UNIT 1 SOFTWARE ENGINEERING AND ITS MODELS

Structure **Page Nos.**

1.0 INTRODUCTION

The field of software engineering is related to the development of software. Large software needs systematic development unlike simple programs which can be developed in isolation and there may not be any systematic approach being followed.

In the last few decades, the computer industry has undergone revolutionary changes in hardware. That is, processor technology, memory technology, and integration of devices have changed very rapidly. As the software is required to maintain compatibility with hardware, the complexity of software also has changed much in the recent past. In 1970s, the programs were small, simple and executed on a simple uniprocessor system. The development of software for such systems was much easier. In the present situation, high speed multiprocessor systems are available and the software is required to be developed for the whole organisation. Naturally, the complexity of software has increased many folds. Thus, the need for the application of engineering techniques in their development is realised. The application of engineering approach to software development lead to the evolution of the area of Software Engineering. The IEEE glossary of software engineering terminology defines the Software Engineering as:

"(a) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, that is, the application of engineering to software. (b) The study of approaches in (a)."

There is a difference between programming and Software Engineering. Software Engineering includes activities like cost estimation, time estimation, designing, coding, documentation, maintenance, quality assurance, testing of software etc. whereas programming includes only the coding part. Thus, it can be said that programming activity is only a subset of software development activities. The above mentioned features are essential features of software. Besides these essential features, additional features like reliability, future expansion, software reuse etc. are also considered. Reliability is of utmost importance in real time systems like flight control, medical applications etc.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- define software engineering;
- understand the evolution of software engineering;
- understand the characteristics of software;
- learn about phases of software development life cycle, and
- understand software development models.

1.2 EVOLUTION OF SOFTWARE ENGINEERING

Any application on computer runs through software. As computer technologies have changed tremendously in the last five decades, accordingly, the software development has undergone significant changes in the last few decades of 20th century. In the early years, the software size used to be small and those were developed either by a single programmer or by a small programming team. The program development was dependent on the programmer's skills and no strategic software practices were present. In the early 1980s, the size of software and the application domain of software increased. Consequently, its complexity has also increased. Bigger teams were engaged in the development of Software. The software development became more bit organised and software development management practices came into existence.

In this period, higher order programming languages like PASCAL and COBOL came into existence. The use of these made programming much easier. In this decade, some structural design practices like top down approach were introduced. The concept of quality assurance was also introduced. However, the business aspects like cost estimation, time estimation etc. of software were in their elementary stages.

In the late 1980s and 1990s, software development underwent revolutionary changes. Instead of a programming team in an organisation, full-fledged software companies evolved (called software houses). A software houses primary business is to produce software. As software house may offer a range of services, including hiring out of suitably qualified personnel to work within client's team, consultancy and a complete system design and development service. The output of these companies was 'Software'. Thus, they viewed the software as a *product* and its functionality as a *process*. The concept of software engineering was introduced and Software became more strategic, disciplined and commercial. As the developer of Software and user of Software became separate organisation, business concepts like software costing, Software quality, laying of well-defined requirements, Software reliability, etc., came into existence. In this phase an entirely new computing environments based on a knowledge-based systems get created. Moreover, a powerful new concept of object-oriented programming was also introduced.

The production of software became much commercial. The software development tools were devised. The concept of Computer Aided Software Engineering (CASE) tools came into existence. The software development became faster with the help of CASE tools.

The latest trend in software engineering includes the concepts of software reliability, reusability, scalability etc. More and more importance is now given to the quality of the software product. Just as automobile companies try to develop good quality automobiles, software companies try to develop good quality Software. The software creates the most valuable product of the present era, i.e., information.

The following *Table* summarises the evolution of software:

1960s Infancy	Machine Code
1970s Project Years	Higher Order Languages
1980s Project Years	Project Development
1990s Process and Production Era	Software Reuse

The problems arising in the development of software is termed as crisis. It includes the problems arising in the process of development of software rather than software functioning. Besides development, the problems may be present in the maintenance and handling of large volumes of software. Some of the common misunderstandings regarding software development are given below.

1. Correcting errors is easy. Though the changes in the software are possible, but, making changes in large software is extremely difficult task.

2. By proper development of software, it can function perfectly at first time. Though, theoretically, it seems correct, but practically software undergoes many development/coding/testing passes before becoming perfect for working.

3. Loose objective definition can be used at starting point. Once a software is developed using loose objective, changing it for specific objectives may require complete change.

4. More manpower can be added to speed up the development. Software is developed by well coordinated teams. Any person joining it at a later stage may require extra efforts to understand the code.

Software Standards

Various terms related to software engineering are regularly standardised by organisations like IEEE (Institute of Electrical and Electronics Engineers), ANSI (American National Standards Institute), OMG (Object Management Group), CORBA (Common Object Request Broker Architecture).

IEEE regularly publishes software development standards.

OMG is international trade organisation (http://www.omg.org) and is one of the largest consortiums in the software industry. CORBA defines the standard capabilities that allow objects to interact with each other.

1.3 SOFTWARE DEVELOPMENT MODELS

Software Engineering deals with the development of software. Hence, understanding the basic characteristics of software is essential. Software is different from other engineering products in the following ways:

1. Engineering products once developed cannot be changed. To modifications the product, redesigning and remanufacturing is required. In the case of software, ultimately changes are to be done in code for any changes to take effect.

2. The Other Engineering products are visible but the software as such is not visible. That’s why, it is said that software is developed, but not manufactured. Though, like other products, it is first designed, then produced, it cannot be manufactured automatically on an assembly line like other engineering products. Nowadays, CASE (Computer Aided Software Engineering) tools are available for software development. Still it depends on the programmer’s skill and creativity. The creative skills of the programmer is difficult to quantify and

standardise. Hence, the same software developed by different programmers may take varying amount of time, resources and may have variable cost.

3. Software does not *fail* in the traditional sense. The engineering products has wear and tear in the operation. Software can be run any number of times without wear and tear. The software is considered as *failed* if:

   a) It does not operate correctly.
   b) Does not provide the required number of features.

4. Engineering products can be perfectly designed, but in the case of software, however good the design, it can never be 100% error free. Even the best quality software is not completely error free. A software is called good quality software if it performs the required operation, even if it has a few errors.

5. The testing of normal engineering products and software engineering products are on different parameters. In the former, it can be full load testing, etc., whereas in the case of software, testing means identification of test cases in which software may fail. Thus, testing of software means running of software for different inputs. By testing, the presence of errors is identified.

6. Unlike most of the other engineering products, software can be reused. Once a piece of code is written for some application, it can be reused.

7. The management of software development projects is a highly demanding task, since it involves the assessment of the developers creative skills. The estimation regarding the time and cost of software needs standardisation of developers creativity, which can be a variable quantity. It means that software projects cannot be managed like engineering products. The correction of a bug in the case of software may take hours But, it may not be the case with normal engineering products.

8. The Software is not vulnerable to external factors like environmental effects. But the same external factors may harm hardware. The hardware component may be replaced with spare parts in the case of failure, whereas the failure of a software component may indicate the errors in design.

Thus, the characteristics of software are quite different from other engineering products. Hence, the software industry is quite different from other industries.

1.3.1 Importance of Software Engineering

As the application domains of software are becoming complicated and design of big software without a systematic approach is virtually impossible, the field of software engineering is increasingly gaining importance. It is now developing like an industry. Thus, the industry has to answer following or similar queries of clients:

1) What is the best approach to design of software?
2) Why the cost of software is too high?
3) Why can’t we find all errors?
4) Why is there always some gap between claimed performance and actual performance?

To answer all such queries, software development has adopted a systematic approach.

Software development should not remain an art. Scientific basis for cost, duration, risks, defects etc. are required. For quality assurance, product qualities and process qualities and must be made measurable as far as possible by developing metrics for them.

1.3.2 Various Development Models

The following are some of the models adopted to develop software:

(i) Build and Fix Model

It is a simple two phase model. In one phase, code is developed and in another, code is fixed.

Figure 1.1 depicts the Build and Fix model.

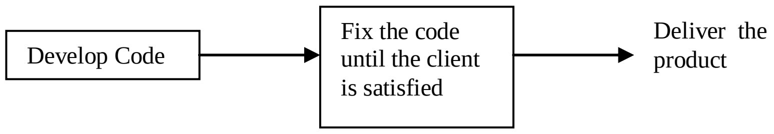

Figure 1.1 : Build and fix model

(ii) Waterfall Model

It is the simplest, oldest and most widely used process model. In this model, each phase of the life cycle is completed before the start of a new phase. It is actually the first engineering approach of software development.

Figure 1.2 depicts Water Fall Model.

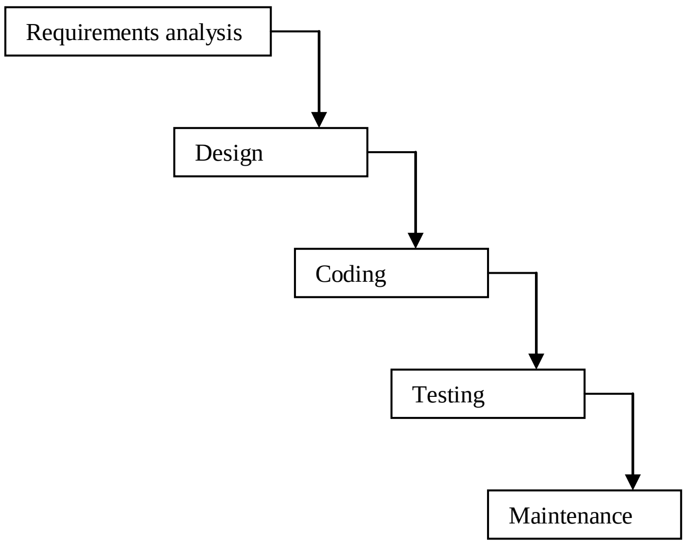

Figure 1.2 : Water fall model

The functions of various phases are discussed in software process technology.

