Printf("Number of blanks are...%d\n", bcount);
    Printf("Number of tabs are...%d\n", tcount);
    Printf("Number of characters are...%d\n", wcount);
    Printf("Total No. of characters are...%d\n", sum);
}
```

For this program identify all

(a) dp-path
(b) du-path
(c) dc-path

8. For the program given in question 7 compute cyclomatic complexity factor.
9. Differentiate between Alpha and Beta testing.
10. Consider the modular structure shown in Fig 8.15

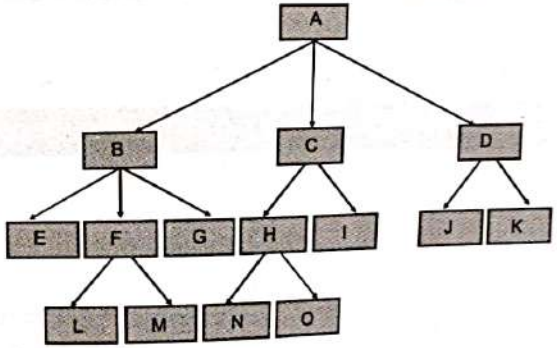


Fig. 8.15

Illustrate the steps involved while doing integration testing using

(a) Top down testing
(b) Bottom up testing
(c) Sandwich testing

11. What is the difference between a stub and a driver?
12. Consider the following code

```
for(i = 1; i < 10; i++)
        for (j = 1;j < 10; j++) {
            sum = i + j;
            prod = i * j;
        }
```

Identify test cases to perform loop testing.

13. For the C code given in problem 12 generate at least 5 mutants.
14. A test case consists of what information?
15. Why special techniques are required to test an object oriented program?
16. What are the advantages of using automated testing tools?
17. Write short notes on
    (a) Capture/Playback tools
    (b) Test case generators
18. How is the activity of debugging different from testing?

○○○

Chapter 9

CONFIGURATION MANAGEMENT

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- What is Configuration Management?
- Baseline
- Configuration Control Board(CCB)
- Version Control
- Software Configuration Management Plan
- Configuration Management Tools

9.1 WHAT IS CONFIGURATION MANAGEMENT?

During the software development life cycle and after its release, software is changed several times. There can be various reasons for this evolution. Some of the reasons worth mentioning are:

- Different hardware platforms
- Change in requirements
- Different operating systems e.g., a complier can exist for Window, Unix or Linux operating system
- Introduction of new technologies

- Change in government laws
- Introduction of new standards
- Feedback from the customer/field
- Faults discovered during testing at various levels.
- Delivery of same product to different customers with minor changes and many more.

As a result of the changes made, multiple versions of the software systems exist at one time. In order to have reliable systems, these changes need to be managed, controlled and documented carefully. If not managed properly, these uncontrolled changes can result into several problems like delivery of defective product, product not matching the requirements of the customer, inability to find and correct defects and of course dissatisfied customers.

Configuration management(CM) helps the developers to manage these changes in the evolving software systematically by applying procedures and standards. In this process automated tools are also used for:

- Storing different versions of system components.
- Building different version of systems for the existing components.
- Releasing appropriate system version to the right end users/customers.
- Controlling update of a software component by multiple persons.

In simple words configuration management is an activity which is in operation throughout the software life cycle. Hence it can be defined as

According to [IEEE Std 1042-1987], software configuration management is the discipline of managing and controlling change in the evolution of software systems.

According to (kelkar07) Configuration Management comprises the documented procedures for applying technical and administrative directions and surveillance to:

- *Identify and document the characteristics of an item or system*
- *Control changes to such characteristics*
- *Record and report changes and their implementation status*
- *Audit items/systems to verify conformance to requirements.*

In short configuration management deals with controlling and documenting the changes made to different artifacts of the software project.

9.2 CONFIGURATION MANAGEMENT TERMINOLOGY

Baseline – Though multiple versions of project components exist in the tool repository there will be only one official version of project components at any point of time. This

set of components is known as baseline. Baseline plus the approved changes defines the current configuration of the system and serves as the basis for further development.

Configuration item – The components of base line are known as configuration items. With time more and more configuration items are added to the baseline. Some of the possible configuration items existing in the baseline are:

- Project plans
- Software configuration management plans
- Requirement specification document
- Design document
- Source code
- Test plans
- Reusable components
- User manuals
- Maintenance plans and many more.

A configuration item is normally treated as single entity by configuration management process.

Version – An instance of a system's state at a point of time during product life cycle which is functionally different in some way from other instances.

Promotion – A version for use of other project member

Release – A version released to end users or customers

Change request – A formal request from a developer or user for change of configuration.

Configuration Control Board (CCB) – A body consisting of people specializing in different phases of software life cycle to ensure authorized change to configuration items or baseline as a whole.

9.3 HANDLING CHANGES: CHANGE MANAGEMENT

Changes made to baseline are subject to approval of configuration control board. Different types of change requests can arise ranging from introduction of new requirements in the SRS to correcting the defect found in some module. Following steps are followed in order to handle change request:

(a) Submit change request along with detailed information about financial implication, affect on other modules of the system, resources required etc. This is done by filling up the form and submitting it to CCB.

(b) Access the change request after evaluating it properly.

(c) Depending upon the results of step (b) change request is either accepted or rejected. If some additional information is required by CCB the change request is deferred for the future assessment.

(d) If the change request is accepted, necessary planning is done to implement the change by assigning work to the developer.

(e) Once changes are made, after validating by quality control team, all the configuration items are updated in the configuration management system library.

A flow chart showing all the steps is shown in Fig 9.1.

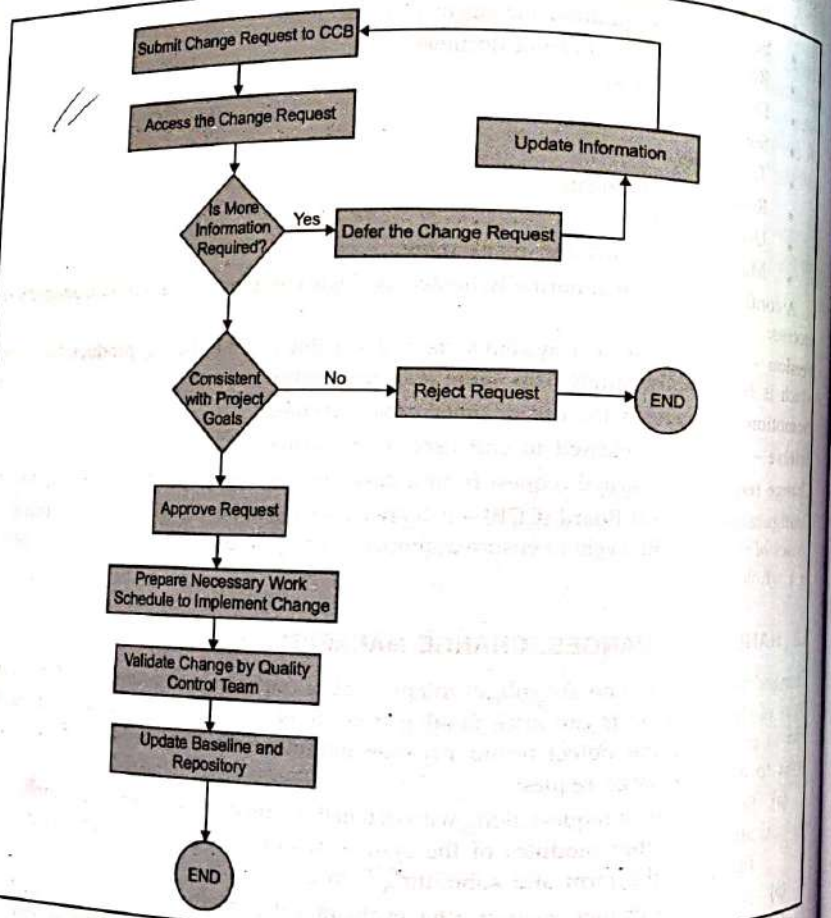


Fig. 9.1 Change Control Procedure

9.4 VERSION CONTROL

A number of different schemes can be used to assign unique identification numbers called version identifier to system and configuration items. One way to do this is by using some kind of numbering scheme e.g., a three digit scheme can be used to indicate major version (major change in the product), minor version (minor change in functionality of the product) and corrections or revisions. A configuration item X during its evolution will go through following steps:

X.1.0.0 (First major release)

X.1.1.0 (First minor release)

X.1.2.3 (Second minor release with number of defect fixing)

The products released for Alpha/Beta testing are denoted using version before 1.0.0 e.g., 0.0.1. In large organizations, even different branches of revisions can be created for parallel development paths. Configuration management systems can store all the versions of the configuration items (version oriented model) or store only the changes made to different versions (change oriented model). These changes are called deltas.

9.5 CONFIGURATION MANAGEMENT PLAN AND TOOLS

Configuration Management plan specifies all the procedures for configuration management to be followed during software life cycle. IEEE standard 828-1990 is a standard that describes the guidelines for writing configuration management plan which can be applied to any kind of project. A sample template based on IEEE standard is described below.

Project Name

Version Number

Revision History

1. Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms, and Abbreviations

1.4 References to other documents used in the plan

1.5 Overview

2. Software Configuration Management

2.1 Organization

2.2 Responsibilities

2.3 Relationship of CM to the software process life cycle

2.3.1 Interfaces to other organizations on the project

2.3.2 Other project organizations CM responsibilities

2.4 Tools, Environment, and Infrastructure: This section describes the computing environment and software tools to be used in fulfilling the CM functions throughout the project or product lifecycle.

3. **The Configuration Management Program**

3.1 Configuration Identification

3.1.1 Identification Methods: Describe the naming and numbering schemes for different project or product artifacts.

3.1.2 Project Baselines: In this section various baselines are identified for the project. For each baseline created following information is provided.

- How and when it is created
- Who authorizes and who verifies it
- The purpose
- What goes into it (software and documentation)

3.2 Configuration and Change Control

3.2.1 Change Request Processing and Approval: Describes the process by which problems and changes are submitted, reviewed, and dispositioned

3.2.2 Change Control Board (CCB): Describes the CCB membership and the procedures for processing change requests and approvals to be followed by the CCB.

3.3 Configuration Status Accounting

3.3.1 Project Media Storage and Release Process: This section describes the policies for backup and the recovery plans. Additionally it also specifies how releases are to be made.

3.3.2 Reports and Audits: Describes the content, format, and purpose of the requested reports and configuration audits.

4. **Milestones:** Defines all CM project milestones (e.g., baselines, reviews, audits) and how the CM milestones tie into the software development process. Further this section also identifies the criteria for reaching each milestone.
5. **Training and Resources:** Identifies the kind and amount of training or tools required for CM activities.
6. **Subcontractor/Vendor Support:** This section identifies if any subcontractor and/or vendor support and interfacing is required.

Some popular configuration management tools are PVCS, VSS (Visual Source Safe), SCCS and MAKE (supplied with UNIX), RCS, Clear Case and CVS.

SUMMARY

- The configuration management is an activity which is in operation throughout the software life cycle.
- Configuration management helps the developers to manage the changes in the evolving software systematically by applying procedures and standards.
- An official version of project components at any point of time is called the baseline.
- Configuration Control Board(CCB) is a body consisting of people specializing in

different phases of Software Product life cycle to ensure authorized changes to configuration items or baseline as a whole.

- IEEE standard (IEEE Std 828-1990) is a standard that describes the guidelines for writing configuration management plan and can be applied to any kind of project.

REVIEW PROBLEMS

1. Define (a) configuration management (b) baseline (c) configuration item (d) version (e) change request (f) configuration control board
2. Explain the scheme to handle changes in the software.
3. Explain the main contents of a configuration management plan.
4. One scheme for version control is defined in the chapter. Identify and describe other schemes being used in the projects for version control.
5. List some factors which trigger the changes in the project.
6. Why Configuration audit is required?

○○○

Chapter 10

SOFTWARE QUALITY

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- What is quality?
- Different models of quality
- Software quality metrics e.g., LOC, Halstead's software science metrics, Mccabe's cyclomatic complexity metric, information flow metric etc.
- Object oriented metrics
- Software reliability
- Software quality assurance
- Software quality frameworks like CMM, ISO9001, Six SIGMA etc.

10.1 WHAT IS QUALITY?

Quality is something which no organization and no customer/end user would like to make compromise with. Poor quality software results into number of problems e.g., inconsistent product, unreliability, poor performance, difficulties in debugging and maintaining the product etc. Developing high quality software is an issue of great importance. The question which therefore immediately comes to our mind is — what exactly do we mean by quality. A number of definitions exist in literature.

According to one definition quality is something which is hard to define, impossible to measure and at the same time easy to recognize.

According to International Standards Organization (ISO1986) quality is defined as: "The totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs."

(Kitchenhan89) defines, the software quality as "fitness for needs."

(Alan97) provides following insights about quality:

- Quality is not absolute.
- Quality is multidimensional.
- Quality is subject to constraints.
- Quality is about acceptable compromises.
- Quality criteria are not independent but interact with each other causing conflicts.

Software quality also has some inherent problems as compared to other areas of manufacturing. They are:

- Users not clear about requirements at the start of the project.
- Requirements changing quite frequently.
- Technology changing very fast.
- High expectations of clients as far as adaptability to new environment is concerned.
- Lack of good quality processes.
- No physical existence of the software.

As quality is quite subjective and multidimensional, a number of views and models of quality are proposed in the literature. Next we discuss some of these views briefly.

10.2 DIFFERENT VIEWS/MODELS OF QUALITY

10.2.1 Garvin's Five Views of Quality

Garvin's five views of quality are (Garvin84):

- The transcendent view
- The product based view
- The user based view
- The value based view
- The manufacturing view
- The value based view

The transcendent view of quality relates the product quality to innate excellence. i.e., one strives towards an ideal but may never be able to achieve it.

The product based view looks at inherent product characteristics and says that cost of

the product is proportional to quality of the product. Higher is the quality, higher will be the cost of product. The product based view of quality asks, "are we building the product right?"

The user based view relates the user requirements to the product design.

The manufacturing view of quality is concerned with conformance to specification and asks the question "are we building the right product?"

The value based view of quality relates the product quality to customer's willingness to pay. It is defined as ability to provide quality to customers at price that is within their budget.

10.2.2 McCall's Model of Quality

McCall's model of quality (McCall77) was an attempt to synchronize and relate the user's view of quality to developers view. It is a type of hierarchical model where each quality criteria has a set of metrics associated with it. McCall looks at software quality from three perspectives each having specific characteristics as shown in Fig. 10.1.

The three perspectives are– product revision, product transition and product operation. The attributes mentioned in Fig. 10.1 are the attributes of quality from user's perspective. These attributes are decomposed so as to relate the user's view to developer's view into lower level attributes called quality criteria as shown in table 10.1.

A further level decomposition is also possible to relate each of these quality criteria to directly measurable attributes called quality metrics. For example with quality criteria correctability, metrics associated are effort and fault count.

Table 10.1 McCall's Model Relating Quality Factors to Quality Criteria

S.No.	User's Perspective (Quality factors)	(Developer's Perspective) (Software Quality Criteria)
1.	Usability	Operability, Training, Communicativeness, I/O Volume, I/O rate
2.	Integrity	Access control, Access audit
3.	Efficiency	Storage efficiency, Execution efficiency
4.	Correctness	Traceability, Completeness, Consistency
5.	Reliability	Accuracy, Error tolerance, Consistency, Simplicity
6.	Maintainability	Consistency, Simplicity, Conciseness, Self Descriptiveness, Modularity
7.	Testability	Simplicity, Instrumentation, Modularity, Self descriptiveness
8.	Flexibility	Expendability, Generality, Modularity, Self descriptiveness

Contd...

9.	Reusability	Generality, Self descriptiveness, Modularity, Machine independence, Software system independence
10.	Portability	Self descriptiveness, Modularity, Machine independence, Software system independence.
11.	Interoperability	Modularity, Data commonality, Communications commonality

Much of the quality work done by researchers is based on the McCall's quality model.

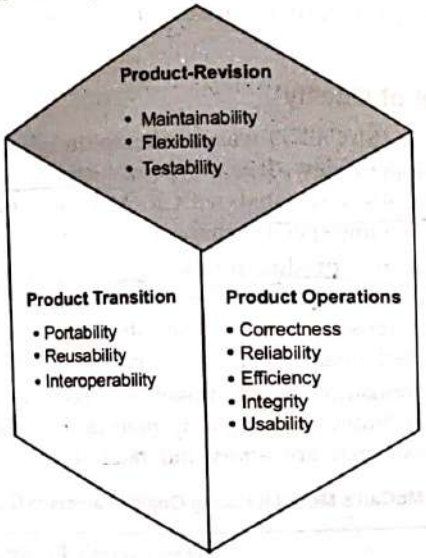


Fig. 10.1 McCall's Quality Factors

10.2.3 Boehm's Software Quality Model

Boehm's quality model also by following decomposition approach provides a well defined characteristics of software quality as shown in Fig. 10.2. This model is also hierarchical in nature consisting of quality criteria and metrics. Quality criteria is shown in Fig. 10.2 and corresponding metrics are given in Table 10.2.

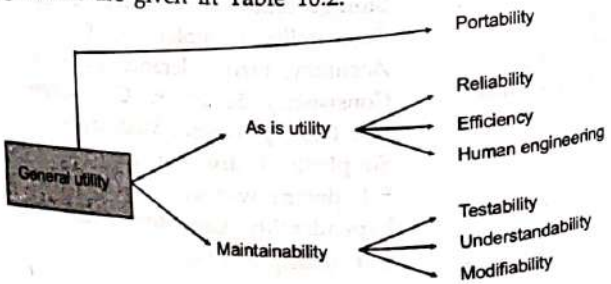


Fig. 10.2 Boehm's Quality Criteria

Table 10.2 Boehm's Quality Criteria and Associated Quality Metrics

S.No.	Quality Criteria	Quality Metrics
1.	Portability	Device-independence, Self containedness
2.	Reliability	Self containedness, Accuracy, Completeness, Robustness, Consistency
3.	Efficiency	Device Efficiency
4.	Human Engineering	Robustness, Accessibility, Communicativeness
5.	Testability	Accessibility, Communicativeness, Structuredness, Self descriptiveness
6.	Understandability	Consistency, Structuredness, Conciseness, Legibility
7.	Modifiability	Structuredness, Augmentability

10.2.4 ALAN's View of Quality

(Alan97) believes that software quality is all about people because of following reasons:

- Problems as well as solutions are specified by the people.
- All the designs and code is implemented by people.
- Testing of code is done by people.
- Final use of systems and overall quality of the product is also judged by people.
- Use of automated tools, quality management systems etc., also depends an capability of people using it.

He therefore emphasized on human aspects of quality.

10.2.5 ISO9126 Quality Model

ISO9126 standard was published in 1991 due to user's need for a common quality model and lack of commonly agreed definitions of quality of software among the users and developer community. The model describes quality in terms of following characteristics:

- Functionality
- Reliability
- Efficiency
- Usability
- Maintainability
- Portability

FREUMP

Additionally to clarify these characteristics, each of these are described in terms of sub characteristics e.g., usability can be expressed in terms of understandability, learnability and operability. A process for evaluating the software quality is also defined by the standard.

10.3 SOFTWARE QUALITY METRICS

As quality is very subjective, there must be some way to measure the quality of software product. It is here that metrics come into picture and help us to predict and control the software product quality. Software metrics are units of measurement, used to characterize software engineering products, processes and people. In other words, software metrics is the term applied to the field of measurement within software engineering. It is to be noted that software engineering metrics are applicable to all the phases of software development life cycle starting from conception to maintenance phase.

According to (Goodman93) Software metrics can be defined as "The continuous application of measurement based techniques to the software development processes and products to supply meaningful information together with use of those techniques to improve software development process and its products."

According to (Fenton87) software metrics are quantitative measures which can be derived from any attribute of software life cycle in a mechanical and algorithmic fashion. For example, if a system has reliability of 0.95, it means that it will operate for 95 periods of time out of 100.

If used properly, software metrics can be used to

(a) Predict success and failure quantitatively of a product, person or process
(b) Control quality of the software.
(c) Make important and meaningful decisions.
(d) Improve code quality and maintainability.
(e) Predict quantified estimates.
(f) Improve scheduling and productivity.

10.4 SOFTWARE MEASUREMENT BASICS

Measurement is the process of assigning numbers to attributes of entities according to some defined rules.

An entity can be an object of interest or an event e.g. source program, concerto, schema, design document etc. In context with software, three types of entities are identified. They are products, processes and resources. Resources and products are associated with the process. Resources are inputs to process and outputs are products. An attribute is a property of an entity e.g. length of program. Attributes can be classified as:

- Internal attributes
- External attributes

Internal attributes of an entity (product, process and resource) are measured in terms of entity itself. They can be measured statically i.e., without execution. Examples of internal attributes are length, modularity etc. ***External attributes*** on the other hand are measured depending upon the way an entity responds to external environment e.g., maintainability, reliability etc. It is to be noted that internal attributes are used to predict the external attributes e.g., a software with poor modularity will result into a software which is difficult to maintain.

10.4.1 Types of Measurement

Measurement can be classified into two types:

- Direct measurement.
- Indirect measurement.

Direct measurement are numbers that can be directly derived from the entity without using any other information e.g., length of program which can be measured by counting number of lines or developer's effort on a project which in turn can be measured in terms of man-months worked. ***Indirect measurement*** are the numbers on the other hand that are computed by combining two or more direct measures e.g.,

$$\text{Developers productivity} = \frac{\text{Lines of code written}}{\text{Man months of effort}}$$

10.5 METRIC CLASSIFICATION

Software quality metrics can be classified in to three categories. They are:

- Product Metrics
- Process Metrics
- Project Metrics

Product Metrics: These metrics describe the internal and external product characteristics e.g., size, complexity, reliability etc. Most of the software metrics come under this category.
Process Metrics: These metrics are used to measure the software development and maintenance process. These metrics help in improving the process e.g., response time for fixing the bug.
Project Metrics: These metrics describe the characteristics of a project e.g., project schedule, budget, team members, productivity etc.

In the following sections we would be discussing some popular product metrics.

10.5.1 Size Metric-Lines of Code (LOC)

The basic and simplest metric for the size is Lines of Code (LOC) often quoted in 1000's (KLOC). In a real sense this metric measures the length of a program which is a logical characteristic but not size which is physical characteristic of the program. While counting number of lines, normally researchers are of the view that comments and blank lines should not be counted. Comments used in the program make the program understandable

and easier to maintain but effort required for writing the comments is not as much as writing the code. According to some researchers LOC may be computed by counting the new-line characters in the program. This metric can also be used in other indirect measures as well, such as

$$\text{Productivity} = \text{LOC/Effort}$$

Other similar measures used are KDSI (1000's of delivered source instruction), NLOC, (Non-comment lines of Code) etc. The problem with LOC metric is that it can be measured only at the end of life cycle.

10.5.2 Halstead's Software Science Metrics

Maurice Halstead was one of the prominent researcher of late 60's who contributed significantly in the area of software metrics. His theory of software science is one of the best theory for measuring software complexity (Halstead77).

Halstead proposed that program may be considered as a collection of lexical tokens, each of which can be classified as operator or operand. Operands are that tokens which have value. Data variables and constants therefore constitute the operands. Operators are:

- Commas
- Parenthesis
- Keywords
- Arithmetic operators
- Functions

and so on. Tokens that appear in pairs are counted as one token.

The primitive measures proposed by Halstead are

η_1 — The number of distinct/unique operators that appear in the program.
η_2 — The number of distinct/unique operands that appear in the program.
N_1 — The total number of operator occurrences.
N_2 — The total number of operand occurrences.

These measures are then used to develop expression for overall program length, program volume etc.

The Vocabulary of the program η is given as

$$\eta = \eta_1 + \eta_2$$

and length of the program is given by

$$N = N_1 + N_2$$

Using η_1, η_2, N_1 and N_2 as basis, Halstead proposed a theory of software complexity and effort. The estimated program length N in terms of tokens is given by

$$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

This value can be estimated before the program is written.