The waterfall model provides a systematic and sequential approach to software development and is better than the build and fix approach. But, in this model, complete requirements should be available at the time of commencement of the project, but in actual practice, the requirements keep on originating during different phases. The water fall model can accommodate the new requirements only in maintenance phase. Moreover, it does not incorporate any kind of risk assessment. In waterfall model, a working model of software is not available. Thus, there is no methods to judge the problems of software in between different phases.

A slight modification of the waterfall model is a model with feedback. Once software is developed and is operational, then the feedback to various phases may be given.

Figure 1.3 depicts the Water Fall Model with feedback.

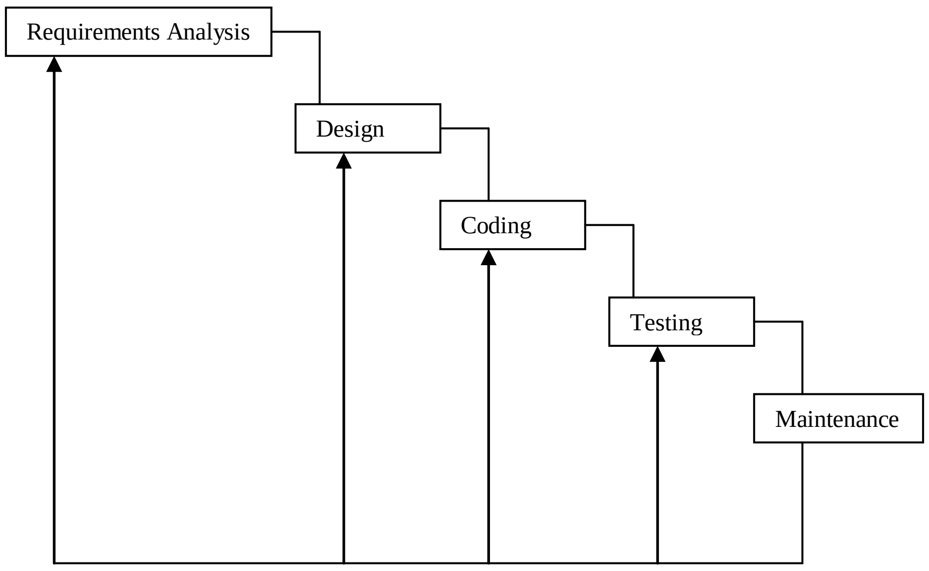

Figure 1.3 : Water fall model with feedback

(iii) Iterative Enhancement Model

This model was developed to remove the shortcomings of waterfall model. In this model, the phases of software development remain the same, but the construction and delivery is done in the iterative mode. In the first iteration, a less capable product is developed and delivered for use. This product satisfies only a subset of the requirements. In the next iteration, a product with incremental features is developed. Every iteration consists of all phases of the waterfall model. The complete product is divided into releases and the developer delivers the product release by release.

Figure 1.4 depicts the Iterative Enhancement Model.

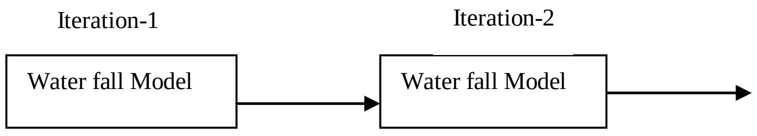

Figure 1.4 : Iterative enhancement model

This model is useful when less manpower is available for software development and the release deadlines are tight. It is best suited for in-house product development, where it is ensured that the user has something to start with. The main disadvantage of this model is that iteration may never end, and the user may have to endlessly wait for the final product. The cost estimation is also tedious because it is difficult to relate the software development cost with the number of requirements.

(iv) Prototyping Model

In this model, a working model of actual software is developed initially. The prototype is just like a sample software having lesser functional capabilities and low reliability and it does not undergo through the rigorous testing phase. Developing a working prototype in the first phase overcomes the disadvantage of the waterfall model where the reporting about serious errors is possible only after completion of software development.

The working prototype is given to the customer for operation. The customer, after its use, gives the feedback. Analysing the feedback given by the customer, the developer refines, adds the requirements and prepares the final specification document. Once the prototype becomes operational, the actual product is developed using the normal waterfall model. *Figure 1.5* depicts the prototyping model.

The prototype model has the following features:

(i) It helps in determining user requirements more deeply.
(ii) At the time of actual product development, the customer feedback is available.
(iii) It does consider any types of risks at the initial level.

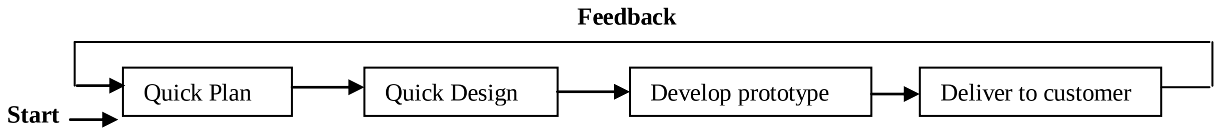

Figure 1.5: Prototyping model

(v) Spiral Model

This model can be considered as the model, which combines the strengths of various other models. Conventional software development processes do not take uncertainties into account. Important software projects have failed because of unforeseen risks. The other models view the software process as a linear activity whereas this model considers it as a spiral process. This is made by representing the iterative development cycle as an expanding spiral.

The following are the primary activities in this model:

- **Finalising Objective:** The objectives are set for the particular phase of the project.
- **Risk Analysis:** The risks are identified to the extent possible. They are analysed and necessary steps are taken.
- **Development:** Based on the risks that are identified, an SDLC model is selected and is followed.
- **Planning:** At this point, the work done till this time is reviewed. Based on the review, a decision regarding whether to go through the loop of spiral again or not will be decided. If there is need to go, then planning is done accordingly.

In the spiral model, these phases are followed iteratively. *Figure 1.6* depicts the Boehm's Spiral Model (IEEE, 1988).

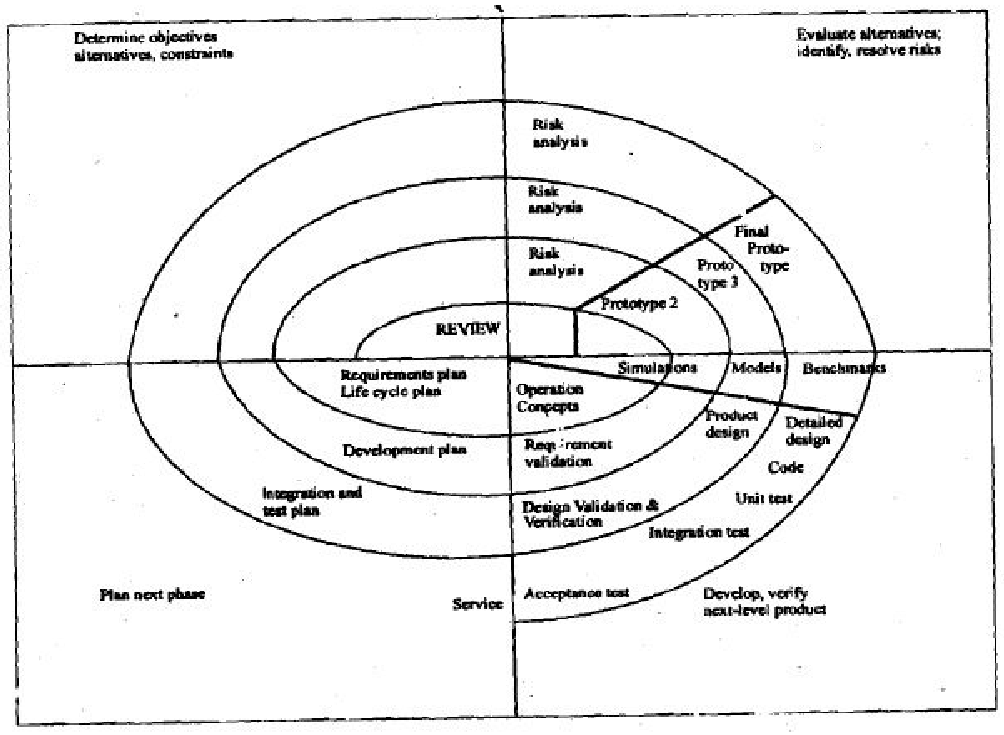

Figure 1.6: Spiral model

In this model Software development starts with lesser requirements specification, lesser risk analysis, etc. The radical dimension this model represents cumulative cost. The angular dimension represents progress made in completing the cycle.

The inner cycles of the spiral model represent early phases of requirements analysis and after prototyping of software, the requirements are refined.

In the spiral model, after each phase a review is performed regarding all products developed upto that point and plans are devised for the next cycle. This model is a realistic approach to the development of large scale software. It suggests a systematic approach according to classical life cycle, but incorporates it into iterative framework. It gives a direct consideration to technical risks. Thus, for high risk projects, this model is very useful. The risk analysis and validation steps eliminate errors in early phases of development.

(vi) RAD Approach

As the name suggests, this model gives a quick approach for software development and is based on a linear sequential flow of various development processes. The software is constructed on a component basis. Thus multiple teams are given the task of different component development. It increases the overall speed of software development. It gives a fully functional system within very short time. This approach emphasises the development of reusable program components. It follows a modular approach for development. The problem with this model is that it may not work when technical risks are high.

☞ Check Your Progress 1

1) Indicate various problems related with software development.

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

2) Give a comparative analysis of various types of software process models.

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3) What are various phases of software development life cycle?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

1.4 CAPABILITY MATURITY MODELS

The process models are based on various software development phases whereas the capability models have an entirely different basis of development. They are based upon the capabilities of software. It was developed by Software Engineering Institute (SEI). In this model, significant emphasis is given to the techniques to improve the “software quality” and “process maturity”. In this model a strategy for improving Software process is devised. It is not concerned which life cycle mode is followed for development. SEI has laid guidelines regarding the capabilities an organisation should have to reach different levels of process maturity. This approach evaluates the global effectiveness of a software company.

1.4.1 Maturity Levels

It defines five maturity levels as described below. Different organisations are certified for different levels based on the processes they follow.

Level 1 (Initial): At this maturity level, software is developed an ad hoc basis and no strategic approach is used for its development. The success of developed software entirely depend upon the skills of the team members. As no sound engineering approach is followed, the time and cost of the project are not critical issues. In Maturity Level 1 organisations, the software process is unpredictable, because if the developing team changes, the process will change. The testing of software is also very simple and accurate predictions regarding software quality are not possible. SEI's assessment indicates that the vast majority of software organisations are Level 1 organisations.

Level 2 (Repeatable): The organisation satisfies all the requirements of level-1. At this level, basic project management policies and related procedures are established. The institutions achieving this maturity level learn with experience of earlier projects and reutilise the successful practices in on-going projects. The effective process can be characterised as practised, documented, implemented and trained. In this maturity level, the manager provides quick solutions to the problem encountered in software development and corrective action is immediately taken. Hence, the process of development is much disciplined in this maturity level. Thus, without measurement, sufficiently realistic estimates regarding cost, schedules and functionality are performed. The organisations of this maturity level have installed basic management controls.

Level 3 (Defined): The organisation satisfies all the requirements of level-2. At this maturity level, the software development processes are well defined, managed and documented. Training is imparted to staff to gain the required knowledge. The standard practices are simply tailored to create new projects.

Level 4 (Managed): The organisation satisfies all the requirements of level-3. At this level quantitative standards are set for software products and processes. The project analysis is done at integrated organisational level and collective database is created. The performance is measured at integrated organisation level. The Software development is performed with well defined instruments. The organisation's capability at Level 4 is "predictable" because projects control their products and processes to ensure their performance within quantitatively specified limits. The quality of software is high.

Level 5 (Optimising): The organisation satisfies all the requirements of level-4. This is last level. The organisation at this maturity level is considered almost perfect. At this level, the entire organisation continuously works for process improvement with the help of quantitative feedback obtained from lower level. The organisation analyses its weakness and takes required corrective steps proactively to prevent the errors. Based on the cost benefit analysis of new technologies, the organisation changes their Software development processes.

1.4.2 Key Process Areas

The SEI has associated key process areas (KPAs) with each maturity level. The KPA is an indicative measurement of goodness of software engineering functions like project planning, requirements management, etc. The KPA consists of the following parameters:

Goals: Objectives to be achieved.

Commitments: The requirements that the organisation should meet to ensure the claimed quality of product.

Abilities : The capabilities an organisation has.

Activities : The specific tasks required to achieve KPA function.

Methods for varying implementation: It explains how the KPAs can be verified.

18 KPAs are defined by SEI and associated with different maturity levels. These are described below:

Level 1 KPAs : There is no key process area at Level 1.

Level 2 KPAs:

1) **Software Project Planning:** Gives concrete plans for software management.

2) **Software Project Tracing & Oversight:** Establish adequate visibility into actual process to enable the organisation to take immediate corrective steps if Software performance deviates from plans.

3) **Requirements Management:** The requirements are well specified to develop a contract between developer and customer.

4) **Software Subcontract Management:** Select qualified software subcontractors and manage them effectively.

5) **Software Quality Assurance (SQA):** To Assure the quality of developed product.

6) **Software Configuration Management (SCM):** Establish & maintain integrity through out the lifecycle of project.

Level 3 KPAs:

1) **Organisation Process Focus (OPF)**: The organisations responsibility is fixed for software process activities that improve the ultimate software process capability.

2) **Training Program (TP)**: It imparts training to develop the skills and knowledge of organisation staff.

3) **Organisation Process Definition (OPD)**: It develops a workable set of software process to enhance cumulative long term benefit of organisation by improving process performance.

4) **Integrated Software Management (ISM)**: The software management and software engineering activities are defined and a tailor made standard and software process suiting to organisations requirements is developed.

5) **Software Product Engineering (SPE)**: Well defined software engineering activities are integrated to produce correct, consistent software products effectively and efficiently.

6) **Inter group co-ordination (IC)**: To satisfy the customer's needs effectively and efficiently, Software engineering groups are created. These groups participate actively with other groups.

7) **Peer reviews (PR)**: They remove defects from software engineering work products.

Level 4 KPAs:

1) **Quantitative Process Management (QP)**: It defines quantitative standards for software process.

2) **Software Quality Management (SQM)**: It develops quantitative understanding of the quality of Software products and achieves specific quality goals.

Level 5 KPAs:

1) **Defect Prevention (DP)**: It discovers the causes of defects and devises the techniques which prevent them from recurring.

2) **Technology Change Management (TCM)**: It continuously upgrades itself according to new tools, methods and processes.

3) **Process Change Management (PCM)**: It continually improves the software processes used in organisation to improve software quality, increase productivity and decrease cycle time for product development.

☞ **Check Your Progress 2**

1) What are different levels of capability maturity model?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What is the level of CMM with which no KPA is associated?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

1.5 SOFTWARE PROCESS TECHNOLOGY

The software industry considers software development as a process. According to Booch and Rumbaugh, “A process defines who is doing what, when and how to reach a certain goal?” Software engineering is a field which combines process, methods and tools for the development of software. The concept of process is the main step in the software engineering approach. Thus, a software process is a set of activities. When those activities are performed in specific sequence in accordance with ordering constraints, the desired results are produced. A software development project requires two types of activities viz., development and project management activities. These activities together comprise of a software process. As various activities are being performed in software process, these activities are categorized into groups called phases. Each phase performs well defined activities.

The various steps (called phases) which are adopted in the development of this process are collectively termed as Software Development Life Cycle (SDLC). The various phases of SDLC are discussed below. Normally, these phases are performed lineally or circularly, but it can be changed according to project as well. The software is also considered as a product and its development as a process. Thus, these phases can be termed as Software Process Technology. In general, different phases of SDLC are defined as following:

- Requirements Analysis
- Design
- Coding
- Software Testing
- Maintenance.

Let us discuss these steps in detail.

Requirements Analysis

Requirements describe the "What" of a system. The objectives which are to be achieved in Software process development are the requirements. In the requirements analysis phase, the requirements are properly defined and noted down. The output of this phase is SRS (Software Requirements Specification) document written in natural language. According to IEEE, requirements analysis may be defined as (1) the process of studying user's needs to arrive at a definition of system hardware and software requirements (2) the process of studying and refining system hardware or software requirements.

Design

In this phase, a logical system is built which fulfils the given requirements. Design phase of software development deals with transforming the customer's requirements into a logically working system. Normally, design is performed in the following two steps:

i) **Primary Design Phase:** In this phase, the system is designed at block level. The blocks are created on the basis of analysis done in the problem identification phase. Different blocks are created for different functions emphasis is put on minimising the information flow between blocks. Thus, all activities which require more interaction are kept in one block.

ii) **Secondary Design Phase:** In the secondary design phase the detailed design of every block is performed.

The input to the design phase is the Software Requirements Specification (SRS) document and the output is the Software Design Document (SDD). The general tasks involved in the design process are the following:

i) Design various blocks for overall system processes.

ii) Design smaller, compact, and workable modules in each block.

iii) Design various database structures.

iv) Specify details of programs to achieve desired functionality.

v) Design the form of inputs, and outputs of the system.

vi) Perform documentation of the design.

vii) System reviews.

The Software design is the core of the software engineering process and the first of three important technical activities, viz., design, coding, and testing that are required to build software. The design should be done keeping the following points in mind.

i) It should completely and correctly describe the system.

ii) It should precisely describe the system. It should be understandable to the software developer.

iii) It should be done at the right level.

iv) It should be maintainable.

The following points should be kept in mind while performing the design:

i) **Practicality:** This ensures that the system is stable and can be operated by a person of average intelligence.

ii) **Efficiency:** This involves accuracy, timeliness and comprehensiveness of system output.

iii) **Flexibility:** The system could be modifiable depending upon changing needs of the user. Such amendments should be possible with minimum changes.

iv) **Security:** This is an important aspect of design and should cover areas of hardware reliability, fall back procedures, security of data and provision for detection of fraud.

Coding

The input to the coding phase is the SDD document. In this phase, the design document is coded according to the module specification. This phase transforms the SDD document into a high level language code. At present major software companies adhere to some well specified and standard style of coding called coding standards. Good coding standards improve the understanding of code. Once a module is developed, a check is carried out to ensure that coding standards are followed. Coding standards generally give the guidelines about the following:

i) Name of the module

ii) Internal and External documentation of source code

iii) Modification history

iv) Uniform appearance of codes.

Testing

Testing is the process of running the software on manually created inputs with the intention to find errors. In the process of testing, an attempt is made to detect errors, to correct the errors in order to develop error free software. The testing is performed keeping the user's requirements in mind and before the software is actually launched on a real system, it is tested. Testing is the process of executing a program with the intention of finding errors.

Normally, while developing the code, the software developer also carries out some testing. This is known as debugging. This unearths the defects that must be removed from the program. Testing and debugging are different processes. Testing is meant for finding the existence of defects while debugging stands for locating the place of errors and correcting the errors during the process of testing. The following are some guidelines for testing:

i) Test the modules thoroughly, cover all the access paths, generate enough data to cover all the access paths arising from conditions.

ii) Test the modules by deliberately passing wrong data.

iii) Specifically create data for conditional statements. Enter data in test file which would satisfy the condition and again test the script.

iv) Test for locking by invoking multiple concurrent processes.