Volume V i.e., number of bits required to encode the program being measured is given by

$$V = N \log_2 (\eta)$$
$$= N \log_2 (\eta_1 + \eta_2)$$

The effort E required to implement the program P is given by

$$E = \left(\frac{\eta_1 N_2}{2\eta_2}\right)(N \log_2 \eta)$$

Halstead used the notion of elementary mental discriminators (emd) as units for effort E.

Halstead also estimated the time T required to implement the program as

$$T = (E/18) \text{ seconds}$$

The $\log_2$ of X is the exponent to which 2 must be raised to give value equal to X i.e., $\log_2$ of 8 is 3. Applications of Halstead's concepts to a program are discussed in Example 10.1.

Example 10.1 Consider the program code given below

```
int sum, i;
sum = 0;
for   (i = 1; i <=20; i++)
      sum = sum + i;
printf(sum);
```

For this program compute Halstead software science metrics i.e., η_1, η_2, N_1, N_2, η, N, V, E and T.

Solution. List of operators and operands of the program along with the number of occurrences of each of these is given below:

Operators			Operands		
int	—	1	sum	—	5
;	—	6	i	—	5
=	—	3	0	—	1
( )	—	2	1	—	1
<=	—	1	20	—	1
++	—	1			
for	—	1			
+	—	1			
Print	—	1			
	—	1			

Contd...

Here number of distinct operators are 10 and number of distinct operands are 5. Similarly total number of operator occurrences is 18 and total number of operands occurrences is 13. Therefore values of η_1, η_2, N_1 and N_2 are:

$\eta_1 = 10,\ \eta_2 = 5$

$N_1 = 18,\ N_2 = 13$

Therefore, Vocabulary η of the program is given by:

$$\eta = \eta_1 + \eta_2 = 10 + 5 = 15$$

Length of the program $N = N_1 + N_2 = 18 + 13 = 31$

Estimated program length in tokens is $= \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

$= 10 \log_2 10 + 5 \log_2 5$

$= 10 * 3.32 + 5 * 2.32$

$= 44.8$

Volume V $= N \log_2 \eta$

$= 31 \log_2 15$

$= 31 * 3.91$

$= 121.21$ bits

$$\text{Effort } E = \left(\frac{\eta_1 N_2}{2\eta_2}\right)(N \log_2 \eta) = \left(\frac{10 \times 13}{2 \times 5}\right)(31 \log_2 15)$$

$= 403 \log_2 15$

$= 403 * 3.93$

$= 1583.79$ emd

Therefore, Time T $= E/18$

$= 1583.79/18$

$= 87.98$ seconds

10.5.3 McCabe's Cyclomatic Complexity Metric

McCabe's Cyclomatic Complexity Metric (McCabe76) is one of the popular software metric and is based on graph theory. Given a flow graph G with *e* edges and *n* nodes for a program P, the cylcomatic complexity of program P is given by

$$C(P) = e - n + 2$$

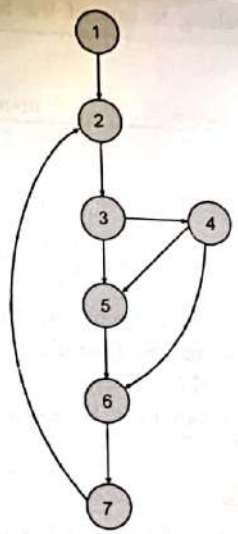


Fig. 10.3

The value of C (P) gives us the number of independent paths in the graph G. Consider the control flow graph shown in Fig. 10.3. In this control flow graph

number of nodes (n) = 7

number of edges (e) = 9

Therefore Cyclomatic complexity C is given by

$C = e - n + 2$

$= 9 - 7 + 2$

$= 4$

For detailed discussion on McCabe's Cyclomatic complexity metric refer to section 8.8.5 of chapter 8.

10.5.4 Measuring Functionality—Function Point Analysis

Many researchers proposed that functionality of product gives a better idea about the product size. A number of approaches are proposed to measure functionality. One such approach is Albrecht's Function Point Analysis (FPA) which measures the product functionality in terms of function points. Steps in calculating the function points are given below.

1. Compute Unadjusted Function Count (UFC) using the formula

$$UFC = \sum_{n=1}^{5} (\text{complexity weight}) * \text{No. of items of type n}$$

Where software items can belong to one of the following types

- External inputs
- External outputs
- Internal files
- External files
- External inquiries

2. Compute Technical Complexity Factor (TCF)

$$TCF = 0.65 + 0.01 \sum_{i=1}^{14} Factor_i$$

where $Factor_i$ represents one of 14 technical complexity factors.

3. Compute Function Points using the formula

$$FP = UFC \times TCF$$

The function points computed can now be used for effort estimation. For details refer to section 2.4.3 of chapter 2.

10.5.5 Information Flow Metrics

Information Flow metrics proposed by Henry and Kafura (Henry81) measure the level of information flow among different modules. In other words they measure the inter module complexity. For a module X let Fan-In is the count of all information that flows into the module and Fan-Out is the count of all information that flows out of the module.

Information flow index for module X is then given by:

$$IF(X) = [Fan\text{–}In(X) * Fan\text{–}Out(X)]^2$$

The total information flow measure for the software S consisting of n modules is:

$$IF(S) = \sum_{i=1}^{n} IF(X_i)$$

This is illustrated in Example 10.2.

Example 10.2 For the structure chart given in Fig. 10.4 calculate the Information flow index of individual modules as well as whole software.

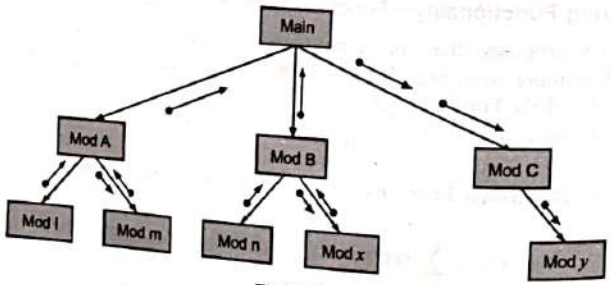


Fig. 10.4

Solution:

Module	Fan-in	Fan-out	Information flow index of module (Fan-in * Fan-out)²
main	2	2	16
A	2	2	16
B	2	2	16
C	2	1	4
l	0	1	0
m	1	1	1
n	0	1	0
x	1	1	1
y	1	0	0

Information flow index of individual module is computed. The total information flow index is given by

$$\begin{aligned} IF &= IF(main) + IF(A) + IF(B) + IF(C) + IF(l) \\ &\quad + IF(m) + IF(n) + IF(x) + IF(y) \\ &= 16 + 16 + 16 + 4 + 0 + 1 + 0 + 1 + 0 \\ &= 54 \end{aligned}$$

10.6 OBJECT ORIENTED SOFTWARE METRICS

In recent years, introduction of complex systems has made object oriented design very popular. Centre of object oriented design is the notion of class. As a result quality of class component is an important issue. In this context a number of software metrics are also proposed for measuring and controlling the quality of class components. A number of models are also proposed for class component measurements. A metric suite is proposed by (Chidamber94) for evaluating class component quality by measuring the complexity of design. We would briefly discuss some of these object oriented metrics proposed by (Chidamber94).

1. **Depth of Inheritance Tree (DIT).** This metric for each class evaluates the maximum length of inheritance tree from root to that child class. More is the value of this metric, more is the complexity. A value of 0 indicates that there is no inheritance.
2. **Number of Children (NOC).** For each class, this metric evaluates the number of child classes. Average value of metric is computed as follows.

   Let C_i be number of children for i^{th} class

   N = (number of total classes – number of child classes at lowest level)

then for n classes average value of NOC is given by

$$[NOC]_{AVG} = \frac{\left(\sum_{i=1}^{n} C_i\right)}{N}$$

The value of NOC > 4 indicates improper abstraction.

3. **Number of Methods per Class.** Number of Methods(NOM) per class contributes significantly for measuring the complexity of the object oriented software. This is done by counting the number of methods of each class plus number of methods inherited from parent classes. A very high average value of number of methods per class demands further object oriented decomposition.

   Let M_i = number of methods for ith class
   C_i = complexity of the class (not mentioned in the paper)
   N = Number of classes.

   Assuming $C_i = 1$ for all classes in the model, Average value for NOM is

$$NOM_{AVG} = \frac{\sum_{i=1}^{N} M_i}{N}$$

4. **Coupling between Object Classes (CBO).** Coupling between Object classes is the measure of interdependence between classes of object model. In other words this metric indicates coupling between the classes. This is done by counting the number of non-inheritance relationships (methods of one class use methods or attributes of other class) between different classes. This metric is computed for all classes. Higher value of CBO for a class indicates tight coupling with other classes. This also requires rigorous testing and prevents reuse. A value of 0 indicates that a class is not related to any other class of the model and therefore must be discarded.

5. **Response for a Class (RFC):** The Response Set (RS) of a class is the set of methods that can potentially be executed in response to a message received by an object of the class. The RS consists of therefore all methods in the class and all methods called by methods in the class.

   $RFC = |RS|$

   Higher is the value of RFC, higher is the class complexity, which requires rigorous testing of class.

6. **Lack of Cohesion in Methods (LCOM).** This metric measures lack of cohesiveness within classes. Let for each class C_i, I_i is set of instance variables for corresponding method M_i. Therefore for n such methods there will be n such types of sets. *i.e.*, $\{I_1\}$, ... $\{I_n\}$ for each class.

   Let $A = \{(I_i, I_j) \mid I_i \cap I_j = \Phi\}$ for $i \neq j$

   and $B = \{(I_i, I_j) \mid I_i \cap I_i \neq \Phi\}$ for $i \neq j$.

Then Lack of Cohesion in Methods (LCOM) is defined as

$$LCOM = \begin{cases} |A| - |B| & \text{if } |A| > |B| \\ 0 & \text{otherwise} \end{cases}$$

In other words

LCOM = Number of null interactions – Number of non-empty interactions

Higher is the value of LCOM, more cohesive is the class.

Example.10.3 *Calculate the chidamber metrics for a C++ program given below. The program enters the data of a student and displays it using multiple inheritance technique.*

```
#include <iostream.h>
#include <coino.h>
class student
{
  Private:
        int   student_id;
        char  sname[30];
  public:
        void get_data()
        {
              cout << "Enter Student Id...";
              cin >> student_id;
              cout << "Enter Student Name...";
              cin >> sname;
        }
        void put_data()
        {
              cout << "Student Id is...;" << student_id ; << endl;
              cout << "Student Name is...;" << sname << endl;
        }
};
class marks
{
  Private:
        int   maths;
        int   science;
  public:
```

```
        void get_data()
        {
            cout << "Enter Mathematics marks ...";
            cin >> maths;
            cout << "Enter Science marks ...";
            cin >> science;
        }
        void put_data()
        {
            cout << "Mathmatics marks are..." << maths << endl;
            cout << "Science marks are..." << science << endl;
        }
};
class studentresult : public student, public marks
{
  public:
        void get_data()
        {
            student::get_data();
            marks::get_data();
        }
        void put_data()
        {
            student::put_data();
            marks::put_data();
        }
};
void main()
{
  class studentresult s1;
  clrscr();
  cout << "Enter the Student information ..." << endl;
  s1.get_data();
  cout << "Student Details are ... " <<endl;
  s1.put_data();
  getch();
}
```


Solution. The Chidamber metrics are computed below:

Class	Depth of inheritance	No. of children	Weighted methods per class
Student	0	1	2
Marks	0	1	2
Students result	1	0	2

Coupling Between Object Classes

To compute the coupling between classes let us represent the classes as shown in Fig. 10.5 which shows which function is called by which function of other classes.

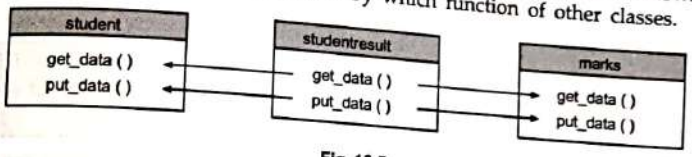


Fig. 10.5

From the Fig. 10.5 it is clear that get-data() function of class studentresult calls get-data() function of both student and marks classes and hence CBO of class studentresult is 2. Similarly CBO for student and marks class is 1.

Response for a Class (RFC)

Class	Response set	RFC
student	get_data(), put_data()	2
marks	get_data(), put_data()	2
studentresult	get_data(), put_data(), student : get_data(), student : put_data(), marks : get_data(), marks : put_data()	6

Lack of Cohesion in Methods (LCOM)

Let us first consider class student with methods M_1 and M_2 i.e., get_data() and put_data() respectively.

Let $\{I_1\}$ = {student_id, s_name}

$\{I_2\}$ = {student_id, s_name}

represent set of variables used by M_1 and M_2 respectively.

$\{I_1\} \cap \{I_2\}$ is non-empty.

Let P = number of null interactions

Q = number of non-null interaction

For class student

P = 0

Q = 1

Therefore, LCOM = max(0, (0 – 1)) = 0

Similarly for class marks, LCOM = max(0, (0 – 1)) = 0

and for class studentresult, LCOM = max(0, (0 – 1)) = 0

10.7 SOFTWARE RELIABILITY

All the quality models discussed in section 10.2 have given lot of importance to reliability characteristic of software quality. Understanding software reliability is an issue of great importance and requires proper knowledge and skills.

In simple language software reliability can be defined as the probability, the software will operate failure free for a specified duration under specified operating conditions.

Failures are deviation of operational behavior of software from customer expectations. Since 70's, number of software reliability models are proposed.

10.7.1 Hardware Vs Software Reliability

Hardware reliability has long history and is much better understood as compared to software reliability. The main reason of hardware failure is the physical changes that take place in the hardware over a period of time i.e., the components deteriorate (due to overheating, corrosion etc.) and finally wear out. However there is no wear out phenomenon in case of software and failures mainly take place due to its exposure to wrong environment. As compared to hardware, software can be replicated to high standards of quality. In case of hardware failure, it can be corrected by simply replacing the failed components with new components. The system after correction restores to same reliability level. In other words hardware reliability has a tendency towards constant value. In software it is not the same. Once the failures are fixed, repaired, software reliability improves. It means that it has a tendency to change and improve specially during testing periods. As both hardware and software are integral part of a system, specially in today's scenario, both software and hardware reliability can be combined in order to make a reliable system. Over a period of time therefore, software reliability theory is developed so as to make it compatible with hardware reliability theory.

10.7.2 Reliability Theory Basics

The whole theory of reliability revolves around predicting the failure. This can be done by assigning a probability with the product or defining a probability density function $f(t)$ of time t for a component. The probability of failure for a component between time interval t_1 and t_2 can be computed by integrating the probability density function (Pdf). Hence probability of failure between time

$$t_1 \text{ and } t_2 = \int_{t_1}^{t_2} f(t)dt$$

Let us call it F(t). Reliability function R(t) can now therefore be expressed as

$$R(t) = 1 - F(t)$$

The function R(t) computes the probability that component (software in this case) will operate without failure between time t_1 and t_2. If starting time t_1 is taken as time 0 and t_2 equal to t, R(t) computes the probability of failure free operation up to time t. Other parameters of interest are:

(a) hazard rate $h(t) = \frac{f(t)}{R(t)}$

(b) The mean time to failure (MTTF) = $\int tf(t)dt$

(c) The mean time between failure (MTBF) expressed as

$$MTBF = MTTF + MTTR$$

where MTTR is mean time to repair a component.

(d) Software availability is the probability that software or a component is operating successfully at a given point in time and is expressed as

$$\text{Software availability} = \frac{MTTF}{MTTF + MTTF} \times 100\%$$

10.7.3 Software Reliability Terminology

In context with software reliability, following terms are commonly used (Robert02), (Musa87):

- **Error**—Error is defined as a human action that results into an incorrect result. It is a problem found in current phase.
- **Defect**—Defect is a problem which is found in a phase other than where it was introduced.
- **Fault**—A fault is defined as a defect in the program, that when executed under particular conditions causes a failure.
- **Failure**—Departure of program operation from requirements is called failure. Failure is dynamic in nature.
- **Fault-tolerant**—Fault tolerant is the property of the system to defend against certain faults and keep operating.
- **Execution time**—The execution time of a program is the actual processor time used for executing the program.
- **Calendar time**—It is the time which we normally experience.
- **Robustness**—It is the property of a system which shows its tolerance to bad inputs.

10.7.4 Software Reliability Models

Software reliability focuses on predicting when a system will eventually fail which is normally done by building a model.

According to (Ali96) a model is a representation of reality through a mathematical expression involving one or more measurable variables or parameters.

It is seen that different proposed models predict differently for the same set of failure data. Some desirable characteristics/criteria for choosing a model are (Iannino84), (Ali96):

- The capability of model to predict future failure behaviour from known characteristics of the software i.e. lines of code, language used, present and past failure data etc.
- The degree of model simplicity in terms of use, automation etc.
- The ability of a model to estimate with satisfactory accuracy the quantities needed for planning and managing software project.
- The degree of model completeness i.e., all necessary parameters and constraints etc., are identified and included in the model.
- The quality of modeling assumptions made.
- The degree by which parameters can be related to program information prior to its execution.
- The degree of model validity.

Software reliability models can be categorized into two categories. They are

- Time dependent software reliability models.
- Time independent software reliability models.

Time dependent reliability models are the popular category of models which express reliability relationships as a function of time. Time independent reliability models on the other hand establish relationship between detected and undetected faults in term of static dependencies. Based on these approaches, several reliability models are proposed in the literature, (Jelinski72), (Moranda75), (Shooman72), (Shooman76), (Schick72), (Musa75), (Goel79), (Mazzuchi88) etc. Discussing all these models is beyond the scope of this book. However we would be discussing few popular simple reliability models. For detailed study readers can refer to handbook of software reliability engineering (Lyu95) which covers all the topics in detail.

Basic Execution Time Model

The basic execution time model (Musa75) is simple, easy to use and understand and is applied to several projects successfully. Failure intensity function which is defined as number of failures per unit time is expressed as:

$$\lambda(\mu) = \lambda_0 (1 - \mu / v_0) \qquad \ldots(i)$$

where λ_0 = failure intensity at the start of execution i.e., at time 0

μ = mean or expected number of failures experienced

v_0 = number of failures that occurs in infinite time

The slope of failure intensity function is given by

$$d\lambda / d\mu = -\lambda_0 / v_0 \qquad \ldots(ii)$$

Both λ and μ are functions of execution time τ. Number of failures as a function of execution time can be expressed as

$$\mu(\tau) = v_0 \left[1 - \exp\left(\frac{-\lambda_0}{V_0}\tau\right)\right] \qquad \ldots(iii)$$

and failure intensity as a function of execution time is given by

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0}{V_0}\tau\right) \qquad \ldots(iv)$$

Logarithmic Poisson Execution Time Model

Though both basic and Poisson execution time models assume that failures occur randomly, the main difference between the basic execution time model and logarithmic Poisson execution time model [Musa84] is in terms of decrement per failure or slope of failure intensity function. In case of Poisson execution time model decrement per failure decreases exponentially but in the basic model it remains constant. The failure intensity function is expressed as

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu) \qquad \ldots(v)$$

where θ = failure intensity delay parameter

Similarly $$\frac{d\lambda}{d\mu} = -\theta\lambda_0 \exp(-\theta\mu)$$

$$= -\theta\lambda$$

$$\mu(\tau) = \frac{1}{\theta} \ln(\lambda_0\theta\tau + 1) \qquad \ldots(vi)$$

and $$\lambda(\tau) = \lambda_0 / (\lambda_0\theta\tau + 1) \qquad \ldots(vii)$$

Similarly the quantity expected no. of failures to reach the failure intensity objective is given by

$$\Delta\mu = \frac{1}{\theta} \ln \frac{\lambda_p}{\lambda_F} \qquad \ldots(viii)$$

where λ_P = current failure intensity

λ_F = failure intensity objective

and additional execution time $\Delta\tau$ required to reach the failure intensity objective is expressed as

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_p}\right] \qquad \ldots(ix)$$

$$\ldots(x)$$

For basic execution time model, these quantities are

$$\Delta\mu = \frac{V_0}{\lambda_0}[\lambda_p - \lambda_F] \quad \ldots(xi)$$

and

$$\Delta\tau = \frac{V_0}{\lambda_0}\ln\frac{\lambda_p}{\lambda_F} \quad \ldots(xii)$$

In both the models the problem the engineer faces is the estimation of parameters. This can be done by recording first N failures at different points in time, and from these deriving the model parameters using techniques like least squares. Then using the formulas discussed above predictions can be made. Finally in all the formulas notion of execution time is used which can be related to calendar time.

Example 10.4 For basic execution time model

Given λ_0 = 30 failures/CPU hr

and v_0 = 120 failures

Compute additional failures and additional execution time required to reach a failure intensity objective of 10 failures/CPU hr.

Solution: For basic execution time model

$$\Delta_\mu = \frac{V_0}{\lambda_0}(\lambda_p - \lambda_F)$$

$$= \frac{120}{30}(30\text{–}10) = 80 \text{ failures}$$

$$\Delta_\tau = \frac{V_0}{\lambda_0}\ln\frac{\lambda_p}{\lambda_F} = \frac{120}{30}\ln\frac{30}{10}$$

$$= 4 \ln 3 \text{ CPU hr}$$

$$= 4.394 \text{ CPU hr}$$

Example 10.5 Given number of failures experienced by a program in infinite time is 120 and mean failures experienced are 60. The initial failure intensity was 12 failures/CPU hr. Compute the current failure intensity for basic execution time model.

Solution: Given $\mu = 60$

$v_0 = 120$

$\lambda_0 = 12$

The current failure intensity is given by

$$\lambda(\mu) = \lambda_0 (1 - \mu/v_0)$$

$$= 12 (1 - 60/120)$$

$$= 6 \text{ failures/CPU hr.}$$

10.7.5 Approaches to Build High Reliable Software

(Robert02) has discussed four approaches in order to achieve highly reliable software. They are fault forecasting, fault prevention, fault removal and fault tolerance. Fault forecasting focuses on determining the functional profile, defining the failures, identifying the customer reliability needs and setting the reliability objectives using reliability models, historic data analysis etc. This is done normally at requirements definition phase and system exploration phase. Fault prevention emphasizes on refining the system requirements using techniques like reviews, inspections and other formal methods. Fault removal activity starts the moment faults are discovered. Additional testing is required here in order to ensure that reliability objectives are met. Finally fault tolerance activity concerns with proper system functions after the faults in the delivered systems manifest themselves. Further all these activities are supported by automated reliability tools.

Another discipline is software reliability engineering which is a sub discipline of software engineering. According to (Musa94) Software Reliability Engineering (SRE) is a quantitative study of how well the operational behavior of software-based systems meet user requirements. *SRE* makes use of several techniques like prediction, statistical estimation and modeling, measurement etc. It also emphasizes on finding how and where faults are introduced and ways to reduce these faults thereby resulting in highly reliable software. It can also be used to develop continuous improvement cycle for software development process. It also helps in specifying the requirements more precisely with respect to attributes, achieving optimum trade off for customer with respect to schedules, budget and reliability and planning the maintenance operations accurately.

10.8 SOFTWARE QUALITY ASSURANCE (SQA)

Software quality assurance can be thought of as a framework which ensures that quality standards and practices have been used to develop a good quality software product which conforms to the end user/customer requirements. This framework applies both to the product as well as the processes followed during the software development life cycle.

According to IEEE standard for Software Quality Assurance plans [STD 730.1 – 1995], software quality assurance can be defined as—

- **the planned and systematic pattern of all actions necessary to provide adequate confidence that end product conforms to established technical requirements.**
- **a set of activities designed to evaluate the process used for developing and manufacturing products.**

Capability Maturity Model (CMM) has also proposed Software Quality Assurance as one of the key process area at the repeatable level i.e., level 2. The main objective of SQA activities is reviewing all products (intermediate as well as end product) and processes used during software life cycle. Checklists etc., can also be used during review. Software metrics can also be identified and used to support software quality assurance program.

A number of standards have been proposed for documenting SQA plans. The standard developed by IEEE standards association for SQA plan is IEEE 730-1998. The main constituents of the plan are:

Template for SQA Plan based on IEEE 730-1998.

1. Purpose of SQA plan.
2. List of reference documents.
3. Management tasks, responsibilities etc.
4. Documentation of all required documents.
5. Standards, practices, metrics etc., applied and used during development.
6. Details of reviews, audits etc., to be used.
7. Details of tests unique to SQA.
8. Details of procedures for reporting problems and means to resolve the problems.
9. Identification of tools, techniques and methodologies and their use in SQA.
10. Description of techniques for code control.
11. Description of methods for media control.
12. Provisions for supplier control in case of outsourcing.
13. Documentation of all project records.
14. Training details.
15. Procedures for risk management.

It is to be noted that SQA activities must be applied to all the stages of software life cycle starting from requirement analysis till the maintenance. If applied sincerely by the organization it will result in detecting more than 70% errors before the final product is delivered to the customer.

10.9 SOFTWARE PROCESS MATURITY FRAMEWORKS AND QUALITY STANDARDS

In chapter 1 several software process models have been described. We have seen that use of any particular process model is project specific. The success or failure of any software project depends not only on the process model used but also the degree to which processes are implemented. In other words process maturity of a project or an organization plays a very important role. In this section we would discuss various frameworks and quality standards used for accessing the process maturity of any organization. Some of them are specific to only software processes whereas some quality standards are applicable to all industries.

10.9.1 Capability Maturity Model (CMM)

The CMM has been developed by Software Engineering Institute (SEI) at Carnegie-Mellon University around 1987 and is still undergoing the process of refinement. It is not a software process model. Instead it is a framework used to assess the processes followed

by any organization to develop the software and provides guidelines to enhance their maturity further. In other words the CMM prescribes a systematic path from an adhoc process to a mature process by moving through five levels. The model is based on the best practices followed in the organizations world wide. Extensive feedback was also taken both from government and industry while developing and refining this model. The five CMM levels of maturity (in increasing order) are:

1. **Initial**—Characterized by adhoc processes.
2. **Repeatable**—Basic Project Management techniques are established.
3. **Defined**—Focuses on documentation of Software Engineering Process.
4. **Managed**—Quantitative quality goals are set for software process and product.
5. **Optimizing**—Focuses on continuous process improvement for better efficiency.

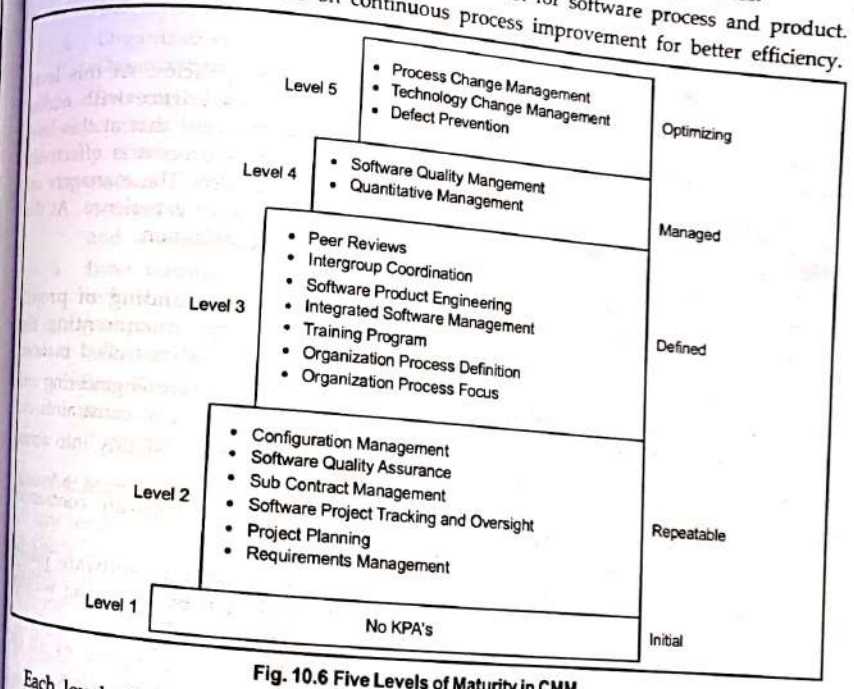


Fig. 10.6 Five Levels of Maturity in CMM

Each level of maturity shows a process capability level. All the levels except level 1 are further decomposed into number of Key Process Areas (KPAs). The key process areas at any level address the issues/targets that must be satisfied by the organization to achieve that maturity level. For example for maturity level 4 i.e., managed level, an organization must address all KPA's at levels 2, 3 and 4. Each of the KPAs at various

levels identifies some goals which must be achieved by the organization in order to satisfy the KPA.

Next we briefly discuss each of these five maturity levels.

Level 1: Initial

At the level 1 the processes followed by the organization are not well defined. They are completely adhoc and immature. As a result a stable environment is not available for software development. Further success and failure of any project depends on the team member's competence. There is no basis for predicting the product quality, time for completion etc. Unless the people are available for a new project of similar nature, processes followed in earlier successful projects cannot be repeated. No KPA's are defined at level 1.

Level 2: Repeatable

The level 2 focuses on establishing basic project management policies. At this level organization follows the processes at project level. Therefore experience with earlier projects is used for managing new projects of similar nature. It means that at this level standards for projects are defined and followed also. A project's process is effectively controlled by the project management system and is repeatable also. The managers are able to predict the cost and schedule of the project based on earlier experience. At this level organization can be called a disciplined and organized organization.

The key process areas to achieve this level of maturity are

- Requirement Management. Establishes common understanding of project established between the developer and customer by documenting the requirements. Requirements are approved and changed in a controlled manner.
- Project Planning. Establishes plans for performing the software engineering and for managing the software project, by defining resources, goals, constraints etc.
- Software Project Tracking and Oversight. Establishes proper visibility into actual progress of software project.
- Subcontract Management. Focuses on selection of qualified software contractors and their effective management.
- Software Quality Assurance. Provides adequate visibility into software project process and the products being built by the management by following suitable quality standards.
- Configuration Management. Focuses on maintaining the integrity of all the products of the software project throughout the lifecycle of the project.

Level 3: Defined

At this level standard processes to be used across the organization are defined and documented. This is referred as organization's *Standard Software Process* and includes both

software engineering as well as management process. Standardization of organization wide processes helps the manager and other team members to perform efficiently. Training programs are also implemented to enhance the manager's and staff member skills. Organization wide standard software process can be further instantiated to meet a project's specific requirements and is called the project's defined software process. It is a well defined integrated set of project specific software engineering and management processes. A process is well defined, there is now visibility into the construction process. Compared to Level 2 defects are down 20% and project cycle wise is down 10%. The main features at level 3 are:

- **Organization Process Focus** – Focuses on improvement of organization's overall software process capabilities.
- **Training Program** – Focuses on development of knowledge and skills of individuals by providing suitable training so as to increase their efficiency.
- **Organization Process Definition** – Focuses on development and maintenance of Organization Standard Process along with set of software process assets.
- **Integrated Software Management** – Focuses on integration of software engineering and management activities into well defined coherent software process.
- **Software Product Engineering** – Focuses on consistent performance of a well defined engineering process, integrating all activities in order to produce correct and consistent software products efficiently and effectively.
- **Inter Group Coordination** – Establishes means by planning interactions and technical interfaces between the different groups so that different engineering groups of a project participate/coordinate with each other in order to meet the customer needs efficiently.
- **Peer Reviews** – Focuses on removing the defects efficiently from the different software work products early in the life cycle. This is done by using a number of review methods like inspections, walkthrough etc.

Level 4: Managed

At this level of CMM, quantitative quality goals are set for the organization for software products as well as software processes. For all projects defined processes, quality and productivity is measured and maintained by organization wide software process database. It is essential to give feedback to the management at each process step. Efforts focus on software product quality expectations. The measurements made also help the organization to predict the product and process quality within some limits defined quantitatively. In case, quality is not met, necessary action is taken to improve the quality. The key properties associated at the managed level are described below:

- **Quantitative Process Management.** Focuses on controlling the software project process performance quantitatively.

- **Software Quality Management.** Focuses on developing a quantitative understanding of software project's product quality by establishing strategies and plans.

Level 5: Optimizing

This is the highest level of process maturity in CMM and focuses on continuous process improvement in the organization using quantitative feedback. New tools, techniques and technology is introduced to see their effect on software product and process. Defects in the products are analyzed to know their causes. Similarly, software processes are evaluated to avoid recurrence of known defects. Organizations continue to improve the process capability in order to improve their project's process performance. The main features at level 5 are described below:

- **Defect Prevention.** Focuses on identification of causes of defects and to prevent them from recurring in future projects by improving project defined process.
- **Technology Change Management.** Focuses on identification of new technologies i.e., tools, methods and processes and transferring the same into the organization in a systematic way in order to improve product quality and decrease the product development time.
- **Process Change Management.** Focuses on continuous improvement of software processes used in the organization in order to improve productivity, quality and cycle time for software product development.

10.9.2 ISO 9000 Series Standards

ISO 9000 series of standards developed by International Standards Organization(ISO) are generic standards for quality management. As compared to CMM, which is mainly developed for software industry, these standards are not industry specific. In fact, it is a series of five standards (ISO 9000, ISO 9001, ISO 9002, ISO 9003 and ISO 9004) which can be applied to a wide range of industries. Out of these five series, standard ISO 9001 is the standard which is most suitable for software development and maintenance and is described by the guideline, ISO 9000-3. Next we briefly discuss the ISO 9001 standard.

ISO 9001

There are 20 clauses that define the minimum requirements for setting up a quality management system for use in software development and maintenance. These 20 clauses are listed below along with their important features.

1. **Management Responsibility** – Requires documenting, understanding, implementation and maintenance of quality policy and specifying responsibilities and authorities for all personnel monitoring quality.
2. **Quality System** – Establishing documented quality system including detailed procedures and instructions.

3. **Contract Review** – Review of contracts in order to ensure that requirements are defined properly, agree with the bid, and are implement able.
4. **Design Control** – Establishing procedures for controlling and verifying the design related activities.
5. **Document and Data Control** – Control of distribution and modification of document and data.
6. **Purchasing** – Ensuring that the purchased products are as per the specified requirements and evaluating potential subcontractors.
7. **Control of Customer Supplied Product** – Verification, control and maintenance of customer supplied product.
8. **Product Identification and Traceability** – Identification and traceability of product at all stages of production, delivery and installation.
9. **Process Control** – Defining and planning the production processes and to ensure that production is carried out under controlled condition.
10. **Inspection and Testing** – Inspection and verification of incoming materials before use. Performing in process inspections and testing and maintaining their records.
11. **Control of Inspection, Measuring and Test Equipment** – Controlling, calibrating and maintaining the equipment used for demonstrating conformance.
11. **Inspection and Test Status** – Maintaining the status of test and inspections of item.
13. **Control of Non-conforming Product** – Controlling the non-confirming product so as to avoid inadequate use or installation.
14. **Corrective and Preventive Action** – Identifying the reasons of non-conforming product. Eliminating causes through corrective and preventive action.
15. **Handling, Storage, Packaging and Preservation Delivery** – Establishing & maintaining procedures for handling, storage, packaging and delivery.
16. **Control of Quality Records** – Collecting and maintaining the quality records.
17. **Internal Quality Audits** – Planning and performing the audits and correcting the deficiencies found during audit.
18. **Training** –
    - Identifying training needs of organization.
    - Providing the training to the concerned personnel and maintaining training records.
19. **Servicing** – Performing servicing activities as per specification. In other words it refers to maintenance.
20. **Statistical Techniques** – Identifying the statistical techniques and using these techniques for verification of process capability as well as product characteristics.

In order to obtain ISO 9001 certification a formal audit is done and audit outcome

must be positive. It is to be noted that there is a strong relationship/correspondence between the CMM and ISO 9001.

10.9.3 Comparing ISO 9001 and the CMM

A comparison between the two has been done by (Paulk94). While CMM focuses on continuous software process improvement, ISO 9001 talks about minimum requirement for acceptable quality system. ISO 9001 can be used for software, hardware, processed materials, whereas CMM focuses only on software services (Marquardt91). Both of them also emphasize the documentation of processes for later use. As each of the Key Process Areas in CMM are described in terms of Key practices, these Key practices can be related to ISO 9001 clauses for proper comparison. It is to be noted that KPAs at Level 2 are strongly related to ISO 9001. Delivery and Installation clause of ISO 9001 is not adequately covered in CMM and is under consideration. For more detailed description and comparison of CMM and its comparison with ISO 9001, readers can refer to the book *"The Capability Maturity Model: Guidelines for Improving the Software Process"* by Carnegie Melon University and Software Engineering Institute Published by Pearson Education Asia. Pankaj Jalote in his book *"CMM in Practice"* has also explained in detail the processes followed at Infosys and is must for all those who want to gain in-depth knowledge of the subject.

10.9.4 Software Process Improvement and Capability Determination (SPICE)

The SPICE is inspired by various process models around the world. Some of them worth mentioning are CMM, ESPIRIT's Bootstrap (Koch93), Bell Canada's Trillium (Trillium) etc. The SPICE methodology gives lot of importance to the way process assessments are done. It has now become an international standard (ISO/IEC 15504). The SPICE has mainly two dimensions viz process dimension and capability dimension. The process dimension consists of 5 categories of processes. They are:

- **Customer Supplier Processes.** Deal with acquiring software, managing customer needs, supplying software etc.
- **Engineering Processes.** Deal with specification, implementation and maintenance of software product.
- **Support Processes.** Deal with processes supporting processes related to any phase of life cycle e.g., software quality assurance, configuration management, audits etc.
- **Management Processes.** Focus on risk management, sub contract management, quality management.
- **Organization Processes.** Focus on development of company resources and assist the organization in meeting its business goals.

The Capability level consists of 6 levels and few sub levels. The six levels are:

Level		
5	—	Optimizing Process
4	—	Predictable Process
3	—	Established Process
2	—	Managed Process
1	—	Performed Process
0	—	Not Performed.

With each process, measurable process attributes are associated which are further rated as F (Fully), L (Largely), P (Partial) or N (Not achieved). To obtain level n requires that all Process Attributes (PAs) below level n to be rated F and those at level n to be either F or L.

10.9.5 SIX SIGMA

Six Sigma (6σ) originated at Motorola in the early 1980's. The focus was to achieve tenfold reduction in product-failure levels in five years. It is a business-driven, multi-faceted approach to process improvement, high customer satisfaction, reduced costs, and increased profitability by reducing defects in the product/process. Ideally it aims to achieve virtually defect-free processes and products (3.4 or fewer defective parts per million (ppm)). The Six Sigma methodology, consists of following steps:

- Define
- Measure
- Analyze
- Improve
- Control

The Six Sigma method, along with its toolkit, can be easily integrated with existing models of software process implementation. Several software companies are using Six Sigma along with CMM successfully for improving the processes used for developing the software. A detailed discussion on Six Sigma is beyond the scope of this book.

SUMMARY

- Developing high quality software is an issue of great importance.
- Lack of quality results into inconsistent product, poor performance, unreliability, maintenance problems etc.
- A number of quality models are proposed to relate user's view of quality to developer's view e.g., McCall's quality model, Boehm's software quality model.
- Software quality metrics are used to measure and control software quality.
- Three types of metrics are—product metrics, process metrics and project metrics.
- Software reliability is one of the important quality feature and requires proper knowledge and skills.

- Software Quality Assurance is a framework which ensures that quality standards and practices have been used to develop a good quality software which conforms to the end user requirements.
- Several process maturity frameworks and quality standards are proposed to improve software quality e.g., CMM, SPICE, Bootstrap, ISO 9001 etc.

REVIEW PROBLEMS

1. Define (a) software metrics (b) measurement (c) software quality assurance.
2. List three perspectives of McCall's quality model and important quality criteria of each of these perspective.
3. What are the advantages of software metrics? List at least five advantages.
4. What is the difference between direct and indirect measurement?
5. Calculate McCabe's Cyclomatic complexity for the flow graph shown in Fig. 10.7.

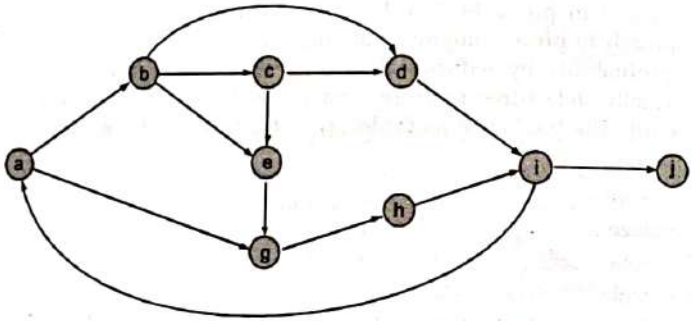


Fig. 10.7

6. Study the program given below

```
/*This program calculates volume of a cylinder*/
#include <stdio.h>
#define PI 3.142
float cylinder(float rad, float ht);
main()
{
    float volume, radius, height;
    radius = 5.0;
    height = 10.0;
    volume = cylinder(radius,height);
        printf("Volume of cylinder is %f\n", volume);
}
float cyclinder(float rad, float ht);
{
        float result;
        result = (22/7)* rad * rad * ht;
        return result;
}
```

For this program compute

(a) Halstead's metrics

(b) Lines of Code.

7. For the structure chart given in Fig. 10.8, calculate the information flow index of individual module as well as whole software.

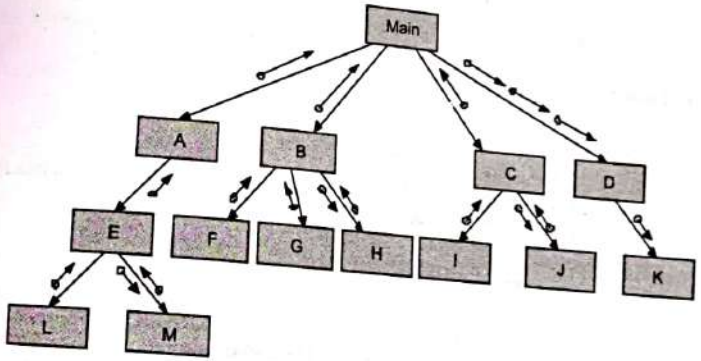


Fig. 10.8

8. Explain briefly the Object Oriented metrics proposed by Chidamber.
9. List important Constituents of Software Quality Assurance plan.
10. Study the C++ program given below and compute Chidamber's 0-0 metrics for the same.

```
#include <iostream.h>
class employee
{
    private:
        char  name[40];
        long  emp_id;
        int   age;
    public:
        void get_data()
```

```
        {
                cout << "Enter Employee Name...";
                cin >> name
                cout << "Enter Employee Id...";
                cin >> emp_id;
                cout << "Enter Employee age...";
                cout >> age;
        }
        void put_data()
        {
                cout << Employee Name is..." << name << endl;
                cout << Employee Id is..." << emp_id << endl;
                cout << Employee age is..." << age << endl;
        }
};
class faculty:public employee
{
    private:
            char    departmet[30];
            char    designation[30];
            int     num_of_pub;
    public:
            void get_data()
            {
                    employee::get_data();
                    cout << "Enter Faculty Department...";
                    cin >> department;
                    cout << "Enter Designation...";
                    cin >> designation;
                    cout << "Enter Number of Publications...";
                    cin >> num_of_pub;
            }
            void put_data()
            {
                    employee::put_data();
                    cout << "Department is..." << department << endl;
                    cout << "Designation is..." << designation << endl;
                    cout << "No. of publications are..." << num_of_pub << endl;
            }
};
class staff:public employee
{
```



```
    private:
            int     division[30];
            int     designation[30];
    public:
            void get_data()
            {
                    employee :: get_data();
                    cout << "Enter Administration Division...";
                    cin >> division;
                    cout << "Enter Designation...";
                    cin >> designation;
            }
            void put_data()
            {
                    employee :: put_data();
                    cout <<"Administrative Division is..." << division << endl;
                    cout <<"Designation is..." << designation << endl;
            }
};
class consultant : public faculty
{
    private:
            char    socialization[40];
            int     num_of_projects;
    public:
    void get_data()
    {
            faculty :: get_data()
            cout << "Enter Area of specialization...";
            cin >> specialization;
            cout << "Enter Number of projects handled...";
            cin >> num_of_projects;
    }
    void put_data()
    {
            faculty :: get_data();
            cout << "Area of specialization..." << specialization << endl;
            cout << "Projects handled..." << num_of_projects << endl;
    }
};
```

```
void main()
{
        class  faculty f1;
        class  staff s1;
        class  consultant c1;
        count << "Enter data for staff member...\n";
        s1.get_data();
        cout << "Enter data for a faculty member...\n";
        f1.get_data();
        cout << "Enter data for consultant...\n";
        c1.get_data();
        cout <<"Data for staff member...\n";
        s1.put_data();
        cout << "Data for a faculty member...\n";
        f1.put_data();
        cout << "Data for consultant...\n";
        c1.put_data();
        getch();
}
```

11. Define software reliability. How software reliability is different from hardware reliability?
12. Define: (a) software availability (b) error (c) fault (d) failure (e) execution time (f) software reliability engineering.
13. For a logarithmic Poisson execution time model initial failure intensity is 12 failures CPU hr, failure intensity decay parameter is 0.03/failure and failures experienced are 60. Compute the current failure intensity.
14. What are five levels of maturity of CMM? Explain briefly?
15. List five differences between organization following immature and mature processes respectively.
16. Explain briefly the various KPA's at maturity level of CMM.
17. List 20 clauses of ISO 9001.
18. Do a comparative study of CMM and Six Sigma.
19. What are the 5 levels of P-CMM? List important features of each of these levels after studying the literature.
20. Write a short note on SPICE.

○○○

Chapter 11

SOFTWARE MAINTENANCE

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- Different types of software maintenance
- Different maintenance models
- Reverse engineering
- Software reengineering
- Software reuse
- Software restructuring

11.1 INTRODUCTION

Once the software is delivered and installed at the customer's premises, the process of maintenance begins. It has been found that software maintenance takes up most of the software life cycle costs and hence total cost of software maintenance is quite huge.

According to (Bennett91), software maintenance can be defined as set of activities undertaken on a software system following its release for operational use.

According to standard for software maintenance-1992, software maintenance is modification of a software product after delivery to correct faults, to improve performance of other attributes or to adopt the product to a modified environment.

It is clear from the software life cycle phases that operation and maintenance are at the end of the life cycle. But for large scale software systems, this phase runs longer than the previous phases. According to (Arthur88) at the most one-third of total costs are for software development and the rest is spend for its maintenance.

There are many reasons for maintaining the software. Some of them are:

- Over a period of time, software's original requirements may change to reflect the customer's needs.
- Errors undetected during software development may be found during use and require correction.
- With time new technologies are introduced such as new hardware, operating system etc. The software therefore must be modified to adopt the new operating environment.

The changes in the software may be therefore:

- Simple changes to repair coding problems.
- More extensive changes to repair design errors.
- Changes to accommodate the new requirements.
- Changes to the architecture of the software.

In today's economy and business environment software is an integrated and critical part of the system. Companies all over the world have made huge investment in these systems. Therefore managing changes to these systems is an important and challenging task.

Maintenance activities can be performed by the developer itself or by a separate agency. Each option has advantages as well as disadvantages (Parikh86). The first option i.e., maintenance done by the developer is more suitable for small systems and there is no need of elaborate documentation. However if the expert leaves, the organization may be in trouble. Similarly advantages of maintenance being performed by the separate organization are:

(a) Better documentation of the software.
(b) Establishment of proper procedures for incorporating changes.
(c) Identification of weak and strong points of system by the maintenance programmers.

However this approach has disadvantages also like funding problems, need for proper training, assigning job to incompetent people etc. According to (McClure83) large organizations must establish a separate maintenance department to ensure better maintenance of the product. It is seen that a well organized maintenance organization following well defined maintenance procedures and change management is more effective. If the maintenance processes followed by organization are uncontrolled it will increase the maintenance cost even if the product is a good quality product.