The following objectives are to be kept in mind while performing testing:

i) It should be done with the intention of finding the errors.

ii) Good test cases should be designed which have a probability of finding, as yet undiscovered error.

iii) A success test is one that uncovers yet undiscovered error(s).

The following are some of the principles of testing:

i) All tests should be performed according to user requirements.

ii) Planning of tests should be done long before testing.

iii) Starting with a small test, it should proceed towards large tests.

The following are different levels of testing:

Large systems are built out of subsystems, subsystems are made up of modules, modules of procedures and functions. Thus in large systems, the testing is performed at various levels, like unit level testing, module level testing, subsystem level, and system level testing.

Thus, testing is performed at the following levels. In all levels, the testing are performed to check interface integrity, information content, performance.

The following are some of the strategies of testing: This involves design of test cases. Test case is set of designed data for which the system is tested. Two testing strategies are present.

i) **Code Testing:** The code testing strategy examines the logic of the system. In this, the analyst develops test cases for every instruction in the code. All the paths in the program are tested. This test does not guarantee against software failures. Also, it does not indicate whether the code is according to requirements or not.

ii) **Specification Testing:** In this, testing with specific cases is performed. The test cases are developed for each condition or combination of conditions and submitted for processing.

The objective of testing is to design test cases that systematically uncover different classes of errors and do so with the minimum amount of time and effort. Testing cannot show the absence of errors. It can only find the presence of errors. The test case design is as challenging as software development. Still, however effective the design is, it cannot remove 100% errors. Even, the best quality software are not 100 % error free. The reliability of software is closely dependent on testing.

Some testing techniques are the black box and the white box methods.

White box testing: This method, also known as glass box testing, is performed early in the testing process. Using this, the software engineer can derive a tests that guarantees that all independent paths within the module have been exercised at least once. It has the following features:

i) Exercise all logical decisions on their true and false sides.

ii) Execute all loops at their boundaries and within their operational bounds.

iii) Exercise internal data structures to assure their validity.

Black box testing: This is applied during the later stage of testing. It enables the software developer to derive a set of input conditions that will fully exercise the functional requirements of a program. It enables him to find errors like incorrect or missing functions, interface errors, data structures or external data base access errors and performance errors etc.

Maintenance

Maintenance in the normal sense means correcting the problems caused by wear and tear, but software maintenance is different. Software is either wrong in the beginning or later as some additional requirements have been added. Software maintenance is done because of the following factors.

i) To rectify the errors which are encountered during the operation of software.

ii) To change the program function to interface with new hardware or software.

iii) To change the program according to increased requirements.

There are three categories of maintenance:

i) Corrective Maintenance

ii) Adaptive Maintenance

iii) Perfective Maintenance

Software maintenance is a very broad activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities, and optimisation [STEP 78]. Hence, once the software becomes operational, whatever changes are done are termed as maintenance. As the software requirement change continuously, maintenance becomes a continuous process. In practice, the cost of software maintenance is far more than the cost of software development. It accounts for 50% to 80% of the total system development costs. The Software maintenance has the following problems:

i) It is very cumbersome to analyse and understand the code written by somebody.

ii) No standards for maintenance have been developed and the area is relatively unexplored area.

iii) Few tools and techniques are available for maintenance.

iv) It is viewed as a necessary evil and delegated to junior programmers.

The various phases of the software development life cycle are tightly coupled, and the output of one phase governs the activity of the subsequent phase. Thus, all the phases need to be carefully planned and managed and their interaction requires close monitoring. The project management becomes critical in larger systems.

1.6 SUMMARY

Software engineering covers the entire range of activities used to develop software. The activities include requirements analysis, program development using some recognised approach like structured programming, testing techniques, quality assurance, management and implementation and maintenance. Further, software engineering expects to address problems which are encountered during software development.

1.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The following are major problems related with software development:

   - There may be multiple models suitable to do project. How to decide the best one?
   - The understanding of user requirements may be incomplete.
   - The problem of less documentation or minimal structured techniques may be there.
   - The last minute changes in implementation issues give rise to problems elated with hardware.

2) Out of all SDLC models, the most popular one is waterfall model. But, it insists on having a complete set of requirements before commencement of design. It is often difficult for the customer to state all requirements easily. The iterative enhancement model, though better than waterfall model, in customised software development where the client has to provide and approve the specification, it may lead to time delays for software development. Prototype model provides better understanding of customer’s needs and is useful for large systems, but, in this, the use of inefficient inaccurate or dummy functions may produce undesired results. The spiral model accommodates good features of other models. In this, risk analysis and validation steps eliminate errors in early phases of development. But, in this model, there is a lack of explicit process guidance in determining objectives.

3) Various phases of software development life cycle are:

   - Requirement analysis
   - Design
   - Coding
   - Testing
   - Maintenance

Check Your Progress 2

1) Initial, Repeatable, Defined, Managed, Optimizable

2) Level-1

1.8 FURTHER READINGS

1) *Software Engineering, Sixth Edition, 2001*, Ian Sommerville; Pearson Education.

2) *Software Engineering – A Practitioner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.

Reference websites

http://www.rspa.com
http://www.ieee.org
http://standards.ieee.org

UNIT 2 PRINCIPLES OF SOFTWARE REQUIREMENTS ANALYSIS

Structure **Page Nos.**

2.0 INTRODUCTION

In the design of software, the first step is to decide about the objectives of software. This is the most difficult aspect of software design. These objectives, which the software is supposed to fulfill are called *requirements*.

The IEEE [IEEE Std 610.12] definition of requirement is:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system component, to satisfy a contract formally imposed document.
3. A documented representation of a condition or capability as in (1) or (2).

Thus, requirements specify "what the system is supposed to do?" These requirements are taken from the user. Defining the requirements is most elementary & most difficult part of system design, because, at this level, sometimes, the user himself is not clear about it. Many software projects have failed due to certain requirements specification issues. Thus, overall quality of software product is dependent on this aspect. Identifying, defining, and analysing the requirements is known as requirements analysis. Requirements analysis includes the following activities:

1. Identification of end user's need.
2. Preparation of a corresponding document called SRS (Software Requirements Specification).

3. Analysis and validation of the requirements document to ensure consistency, completeness and feasibility.

4. Identification of further requirements during the analysis of mentioned requirements.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- understand the significance of requirements analysis;
- develop SRS, and
- know various tools and techniques used for requirements analysis.

2.2 ENGINEERING THE PRODUCT

Due to the complexity involved in software development, the engineering approach is being used in software design. Use of engineering approach in the area of requirements analysis evolved the field of Requirements Engineering.

2.2.1 Requirements Engineering

Requirements Engineering is the systematic use of proven principles, techniques and language tools for the cost effective analysis, documentation, on-going evaluation of user's needs and the specification of external behaviour of a system to satisfy those user needs. It can be defined as a discipline, which addresses requirements of objects all along a system development process.

The output of requirements of engineering process is Requirements Definition Description (RDD). Requirements Engineering may be defined in the context of Software Engineering. It divides the Requirements Engineering into two categories. First is the requirements definition and the second is requirements management.

Requirements definition consists of the following processes:

1. Requirements gathering.
2. Requirements analysis and modeling.
3. Creation of RDD and SRS.
4. Review and validation of SRS as well as obtaining confirmation from user.

Requirements management consists of the following processes:

1. Identifying controls and tracking requirements.
2. Checking complete implementation of RDD.
3. Manage changes in requirements which are identified later.

2.2.2 Types of Requirements

There are various categories of the requirements.

On the basis of their priority, the requirements are classified into the following three types:

1. Those that should be absolutely met.
2. Those that are highly desirable but not necessary.
3. Those that are possible but could be eliminated.

On the basis of their functionality, the requirements are classified into the following two types:

i) **Functional requirements**: They define the factors like, I/O formats, storage structure, computational capabilities, timing and synchronization.

ii) **Non-functional requirements**: They define the properties or qualities of a product including usability, efficiency, performance, space, reliability, portability etc.

2.2.3 Software Requirements Specification (SRS)

This document is generated as output of requirement analysis. The requirement analysis involves obtaining a clear and thorough understanding of the product to be developed. Thus, SRS should be consistent, correct, unambiguous & complete, document. The developer of the system can prepare SRS after detailed communication with the customer. An SRS clearly defines the following:

- External Interfaces of the system: They identify the information which is to flow '*from and to'* to the system.
- Functional and non-functional requirements of the system. They stand for the finding of run time requirements.
- Design constraints:

The SRS outline is given below:

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 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 Interfaces
   - 3.2 Functional requirements
   - 3.3 Performance requirements
   - 3.4 Logical Database requirements
   - 3.5 Design Constraints
   - 3.6 Software system attributes
   - 3.7 Organising the specific requirements
   - 3.8 Additional Comments
4. Supporting information
   - 4.1 Table of contents and index
   - 4.2 Appendixes

2.2.4 Problems in SRS

There are various features that make requirements analysis difficult. These are discussed below:

1. Complete requirements are difficult to uncover. In recent trends in engineering, the processes are automated and it is practically impossible to understand the complete set of requirements during the commencement of the project itself.
2. Requirements are continuously generated. Defining the complete set of requirements in the starting is difficult. When the system is put under run, the new requirements are obtained and need to be added to the system. But, the project schedules are seldom adjusted to reflect these modifications. Otherwise, the development of software will never commence.
3. The general trends among software developer shows that they have over dependence on CASE tools. Though these tools are good helping agents, over reliance on these Requirements Engineering Tools may create false requirements. Thus, the requirements corresponding to real system should be understood and only a realistic dependence on tools should be made.
4. The software projects are generally given tight project schedules. Pressure is created from customer side to hurriedly complete the project. This normally cuts down the time of requirements analysis phase, which frequently lead to disaster(s).
5. Requirements Engineering is communication intensive. Users and developers have different vocabularies, professional backgrounds and psychology. User writes specifications in natural language and developer usually demands precise and well-specified requirement.
6. In present time, the software development is market driven having high commercial aspect. The software developed should be a general purpose one to satisfy anonymous customer, and then, it is customised to suit a particular application.
7. The resources may not be enough to build software that fulfils all the customer's requirements. It is left to the customer to prioritise the requirements and develop software fulfilling important requirements.