11.2 SOFTWARE EVOLUTION

In a way changes made to the software can also be termed as software evolution. (Lehman80) has also proposed certain laws based on evolution of number of large software systems. They are:

(a) **Law of Continuing Change** – This law states that any software system that represents some real world reality undergoes continuous change or becomes progressively less useful in that environment.
(b) **Law of Increasing Complexity** – This law states that as software evolves, its complexity also increases. Hence measures must be taken to simplify the structure.
(c) **Law of Program Evolution** – This law states that software evolution is, self regulating with statistically determinable trends and metrics.
(d) **Law of Conservation of Organization Stability** – This law states that during the active lifetime of a software system, regardless of the resources used the work output is almost constant.
(e) **Law of Conservation of Familiarity** – This law states that during the active life time of the program, changes made in successive releases is almost constant. This is because each change made in the system results into new faults in the system.

It has been suggested by the researchers that while planning the organization's maintenance process, these laws must be kept in mind. A number of strategies have been proposed for software change and software maintenance is one of them. Other strategy of interest is software reengineering. The major difference between the two is that in case of software maintenance the overall structure of the software does not change whereas in the case of software re-engineering the structure is modified and improved so that the software becomes easier to understand, maintain and evolve. The software reengineering is discussed in section 11.9. In still another software change strategy the focus is to change the complete architecture of the software. As a result of changes made to software, different versions of software and its components are generated. In order to keep track and manage different versions, software configuration management is used which is discussed in chapter 9.

11.3 TYPES OF SOFTWARE MAINTENANCE

Software maintenance can be categorized into following four types:

(a) Corrective maintenance
(b) Adaptive maintenance
(c) Perfective maintenance
(d) Preventative maintenance

11.3.1 Corrective Maintenance

This type of maintenance is also called bug fixing and deals with fixing the reporting errors while the software is in use.

Reporting errors can be

(*a*) coding errors

(*b*) design errors and

(*c*) requirements errors i.e., incomplete specifications.

Out of these, requirements related errors are most expensive to correct because of the extensive redesign involved. The corrective maintenance sometimes also includes temporary patches and workarounds which can cause great problems for subsequent maintenance work.

11.3.2 Adaptive Maintenance

This type of maintenance concerns external changes. Even if the software is error free, it is possible that the environment in which the software system works will often change. The changes can be introduction of new versions of operating systems, new hardware or removing support for existing facilities. As a result the software must be ported to this new changed environment and made operational.

The modifications made to the software as a result of changes to external environment is called adaptive maintenance.

11.3.3 Perfective Maintenance

This type of maintenance concerns improving the delivered software as a result of change in user requirements or efficiency improvements. For example, a salary computation software may be altered to reflect new income tax laws, or change in salary structures. According to a study made in the industry it is found that around 10% of changes will be made every year to the software as a result of change in user requirements.

11.3.4 Preventative Maintenance

This type of maintenance is done to anticipate future problems and to improve the maintainability using techniques like documenting, commenting or even re-implementing some part of software using modern software engineering tools and techniques.

11.4 COST OF SOFTWARE MAINTENANCE

Cost of software maintenance is very high. More than 65% of software life cycle costs are expended in the maintenance activities. At a workshop conducted at university of Durham, UK in 1987 it was reported that in UK alone, £1 billion is spend annually on

software maintenance. The mean distribution of effort spent on maintenance activit was as follows:

- Perfective
  - user enhancements – 50 %
  - efficiency – 43 %
  - others – 3 %
- Adaptive – 4 %
- Corrective – 25 %
- Preventative – 25 %
- 4 %

It is clear that preventative maintenance is not done significantly in the computer industry. The main factors/reasons which contribute to high cost of maintenance are discussed below:

(*a*) In today's scenario, salaries of the programmers, analyst, testers etc., consume major part of an organization budget. As maintenance is a labor intensive activity, developers spend most of the time on maintenance. This in turn partially accounts for high cost of maintenance.

(*b*) Most of the software are 10 to 20 years old (called legacy systems) and lack structure. These systems were designed without using efficient techniques and processes which resulted into poor quality of documentation, poor coding and design. Maintaining these legacy systems again requires lot of rework. It has been found by the researchers and industry people that cost of enhancing an existing system is much higher than developing a new system if the base system is not designed properly.

(*c*) Often required items are not included in the initial stages of software development due to budget or schedule constraints and are deferred until the maintenance phase, thereby increasing the maintenance costs.

Organizations must record properly the actual number of bugs fixed, enhancements made and distribution of effort.

11.5 SOFTWARE MAINTENANCE MODELS

A number of software maintenance models are proposed since 1970's. All of these models focus on three main activities: understanding the software structure, modifying and changing the software and revalidating the software. Since 1980's a class of process oriented maintenance models were also proposed. These maintenance models viewed the maintenance process in terms of activities performed and the order of these activities. A process for managing and executing maintenance activities is also proposed by IEEE standard for software maintenance. Every organization must use a maintenance model in order to improve the maintenance process. Next we will discuss some popular software maintenance models.

11.5.1 Quick-fix Model

This is one of the simplest and ad hoc model used for maintaining software. In this model changes are made at the code level as early as possible without anticipating future maintenance problems. As a result the structure of software degrades rapidly. This model is not suitable for large software systems but can be used if

(a) Software is small, developed and maintained by single person

(b) Customers are not ready to wait and want fixing of bugs to be done immediately. In such cases temporary fixing of bugs can be done to make the software operational. Parallely proper correction and enhancement work is carried out for upgrading the software in future.

11.5.2 Iterative Enhancement Model

The iterative enhancement model incorporates changes in the software based on the analysis of the existing system and assumes that the complete documentation of the software is available in the beginning. The model attempts to control complexity and tries to maintain good design as a result of changes made. The document of each software life cycle phase i.e. SRS, design document, testing documents etc. are also modified so as to successfully support the next iteration.

11.5.3 Full-reuse Model

The full-reuse model of software maintenance is based on reuse of existing software components. This model starts with defining requirements for the new system and reusing parts of old system as much as possible. To be successful, this model therefore needs a mature reuse culture.

11.5.4 Yau and Collofello's Model

(Yau80) proposed a more specific maintenance model consisting of several phases and belonged to category of process oriented maintenance models. The main steps/phases of the model are:

- Identification of maintenance objective
- Understanding the program structure
- Generating the maintenance change
- Account for the ripple effect as a result of maintenance change
- Conducting the regression testing.

In this model chances of having long term problems is reduced substantially.

11.5.5 Taute's Maintenance Model

Taute Maintenance model is also a type of process oriented maintenance model and is quite easy to understand. This model was presented by Barbasa Taute (Taute83) and discussed by (Parikh86). The maintenance activity in this model is looked upon as a

complete independent cycle consisting of eight phases. The cycle starts with a change request and towards the end of the cycle modified product is delivered to the user. The model consists of eight phases as shown in Fig 11.1.

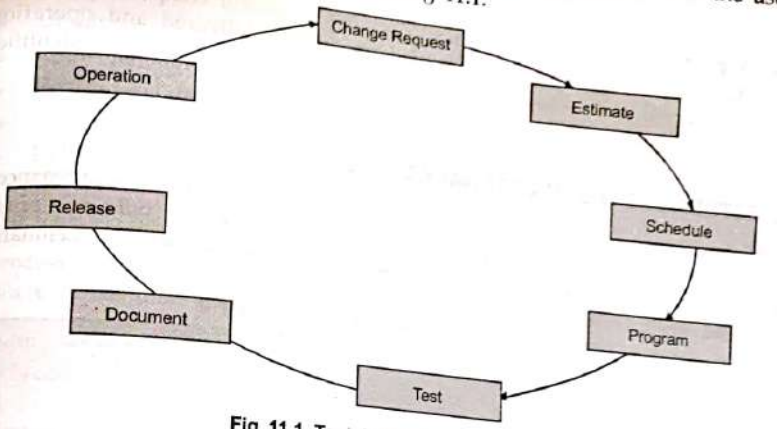


Fig. 11.1 Taute's Maintenance Model

Brief description of various phases of model is as follows:

- **Request Phase.** A user of the product or programmer initiates the maintenance process by requesting a change. Request is identified as corrective, adaptive or perfective. At this stage configuration management activities are used for storage and tracking the request.
- **Estimate Phase.** This phase focuses on estimation of time and resources required to implement change request. Analysis is also done to find out its impact on the existing system.
- **Schedule Phase.** This is one of the important phases and concerns with identification of change requests for the next schedule release and developing the planning documents.
- **Programming Phase.** This phase of the Taute's maintenance model focuses on creating test version of the production source code, designing changes and coding. Necessary documents are also updated at this stage.
- **Testing Phase.** This phase focuses on the proper functioning of new production system and ensures that new changes as requested by user are incorporated properly. A number of testing techniques like inspections, code walkthroughs, system testing, acceptance testing as discussed in chapter 8 are used here.
- **Documentation Phase.** All the relevant documentation is updated in this phase before the system release. This is necessary for trouble-free maintenance of product in future.

- **Release Phase.** The new changed system along with updated documents are delivered to the users during this phase for operation, who also conduct the acceptance testing.
- **Operation Phase.** In this phase after the successful completion of acceptance testing by the user, new changed product is delivered and operations are conducted. As system is used, new requirements and bugs are identified and new change requests are submitted.

11.6 ESTIMATING THE MAINTENANCE COST

Several models are proposed in the literature to estimate the cost of maintenance which normally varies from project to project. Also these models give only an approximate idea about maintenance cost. The two popular models are Belady and Lehman model (Bel76) and Boehm's model.

The basic equation proposed by Belady and Lehman model is

$$M = P + Ke^{(c-d)}$$

where

M = Total Effort
P = Effort done in analysis, design, coding, testing and evaluation
K = Constant
c = complexity measure
d = degree of familiarity of maintenance team with software

value of c will be high if software is developed without following well defined process. Similarly if maintenance team does not have proper knowledge of the software, value of d will be low.

(Boehm81) as a part of his COCOMO cost estimation model proposed a formula for estimating the maintenance cost in terms of Annual Change Traffic(ACT)

ACT is defined as the fraction of a software product's source instructions that undergo change either through adding the new lines or deleting the existing lines.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

Maintenance is then calculated by multiplying the ACT with the total development cost.

11.7 SOFTWARE MAINTENANCE STANDARDS

Two standards which are used for maintenance of software in the organizations are:

1. Standard for software maintenance (1993) developed by P1219 Software Maintenance Standard Working Group under the IEEE computer Society.
2. Standard for Software Quality Metrics Methodology (1992) developed by Software Engineering Standards subcommittee of IEEE Computer Society.

The Standard for software maintenance has also proposed a maintenance process consisting of following stages:

- Problem Identification
- Analysis
- Design
- Implementation
- Regression/System Testing
- Acceptance Testing
- Delivery

11.8 REVERSE ENGINEERING

The process of reverse engineering is used to extract information about the software and is a very common analysis process used by the organizations.

According to (Chikofsky90) software reverse engineering is defined as the process of analyzing a system to identify the system components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction.

Input to the reverse engineering process is source code. In case even the source code is not available, input is executable code and output is the high level representation of system as shown in Fig 11.2.

The motivation for using reverse engineering is due to problem of re-implementing an existing software in different languages to meet the needs of time. (Converse88) has proposed following three approaches to solve this problem:

(*a*) Rewrite the existing system manually.
(*b*) Use automatic language translator.
(*c*) Redesign and re-implement the system.

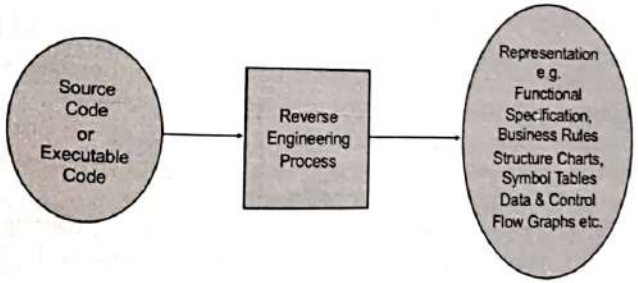


Fig. 11.2 Block Diagram of Reverse Engineering Process

The first approach allows changes in program structure but is very time consuming. New system is influenced by the style of existing code and new system must be properly tested to ensure that it is functionally equivalent to the original program. Use of Automatic language translator though generates the code quickly but is not well structured, is of poor quality, difficult to understand, incomplete and hence results into increased maintenance costs. The third approach though has high building cost, redesigns and builds the completely new system starting from system requirements. It therefore results into good quality design and lower maintenance cost. But the problem comes if there is no valid SRS of the system. It is here the reverse engineering techniques play an important role to capture the functionality of the system and generate the reconstructed design in order to implement the system using new language. If the reconstructed design is further restructured or used for incorporating new requirements, the process is called software re-engineering which would be discussed in detail in section 11.9.

Reverse engineering can be used by the organizations to improve their existing products. Reverse engineering is also an efficient approach to understand and maintain large software systems called legacy systems. The organizations heavily depend upon these systems for successful working of the organization. But with time these systems must be changed to meet rapidly changing business needs.

11.8.1 Objectives of Reverse Engineering

(Chikofsky90) has identified following seven objectives which reverse engineering helps organizations/developers to fulfill. They are:

(*a*) **Coping with complexity** – Reverse engineering techniques supported by automated tools help the companies to understand the complexity of the systems and help in extracting relevant information.

(*b*) **Generating alternate views** – Reverse engineering helps in generating the graphical representation of the system from different perspectives i.e. structural representation, functional views or dynamic views in terms of ER diagram, DFD, state transition diagrams, structure chart, class diagram etc.

(*c*) **Recovering lost information** – Reverse engineering techniques help the analyst in generating useful lost information about the old systems.

(*d*) **Detecting side effects and analyzing quality** – Over a period of time modifications made to the program can lead to lot of unexpected problems. The reverse engineering techniques can be used to detect these anomalies.

(*e*) **Facilitating reuse** – The reverse engineering techniques can also be used for identifying reusable software components from the existing systems. These components can then be used for building new systems.

(*f*) **Populating a repository or knowledge base** – Reverse engineering can be used to identify reusable components and storing these artifacts in the repository for analysis and future reuse.

(*g*) **Synthesizing higher abstractions** – Reverse engineering techniques also help the analyst to build higher level abstractions for the information collected about the code using CASE tools. For example from the information collected about individual module, information can be generated at system level about interaction between modules.

11.8.2 Reverse Engineering Process

The traditional software life cycle focuses on mainly three activities. They are: requirement analysis, design and coding. This process can be looked upon as moving from high level representations of the system to the actual physical implementation and is often termed as forward engineering. The reverse engineering is just the opposite i.e., starting from code, generating representations that are implementation independent as shown in Fig 11.3.

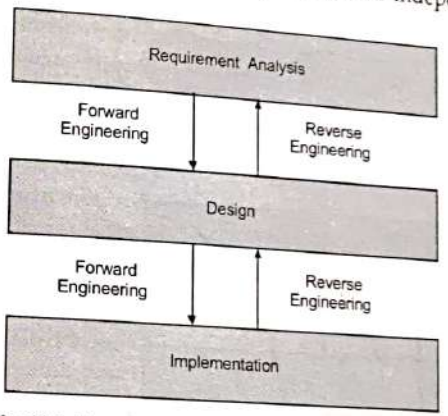


Fig. 11.3 Forward Engineering Vs Reverse Engineering

According to (Byrne91), reverse engineering process consists of following steps:

1. **Collecting and examining the information** – This step focuses on collecting all possible information (i.e., source code, design documents, etc.) about the software and to study so as to get familiar with the system.
2. **Extracting the structure** – This step concerns with identification of program structure in the form of structure chart where each node corresponds to some routine.
3. **Recording the functionality** – During this step processing details of each module of the structure charts are recorded using structured language, PDL, decision tables etc.

4. **Recording data flow** – From the information extracted at step 2 and step 3, set of data flow diagrams are derived to show flow of data among the processes.
5. **Recording control flow** – High level control structure of the software is recorded.
6. **Review extracted design** – Design document extracted is reviewed several times to ensure consistency and correctness. It is also ensured that the design represents the program.
7. **Generate documentation** – Finally in this step the complete documentation including SRS, design document, history, overview etc., is recorded for future use.

The block diagram of the reverse engineering process is shown in Fig. 11.4.

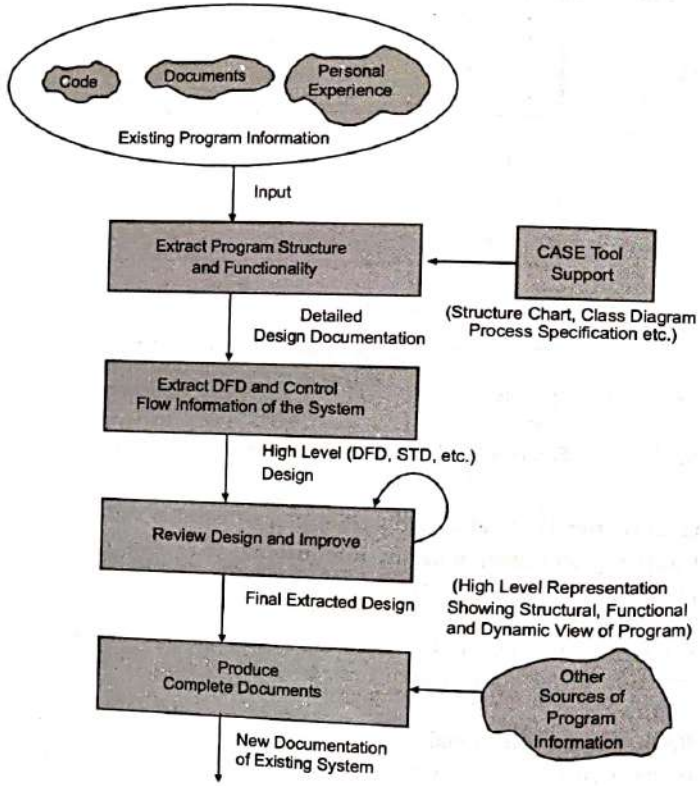


Fig. 11.4 Reverse Engineering Process

1.3 Reverse Engineering Tools

reverse engineering if done manually would consume lot of time and human labor hence must be supported by automated tools. As a result numerous reverse neering tools have been developed to assist the reverse engineering activities. Some of the tools supporting reverse engineering activities are listed below:

- **CIAO and CIA** – A graphical navigator for software and web repositories along with a collection of reverse engineering tools. Also includes an extractor (Parsar) for C.
- **PBS** – Software Bookshelf Tools for extracting and visualizing the architecture of programs.
- **GEN++** – An application generator to support development of analysis tools for the C++ language.
- **Rigi** – A visual software understanding tool from University of Victoria, Canada.
- **Bunch** – A software clustering/modularization tool from Drexel University, Philadelphia.

1.9 SOFTWARE REENGINEERING

the initial years i.e., seventies, software were written using non-conventional and ad r techniques. There was no systematic approach followed to develop these software. fact many organizations all over the world are still using these systems for their day day working. So it is not simply possible to reject these software and replace with w software which again calls for lot of financial investment on the company's part. ese old systems are called legacy systems. These systems also contain lot of crucial rmation about the organization i.e., business rules, business policies etc., which are so not documented separately. Hence these systems are valuable assets of the ganizations, but at the same time difficult and costly to maintain.

The main problems of maintaining these old systems are (Bennett91):

- Most of the existing software is unstructured, written using non-structured programming methods. Hence it is difficult to understand.
- Most of these software is written in assembly language, machine code or languages like COBOL which are out-dated.
- There is no documentation of these systems, even if available either it is incomplete or outdated.
- Involvement of maintenance programmers in developing the product is very less and hence they cannot map the working of program to its source code efficiently.
- The effect of changes if to source code is very difficult to predict and usually results in incre ntenance costs.
- Multiple concurren source code are difficult to handle.

The answer to majority of these problems is reengineering.

Idea behind reengineering the software is to understand the existing system and replace the system partially or fully with latest technology. The resulting software then becomes easier to maintain.

Synonyms for reengineering are software renewal and software renovation. At a higher level of abstraction reengineering can be looked upon as consisting of two main stages reverse engineering and forward engineering as shown in Fig. 11.5.

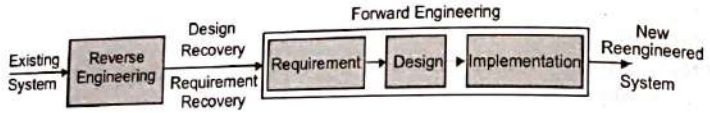


Fig. 11.5 Reengineering Model

Reverse engineering as discussed in detail in section 11.7 is used to understand the design and requirements of the system and forward engineering is done to modify the system partially or fully. IEEE however looks upon re-engineering as an umbrella technology consisting of program restructuring, reverse engineering and forward engineering.

The decision to reengineer an existing system is very difficult to make as it involves lot of financial implications. National Bureau of Standards (NBS1983) have proposed certain guidelines to help in deciding, when to reengineer the existing system. They are:

- Frequency of system failures is high.
- Code is more than 7 years old.
- Structure of program and logic is too complex to understand.
- Programs are written to support previous generation of hardware (hardware changes very fast).
- Programs are running in emulation mode.
- Modules or program subroutines have grown excessively large.
- Company finds difficulty in retaining the maintainers.
- Documentation has become outdated.
- Design specification incomplete, obsolete or missing.
- Hard-code parameters are subject to change.

11.9.1 Software Reengineering Process

The software reengineering process can be illustrated with the help of Fig. 11.6. The reengineering process as shown in Fig. 11.6 is self explanatory. It is to be noted that the new system produced at the end is again open for reengineering.

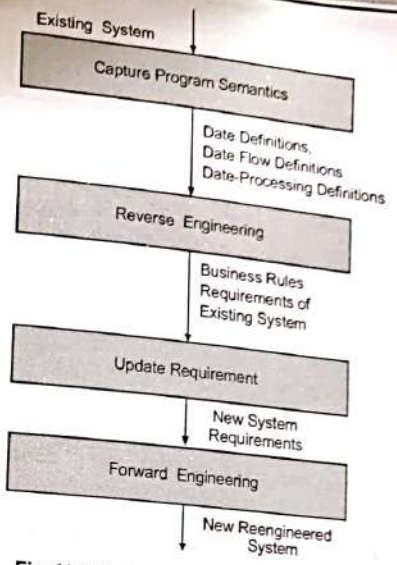


Fig. 11.6 Software Reengineering Process

11.10 SOFTWARE RESTRUCTURING

An important solution to control the software maintenance costs is to modify the software, also called software restructuring.

In simple language software restructuring can be defined as making changes to the software so that it becomes easier to understand and change.

The term software here is used for the source code as well as the related documents. According to (Arnold 89), the benefits of software restructuring are:

- Reduced complexity.
- Enhanced understandability.
- Easy testing.
- Increased programmer productivity.
- Reduced dependence on individuals who understand the software.
- Less time needed by maintenance programmer to gain familiarity with a system.
- Easy debugging.
- Increased software life time.
- Easy to enhance the functionality of software.

Existing C Code

```
if (rate_type = = 1)
    Calculate_Lowtax (Salary);
else if (rate_type = = 2)
    Calcmidtax(Salary);
else if (rate_type = = 3)
    Calc_hightax(Salary);
else
    Calc_sen_citizen(Salary);
```

Restructured C Code

```
Switch(rate_type)
  {
    Case 1 : Calaculate_Low_tax(Salary); break;
    Case 2 : Cal_mid_tax(Salary); break;
    Case 3 : Calc_high_tax(Salary); break;
    default : Calc_sen_citizen(Salary); break;
  }
```

Fig. 11.7 An Example of Restructured Code

(Bennett91) also proposed that restructuring techniques can be applied at five levels. They are code, documentation, programming environment tools, software engineers and management/policies. At the code level techniques like modifying code without altering control and data structure, using reusable code, using algorithms and procedures for restructuring program control flow graphs, and restructuring data can be used. At documentation level documentation can be upgraded, reverse engineering techniques can be applied or system can be made more modular. Similarly Integrated tools can be used which indirectly improve the ability of developers to deal with software. Experienced software engineers and policy decisions of the company also effect how software is perceived. An example illustrating restructuring a part of C code is shown in Fig 11.7.

11.11 SOFTWARE REUSE

Software reuse is another area of research to control the software costs during development, maintenance and reengineering as well. Additionally reuse of product and process components from existing systems also results into increased productivity and quality.

According to (Braun92) reuse is defined as the use of an existing software component in a new context, either elsewhere in the same system or in another system and reusability is the extent to which a software component is able to be used.

Braun also stressed that conformance to design and programming standards further increases the component's reusability.

According to (McClure97) software reuse is the process of building or assembling software systems from predefined software components that are designed for reuse.

Object Oriented technology through the concept of reuse helps in building high quality software within budget and schedule. In context with software development, various reusable software components can be

- Software specifications
- Designs
- Source code
- Test plans/test cases
- Project plans
- Documentation
- Object frameworks etc.

Out of these design components and test components are the ones which are reused most.

11.11.1 Reuse Advantages

The major advantages of software reuse are:

- Improved software productivity.
- Reduced software costs.
- Improved quality, reliability and performance.
- Reduced software development time.
- Software development possible with small team.
- Improved system interoperability.
- Easy movement of person, tools and methods from project to project.
- Better standardization of products.
- Uniform system architectures and consistent algorithms.
- Better support for rapid prototyping.

Though software reuse offers number of advantages, it has some challenges also. Main problem is that existing systems were designed without keeping reuse in mind. Hence despite the existence of large amount of software, there is shortage of reusable components. Therefore one of the biggest issues is the identification of reusable components from these existing systems and store them in a repository for future reuse.

11.11.2 Identifying and Storing Components for Reuse

Key to deliver cost-effective solutions is a well organized reusable components library.

Hence to develop such kind of library, organization must extract components from the existing systems and store them in the library using some proper classification scheme. (Caldiera91) has proposed a two phase scheme for identification and storage of components from existing systems. The two phases are:

(*i*) **Component identification.** In this step the existing programs are analyzed and possible candidates for reuse are identified. This step can be automated.

(*ii*) **Component qualification.** In this step components extracted in step 1 are studied by a domain expert in detail to check for their reusability. This is done by generating their functional specifications and associating the test cases (components not satisfying test cases are rejected), applying a proper classification scheme for better retrieval, adding information in the manual for future reuse and storing the component in the library for reuse. Further from the feedback received from experts, the component's are modified.

A number of classification schemes are also proposed for storing and retrieving the components efficiently. One such scheme is a hierarchical scheme in which components are divided into categories and sub-sub categories. Another popular scheme is faceted classification scheme proposed by (Prieto87). It consists of defining number set facets like applications area, functionality, language used, operating system, inputs required and outputs produced etc., for each component. User then specifying these values can extract the component from library for reuse. (Tracz90) has proposed a 3C Model (Concept, Content and Context) for describing the software component. Concept describes the purpose of component, content describes means to realize concept and context places the component in its applicable domain. A number of automated systems are also developed to assist the user in searching the components called Component Retrieval Systems. These systems also offer features like metrics about software, data about frequency of reusing a particular component, configuration management of the reusable components etc. Now a days public libraries of software components are also available at nominal price e.g., collections of object packages implanted in object oriented languages such as Java, C++ etc.

11.11.3 Software Life Cycle with Software Reuse

While developing software, an important issue to address is which stages of software process are effected by reuse. Introducing reuse during software development also requires changes to software life cycle. Most of the software methods specially object oriented methods OMT, UML etc., provide sufficient base to incorporate the reuse activities. At every stage of software life cycle if library of software component is existing in the organization, following steps are followed (McClure97):

1. Identify reuse components that can be used at individual stage of software life cycle.
2. Evaluate and short list these components for final use.
3. Modify components to meet the current systems requirements.

4. Integrate these reusable components into the system and generate deliverable for the current system.
5. Give feedback to improve the component's reusability.

Similarly while development, at every stage

1. Identify components to be developed for reuse.
2. Identify the requirements to be met by the reusable components.
3. Build and package the components for software reuse.
4. Store the reusable components in the reuse library for future use.

11.11.4 Guidelines for Building Reusable Software

A number of guidelines are proposed by the researchers to build reusable components. Some of them are:

1. Component must be easy to use in a new environment. Hence it must have high cohesion and minimum coupling.
2. Using principle of abstraction components must be designed to support similar functionality.
3. Well defined, documented and standard interfaces must be incorporated in the component so as to allow its easy reuse by other system.
4. Use object oriented design approach to build components which integrate the statics and dynamics in one cohesive unit.
5. As far as possible use standard architecture models while designing systems in order to increase software reuse.
6. Establish organization wide set of coding standards.
7. Develop code so as to conform to maintainability standards
8. Components must be supported by good documentation in terms of details like functionality, constraints, installation, so that they are easily understandable by the user.
9. Address all the legal issues properly.

SUMMARY

- The activity of maintenance starts after the software is released for operational use.
- The cost of maintenance is very huge.
- The maintenance is the process of modification of software after its delivery to correct faults and to adopt software to new operating environment.
- Software maintenance is of four types—corrective maintenance, adaptive maintenance, perfective maintenance and preventative maintenance.

- A number of standards and models are proposed by researchers to maintain the software.
- Reverse engineering is an activity which by analyzing the existing code of the software identifies the system components and their relationships.

REVIEW PROBLEMS

1. Define the terms: (a) Software maintenance (b) Reuse (c) Reusability (d) Reverse engineering (e) Reengineering (f) Software restructuring (g) Forward engineering.
2. What is the need for maintaining the software?
3. State Laws proposed by Lehman based on evolution of large systems.
4. What are different types of maintenance? Explain briefly.
5. Explain the Taute's Maintenance Model.
6. What are the objectives of doing reverse engineering?
7. Write a short note on reverse engineering process.
8. Can reengineering of existing systems be performed independent of reverse engineering?
9. What are legacy systems?
10. Identify a legacy system written using FORTRAN or COBOL. Apply reverse engineering principles to obtain specifications and design documents.
11. What is the need of software reengineering?
12. Draw a block diagram to illustrate the reengineering process.
13. List benefits of software restructuring :
14. Define (a) configuration management (b) baseline (c) configuration item (d) version (e) change request (f) configuration control board
15. What are the benefits and limitations of software reuse?
16. List the steps followed by an organization using software reuse in software life cycle.
17. List few guidelines for building reusable components by the researchers.
18. Identify few organizations which have successfully implemented reuse techniques in their software development process.
19. Explain few classification schemes for easy storage and retrieval of software components.

○○○

Chapter 12

COMPUTER ASSISTED SOFTWARE ENGINEERING (CASE)

AFTER STUDYING THIS CHAPTER YOU WILL LEARN ABOUT

- What is a CASE Tool?
- Different Type of CASE Tools
- CASE Construction Techniques
- Some Popular CASE Tools Used during Software Development Life Cycle.

12.1 INTRODUCTION

The term Computer Assisted Software/System Engineering (CASE) refers to the methods dedicated to an engineering discipline for the development of information systems together with automated tools that can be used in this process (Loucopoulos92). Such automated tools are called CASE tools.

Keith Robinson described the term CASE as harboring two key ideas: "computer assistance in software development and/or maintenance" and "an engineering approach to software development and/or maintenance" (Robinson92).

CASE stands for a large number of applications reaching from simple editing tools to environments supporting the whole life cycle. CASE attacks software productivity problems at both ends of the life cycle by automating many analysis and design tasks, as well as program implementation and maintenance tasks (McClure89). The major task

of a CASE tool is to accept different kinds of specifications, analyze the specifications, transform specifications and maintain a large, ever-growing set of interrelated specifications possibly in several versions. Numerous integrated tools and support environments have been launched to support different stages of software development. (Forte92) pointed out that CASE tools have been successful in automating many routine tasks.

CASE environments help in developing the product in an effective and efficient manner by supporting the analyst starting from requirements stage (since its inception) through design, implementation, testing and maintenance phase. They achieve this by integrating tools, techniques and models to support development of large as well as small scale projects. Some other terms used actively for CASE environments are:

- Software Development Environment (SDE)
- Software Engineering Environment (SEE)
- Programming Support Environment (PSE)
- Integrated Program Support Environment (IPSE)

12.2 CLASSIFICATION OF CASE TOOLS

Existing CASE environments can be classified along different dimensions viz. : life Cycle support, integration, construction and knowledge based CASE. Life cycle support dimension classifies CASE tools according to the activities they support in the information systems life cycle. Integration dimension groups together CASE tools which support integration of different tools to support development of large scale software. The construction dimension classifies CASE tools depending upon the techniques used to build them. Finally Knowledge based CASE dimension is used to classify CASE tools that are intelligent.

12.2.1 Life Cycle Support Dimension

Along life cycle support dimension, CASE tools are classified as either *Upper CASE* or *Lower CASE* tools.

Upper CASE tools support strategic, planning and construction of conceptual level product and ignore the design aspect. They, therefore support traditional diagrammatic languages such as ER diagrams, Class diagrams, DFD and structure charts providing mainly draw, store as well as documentation facilities.

Upper CASE tools support a limited degree of verification, validation and integration of system specification. Examples of such tools are AUTOMATE, Excelerator, BLUES DEFT etc.

Lower CASE tools concentrate on the back end activities of the software life cycle and hence support activities like physical design, debugging, construction, testing, integration of software components, maintenance, reengineering and reverse engineering activities.

Now a days, CASE tools exist in the market that provide support for both front and back end activities e.g., Object Analyst, Turbo Analyst (Tel97).

12.2.2 Integration Dimension

CASE tools along the integration dimension support large scale software production. The principal task of these tools is to provide coordinated services to a group of application engineers working together.

Three main integration environments have been proposed (Evans89): CASE framework, Integrated CASE tools and Integrated Project Support Environment.

- ***CASE Framework*** is an open architecture which allows for integrating different tools to develop a tool set covering the whole development life-cycle. A characteristic of this approach is that various tools from different vendors can be integrated. Although no common standard has been established, proposals like ANSI/IRDS, ISO/IRDS and AD/cycle have been suggested(Evans89).
- ***Integrated CASE (ICASE) tools*** support one well-defined approach throughout the whole development process by emphasizing phase integration. They allow the output of one phase to be directed as input to next phase. The integration is supported by a meta-model. Some commercial ICASE tools are IEF, IEW, Foundation and ORACLE/CASE.
- ***Integrated Project Support Environment(IPSE)*** shares several characteristics with the CASE framework and ICASE such as integrating point tools with a uniform user interface, supporting the whole life cycle and providing flexibility and method independence.

12.2.3 Construction Dimension

The construction dimension classifies CASE tools according to the way they are constructed. CASE tools can either be built specifically for a method from scratch or can be built using **CASE shells.** Early CASE tools were built for specific methods from scratch. Examples are AUTOMATE, Excelerator, BLUES DEFT etc. A CASE shell is a customized environment to instantiate the CASE tool for a given method. It is defined as an environment having mechanisms to support any method specified in it (Martin93) and is based on a meta-model. A method engineer is the person who uses the CASE shell to instantiate the CASE tool by modeling the method. Thereafter the application engineer uses the instantiated CASE tool. A number of CASE shells with combined textual, graphical and matrix representation have appeared in the market like VSF (Pocock91), PARADIGM+, RAMATIC (Bergsten89), METAVIEW with its graphical extension GI (Sorenson88), Graphical Designer by ASTI (Advanced Software Technologies), Toolbuilder by Lincoln Software/IPSYS, Paradigm Plus by Protosoft, ObjectMaker by Mark V systems, Maestro II by SoftLab and MetaEdit+ (Smolander91) by MetaCase Consulting.

12.2.4 Knowledge Based CASE Tools

The fourth dimension i.e. Knowledge based CASE deals with those CASE tools that are "intelligent" and use results from Expert systems, Artificial Intelligence(AI) and natural language processing. The purpose of applying AI techniques to CASE aims at providing system developers with a number of intelligent and active tools that facilitate not only specification process but also management of the development product(Arango87). **Intelligent CASE tools** equipped with knowledge and methods of reasoning which mimic human knowledge and reasoning (Anderson89), (Reu91) can be put along this dimension. Some CASE prototypes, tools sets based on expert system approach are SECSI (Bou85), OICSI (Rolland86), (Cauvet88).

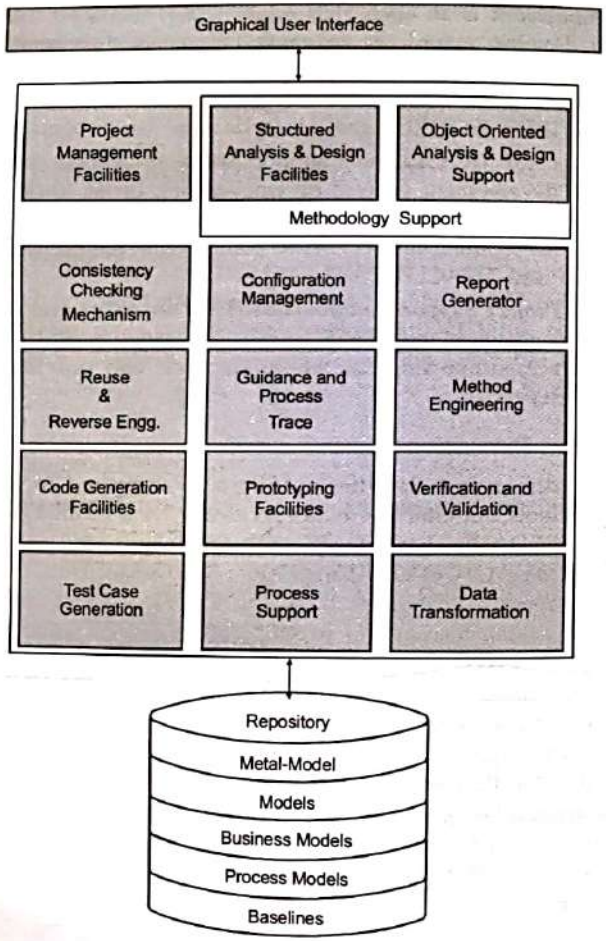


Fig. 12.1 A General CASE Architecture

12.3 CASE ARCHITECTURE

Figure 12.1 shows a general architecture of a CASE environment. Central to this architecture is repository, the most critical and important feature of a CASE environment. The complete information about meta-models, methods, project artifacts, baseline, process models etc., is available in the repository. This information can be shared by different team members efficiently thereby improving the project management as a whole. While implementing, the repository can be either centralized or distributed. Normally this is implemented using a database or a file system. As repository holds different types of data from different phases of software life cycle, it must make the data available to all tools that require them. A Class diagram for example must be available for JAVA/C++ code generation as well as for generating test cases.

Some of the important properties of a CASE environment are discussed below.

(*i*) A CASE environment must have facility to update and retrieve information at any point of development.

(*ii*) It must support project management and planning facilities in order to ensure timely completion of project and to track development in the project.

(*iii*) The CASE environment must also have support for methodology (structured, Object Oriented etc.) used by the organization. As there is no universal method which is applicable to all projects, the environment should support building of new methods from existing method components (method engineering) using concept of meta model. Additionally the environment must be able to instantiate a CASE tool to supply newly engineered method.

(*iv*) The CASE environment must have proper consistency checking mechanisms in order to maintain integrity of the system description.

(*v*) The CASE environment must also have necessary support for tool integration.

(*vi*) The prototyping is an important activity during development in order to understand requirements. As far as possible, CASE environment must have therefore prototyping facility including report generation, animation etc.

(*vii*) A CASE environment must also be able to store data in multiple forms i.e., tables, diagrams, matrix etc. All these forms represent the same semantic information but from different viewpoints. This is because needs of different projects to represent data may be different. Additionally the CASE environment must be able to transform system description from one form to another.

(*viii*) A CASE environment must support integration with other tools in the organization also.

(*ix*) A CASE environment must have some kind of guidance mechanism in order to guide the application engineering during development.

(*x*) A CASE environment should be able to trace product/process components, store them in repository for future reuse.

(xi) In order to provide easy access to user to facilitate his development work, a CASE environment must provide user friendly interface facilities by providing proper graphics and windows environment.

A number of generic systemic architectures have been proposed in the literature. All these architectures are systemic in nature. These architectures identify the static components of a CASE tool and the inter-relationships between them. The systemic architectures which need to be mentioned are the ones proposed by Bubenko (Bubenko92) and Loucopoulos and Karakostas (Loucopoulos95). Bubenko proposed four broad components of the tool architecture viz,. the presentation subsystem, the support subsystem, the repository interface and the repository as shown in Fig 12.2.

Application engineers interact with the *Presentation* subsystem which support graphical, textural and form-based input/output facilities. The *Support* subsystem provides the basic functionality of a CASE tool and consists of knowledge acquisition, verification and validation, design support, development, project management support, and prototyping/ code generation subsystems. Provision exists to add additional subsystems to enhance the *Support* subsystem. The *Repository* interface allows interaction with the *repository* which contains in it the method's object schema, method knowledge, domain knowledge, reusable specifications, application specifications, and executable specifications.

The architecture for CASE tools proposed by Loucopoulos and Karakostas (Loucopoulos95) is also organized into four levels viz., tools level, Interface level, repository level and database level.

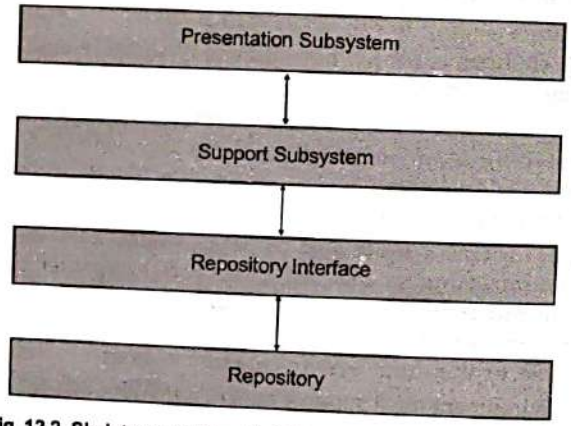


Fig. 12.2 Skeleton of a Generic CASE Architecture (Bubenko92)

Tools level consists of editing, reporting and analysis features. The interfaces level is the principal interface between the application engineer and the CASE tool. Repository is a container of meta-models, domain specifications, and instances. It keeps information about the current status of the development process. In general, it holds data of all

phases in an information system life cycle about individual projects that may be going on concurrently and about their inter-communication needs. Finally, the database level consists of a consistency checking mechanism and an object manager.

Additionally, to support large scale software development and CASE integration, architectures like IPSE and ICASE were proposed. They rest on general frameworks like the Information Resource Dictionary System (IRDS), standard tool interfaces such as the Portable Common Tool Environment (PCTE), data interchange formats such as the CASE Data Interchange Format (CDIF) and IEEE's work on tool interconnectivity.

12.4 FUNCTIONALITY IN CASE

Since CASE tools represent methods in automated form, functionality in CASE can also be looked upon from two aspects: *Product and the Process*. Regarding tool functionality from product point of view, it has been pointed out that CASE tools have been successful in automating many routine software development tasks. Wijers (Wijers91) says that though the possible list of things that CASE tools can do is quite large, they have been essentially successful in providing documentation and verification support. Current tools therefore have often, excellent facilities(Loucopoulos95), (Chikofsky94) for:

- Editing and maintenance of simple graphical specifications
- Verification and validation of the resulting product
- Ability to transform data
- Capability to retrieve information
- Maintaining integrity
- Facilities to update information
- Support for different representation forms
- Capability for analysis in terms of consistency and completeness checks
- Project control and management
- Bridge capability
- User friendly graphical user interfaces etc.
- Data management
- Inter tool communication
- Prototyping

From the process point of view CASE tools don't support features like process tracing and reuse, guidance etc., and represent a stage wise linear model of the process to be followed. A process is concerned with the tasks that have to be performed in order to build a product and expressed in the form of process models. More recently, a number of Process Centered Software Engineering Environments(PSEE) have been developed e.g., EPOS(Reider94), OIKOS(Carlo94), ADELLO-Tempo(Nou94) etc. According to (Ernst95), the PSEEs are CASE tools with great potential for making a large step towards

achieving the ultimate goal of developing high quality software. The key component of PSEE is the process engine which can be loaded with a process model of the activities to be carried out by the staff of the company or project using the PSEE. These activities can be technical or managerial.

12.5 SHORTCOMINGS OF CASE

We consider the shortcomings of CASE from two aspects - functionality and CASE technology. Tool functionality determines the possible actions that can be performed and which must be supported by suitable user interfaces to offer an application engineer-friendly tool. The second facet i.e., *CASE technology is* ***the collection of techniques that must be employed in order to construct a CASE tool***(Prakash96). On the one hand this asks for an articulation of the overall strategy, the systemic and functional architectures, employed in constructing these tools and on the other, there must be a clear identification of the data structures and associated algorithms using which the tool can be constructed.

12.5.1 Functionality

Existing CASE tools lack many of the functional features as projected in generic CASE architectures. These shortcomings are principally in the area of providing process support. According to (Bubenko92) current tools lack many functional features needed to give active support in knowledge acquisition, distributed cooperative work, view integration etc. Current CASE tools also do not support features like reengineering(Bjerknes91), version control(Hahn91) and hypertext features(Cybulski92) properly. According to (Loucopoulos95) important functional features of next generation CASE are process support, process reuse and reverse modeling, providing guidance to the user, validating the requirements model, generating test cases directly from requirements, model evolution etc. (Kelly96) has identified the weakness in CASE tool support as lack of integration mechanisms, insufficient multi-user support, insufficient support for multiple representation paradigms, lack of method modifiability and evolution, limited information retrieval and reporting capability. Managing the modeling process is a feature which is not supported by CASE environments(Tan95).

Additionally, (Huang98) believes that tools are not user requirement-driven and human factors are not considered in today's CASE tools. He has also suggested some possible additions for the next generation of CASE tools. They are process modeling, cross-platform portability, knowledge and learning, active guidance, standardization, access through internet etc. However some of these features like portability, access through internet are not considered as short comings of the CASE tools but are additional features. The common thread in the literature is that of inadequate process support, which is considered next.

(a) Process Model Independence – Methods and existing tools adopt linear and Cartesian paradigm. Under this paradigm, a given task is broken up into subtasks

recursively till atomic tasks are reached. The linear assumption of methods says that the next task is to be performed when the preceding one has been completed. The linear assumption coupled with the Cartesian assumption imposes a discipline on the way an application engineer can construct a product and is too rigid for practical work. In fact, application engineers need to be able to backtrack to an earlier activity to correct errors and omissions as well as to change previously made choices in the light of experience gained in subsequent activities. Further, the strict rule of moving to the next activity only upon completion of the first one is also very restrictive. Thus for example, incremental product development is hampered by it. In other words ***tools are not process model independent.***

(b) Guidance – The research community (Rolland94), (Rolland95), (Dowson94), (Finkelstein94), (Feiler93) has accepted the importance of process guidance: application engineers need to be guided at every step of application development. Guidance has been classified (Dowson94) into *passive guidance* and *active guidance.* Passive guidance suggests possible actions that could be performed next whereas active guidance provides advice on process enforcement to force a process to conform to the process model.

***In (Feiler93) guidance has been considered as the* activity of providing the enacting process agent with assistance regarding the legal steps at any point during the process enactment.**

This definition fits well with the notion of passive guidance. The basic need is to identify legal steps at any given stage in the development process and then to present these so that an appropriate step could be selected by the application engineer.

Methods often use large grained life cycle descriptions and are unsatisfactory because they provide guidance at high level of granularity and don't contain the know-how needed to make the application engineer efficient in practice. Existing CASE tools and CASE shells supporting current ISE methods and SE centered software environments are unable to provide today the kind of heuristics and experience based guidance required in early phases of system development and therefore are passive CASE tools. CASE tools help in capturing, storing and documenting IS products but don't support the application engineers in their development activities (Martin93). No guidance on what would be a good thing to do next is available. It is even difficult to get information on the possible courses of actions available.

Further, each application engineer has his/her own way of working. This way of working is not available as hard data. As a consequence, the process employed to arrive at the product is lost. Need of today is therefore to have CASE tools that can *actively* guide the user to develop the product. A research CASE tool MENTOR(Grosz94) has proposed to represent the process by which application engineers construct information systems specification and use this representation for effective guidance during the development process.