2.2.5 Requirements Gathering Tools

The requirements gathering is an art. The person who gathers requirements should have knowledge of what and when to gather information and by what resources. The requirements are gathered regarding organisation, which include information regarding its policies, objectives, and organisation structure, regarding user staff. It includes the information regarding job function and their personal details, regarding the functions of the organisation including information about work flow, work schedules and working procedure.

The following four tools are primarily used for information gathering:

1. **Record review:** A review of recorded documents of the organisation is performed. Procedures, manuals, forms and books are reviewed to see format and functions of present system. The search time in this technique is more.
2. **On site observation:** In case of real life systems, the actual site visit is performed to get a close look of system. It helps the analyst to detect the problems of existing system.

3. **Interview:** A personal interaction with staff is performed to identify their requirements. It requires experience of arranging the interview, setting the stage, avoiding arguments and evaluating the outcome.

4. **Questionnaire:** It is an effective tool which requires less effort and produces a written document about requirements. It examines a large number of respondents simultaneously and gets customized answers. It gives person sufficient time to answer the queries and give correct answers.

☞ **Check Your Progress 1**

1) Why is it justified to use engineering approach in requirements analysis?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2.3 MODELING THE SYSTEM ARCHITECTURE

Before the actual system design commences, the system architecture is modeled. In this section, we discuss various modeling techniques.

2.3.1 Elementary Modeling Techniques

A model showing bare minimum requirements is called **Essential Model**. It has two components.

1. **Environmental model**: It indicates environment in which system exists. Any big or small system is a sub-system of a larger system. For example, if software is developed for a college, then college will be part of University. If it is developed for University, the University will be part of national educational system. Thus, when the model of the system is made these external interfaces are defined. These interfaces reflect system's relationship with external universe (called environment). The environment of a college system is shown in *Figure 2.1*.

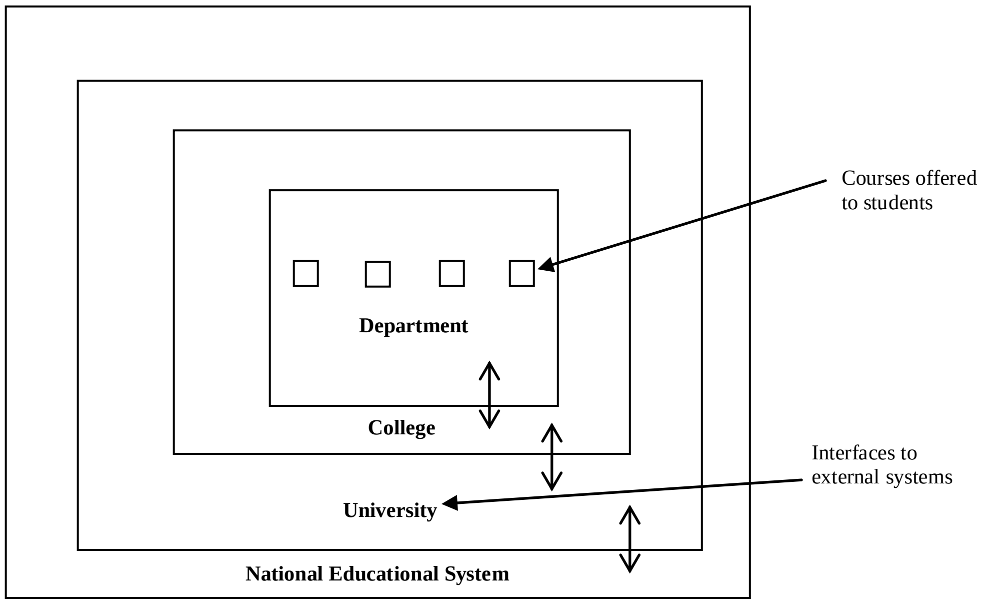
Courses offered to students
Department
College
Interfaces to external systems
University
National Educational System

Figure 2.1 : Environmental model of educational system

In environmental model, the interfaces should clearly indicate the inflow and outflow of information from the system.

The tools of environment model are:

(i) **Statement of purpose**: It indicates the basic objectives of system.

(ii) **Event list**: It describes the different events of system and indicates functionality of the system.

(iii) **Context diagram**: It indicates the environment of various sub-systems.

2. **Behavioural Model:** It describes operational behaviour of the system. In this model, various operations of the system are represented in pictorial form. The tools used to make this model are: Data Flow Diagrams (DFD), E-R diagrams, Data Dictionary & Process Specification. These are discussed in later sections.

   Hence, behavioural model defines:

   Data of proposed system.

   (i) The internal functioning of proposed system,
   (ii) Inter-relationship between various data.

In traditional approach of modeling, the analyst collects great deal of relatively unstructured data through data gathering tools and organize the data through system flow charts which support future development of system and simplify communication with the user. But, flow chart technique develops physical rather than logical system.

In structured approach of modeling the standard techniques of DFD, E-R diagram etc. are used to develop system specification in a formal format. It develops a system logical model.

2.3.2 Data Flow Diagrams (DFD)

It is a graphical representation of flow of data through a system. It pictures a system as a network of functional processes. The basis of DFD is a data flow graph, which pictorially represents transformation on data as shown in *Figure 2.2.*

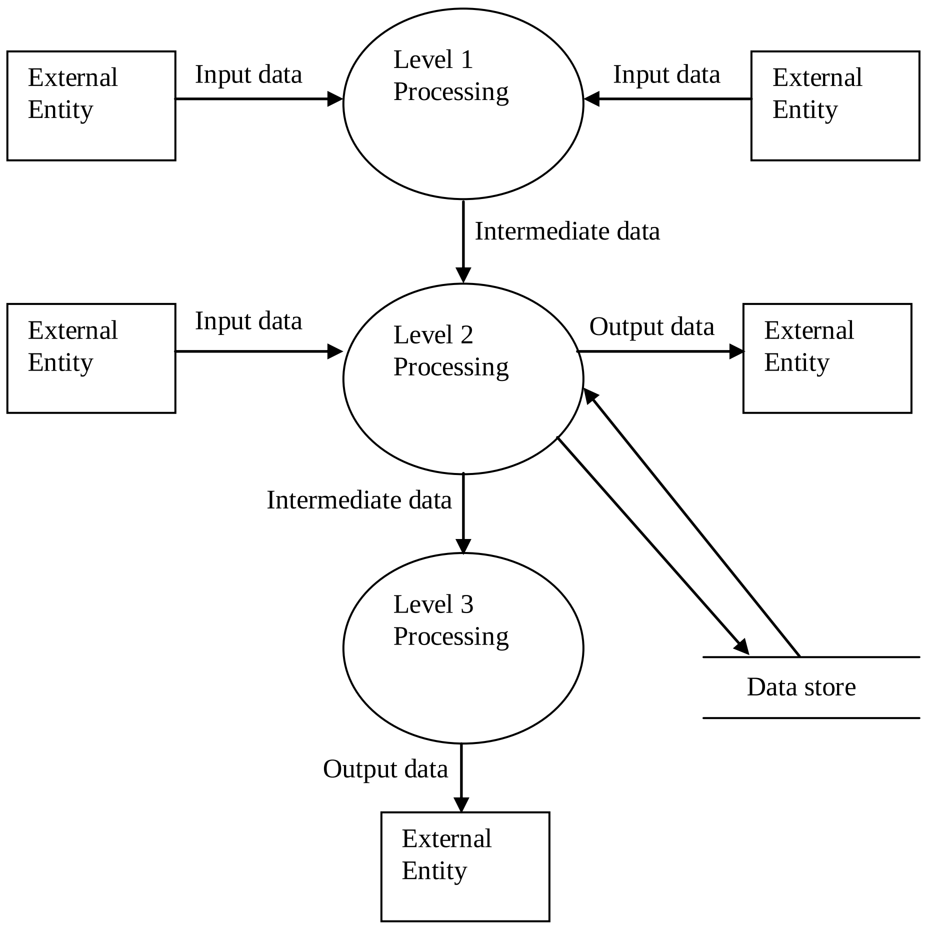

Figure 2.2: Data flow diagram

In this diagram, the external entities provide input data for the processing. During the processing, some intermediate data is generated. After final processing, the final output data is generated. The data store is the repository of data.

The structured approach of system design requires extensive modeling of the system. Thus, instead of making a complete model exhibiting the functionality of system, the DFD's are created in a layered manner. At the first layer, the DFD is made at block level and in lower layers, the details are shown. Thus, level "0" DFD makes a fundamental system (*Figure 2.3*).

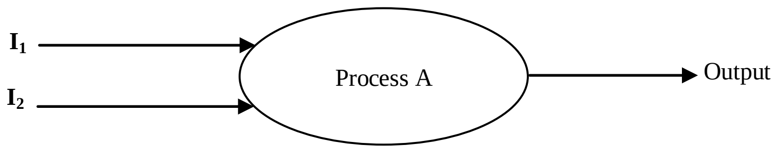

Figure 2.3: Layer 1 depiction of process A

DFD's can represent the system at any level of abstraction. DFD of "0" level views entire software element as a single bubble with indication of only input and output data. Thus, "0" level DFD is also called as Context diagram. Its symbols are shown in *Figure 2.4.*

Symbol	**Name**	**Description**
⟶	Data Flow	Represents the connectivity between various processes
○	Process	Performs some processing of input data
▭	External Entity	Defines source or destination of system data. The entity which receives or supplies information.
═	Data Store	Repository of data

Figure 2.4: Symbols of a data flow diagram

2.3.3 Rules for making DFD

The following factors should be considered while making DFDs:

1. Keep a note of all the processes and external entities. Give unique names to them. Identify the manner in which they interact with each other.

2. Do numbering of processes.

3. Avoid complex DFDs (if possible).

4. The DFD should be internally consistent.

5. Every process should have minimum of one input and one output.

   The data store should contain all the data elements that flow as input and output.

To understand the system functionality, a system model is developed. The developed model is used to analyze the system. The following four factors are of prime concern for system modeling:

1. The system modeling is undertaken with some simplifying assumptions about the system. Though these assumptions limit the quality of system model, it reduces the system complexity and makes understanding easier. Still, the model considers all important, critical and material factors. These assumptions are made regarding all aspects like processes, behaviors, values of inputs etc.

2. The minute details of various components are ignored and a simplified model is developed. For example, there may be various types of data present in the system. The type of data having minute differences are clubbed into single category, thus reducing overall number of data types.

3. The constraints of the system are identified. Some of them may be critical. They are considered in modeling whereas others may be ignored. The constraints may be because of external factors, like processing speed, storage capacity, network features or operating system used.

4. The customers mentioned preferences about technology, tools, interfaces, design, architecture etc. are taken care of.

Example 1: The 0th and 1st levels of DFD of Production Management System are shown in *Figure 2.5 (a) and (b)*

Let us discuss the data flow diagram of Production Management System.

// Copy Figures 2.5 (a), (b)

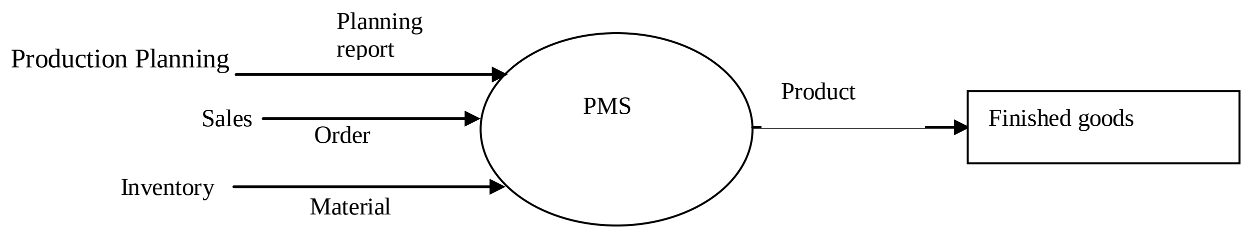

Figure 2.5 (a) : Level 0 DFD of PMS

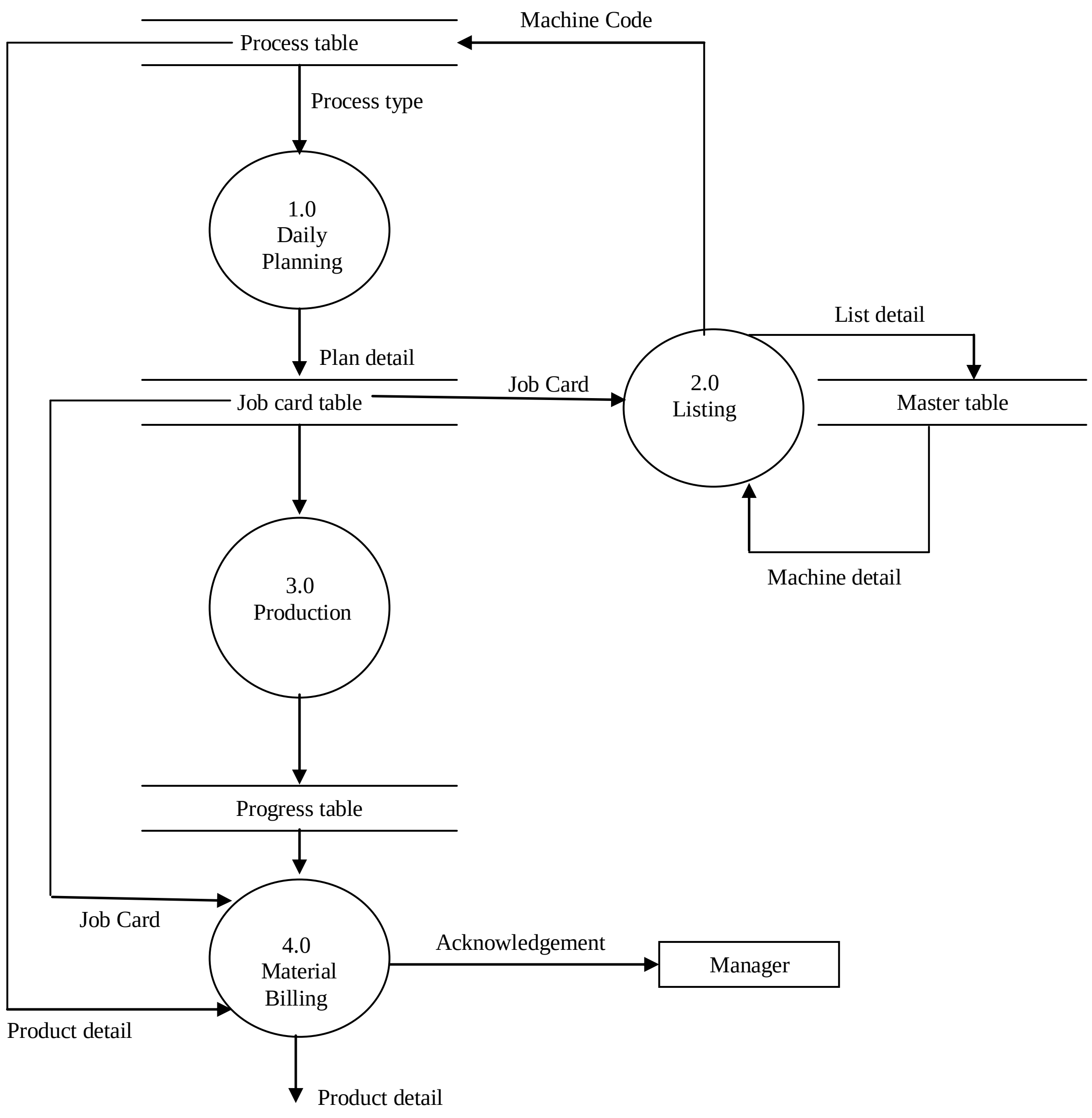

Figure 2.5 (b): Level 1 DFD of PMS

2.3.4 Data Dictionary

This is another tool of requirement analysis which reduces complexity of DFD. A data dictionary is a catalog of all elements of a system. DFD depicts flow of data whereas data dictionary gives details of that information like attribute, type of attribute, size, names of related data items, range of values, data structure definitions etc. The name specifies the name of attribute whose value is collected. For example, fee deposit may be named as FD and course opted may be named as CO.

Related data items captures details of related attributes. Range of values store total possible values of that data. Data structure definition captures the physical structure of data items.

Some of the symbols used in data dictionary are given below:

X= [a/b]	x consists of either data element a or b
X=a	x consists of an optional data element a
X=a+b	x consists of data element a & b.

X=y{a}z	x consists of some occurrences of data elements a which are between y and z.
|	Separator
**	Comments
@	Identifier
( )	Options

Example 2: The data dictionary of payroll may include the following fields:

PAYROLL	=	[Payroll Details]
	=	@ employee name + employee + id number + employee address + Basic Salary + additional allowances
Employee name	=	* name of employee *
Employee id number	=	* Unique identification no. given to every employee*
Basic Salary	=	* Basic pay of employee *
Additional allowances	=	* The other perks on Basic pay *
Employee name	=	First name + Last name
Employee address	=	Address 1 + Address 2 + Address 3

2.3.5 E-R Diagram

Entity-relationship (E-R) diagram is detailed logical representation of data for an organisation. It is data oriented model of a system whereas DFD is a process oriented model. The ER diagram represent data at rest while DFD tracks the motion of data. ERD does not provide any information regarding functionality of data. It has three main components – data entities, their relationships and their associated attributes.

Entity: It is most elementary thing of an organisation about which data is to be maintained. Every entity has unique identity. It is represented by rectangular box with the name of entity written inside (generally in capital letters).

Relationship: Entities are connected to each other by relationships. It indicates how two entities are associated. A diamond notation with name of relationship represents as written inside. Entity types that participate in relationship is called degree of relationship. It can be one to one (or unary), one to many or many to many.

Cardinality & Optionally: The cardinality represents the relationship between two entities. Consider the one to many relationship between two entities – class and student. Here, cardinality of a relationship is the number of instances of entity student that can be associated with each instance of entity class. This is shown in *Figure 2.6.*

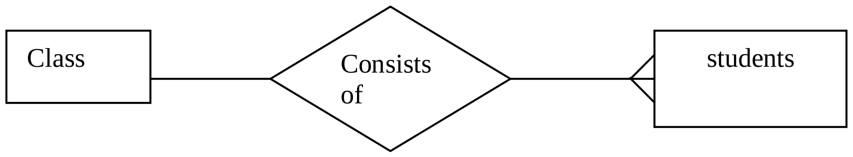

Figure 2.6: One to Many cardinality relationship

The minimum cardinality of a relationship is the minimum number of instances of second entity (student, in this case) with each instance of first entity (class, in this case).