(c) **Process Trace and Reuse** – A number of researchers has (Dowson93) pointed out the importance of tracing the process so that process improvement and reuse can be carried out subsequently. (Ramesh93) considers traceability as an important issue in IS development process for many reasons: product change and reuse support, identification of mission-critical requirements and components, recording of design rationale, project tracking and cost estimates, responsibility and accountability for achieving requirements etc. All of these issues influence the design of traceability models and tracing mechanisms.

The SEI (Humphrey89) has developed a framework known as the *"capability maturity model"* which can be used to assess a development process. Under this model the development process being followed in an organization is rated on a scale of five points. The lowest point characterizes the process as "chaotic", the next higher level as "repeatable" and so on. Assessment studies have shown that the number of processes that can be rated as "chaotic" is very large. Such processes are incoherent and implicit (Dowson93). This leads to lack of predictability and repeatability. The principal reason for incoherent, implicit processes is that the knowledge which forms the basis of a process decision is lost because it is not kept track of in the process itself (Curtis88). In order to move to the next higher process rating, that of "repeatable" processes, therefore, it is necessary to provide means in terms of a tracing model to the information system developer by which this knowledge is traced and made available for subsequent use.

According to (Ramesh98) development of tools to provide traceability across different development environments is considered very important by high end users who emphasize the capture of process related traceability information. Most of the commercial CASE tools of today provide very little tracing support. Although efforts have been made to store process knowledge in chunks(Rolland94) for reuse. NATURE process engineering framework(Jarke94) has also proposed a multi-perspective meta-model by which requirement engineering processes can be defined, guided, traced and improved and has also proposed an architecture to support the framework. In MENTOR(Grosz94) which is a research CASE tool, capitalization of expertise is made possible by a semi-automatic study of process traces. Some tools provide limited support for capturing traceability, but their usefulness is limited by their predefined process models(Ramesh98). Current traceability environments almost neglect the process responsible for captured traces and no systematic procedure is established for guiding and controlling from experiences(Domges98). TOORS(Pih96) and PRO-ART(Pohl97) are examples of research effort that support automated reasoning to enhance the usefulness of traceability information. *Process learning* and *reuse* should be an important feature of the future CASE tools and suitable tracing models and tracing mechanisms/algorithms should exist in order to trace processes for reuse.

12.5.2 CASE Construction Techniques

As mentioned in section 12.2.3, there are two approaches to build a CASE tool. However these techniques are not problem free. Early CASE tools were constructed in an ad hoc

way from scratch based on the experience acquired by their developers. Each new tool meant a new software package that incorporated the whims and fancies of its designer. There was no standardization of techniques nor was it possible to design the tool systematically. This resulted in focus on the construction of application systems but not on the construction procedures/guidelines for building CASE tools. Similar is the case with CASE shells and there is no standardization of techniques/algorithms to build them. There are also problems while building CASE tools using CASE shells which are as following:

- A CASE shell is limited by the facilities of the generic components which it uses. The success of method adaptation in a CASE shell depends on how completely the knowledge of the method and its use can be represented in the meta-modeling language. Due to the poor or semiformal definitions of currently used methods, obtaining sufficient formal representation is rarely a straightforward process. Hence, the adaptation process faces several obstacles. Meta languages also vary from one CASE shell to another which makes their use difficult. Another problem is that often they use rigid and complicated textual languages which can be difficult to learn and apply.
- Most CASE shells are quite complicated to use for method definition because they provide only low level and primitive mechanism for this task. Therefore a limited number of people can actually model methods using CASE shells.
- Often the specification in the CASE shells involves an error prone and laborious compilation of method meta-model, which can include numerous implementation details. Therefore, method modeling and CASE tool generation remains a complex and difficult undertaking (Smolander91). Constructing CASE tools from CASE shells is known to be a difficult task. An expertise of certain level is being required to use the CASE shells.
- Only some of the CASE shells support the development tasks undertaken by the application engineer for example, project MetaPHOR which is based on the notion of meta-activity models of (Brikemper90).
- Further, the construction of CASE shells is itself based on ad hoc techniques reflecting the experience of their designers.

12.6 POPULAR CASE TOOLS SUPPORTING DIFFERENT STAGES OF SOFTWARE LIFE CYCLE

In this section we will discuss some popular commercial CASE tools supporting software life cycle stages:

1. **Software Requirements Tools** – A number of tools are proposed for modeling, tracing and analyzing requirements. CASE diagramming or upper CASE tools represent the system requirements visually using structured or Object Oriented methodologies. Examples are:

Turboanalyst
Oracle's Designer/2000, Argo/UML
Rational ROSE
MS Access
Dia
Excelerator, Statemate

2. **Software Design Tools.** These tools are used to support the system design stage of SDLC and in most of the cases are extensions of CASE tools used for requirements analysis stage. They can be used to support design, verification and optimization.
3. **Software Construction Tools.** Software construction tools are the tools which are used to code and implement the software and hence transform the software requirements into working product. These tools are required even after the product is installed so as to remove defects and maintain it. Broadly they can be classified as

Program editors — They are used for creating and modifying programs. Examples are Visual studio, Emacs, vi etc.

Compilers — These are the software which translate high level languages like C, C++, Java into executable code using linkers, loaders, preprocessors etc.

Interpreters — Interpreters are tools which support line by line execution of programs and hence provide an controlled environment for program execution which is also observable e.g., Perl, QBASIC, Visual Studio, Python, Scripting Languages like HTML etc.

Debuggers — These tools are useful for debugging the programs by inserting break points in the program. Example is Visual Studio.

4. **Software Testing Tools** – These are the automated tools that support various activities of software testing stage of SDLC. They can be classified as:

- **Test Generators.** These are the tools which assist the developers in test case design and their documentation e.g., MS Access, Quick List.
- **Test Execution and Evaluation Tools:** These tools support the process of test case execution and also evaluate the results of execution. The various tools under this category are:
  - Capture/Play back tools to automate execution of test cases e.g. SQA Robot, QA Playback, JavaStar, Ferret, AutoTester, Web etc.
  - Coverage analysis tools to ensure all parts of code i.e., statement, decision, conditions, paths, loops etc. are executed. Examples are Visual Testing, JavaScope, AquaProva etc.

  - Memory testing tools to test memory related problems e.g., BoundsChecker, HeapAgent etc.
  - Simulators replace the actual hardware/software which interact with the software to be tested and also evaluate system. Examples are PerformanceStudio, LoadRunner etc.

5. **Test Management Tools.** These tools support management of different test artifacts. Examples are Visual Source Safe, CVS etc. There are automated tools to support reviews and inspections also e.g., ReviewPro.
6. **Software Maintenance Tools.** Tools under this category are:
   - Comprehension tools to assist in human comprehension and visualization of the programs. Examples are Visual Studio, exref etc.
   - Reengineering tools which allow the change of existing format of a program to a new format i.e., a new language or new database or new technology in general.
7. **Software Quality Tools.** These tools are used to automate techniques like static analysis, reviews, inspections etc. to ensure software quality.
8. **Configuration Management Tools.** These tools support version control and other activities related to configuration management i.e., management and control of changes made to the documents. Examples are ClearCase, Changeman etc.
9. **Project Management Tools.** Tools under this category automate size estimations, cost estimation, schedule estimation, risk management activities etc. Some of the tools are MS-Project, Excel, COCOMO, FPA etc.

SUMMARY

- CASE tools are automated tools used to support development of information systems during software development life cycle.
- CASE tools improve the productivity of the analysts, designers, programmers, testers and all other persons involved in developing the project.
- CASE tools can be classified along different dimensions viz: Life cycle support, Integration, Construction and Knowledge.
- A number of systemic architectures for CASE are proposed in the literature.
- CASE tools can be built either as standalone tools or using CASE shells.
- CASE shells are CASE tool generators based on a meta model which instantiate a CASE tool depending on the method specification which are input to the tool.
- A number of commercial CASE tools are available in the market to support different stages of the software life cycle.

REVIEW PROBLEMS

1. How CASE Environments increase the productivity of the organization?
2. In what way CASE tools differ from CAD tools?
3. What is the role of repository in a CASE environment?
4. Draw a general architecture of a CASE environment. Explain its important characteristics.
5. What is the difference between Upper CASE and Lower CASE tools?
6. What are the different approaches to build CASE tools?
7. What do you think are the shortcomings of existing CASE tools? Explain in detail.
8. List at least two commercial CASE tools supporting different stages of software life cycle.

○○○

Appendix A

A PARTIALLY COMPLETE SAMPLE SRS BASED ON STANDARD IEEE-830 1993

A.1 TEXTUAL PROBLEM DESCRIPTION FOR PATIENT MANAGEMENT SYSTEM

The purpose of Patient Management System(PMS) is to automate the patient admission and patient treatment part of the hospital. For each patient his name, address, age, telephone, sex, medical insurance policy no. (if any), doctor's name, date of admission and discharge has to be recorded. The hospital has several specialized departments to take care of patients needs. Each department has a head of department and several senior/junior doctors attached with it. The system also maintains record of doctors i.e., his name, address, mobile, qualification, specialization, date of joining and department in which he is working. The hospital consists of private rooms (single bed), semiprivate room (2-beds) and general ward consisting of 8 beds. Each of these has fixed rent per day. Each bed is given a unique number. The department has several nurses to attend the patients. At the time of admission, patient as per his requirements is allotted a bed in the hospital and is also required to deposit some advance with the hospital. Once the patient is admitted, the past history of the patient is fed in to the computer along with the doctor's name attending the patient. All the medicines prescribed by the doctor time to time are also recorded. The information is available to all the doctors and head of departments at the terminals installed in their rooms. If doctor recommends the tests, the details along with the test results is also stored in the system. The hospital has its own laboratory where all tests are available. Cost of the test is added to the final bill. Hospital also has a panel of specialists (visiting doctors) which the hospital contacts in case of complicated cases. The hospital also has contacts with other multi specialty hospitals where it can refer the complicated cases. Hospital is funded by several private organizations and individuals. Therefore name of donor, amount donated and date of

donation is also recorded. Every 3rd day, the patient's approximate expenditure on tests, doctor's visit etc., is computed and bill is given to patient to deposit the amount. The mode of payment can be cash or credit/debit card. In case the patient is medically insured all the bills are forwarded to the insurance company with a copy given to the patient. The patients by paying the fees can also consult a doctor after taking prior appointments. Every day in the evening, appointment schedule of doctor is printed and send to them. Also in the evening records of patients are updated concerning medicines prescribed, test prescribed etc.

In the next section partially complete SRS for the patient management system as per *IEEE* format is discussed.

A.2 SOFTWARE REQUIREMENTS SPECIFICATION AS PER IEEE FORMAT

<Patient Management System>
<Author: S. Sabharwal>
<Date 21|03|04>

Table of Contents

1. INTRODUCTION

1.1 Purpose

The purpose of this document is to record the requirements for the design and development of a patient management system of a hospital. This document is used by designers, testers, programmers and others during the development and maintenance of the software. The document will also serve the basis for acceptance testing by the user.

As such it is an evolving document reflecting the current requirements of the project as understood by the project team. Careful review and understanding of the SRS by different stakeholders of the project including the end user will ensure that the requirements outlined are correct and complete and subsequently the software developed will provide the desired level of functionality and consistency.

1.2 Scope

The Patient Management System(PMS) automates the functions specified in section 3.2 of

this document. The purpose of the system is effective management of the patients admitted into the hospital and the staff by automating the different procedures followed by the hospital. Once the system is installed all the details regarding a patient can be retrieved immediately by the doctors/staff of the hospital, all the billing will be computerized, patients will be able to take the appointment of the doctors online etc. The system being developed is a part of Hospital Management System which provides the automation solution for the complete working procedures of a hospital.

The system maintains a large database of Patients, Doctors, procedures and services offered by the hospital for different diseases, complaints, inventory, infrastructure etc., so that useful can be generated and can be reviewed by the hospital management to plan the future policies for the patient's benefit.

1.3 Definitions, Acronyms and Abbreviations

Some important terms used are as follows:

Patient	A Person admitted in the hospital or visiting the doctor
Doctor	A person who treats the patient
Staff	Any employee of the hospital other than nurse or doctor.
Nurse	A person who looks after the patient when he/she is admitted in the hospital
Department	One of several departments of hospital e.g. cardiology department, orthopedics department, cancer department etc.
Ward	A block of rooms where patients are kept
OPD	Out Patient Department
GUI	Graphical User Interface
SRS	Software Requirement Specification
⋮	

1.4 References

None

1.5 Overview

The intended audience for this SRS of Patient Management System are project team members, system analyst, software developers, testers, documentation specialists, designers, system architects, maintenance people and the users/customer of the system who will use this document for developing the system. This document will also be used for acceptance testing by the end user.

Section 1 describes the purpose, scope and motivation behind building the system. General description of the Patient Management System is given in section 2 and detailed requirements are specified in section 3 of the document.

2. GENERAL DESCRIPTION

2.1 Product Perspective

Patient management system is to be used by hospital administration, doctors, nurses and staff members. The external entities are other visiting doctors, multi specialty hospitals, testing laboratory, funding agencies, insurance company and management. The context diagram for the product showing its interaction with the external world is shown in Fig. A.1.

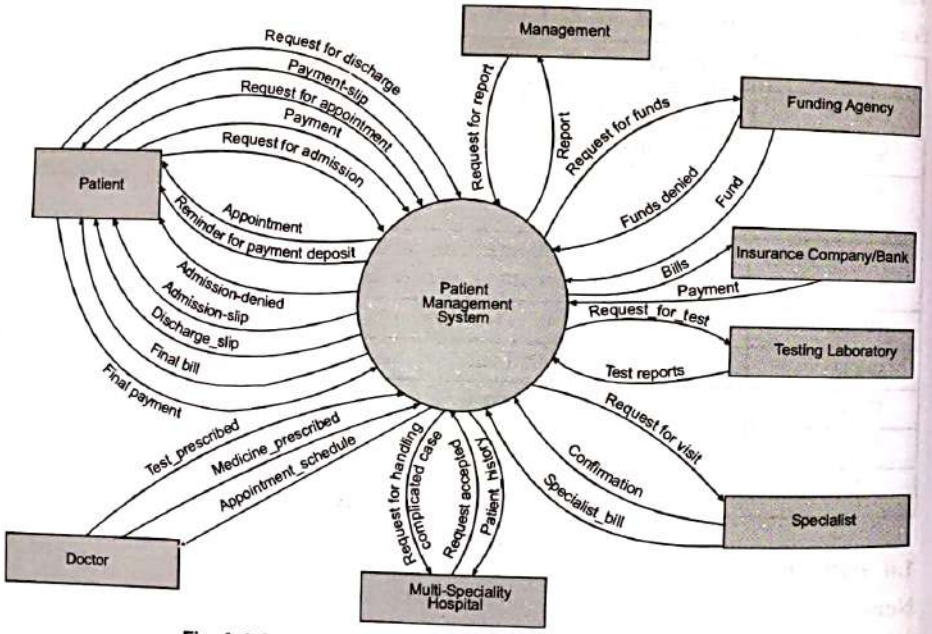


Fig. A-1 Context Diagram for Patient Management System

At present there is no interface to other systems. DBMS Oracle 9*i* will be used to store the information. The system will have maximum capacity of 64 terminals connected through LAN.

2.2 Product Functions

The Patient management system provides 4 types of main functions.

- Functions by which patient's admission/discharge is handled by the hospital staff.
- Functions by which patient's treatment records are updated by the nurses/doctors.
- Functions by which past history and treatment record of patient can be viewed by the doctors.
- Functions by which management can generate different types of reports.

They can be selected by the main menu and have restricted access. Fig. A-2 shows the initial Use Case diagram of the patient management system to illustrate the different functionalities offered by the system to the users.

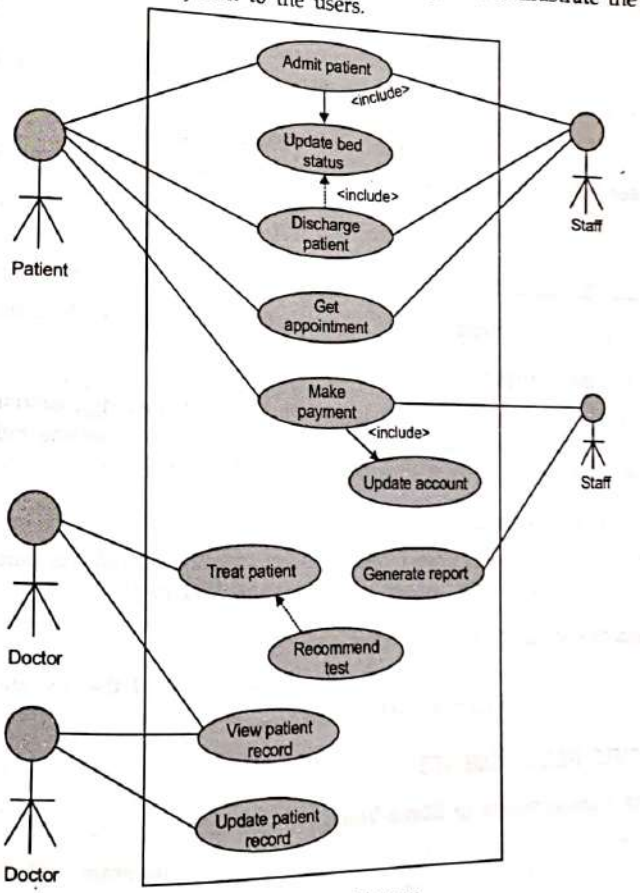


Fig. A-2 Use Case Diagram

Details of the Use Cases of Fig. A-2 are given in document named "Use Cases Description Document". Corresponding sequence diagrams, activity diagrams and Class diagram is given in the design document. A sample Use Case description for viewing the patient records by the doctor is as follows:

Name	:	View Patient record
Author	:	S. Sabharwal
Purpose	:	Enables doctor to retrieve patient record
Assumption	:	Use case starts on demand
Pre-condition	:	Provide a valid patient code
Description	:	1. Doctor enters the patient code/patient name 2. The System displays the details of treatment given to the patient in his last visits/admission
Post conditions	:	1 On successful completion patient information is displayed on the screen. 2. In case of error message is displayed.
Use case termination:		1. Any time user can exit the use case by pressing the cancel key 2. Use case may time out. 3. User presses the exit key.

Readers can write the description of the other Use Cases also in the similar manner.

2.3 User Characteristics

The users of the system may or may not be having knowledge of using automated system of this kind. Therefore the system must give necessary online help instructions. User characteristics can also be described by the Use Cases.

2.4 General Constraints

The access to different functions is restricted e.g., nurses can only update the treatment record of the patients. They cannot allot bed/room to a patient.

2.5 Assumptions and Dependencies

The system is also dependent on some other departments of the hospital for necessary inputs which are not automated yet.

3. SPECIFIC REQUIREMENTS

3.1 Data Requirements or Static View

To represent the static view we will be using the ER diagram. ER diagram for the patient management system is shown in Fig. A-3.

3.2 External Interface Requirements

The software will be used by different employees of the hospital i.e., staff members, doctors, patient, nurse etc. Some of the screen formats for the software are shown in Fig. A-4 and Fig. A-5:

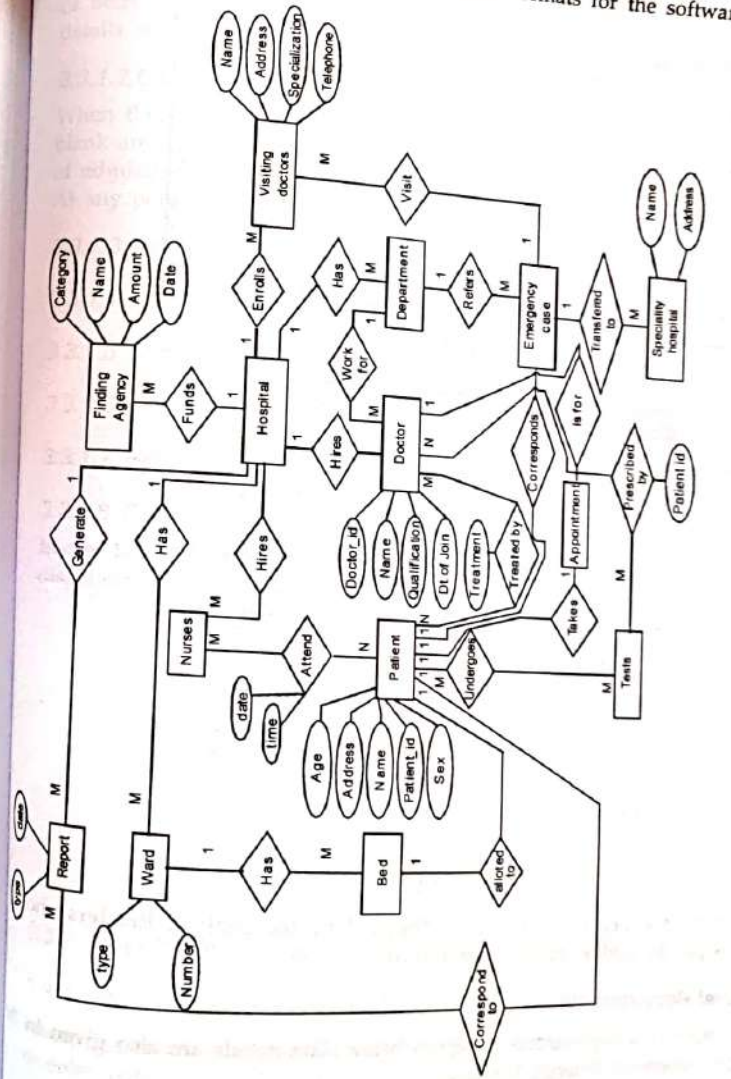


Fig. A-3 Static View

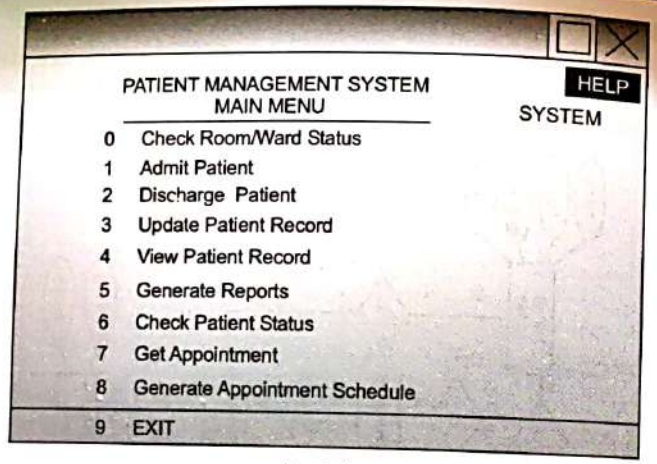


Fig. A-4

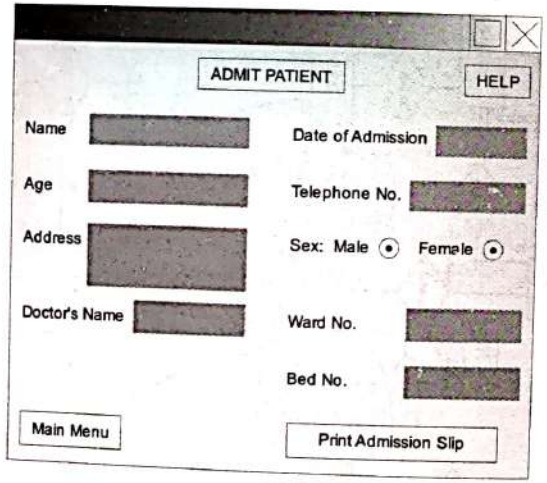


Fig. A-5

Similarly other screen formats can be designed by the analyst. Readers should complete this part by adding more screen formats.

3.3 Functional Requirements

Some of the functional requirements are given below. The details are also given in the Use Case descriptions of different use cases.

3.3.1 Hospital Staff Functions

3.3.1.1 Check Bed Status

When this function is selected by the user, he is offered a screen to check the availability of beds in private, semiprivate and general ward. On clicking at appropriate option, details are displayed on the screen.

3.3.1.2 Check for a Patient Status

When this function is selected by the user he is offered a screen with one fill in the blank area to enter patient's name. In response to this the details of the patient i.e., date of admission, date of discharge, ward no. and bed number are displayed on the screen. At any point user can go back to the main menu.