In a situation where there can be no instance of second entity, then, it is called as optional relationship. For example' if a college does not offer a particular course then it will be termed as optional with respect to relationship 'offers'. This relationship is shown in *Figure 2.7.*

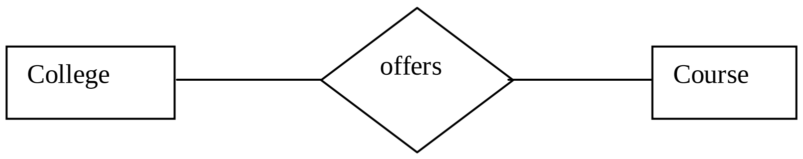

Figure 2.7: Minimum cardinality relationship

When minimum cardinality of a relationship is one, then second entity is called as mandatory participant in the relationship. The maximum cardinality is the maximum number of instances of second entity. Thus, the modified ER diagram is shown in *Figure 2.8.*

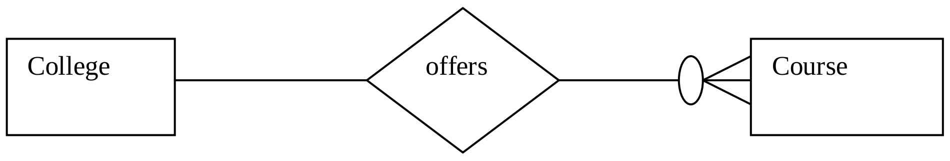

Figure 2.8: Modified ER diagram representing cardinalities

The relationship cardinalities are shown in *Figure 2.9.*

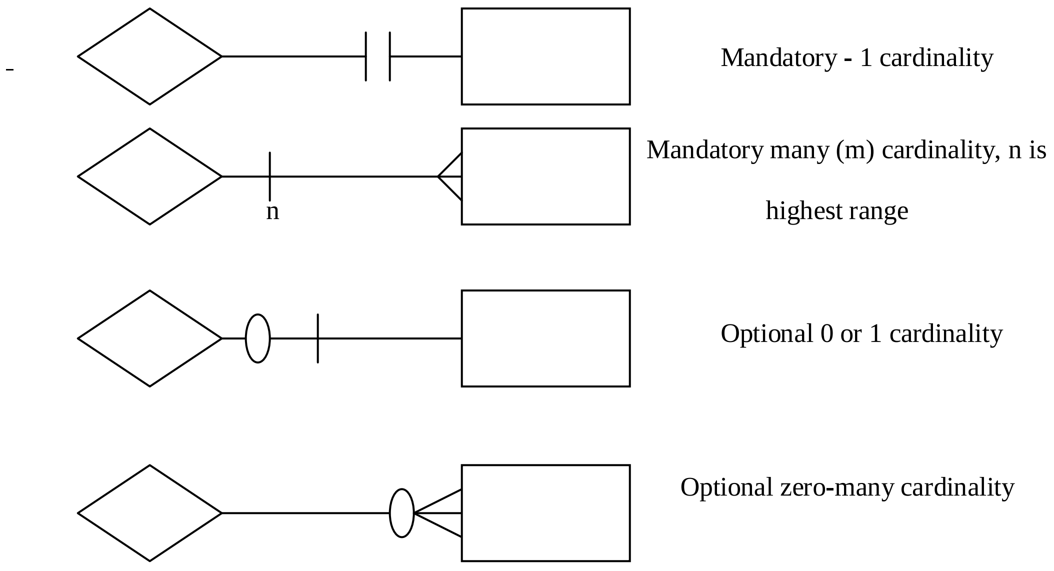
Mandatory - 1 cardinality
n
Mandatory many (m) cardinality, n is highest range
Optional 0 or 1 cardinality
Optional zero-many cardinality

Figure 2.9: Relationship cardinalities

Attributes: Attribute is a property or characteristic of an entity that is of interest to the organisation. It is represented by oval shaped box with name of attribute written inside it. For example, the student entity has attributes as shown in *Figure 2.10.*

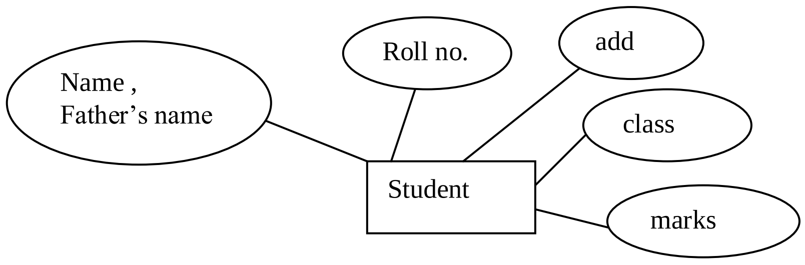

Figure 2.10: Attributes of entity *student*

2.3.6 Structured Requirements Definition

Structured Requirements definition is an approach to perform the study about the complete system and its various sub-systems, the external inputs and outputs, functionality of complete system and its various sub-systems. The following are various steps of it:

Step 1: Make a user level data flow diagram.

This step is meant for gathering requirements with the interaction of employee. In this process, the requirements engineer interviews each individual of organisation in order to learn what each individual gets as input and produces as output. With this gathered data, create separate data flow diagram for every user.

Step 2: Create combined user level data flow diagram.

Create an integrated data flow diagram by merging old data flow diagrams. Remove the inconsistencies if encountered, in this merging process. Finally, an integrated consistent data flow diagram is generated.

Step 3: Generate application level data flow diagram.

Perform data analysis at system's level to define external inputs and outputs.

Step 4: Define various functionalities.

In this step, the functionalities of various sub-systems and the complete system is defined.

☞ Check Your Progress 2

1) What is the advantage of DFD over ER diagram?

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

2) What is the significance specifying functional requirements in SRS document?

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

3) What is the purpose of using software metrics?

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

2.4 SOFTWARE PROTOTYPING AND SPECIFICATION

Prototyping is a process that enables developer to create a small model of software. The IEEE 610.12 standard defines prototype as a preliminary form or instance of a system that serves as a model for later stages for the final complete version of the system.

A prototype may be categorised as follows:

1. A paper prototype, which is a model depicting human machine interaction in a form that makes the user understand how such interaction, will occur.
2. A working prototype implementing a subset of complete features.
3. An existing program that performs all of the desired functions but additional features are added for improvement.

Prototype is developed so that customers, users and developers can learn more about the problem. Thus, prototype serves as a mechanism for identifying software requirements. It allows the user to explore or criticise the proposed system before developing a full scale system.

2.4.1 Types of Prototype

Broadly, the prototypes are developed using the following two techniques:

Throw away prototype: In this technique, the prototype is discarded once its purpose is fulfilled and the final system is built from scratch. The prototype is built quickly to enable the user to rapidly interact with a working system. As the prototype has to be ultimately discarded, the attention on its speed, implementation aspects, maintainability and fault tolerance is not paid. In requirements defining phase, a less refined set of requirements are hurriedly defined and throw away prototype is constructed to determine the feasibility of requirement, validate the utility of function, uncover missing requirements, and establish utility of user interface. The duration of prototype building should be as less as possible because its advantage exists only if results from its use are available in timely fashion.

Evolutionary Prototype: In this, the prototype is constructed to learn about the software problems and their solutions in successive steps. The prototype is initially developed to satisfy few requirements. Then, gradually, the requirements are added in the same prototype leading to the better understanding of software system. The prototype once developed is used again and again. This process is repeated till all requirements are embedded in this and the complete system is evolved.

According to SOMM [96] the benefits of developing prototype are listed below:

1. Communication gap between software developer and clients may be identified.

2. Missing user requirements may be unearthed.

3. Ambiguous user requirements may be depicted.

4. A small working system is quickly built to demonstrate the feasibility and usefulness of the application to management.

5. It serves as basis for writing the specification of the system.

The sequence of prototyping is shown in following *Figure 2.11.*

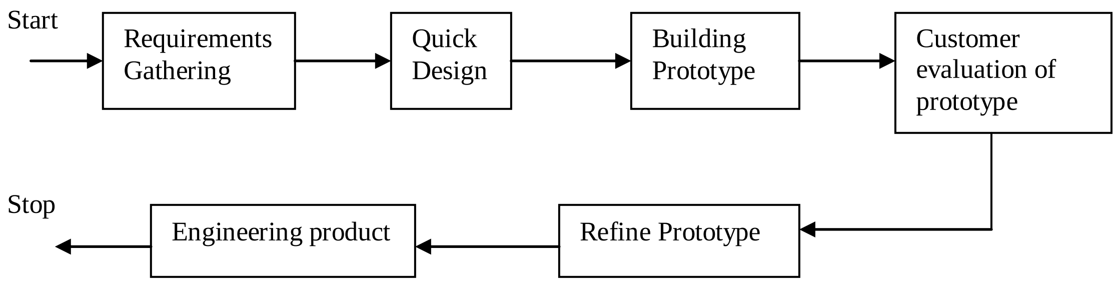

Figure 2.11: Sequence of prototyping

2.4.2 Problems of Prototyping

In some organisations, the prototyping is not as successful as anticipated. A common problem with this approach is that people expect much from insufficient effort. As the requirements are loosely defined, the prototype sometimes gives misleading results about the working of software. Prototyping can have execution inefficiencies and this may be questioned as negative aspect of prototyping. The approach of providing early feedback to user may create the impression on user and user may carry some negative biasing for the completely developed software also.

2.4.3 Advantages of Prototyping

The advantages of prototyping outperform the problems of prototyping. Thus, overall, it is a beneficial approach to develop the prototype. The end user cannot demand fulfilling of incomplete and ambiguous software needs from the developer.

One additional difficulty in adopting this approach is the large investment that exists in software system maintenance. It requires additional planning about the re-engineering of software. Because, it may be possible that by the time the prototype is build and tested, the technology of software development is changed, hence requiring a complete re-engineering of the product.

2.5 SOFTWARE METRICS

Measurement is fundamental to any engineering discipline and software engineering is no exception. Software metric is a quantitative measure derived from the attribute of software development life cycle [Fanton and Kaposi, 1987]. It behaves as software measures.