3.3.1.3 Admit Patient

3.3.1.4 Discharge Patient

3.3.1.5 Schedule Appointment

3.3.1.6 Generate Appointment Schedule

3.3.1.7 Help

3.3.1.8 Exit

In case hospital clerk selects any other function, following message as shown in Fig. A-6 is displayed on the screen.

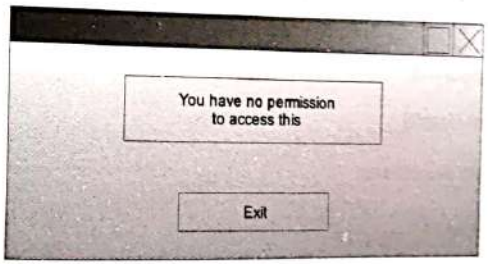


Fig. A-6

3.3.2 Hospital Nurse Functions

3.3.2.1 Enter Patient History First Time

When this function is selected by the nurses, a screen with three fill in the blank areas to enter patient-id, patient name and patient history is displayed on the screen.

3.3.2.2 Enter Medicine Details as Prescribed by Doctor

When this function is selected, a screen with following fields is displayed as shown in Fig. A-7.

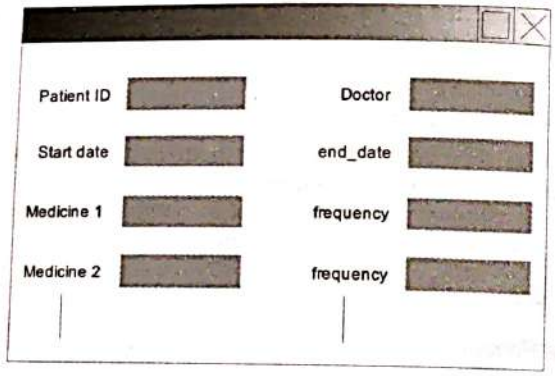


Fig. A-7

3.3.2.3 Enter Test Details Prescribed by the Doctor

3.3.2.4 Print Medicine Record Given to Patients

3.3.3 Doctor Functions

3.3.3.1 View Patient Record

When doctor selects this function a screen with two fill in the blank areas to enter patient name and bed number is displayed. When data is entered, complete history about patient is displayed.

3.3.3.2 Print Patient Details

3.3.4 Management Functions

3.3.4.1 Generate Report

When this function is selected a screen with several options to display and print the reports is displayed.

3.4 Performance Requirements

To start with system will support 20 terminals initially which in future will be increased to 64 terminals. The system must be able to serve all the users simultaneously and all queries in section 3.3 must be answered within maximum 10 seconds.

3.5 Design Constraints

There are no design constraints as such.

3.6 Quality Characteristics

3.6.1. Availability

The system must be available 24 hours.

3.6.2. Correctness

The software must satisfy and fulfill user's specifications.

3.6.3. Security

The system is made secure by providing valid user id and password to different users.

3.6.4. Maintainability

3.7 Other Requirements: NA.

Note: SRS given in this chapter is not complete. Readers are advised to complete the details and prepare a complete SRS on the similar pattern for the problem taken in this appendix or any other problem as a part of class room project. Depending upon the methodology used, reader can used the structured methodology i.e., DFD or UML to model the different details of the problem.

○○○

Appendix B

MULTIPLE CHOICE QUESTIONS

1. Degree to which software or a system is composed of discrete, independent components such that change in one component causes minimum change in other components is called
   (a) Portability (b) Self descriptiveness
   (c) Platform independence (d) Modularity
2. The ease with which software can be transferred from one environment to other environment is called
   (a) Portability (b) Cohesiveness
   (c) Platform independence (d) Modularity
3. Object-oriented modeling and programming techniques have gained popularity because they support
   (a) Platform independence (b) Self-descriptiveness
   (c) Coupling (d) Reusability of code
4. McCall has proposed a popular
   (a) Process model (b) Object model
   (c) Quality model (d) None of above
5. Capability Maturity Model consists of
   (a) 2 levels (b) 3 levels
   (c) 4 levels (d) 5 levels
6. Boehm has proposed
   (a) V model (b) Waterfall model
   (c) Prototyping process model (d) Spiral model

7. Most important feature of Spiral model is
   (a) Requirement analysis (b) Risk management
   (c) Quality management (d) Configuration management
8. Software runaways are result of
   (a) Improper requirement analysis (b) Improper design
   (c) Improper testing (d) Improper maintenance
9. Changes in efficient, cost effective and timely manner in a software are handled through
   (a) Risk management (b) Configuration management
   (c) Software maintenance (d) Both (a) and (b)
10. During a software process, a work product which marks the completion of one activity and starting of new activity is called
    (a) Module (b) Baseline
    (c) Key (d) Configuration Item
11. Seventy percent of the software development time is taken by
    (a) Requirement Analysis (b) System design
    (c) Program coding (d) Testing
12. Which of the following is not used for configuration management auditing
    (a) Critical path method (b) Gantt time allocation charts
    (c) PERT charts (d) Decision tables
13. The outcome of requirement analysis stage of software life cycle is
    (a) Quality assurance plan (b) Design document
    (c) Software requirements specification (d) Testing plan
14. Static view of an application domain can be modeled by
    (a) ER model (b) Data flow diagram
    (c) State transition diagram (d) Petrinet
15. Functional view of an application domain can be modeled with
    (a) Data flow diagram (b) Function decomposition diagram
    (c) Finite State machine (d) Both (a) and (b)
16. A form of finite state machine which is useful in modeling concurrency and asynchronous communication is
    (a) State transition diagram (b) Statechart diagram
    (c) Petrinets (d) Both (b) and (c)
17. A human action that results in a software fault is
    (a) Defect (b) Error
    (c) Failure (d) None of above

18. Reusability estimates can be compared using
    (a) Kiviat diagram (b) Radar diagram
    (c) ER diagram (d) Structure chart
19. Reusability is computed relative to
    (a) Modularity (b) Portability
    (c) Self-descriptiveness (d) (a), (b) and (c)
20. A halt in the working of a system to perform the required operation is called
    (a) Error (b) Defect
    (c) Failure (d) Fault
21. Modularity m is measured as
    (a) m = No. of modules/Total no. of variables
    (b) m = Total no. of variable/No. of modules
    (c) m = No. of modules * Total no. variables
    (d) None of above
22. Main concepts of ER model are
    (a) Entity and Relationship (b) Relationship, Primary key
    (c) Entity, Relationship and Attribute (d) None of above
23. A superkey such that set of its attributes (one or more than one) does not form a superkey is called
    (a) Candidate key (b) Primary key
    (c) Foreign key (d) None of above
24. Main concepts of Data flow diagram are
    (a) Process, Data flow (b) Store, Terminator
    (c) Both (a) and (b) (d) None of above
25. In which software architecture processing elements called filters are connected together
    (a) Process-control (b) Call and return architecture
    (c) Independent process architecture (d) Pipelined architecture
26. Which one of following is example of Call and Return architecture
    (a) Object oriented architecture (b) Layered architecture
    (c) Both (a) and (b) (d) Agent architecture
27. Static testing involves
    (a) Symbolic execution (b) Code walkthrough
    (c) Inspections (d) (a), (b) and (c)
28. Functionality of a software system is tested by
    (a) Black box testing techniques (b) Glass box testing techniques
    (c) White box testing techniques (d) Both (a) and (b)

29. Detailed design of software is tested using
(a) Black box testing (b) Glass box testing
(c) Both (a) and (b) (d) Functional testing

30. Acceptance testing is done by
(a) System analyst (b) System designer
(c) User (d) Tester

31. Schedule of testing activity is shown by
(a) Test plans (b) Test report
(c) Test cases (d) (a), (b) and (c)

32. Internal logic of a software system is tested through
(a) Functional tests (b) Performance tests
(c) Stress tests (d) Structure tests

33. Strengths and limitations of the software are determined through
(a) Performance tests (b) Structure tests
(c) Stress tests (d) Acceptance tests

34. Which one of the following is not a black box testing technique
(a) Syntax-driven testing (b) Boundary value analysis
(c) Cause-effect graphs (d) Data-flow testing

35. Which of the following testing technique supports automatic rerun of some tests for the software whenever a slight change to the product is made
(a) Regression testing (b) Acceptance testing
(c) Exhaustive testing (d) None of above

36. Structured walkthrough is a
(a) Dynamic testing technique (b) Formal static testing technique
(c) Informal static testing technique (d) Acceptance testing technique

37. All modules of the system are integrated and tested as complete system in case of
(a) Bottom up testing (b) Top down testing
(c) Sandwich testing (d) Big-Bang testing

38. COCOMO model was proposed by
(a) Boehm (b) Albert & Gaffney
(c) Codd (d) None of above

39. In basic COCOMO model effort for embedded systems is given by
(a) $3.6 * KDLOC^{1.20}$ (b) $2.4 * KDLOC^{1.05}$
(c) $3.0 * KDLOC^{1.12}$ (d) None of above

40. A system which helps senior level manager to make decision is called
(a) Online system (b) Decision support system
(c) Expert system (d) Real time system

41. The tools which support different stages of software life cycle are
(a) CASE tools (b) CAME tools
(c) CAQE tools (d) CARE tools

42. The process of building scaled down, working version of a desired system is called
(a) Rapid application development (b) Prototyping
(c) Joint application design (d) Both (a) and (b)

43. The extent to which different modules are dependent upon each other is called
(a) Modularity (b) Coupling
(c) Cohesion (d) Decomposition

44. A system which doesn't interact with external environment is called
(a) Open system (b) Closed system
(c) Logical system (d) Physical system

45. The description of project tasks and their sequence is called
(a) Project plan (b) Resource plan
(c) Work breakdown structure (d) Communication plan

46. Graphical representation of the project, showing each task activity as a horizontal bar whose length is proposition to time taken for completion of that activity is called
(a) Gantt chart (b) PERT chart
(c) Structure chart (d) Flow-chart

47. If T_L represents latest expected completion time and T_E represents earliest expected completion time, slack time for an activity is equal to
(a) $T_L - T_E$ (b) $T_E + T_L$
(c) $T_E - T_L$ (d) $2T_L - T_E$

48. All activities which are on critical path have slack time equal to
(a) 0 (b) 1
(c) 2 (d) None of above

49. A repository of all project documentation is called
(a) Project plan (b) Data dictionary
(c) Project work book (d) None of above

50. The property of sticking together of data-elements within a single module is called
(a) Coupling (b) Cohesion
(c) Decomposition (d) Modularity

51. Tools which support analysis and design stage of software development life cycle are called
(a) Upper CASE tools (b) Lower CASE tools
(c) Integrated-CASE tools (d) Cross life cycle CASE tools

52. Tools that support activities across different phases of software life cycle are called
(a) Upper CASE tools (b) Lower CASE tools
(c) Integrated-CASE tools (d) Cross life cycle case tools

53. Tools that can generate reports based upon repository contents are called
(a) Upper CASE tools (b) Code generators
(c) Documentation generators (d) Visual development tools

54. Information analysed and collected during project planning is called
(a) Baseline project plan (b) Test plan
(c) Statement of work (d) Work breakdown structure

55. Items that can be measured in rupees with certainty are called
(a) Intangible benefits (b) Tangible benefits
(c) Recurring costs (d) One time costs

56. Probability that the project will attain its desired objectives is called
(a) Technical feasibility (b) Operational feasibility
(c) Political feasibility (d) Schedule feasibility

57. Understanding the company's ability to develop the proposed system is called
(a) Technical feasibility (b) Operation feasibility
(c) Political feasibility (d) Walkthrough

58. The process by which the existing methods in an organization are replaced by new methods is called
(a) Reverse engineering (b) Business process re-engineering
(c) Conceptual modeling (d) Technical feasibility

59. The flow of data between different processes is shown by
(a) ER-diagram (b) Structure chart
(c) Statechart diagram (d) Data flow diagram

60. Data flows represent
(a) Data at rest (b) Data in store
(c) Data in motion (d) Data in repository

61. Context diagram is also called
(a) level-0 DFD (b) level-1 DFD
(c) level-2 DFD (d) level-3 DFD

62. The processes in a DFD which have only inputs are called
(a) Bubbles (b) Source
(c) Infinite sink (d) None of above

63. The DFD at the lowest level of decomposition is called
(a) Context diagram (b) Logical DFD
(c) Physical DFD (d) Primitive DFD

64. Tools used to illustrate process logic is
(a) Structured English (b) Pre/Post conditions
(c) Decision table (d) (a), (b) and (c)

65. Which one of the following is not a part of decision tables
(a) Condition stubs (b) States
(c) Actions (d) Rules

66. Rules concerning the relationships between different entity types lead to
(a) Referential integrity constraints (b) Entity integrity constraints
(c) Domain constraints (d) None of above

67. An object encapsulates
(a) Data (b) Behavior
(c) State (d) Both Data and Behavior

68. A new instance of a class is created by
(a) Query operations (b) Constructor operation
(c) Update operation (d) Destructor operation

69. A class that has no direct instances but its descendents have direct instances is called
(a) Abstract class (b) Concrete class
(c) Super class (d) Sub-class

70. A class having direct instances is called
(a) Abstract class (b) Meta-class
(c) Concrete class (d) Sub-class

71. Usage of same operation in two or more classes in different ways is called
(a) Overriding (b) Polymorphism
(c) Generalization (d) Aggregation

72. The process of replacement of a method inherited from superclass by a specific implementation in a sub-class is called
(a) Polymorphism (b) Overriding
(c) Multiple classification (d) None of above

73. UML stands for
(a) Unique modeling language (b) Universal modeling language
(c) Unified modeling language (d) None of above

74. UML is proposed by
(a) Booch (b) Jackobson
(c) Booch and Rumbaugh (d) Both (b) and (c)

75. If an object is instance of more than one class it is called
(a) Multiple inheritance (b) Multiple classification
(c) Abstraction (d) None of above

76. Something that happens in the system at a certain point in time is called
(a) Event (b) Trigger
(c) Transition (d) Condition

77. Aggregation represents
(a) is_a relationship (b) part_of relationship
(c) composed_of relationship (d) Both (b) and (c)

78. Overall modular structure of the software is represented through
(a) Structure chart (b) Flow chart
(c) Decision tree (d) Use Case diagram

79. A passive business document which consists of only predefined data and is used only for reading is called
(a) Form (b) Report
(c) Flow chart (d) None of above

80. Exchange of data between two modules in a structure chart is shown by
(a) Data couple (b) Flag
(c) Condition (d) None of above

81. The module that performs the function of getting data into the system is called
(a) Afferent module (b) Efferent module
(c) Central transform (d) None of above

82. The module which performs the function of outputting data from the system is called
(a) Afferent module (b) Co-ordinate module
(c) Efferent module (d) Transform module

83. The module which takes data from superordinate module and after doing some computation on data returns it back to superordinate module is called
(a) Afferent module (b) Efferent module
(c) Transform module (d) Coordinate module

84. When two modules refer to the same global data area they are said to be
(a) Data coupled (b) Common coupled
(c) Content coupled (d) Control coupled

85. Worst type of coupling is
(a) Data coupling (b) Control coupling
(c) Stamp coupling (d) Content coupling

86. Best coupling between modules is
(a) Data coupling (b) Stamp coupling
(c) Control coupling (d) Content coupling

87. The cohesion in which output of one instruction is used as input to another instruction in a module is called
(a) Functional cohesion (b) Sequential cohesion
(c) Procedural cohesion (d) Temporal cohesion

88. Most desirable type of cohesion is
(a) Sequential cohesion (b) Communicational cohesion
(c) Functional cohesion (d) Temporal cohesion

89. The module in which instructions are related through flow of control is said to be
(a) Communicationally cohesive (b) Temporally cohesive
(c) Coincidentally cohesive (d) Sequentially cohesive

90. Worst type of cohesion is
(a) Communicational cohesion (b) Coincidental cohesion
(c) Temporal cohesion (d) Procedural cohesion

91. Changes made to system to repair defects in coding, design or implementation is called
(a) Preventative maintenance (b) Adaptive maintenance
(c) Corrective maintenance (d) Perfective maintenance

92. Changes made to system to reduce the future system failure chances is called
(a) Preventative maintenance (b) Adaptive maintenance
(c) Corrective maintenance (d) Perfective maintenance

93. Changes made to an information system to evolve its functionality to meet changing business requirements is called
(a) Preventative maintenance (b) Adaptive maintenance
(c) Corrective maintenance (d) Perfective maintenance

94. Changes made to information system to add desired but not necessarily required features is called
(a) Preventative maintenance (b) Adaptive maintenance
(c) Corrective maintenance (d) Perfective maintenance

95. Testing method in which each module is tested in order to find errors is called
(a) Unit testing (b) Integration testing
(c) System testing (d) Black box testing

96. A functional dependency between two or more non-key attributes in a relation is called
(a) Multi-valued dependency (b) Transitive dependency
(c) Partial functional dependency (d) None of above

97. Testing of an information system using simulated data is called
(a) Alpha testing (b) Beta testing
(c) Stress testing (d) Sandwich testing

98. Testing of an information system using real data in the real user environment is called
(a) Alpha testing (b) Beta testing
(c) Stress testing (d) Big-Bang testing

99. Main building block of the UML is
(a) Thing (b) Relationships
(c) Diagram (d) (*a*), (*b*) and (*c*)

100. A physical element that exists at run time and represents a computational resource is called a
(a) Component (b) Node
(c) Use case (d) Interface

101. A semantic relationship between two things in which change in one may affect the other is called
(a) Generalization (b) Association
(c) Dependency (d) Realization

102. Which one of the following is not a part of UML
(a) Use case diagram (b) Collaboration diagram
(c) Component diagram (d) Context diagram

103. Configuration of run time processing nodes and the component is shown by
(a) Deployment diagram (b) Component diagram
(c) Sequence diagram (d) Activity diagram

104. In UML class, a contract or an obligation is called
(a) Operation (b) Responsibility
(c) Attribute (d) Activity

105. Which one of the following diagram does not represent the dynamic view of application domain in UML
(a) Statechart diagram (b) Sequence diagram
(c) Collaboration diagram (d) Component diagram

106. Which one of the following diagrams does not represent the static view of application domain in UML
(a) Class diagram (b) Deployment diagram
(c) Collaboration diagram (d) Component diagram

107. The process of transforming a model into code through mapping is called
(a) Forward engineering (b) Reverse engineering
(c) Re-engineering (d) Modeling

108. A collection of operations used to specify a service of a class is called
(a) Component (b) Use Case
(c) Interface (d) Subsystem

109. In generalization, constraint that specifies that objects of the parent can have no more than one of the children as a type is called
(a) Complete (b) Disjoining
(c) Incomplete (d) Overlapping

110. A general purpose mechanism for organizing elements into groups in UML is called
(a) Package (b) Class
(c) Deployment (d) None of the above

111. If every requirement stated in the software requirement specification (SRS) has only one interpretation, SRS is said to be
(a) Correct (b) Consistent
(c) Verifiable (d) Unambiguous

112. In a process oriented approach, system partitioning follows the criteria of
(a) Functional decomposition (b) Classification
(c) Generalization (d) None of above

113. Criteria used for identification of classes is
(a) Multiple objects (b) Similar behavior
(c) Both (a) and (b) (d) Similar attributes

114. The type of model that describes the inheritance relationships between classes is called
(a) Whole-part model (b) Classification model
(c) State-machine (d) None of above

115. A dummy program module used during software testing of higher level module is called
(a) Stub (b) Driver
(c) Test case (d) Both (a) and (b)

116. Which one of the following is not a process model
(a) Spiral model (b) V model
(c) Prototype model (d) CMM

117. Level 2 of CMM is called
(a) Repeatable (b) Defined
(c) Optimized (d) Managed

118. Level 4 of CMM is called
(a) Repeatable (b) Defined
(c) Optimized (d) Managed

119. The model which is used to evaluate the software process of an organization is called
(a) CMM (b) SPICE
(c) ISO 9001 (d) Both (a) and (b)

120. Conceptual model of a method is called
(a) Conceptual schema (b) Meta-model
(c) Data model (d) None of above

121. CORE is an acronym for
(a) Controlled Requirements Engineering
(b) Controlled Requirements Expression
(c) Controlled Requirements Elicitation
(d) None of above

122. CORE is a method for
(a) Requirements elicitation and specification
(b) Software design
(c) Testing
(d) Reverse engineering

123. FODA is an acronym for
(a) Features Oriented Domain Analysis
(b) Form Oriented Domain Analysis
(c) Function Oriented Design and Analysis
(d) None of above

124. FODA methodology for requirements definition is based on
(a) View point analysis (b) Use Case analysis
(c) Domain analysis (d) Goal analysis

125. Which one of the following is not a requirement elicitation technique
(a) Issue based information systems (b) Joint application design
(c) Prototyping (d) Data flow diagram

126. Which one of the following technique is not used for requirements specification
(a) Data flow diagram (b) State transition diagram
(c) Joint application design (d) UML

127. SSM stands for
(a) State system method (b) Soft systems methodology
(c) Structured system methodology (d) None of above

128. IEEE 830-1993 is a IEEE recommended standard for
(a) Software requirement specification (b) Software design
(c) Testing (d) (a) and (b)

129. To estimate size of project, estimater should have clear idea about
(a) Cost of the project (b) Requirement of the project
(c) Project schedule (d) None of above

130. Function point analysis was proposed by
(a) Allan J. Albrecht (b) IBM
(c) Software engineering institute (d) Booch

131. The technique in which size of project is estimated through synergy between different experts is
(a) Function point analysis (b) COCOMO
(c) Mark II function points (d) Delphi's wide band technique

132. In function point analysis no. of general system characteristics used to rate the system are
(a) 10 (b) 14
(c) 20 (d) 12

133. In function point analysis method, unadjusted function points depend on
(a) Internal logical files (b) External interface files
(c) Transitions function (d) All of above

134. Number of general characteristics in Mark II FPA method are
(a) 19 (b) 14
(c) 24 (d) None of above

135. Model used in prototyping projects for using concept of function points to compute size is called
(a) 3D function points (b) Object points
(c) Feature points (d) Quick FPA court

136. ISO 14143 is a standard for
(a) Function size measurement (b) Software requirement specification
(c) Software testing (d) None of above

137. In wide band Delphi estimate technique number of experts required are
(a) only one person (b) 15
(c) 4 to 6 (d) 8

138. COCOMO II estimation model is based on
(a) Algorithmic approach (b) Analog based approach
(c) Bottom up approach (d) Both (b) and (c)

139. Scaling factors used in COCOMO II are
(a) Development flexibility (b) Team cohesion
(c) Process maturity (d) (a), (b) and (c)

140. Which one of the following is used for representing software complexity
(a) Halstead program volume measure
(b) McCabe cyclomatic complexity measure
(c) Both (a) and (b) (d) None of above

141. Cost estimation of a project includes
(a) Hardware and software costs (b) Personnel costs
(c) Communication costs (d) (a), (b) and (c)

142. Halstead program volume measure is defined in terms of
(a) Path of the program (b) Operators of the program
(c) Operands of the program (d) Both (b) and (c)

143. Most important person in Joint Application Design based project is
(a) Sponsor (b) Facilitates
(c) Scriber (d) System specialist

144. No. of clauses used in ISO 9001 to specific quality system requirements are
(a) 15 (b) 20
(c) 25 (d) 27

145. Possible configuration item is
(a) Source code module (b) Test cases
(c) Software requirement specification (d) All of above

146. Which of the following is not a responsibility of configuration or change control board
(a) Assessing purposed change
(b) Updating affected configuration items
(c) Designing test cases
(d) Approving the change request

147. A SWAT stands for
(a) Skilled with advanced tools (b) Skilled worker advanced tools
(c) Systematic worker advanced team (d) None of above

148. A SWAT team consists of _____ members
(a) exactly 10 (b) one
(c) 4–5 (d) 8–10

149. The ease with which inputs and outputs in software can be assimilated is called.
(a) Communicativeness (b) Conciseness
(c) Data commonality (d) Communication commonality

150. The ability to link software modules/components to requirements is called
(a) Completeness (b) Consistency
(c) Modularity (d) Traceability

151. ISO standard ISO-9126 contains definitions of
(a) Quality characteristics (b) Quality subcharacteristics
(c) Quality metric (d) All of above

152. ISO 9000, a series of standards for quality management systems consists of _____ parts
(a) 4 (b) 5
(c) 3 (d) 8

153. The Capability Maturity model is based on the work of
(a) Watts Humphrey (b) Garvin
(c) Booch (d) Boehm

154. Which of the following detailed estimation models are provided by COCOMO II.
(a) Application composition model (b) Post-architecture model
(c) Early design model (d) All of above

155. Which one of the following is not a object oriented analysis and design method
(a) Object modeling technique (b) UML
(c) FUSION (d) None of above

156. Slack Time is computed in
(a) Work breakdown structure (b) Critical path method
(c) Gantt chart (d) All of the above

157. For Organic projects, according to COCOMO method, formula for effort estimation is
(a) $E = 2.4 * (KLOC)^{1.005}$ (b) $E = 3.0 * (KLOC)^{1.120}$
(c) $E = 3.6 * (KLOC)^{1.20}$ (d) $E = 2.7 * (KLOC)^{1.8}$

158. Which one of the following is not a estimation tool
(a) MS-Project (b) SLIM
(c) COSTAR (d) Estimate Professional

159. If P is Risk probability, L is Loss, then Risk exposure (RE) is computed as
(a) RE = P/L (b) RE = P+L
(c) RE = P*L (d) RE = P*L/2

160. An human action resulting into an incorrect result is called
(a) Fault (b) Mistake
(c) Failure (d) Error

161. The amount by which the result is directed from the expected result is called
(a) Mistake (b) Fault
(c) Error (d) Failure

162. Maximum errors are concentrated in which of the following stage of software life cycle.
(a) Requirement analysis (b) System design
(c) Coding (d) Installation

163. Testing the software means
(a) Verification only (b) Validation only
(c) Both verification and validation (d) None of above

164. Which of the following model integrates development and testing process
(a) Spiral model (b) Prototyping model
(c) Incremental model (d) Dotted-U-model

165. Testing which starts after completing the functional design is called
(a) End game testing (b) Partial testing
(c) Full testing (d) Audit level testing

166. Which of the following is not a validation activity
(a) Unit testing (b) System testing
(c) Acceptance testing (d) Walkthrough

167. The activity of managing and coordinating changes is
(a) Reverse engineering (b) Risk management
(c) Configuration management (d) Maintenance

168. Which of the following is not a verification activity
(a) Acceptance testing (b) Inspections
(c) Walkthrough (d) Buddy check

169. Which of the following is not a black box testing.
(a) Syntax testing (b) Cause-effect graph
(c) Path coverage (d) Boundary-value analysis

170. Which of the following validation activities belong to low-level testing
(a) Unit testing (b) Integration testing
(c) System testing (d) Both (a) and (b)

171. ALPHA and BETA testing are forms of
(a) Acceptance testing (b) Integration testing
(c) System testing (d) None of above

172. Testing done to ensure that software bundles the required volume of data is called
(a) Load testing (b) Volume testing
(c) Performance testing (d) Stress testing

173. Acceptance testing is done by
(a) System analyst (b) Programmer
(c) User (d) Tester

174. Approaches to integration testing are
(a) Top down integration (b) Bottom-up integration
(c) Incremental integration (d) (a), (b) and (c)

175. IEEE standard for software test documentation is
(a) IEEE/ANSI standard 829-1983 (b) IEEE/ANSI standard 1008-1987
(c) IEEE/ANSI standard 1028-1988 (d) IEEE/ANSI standard 828-1990

ANSWERS

1. (d)	2. (a)	3. (d)	4. (c)	5. (d)
6. (d)	7. (b)	8. (a)	9. (b)	10. (b)
11. (a)	12. (d)	13. (c)	14. (a)	15. (d)
16. (c)	17. (b)	18. (a)	19. (d)	20. (c)
21. (a)	22. (c)	23. (a)	24. (c)	25. (d)
26. (c)	27. (d)	28. (a)	29. (b)	30. (c)
31. (a)	32. (d)	33. (d)	34. (d)	35. (a)
36. (c)	37. (d)	38. (a)	39. (a)	40. (b)
41. (a)	42. (d)	43. (b)	44. (b)	45. (c)
46. (a)	47. (a)	48. (a)	49. (c)	50. (b)
51. (a)	52. (d)	53. (c)	54. (a)	55. (b)
56. (b)	57. (a)	58. (b)	59. (d)	60. (c)
61. (a)	62. (c)	63. (d)	64. (d)	65. (b)
66. (a)	67. (d)	68. (b)	69. (a)	70. (c)
71. (b)	72. (b)	73. (c)	74. (d)	75. (b)
76. (a)	77. (d)	78. (a)	79. (b)	80. (a)
81. (a)	82. (c)	83. (c)	84. (b)	85. (d)
86. (a)	87. (b)	88. (c)	89. (b)	90. (b)
91. (c)	92. (a)	93. (b)	94. (d)	95. (a)
96. (b)	97. (a)	98. (b)	99. (d)	100. (b)
101. (c)	102. (d)	103. (a)	104. (b)	105. (d)
106. (c)	107. (a)	108. (c)	109. (b)	110. (a)
111. (d)	112. (a)	113. (c)	114. (b)	115. (d)
116. (d)	117. (a)	118. (d)	119. (a)	120. (b)
121. (b)	122. (a)	123. (a)	124. (c)	125. (d)
126. (c)	127. (b)	128. (a)	129. (d)	130. (a)
131. (d)	132. (b)	133. (d)	134. (a)	135. (b)
136. (a)	137. (c)	138. (a)	139. (d)	140. (c)
141. (d)	142. (d)	143. (a)	144. (b)	145. (d)
146. (b)	147. (a)	148. (c)	149. (a)	150. (d)
151. (d)	152. (b)	153. (a)	154. (d)	155. (d)
156. (b)	157. (a)	158. (a)	159. (c)	160. (b)
161. (c)	162. (a)	163. (c)	164. (d)	165. (b)
166. (d)	167. (c)	168. (a)	169. (c)	170. (d)
171. (a)	172. (b)	173. (c)	174. (d)	175. (a)

○○○

Appendix C

LIST OF IMPORTANT STANDARDS

1. **IEEE Std. 730-1989** Standard for software quality assurance plans.
2. **IEEE Std. 1058.1-1987** Standard for software project management plans.
3. **IEEE Std. 730.1-1995** Guide for software quality assurance planning.
4. **IEEE Std. 1228-1994** Standard for software safety plans.
5. **IEEE Std. 828-1990** Standard for software configuration management plans.
6. **IEEE Std. 1012-1986** Standard for software verification and validation plans.
7. **IEEE Std. 1219-1992** Standard for software maintenance.
8. **IEEE Std. 1074-1995** Standard for developing software life cycle processes.
9. **IEEE Std. 1008-1987** Standard for software unit testing.
10. **IEEE Std. 1028-1988** Standard for software reviews and audits.
11. **IEEE Std. 829-1983** Standard for software test documentation.
12. **IEEE Std. 830-1993** Recommended practice for software requirement specifications.
13. **IEEE Std. 1016-1987** Recommended practice for software design description.
14. **IEEE Std. 610.12-1990** Standard glossary for software engineering terminolog
15. **IEEE Std. 1045-1992** Standard for software productivity metrics.
16. **IEEE Std. 1002-1987** Standard taxonomy for software engineering standards.
17. **IEEE Std. 1062-1993** Recommended practice for software acquisition.
18. **IEEE Std. 829-1983** Standard for Software test documentation.
19. **IEEE Std. 1209-1992** Recommended practice for the evaluation and selection CASE tools.

20. **ISO 9000-1: 1994** Quality management and quality assurance standards - Part 1 : Guidelines for selection and use.
21. **ISO 9000-2: 1997** Quality management and quality assurance standards - Part 2 : Generic guidelines for applying ISO 9001, ISO 9002 and ISO 9003.
22. **ISO 9000-3: 1991** Quality management and quality assurance standards - Part 3 : Guidelines for applying 9001 to development, supply, installation and maintenance of computer software.
23. **ISO 9000-4: 1993** Quality management and quality assurance standards - Part 4: Guide to dependability programme management.
24. **ISO 9001: 1994** Quality Systems - Model for quality assurance in design, development, production, installation and servicing.
25. **ISO 9002: 1994** Quality Systems - Model for quality assurance in production, installation and servicing.
26. **ISO 9003: 1994** Quality Systems - Model for quality assurance in final inspection and test.
27. **ISO 10005: 1995** Quality management - Guidelines for quality plans.
28. **ISO 10007: 1995** Quality management - Guidelines for configuration management.
29. **ISO 10013: 1996** Quality management - Guidelines for developing quality manuals.

○○○

Appendix D

LIST OF LABORATORY EXERCISES

1. Identify an application domain say hospital, hotel, library, shopping mall etc. Write down the functional requirements of the problem domain using simple English.
2. Develop the following views of the problem on paper.
   2.1. Static view using ER diagram.
   2.2. Functional view using function decomposition diagram and Data flow diagram.
   2.3. Dynamic view using state transition diagram.
3. Develop the structure chart using the DFD developed in exercise 2.2.
4. If Object Oriented Requirements Analysis is part of course, develop the three views using UML diagrams, i.e., Class diagram, Use Case diagram, Activity diagram, Sequence diagram, Collaboration diagram and Statechart diagram.
5. Explore the various features of a CASE tool available in the lab.
6. Using a project management software e.g., Ms-Project, solve a problem to find critical path.
7. Develop the final SRS of your problem using IEEE std-830 format.
8. Write a C program to convert in a mathematical expression written in infix notating to post fix notation using stack. For this program compute the cyclomatic complexity.
9. For the program in problem 8, write the test cases using boundary value analysis.
10. Write a C++ program using inheritance concept. For this program compute object oriented metrics proposed by chidamber.

GLOSSARY

A

Adaptive system A system which by monitoring the outputs changes the transform process.

Actor An actor is a role played by a person, organization, other system or device which interacts with the system.

Association A relationship that relates a class X to class Y.

Aggregation An abstraction mechanism that supports building of complex objects out of existing objects.

Activity diagram It is one of nine diagrams of UML used for modeling flow of activities for a process.

Acceptance testing Testing done by the user to validate the product.

Activity graph A graph used to depict the interdependence of different activities of a project.

Afferent module The module which receives data from subordinate module and passes it to superordinate module.

Adaptive maintenance Modifications made to the software as a result of change to external operating environment.

B

Brainstorming A group technique to promote creative thinking, used for requirements elicitation.

Black box testing A testing strategy used to validate the functional requirements of the system without considering the internal structure of the program.

Big-Bang testing An integration testing approach in which all the modules after individual testing are combined together and tested in one go.

Basic-Block A set of consecutive statements that can be executed without branching.

Baseline An official version of project components at any point of time.

C

Capability Maturity Model (CMM) A framework developed by Software Engineering Institute to assess the processes followed by an organization to develop the software.

Communication system A system which helps people to exchange information efficiently using tools like Intranet, Fax, e-mail etc.

Class A class represents multiple objects with similar behaviour.

Class diagram A diagram used to show the static view of the system in terms of classes and their relationships.

Collaboration diagram One of the UML diagram that focuses on object interactions irrespective of time.

Cause effect graphing A black box testing technique that establishes relationships between logical input combinations called causes and corresponding actions called effects.

Cyclomatic complexity It is a metric used to measure logical complexity of the program. This value defines the number of independent paths in the program to be executed in order to ensure that all statements in the program are executed at least once.

Condition testing A white box testing technique that tests all logical conditions in a program.

Capture/playback tool An automated testing tool which records and replays the test input scripts number of times.

Critical Path Method (CPM) A method that shows the analysis of paths in an activity graph among different milestones of the project.

Critical activities Activities lying on the critical path.

Critical path Activities with slack time zero constitute critical path. This path gives us the minimum time required to complete the project.

COCOMO Constructive Cost Estimation Model proposed by Boehm to estimate effort.

Context diagram A Level 0 dataflow diagram in which working of whole organization is represented by a single process and its interaction with external agents is shown through exchange of data.

Coordinate module A module which coordinates the working of two or more modules.

Coupling Degree of interdependence between two modules.

Cohesion A glue that keeps the data elements within a single module together.

Corrective maintenance It is the activity of fixing the reporting errors.

Configuration item A component present in the base line.

Configuration management Systematic management of changes made to evolving software.

CASE Computer Assisted/Aided Software Engineering.

CCB Configuration Control Board.

D

Domain requirement The requirements that are specific to an application domain e.g., banking.

Domain analysis A requirement elicitation technique that focuses on reuse of requirements from similar domain.

Decision support system A system used by high level management to take suitable decisions.

Dynamic binding A technique that dynamically binds the message send to the appropriate code at run time so as to activate a polymorphic function call.

Dependency Dependencies are means to show relationship between Use Cases and is of two types include and extend.

Debugging It is a systematic review of program text in order to detect faults.

Driver A program which accepts the test case data to be inputted and printed by the module to be tested.

Decomposition A technique which allows breakup of complex system into sub-systems which are easy to handle.

Data flow diagram A modeling tool used to model functional view of system in term of business processes and flow of data between these processes.

Data dictionary An organized listing of all data elements of the system with their precise and unambiguous definitions.

E

Economic feasibility It is the measure of cost effectiveness of the system and includes cost benefit analysis, income generated etc.

Expert system A system which using the techniques of artificial intelligence simulates the thinking of the expert.

Error Amount of deviation from the correct result.

Equivalence partitioning A black box testing technique that partitions the input domain of the system into a finite number of equivalence classes such that each member of the class behaves in similar fashion.

Estimation The process of reliably predicting size, effort, schedule and cost of project.

EAF Effort Adjustment Factor.

Event Any thing which happens in the system e.g., customer places order.

Efferent module A module that receives data from superordinate module and passes it to subordinate module.

F

Functional requirements The requirements that describe the functionality of the system or the services provided by the product.

Feasibility How beneficial or practical the development of system will be to an organization.

Feasibility analysis A process used to measure the feasibility or necessary study done to ensure that project is worth considering.

Fault It is the outcome of the mistake and can be wrong step, definition etc., in the program.

Failure Failure is the outcome of fault.

FPA Function Point Analysis, a popular method which computes size of software in terms of function points.

FPC Function Point Count.

Function decomposition diagram A diagram that represents working of system in terms of hierarchy of functions.

G

Gantt chart A chart used for representing project plans graphically.

GUI Graphical User Interface.

I

Interview A popular technique used for requirements elicitation. It can be open-ended, structured or can take form of questionnaire.

Inheritance It is the process of creating new classes from the existing base classes.

Independent path A path in the program consisting of at least one new condition or set of processing statements.

Integration testing The process of combining multiple modules systematically for conducting tests in order to find errors in the interface between modules.

IEEE Institute of Electrical and Electronics Engineers.

ISO International Standards Organization.

K

Key process areas Issues/targets at a maturity level of CMM that must be satisfied by the organization to achieve that maturity level.

KLOC One thousand lines of code.

L

Legal feasibility This activity focuses on dealing with legal issues like infringement, contracts, copyright etc.

LOC Lines of Code.

M

Mechanistic system A system that takes a well defined inputs and generates a well defined set of outputs through a transform process.

Man-made systems The systems that are constructed, operated and maintained by humans e.g., financial system. They make use of computers and are also called automated systems.

Mistake A human action leading to incorrect result.

Mutation testing An error based testing technique which looks for presence of errors.

Mutant A program variant generated by introducing known bugs.

Memory-testing tool A testing tool that focuses on testing memory related problems.

Milestone The outcome of an activity.

Module The smallest unit of software encapsulating design.

N

Non-functional requirements Constraints imposed on the system.

Note A dog eared rectangle used for adding comments to the UML diagram.

O

Operational feasibility Operational feasibility focuses on evaluating whether a system will work properly in the organization and also covers issues like efficiency, security, performance etc.

Office-automation system A system which supports day to day processing tasks of an organization and helps in improving its productivity e.g. spread sheets, word processors.

On-line system A system to which inputs are given directly from where they are created and outputs are also returned directly to the concerned persons.

Object oriented requirements analysis A technique for understanding the problem domain and representing the requirements of the software in terms of problem domain objects and interactions among them.

Object It is an instance of a class. It is described as a data structure together with operations that act on that structure.

P

Process A set of activities or steps required to build or produce the product.

Prototype A partially developed product.

Prototyping A process of developing a working model of a system.

Polymorphism It is a property that allows a single function at conceptual level to take different forms depending on the object type on which it is acting i.e., different classes support identically named methods with different arguments.

Package Packages are used to organize the UML diagrams logically.

Predicate node A node with a condition in the flow graph.

Project An enterprise carefully planned to achieve a particular aim.

Project management System of management procedures, techniques, resources, know how, technology etc., required for successful management of the project.

PERT Project Evaluation and Review Technique.

PI Productivity Index.

Perfective maintenance It is the activity of improving the delivered software as a result of change in user requirements.

Preventative maintenance It is done to take care of future problems.

R

Requirement A feature which end user needs in the existing or new system.

Requirements engineering A systematic process of developing requirement specification document.

Requirement specification document Output of first stage i.e., requirements analysis of software life cycle consisting of project requirements.

Requirement elicitation The process of acquiring information about a specific problem domain using various techniques to build the requirements model.

Requirement specification It is an activity which produces formal requirements model.

Requirement validation It is an activity which insures the consistency of requirements model with respect to customer's needs.

Real time system A system that controls the environment quickly by generating and returning the results to the environment after receiving inputs from it.

Review It is human examination of the work product.

Regression testing Regression testing involves automatically running a set of tests whenever changes are made to the software.

Risk An unknown event which if occurs can lead to project failure.

Risk management Process of identifying and understanding the risks that lead to project delay or failure.

RE (Risk exposure) A metric which is computed using the formula $RE = P * L$ where P is the risk probability and L is Loss.

Reverse engineering The process of analyzing a system in order to identify its components and their relationships.

S

Software runaways Large size projects that failed due to lack of usage of systemati techniques and tools.

Software engineering A systematic approach or strategy for producing quality software

Software development life cycle The duration of time that begins with conceptualizatio of software being developed and ends after system is discarded after its usage.

System design A stage of software development life cycle which focuses on developin overall design of software.

Software process A detailed description of process which defines guidelines for softwar engineers to develop the product using methods and tools.

Scenario A story telling session that illustrates how a perceived system will behave.

Software requirement specification A document which contains the complete descriptio of software without saying anything about implementation details. It is the output c requirements engineering process and is also called Requirements Specification Documen

System A system is a collection of interrelated elements or components working togethe in order to achieve a common objective.

Systems engineering It is a process of developing systems through an iterative proces of top down synthesis so as to ensure complete requirements and a balanced developmer schedule.

System analysis Most important activity of system engineering that focuses o identification of goals/objectives of the system.

Schedule feasibility It is an activity that concerns building the proper development schedule of the project as per customer deadline.

Sequence diagram It is one of UML diagrams that shows how operations are carried out and how objects are created and manipulated according to time.

Statechart diagram A diagram that shows all the possible states in an object's life time and events that trigger the change in states.

Software testing It is the process of testing the software in order to find defects and to improve its quality.

Syntax-driven testing A block box testing technique used to test systems that can be represented syntactically by some grammar.

Statement coverage A white box testing technique which ensures that each statement of the program is executed.

Stub A software program that simulates the module called by a module under test.

Slack time The difference between available time to finish the activity and the actual time required to finish the activity.

SEI Software Engineering Institute.

State State is described by set of attribute values at a particular instant of time.

Structure chart A chart showing overall modular structure of the program.

Superordinate module The calling module is called superordinate module.

Software quality It is the characteristic of the product to satisfy the specific needs.

Software metric A unit of measurement characterizing software engineering product, process and people.

SPICE Software Process Improvement and Capability dEtermination project.

Software reliability Probability that the software will operate failure free for a specified time under specified operating conditions.

Software quality assurance A framework which ensures that quality standards and practices have been used to develop a good quality software which meets end user's requirements.

Software maintenance It is the modification of software after its delivery to correct faults and to adopt the product to a modified environment.

Software reengineering The activity of understanding the existing system and replacing the system partially or fully using new techniques and technology.

Software restructuring It is the activity of making changes to the software so that it becomes easier to understand and change.

Software reuse Use of an existing software component in the same system or a new system.

SADT Structured Analysis and Design Technique.

T

Task-analysis A requirement elicitation technique which decomposes the problem domain into a hierarchy of tasks and subtasks.

Technical feasibility Technical feasibility is a technique to assess that whether available technology meets the system needs and also concerns issues like risks involved, resources available etc.

Transaction It is an event taking place in the organization that modifies the existing data or creates new data.

Transaction processing system A system that collects, stores and processes data about different transactions taking place in the organizations.

Tester A person whose job is to find faults in the product.

Testware A work product produced by software test engineers or testers consisting of checklists, test plans, test cases, test reports, test procedures etc.

Test case It is a set of inputs (pre conditions plus actual inputs) and expected outputs (post conditions and actual outputs) for a program under test.

Test case generator An automated testing tool used for generating test cases from SRS, program or design languages.

Test comparator A tool which compares the results of software under test with the expected result and generates the discrepancy reports.

Test database A sample of database being manipulated when the software under test is executed.

Temporal event An event that takes place at fixed day and time.

Transform module A module that receives data from superordinate module and after processing sends it back to superordinate module.

Transform analysis A structured process that converts a transform centered DFD into a structure chart.

U

Unified modeling language A standard language used for developing software blueprints using nine different diagrams.

Use case diagram A diagram that represents the system from user's perspective in terms of features that a user expects the system to provide. It is a collection of Use Cases, actors and their communication.

Use case A Use Case in a Use Case diagram shows the required features of the system which the actors would like the system to support without saying anything about how it will be achieved.

Use case narrative Textual description used to describe a Use Case.

Unit testing A testing technique which focuses on testing smallest component of software using white box testing techniques.

V

Verification The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase. (IEEE definition)

Validation The process of evaluating a system or component during or at the end of development process to determine whether it satisfies specified requirements. (IEEE/ ANSI definition). Validation requires execution of final end product.

W

White box testing A testing strategy used for testing the internals of the program.

Work breakdown structure A tree type structure used to depict project in terms of phases. Each of the phase can be represented by number of steps.

○○○

REFERENCES

[Albrecht79] A. Albrecht, *"Measuring Application Development Productivity"*, Proc. IBM Application Development Symposium, Monterey, California, October 14 - 17, 1979.

[Arango87] Arango G., Baxter I., Freeman P. (1987), *"A framework for Incremental Progress, the Application of AI to Software Engineering"*, Research Report, Department of Information and Computer science, University of California, Irvine, CA, May 1987.

[Arthur88] L.J. Arthur, *"Software Evolution: The Software Maintenance Challenge"*, John Wiley & Sons Inc., New York, 1988.

[Andersen89] Anderson J. and Fickas S.A. (1989), *"Proposed Perspective Shift: Viewing Specifications Design as a Planning Problem"*, In Proc. 5th International Workshop on Software Specification and Design, IEEE, Pittsburg, PA.

[Arnold89] Robert S. Arnold, *"Software Restructuring"*, Proc. IEEE, Vol. 77, No. 4, pp. 607 - 617, April 1989.

[Ali96] Ali Behforooz and Fredrick J. Hudson, *"Software Engineering Fundamentals"*, Oxford University Press.

[Alan97] Alan C Gillies, *"Software Quality Theory and Management"*, published by International Thomson Computer Press, 1997.

[Aggarwal05] Aggarwal K. K. and Yogesh Singh, *"Software Engineering: Programs, Documentation, Operating System"*. New Age International Publishers, 2005.

[Boehm78] B.W. Boehm, J.R. Brose, H. Kaspar, M. Lipw, G.J. Macleod and M.J. Merritt, *"Characteristics of Software Quality"*, Amsterdan, North Holland 1978.

[Boehm81] B.W. Boehm, *"Software Engineering Economics"*, published by Prentice Hall, New Jersey, 1981.