A software measure is a mapping from a set of objects in the software engineering world into a set of mathematical constructs such as numbers or vectors of numbers.

Using the software metrics, software engineer measures software processes, and the requirements for that process. The software measures are done according to the following parameters:

- The objective of software and problems associated with current activities,
- The cost of software required for relevant planning relative to future projects,
- Testability and maintainability of various processes and products,
- Quality of software attributes like reliability, portability and maintainability,

- Utility of software product,
- User friendliness of a product.

Various characteristics of software measures identified by Basili (1989) are given below:

- **Objects of measurement**: They indicate the products and processes to be measured.
- **Source of measurement**: It indicates who will measure the software. For example, software designer, software tester and software managers.
- **Property of measurement**: It indicates the attribute to be measured like cost of software, reliability, maintainability, size and portability.
- **Context of measurement**: It indicates the environments in which context the software measurements are applied.

Common software measures

There are significant numbers of software measures. The following are a few common software measures:

Size : It indicates the magnitude of software system. It is most commonly used software measure. It is indicative measure of memory requirement, maintenance effort, and development time.

LOC : It represents the number of lines of code (LOC). It is indicative measure of size oriented software measure. There is some standardisation on the methodology of counting of lines. In this, the blank lines and comments are excluded. The multiple statements present in a single line are considered as a single LOC. The lines containing program header and declarations are counted.

☞ Check Your Progress 3

1) Explain rapid prototyping technique.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) List languages used for prototyping.

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

SUMMARY

This unit discusses various aspects of software requirements analysis, the significance of use of engineering approach in the software design, various tools for gathering the requirements and specifications of prototypes.

Due to the complexity involved in software development, the engineering approach is being used in software design. Use of engineering approach in the area of requirements analysis evolved the field of Requirements Engineering. Before the actual system design commences, the system architecture is modeled. Entity-relationship (E-R) diagram is detailed logical representation of data for an organisation. It is data-oriented model of a system whereas DFD is a process oriented model. The ER diagram represent data at rest while DFD tracks the motion of data. ERD does not provide any information regarding functionality of data. It has three main components–data entities, their relationships and their associated attributes. Structured Requirements definition is an approach to perform the study about the complete system and its various subsystems, the external inputs and outputs, functionality of complete system and its various subsystems. Prototyping is a process that enables developer to create a small model of software. The IEEE 610.12 standard defines prototype as a preliminary form or instance of a system that serves as a model for later stages for the final complete version of the system. Measurement is fundamental to any engineering discipline and software engineering is no exception. Software metric is a quantitative measure derived from the attribute of software development life cycle.

2.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) In the present era, the development of software has become very systematic and the specification of requirements is very important. Accordingly, to analyse requirements completely, the engineering approach is used in requirements analysis.

Check Your Progress 2

1) The DFD presents more pictorial representation of flow of data. Moreover, by creating different levels of DFD, the accurate and in depth understanding of data flow is possible.

2) By specifying functional requirements, general understanding about overall working of system is possible.

3) The purpose of using software metrics is to achieve basic objectives of the system with low cost and high quality. Metrics provide scale for quantifying qualities and characteristics of a software. An analogy real life is that meter is the metric of length but to determine the length of cloth, one requires the measuring means like meter-tape etc. Similarly, in software measurement, method must be objective and should produce the same result independent of various measures.

Check Your Progress 3

1) Rapid prototyping techniques are used for speedy prototype development. There are three techniques used for this purpose.

   - Dynamic high level language development
   - Dynamic programming
   - Component and application assembly.

2) High level languages that are used for prototyping are as follows:

   Java, Prolog, Small talk, Lisp etc.

2.8 FURTHER READINGS

1) *Effective Requirements Practices, 2001*, Ralph R. Young; Artech House Technology Management and Professional Development Library.

2) *Software Requirements and Specifications, 1995,* M.Jackson; ACM Press.

3) *Software Architecture in practice, 2003,* Len Bass, Paul Clements, Rick Kazman; Addison-Wesley Professional.

Reference Websites:

http://standards.ieee.org
http://www.rspa.com

UNIT 3 SOFTWARE DESIGN

Structure **Page Nos.**

3.0 INTRODUCTION

Software design is all about developing blue print for designing workable software. The goal of software designer is to develop model that can be translated into software. Unlike design in civil and mechanical engineering, software design is new and evolving discipline contrary classical building design etc. In early days, software development mostly concentrated on writing code. Software design is central to software engineering process. Various design models are developed during design phase. The design models are further refined to develop detailed design models which are closely related to the program.

3. 1 OBJECTIVES

After going through this unit, you should be able to:

- design data;
- understand architectural design;
- develop modular design, and
- know the significance of Human Computer Interface.

3.2 DATA DESIGN

To learn the process of system design involved to translate user requirement to implementable models.

- Designing of Data
- Architectural design which gives a holistic architecture of the software product
- Design of cohesive and loosely coupled module
- Design of interface and tips for good interface design
- Design of Human interface

Software Design Process

Software design is the process of applying various software engineering techniques to develop models to define a software system, which provides sufficient details for actual realisation of the software. The goal of software design is to translate user requirements into an implementable program.

Software design is the only way through which we can translate user requirements to workable software. In contrary to designing a building software design is not a fully developed and matured process. Nevertheless the techniques available provides us tools for a systematic approach to the design of software. *Figure 3.1* depicts the process of software design.

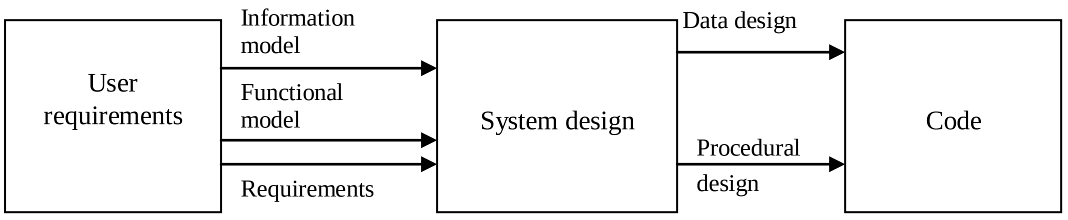

Figure 3.1 : Process of software Design

During the process of software design, the information model is translated to data design. Functional model and behavioural model are translated to architectural design, which defines major component of the software. Keeping in view of the importance of design, it should be given due weightage before rushing to the coding of software.

Software design forms the foundation for implementation of a software system and helps in the maintenance of software in future too. Software quality and software design process are highly interrelated. Quality is built into the software during the design phase.

High level design gives a holistic view of the software to be built, where as low level refinements of the design are very closely related to the final source code. A good design can make the work of programmer easy and hardly allow the programmer to forget the required details. Sufficient time should be devoted to design process to ensure good quality software.

The following are some of the fundamentals of design:

- The design should follow a hierarchical organisation.
- Design should be modular which are logically partitioned into modules which are relatively independent and perform independent task.
- Design leading to interface that facilitate interaction with external environment.
- Step wise refinement to more detailed design which provides necessary details for the developer of code.
- Modularity is encouraged to facilitate parallel development, but, at the same time, too many modules lead to the increase of effort involved in integrating the modules.

Data design is the first and the foremost activity of system Design. Before going into the details of data design, let us discuss what is data? Data describes a real-world information resource that is important for the application. Data describes various entities like customer, people, asset, student records etc.

Identifying data in system design is an iterative process. At the highest level, data is defined in a very vague manner. A high level design describes how the application handles these information resources. As we go into more details, we focus more on the types of data and it’s properties. As you keep expanding the application to the business needs or business processes, we tend to focus more on the details.

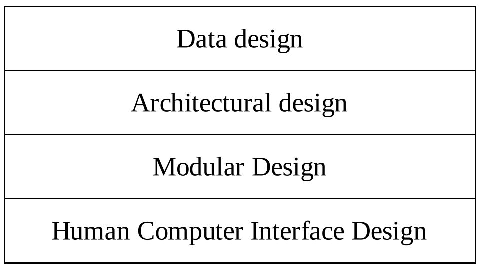

Figure 3.2 : Technical aspects of design

The primary objective of data design is to select logical representation of data items identified in requirement analysis phase. *Figure 3.2* depicts the technical aspects of design. As we begin documenting the data requirements for the application, the description for each item of data typically include the following:

- Name of the data item
- General description of the data item
- Characteristics of data item
- Ownership of the data item
- Logical events, processes, and relationships.

Data Sructure

A data structure defines a logical relationship between individual elements of a group of data. We must understand the fact that the structure of information affects the design of procedures/algorithms. Proper selection of data structure is very important in data designing. Simple data structure like a scalar item forms the building block of more sophisticated and complex data structures.

A scalar item is the simplest form of data. For example, January (Name of the Month), where as the collection of months in a year form a data structure called a vector item.

Example:

Month as string

months_in_year = [Jan, …, Dec]

The items in the vector called array is sequenced and index in a manner so as to retive particular element in the arry.

Month_in_year[4] will retrieve Apr

The vector items are in contiguous memory locations in the computer memory. There are other variety of lists which are non-contiguously stored. These are called linked lists. There are other types of data structures known as hierarchical data structure such as tree. Trees are implemented through linked lists.

Check Your Progress 1

1) Quality is built into the software during the design phase explain?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) Structure of information effects the design of algorithm. True ☐ False ☐

3) Design form the foundation of Software Engineering, Explain?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3.3 ARCHITECTURAL DESIGN

The objective of architectural design is to develop a model of software architecture which gives a overall organisation of program module in the software product. Software architecture encompasses two aspects of structure of the data and hierarchical structure of the software components. Let us see how a single problem can be translated to a collection of solution domains (refer to *Figure 3.3*).

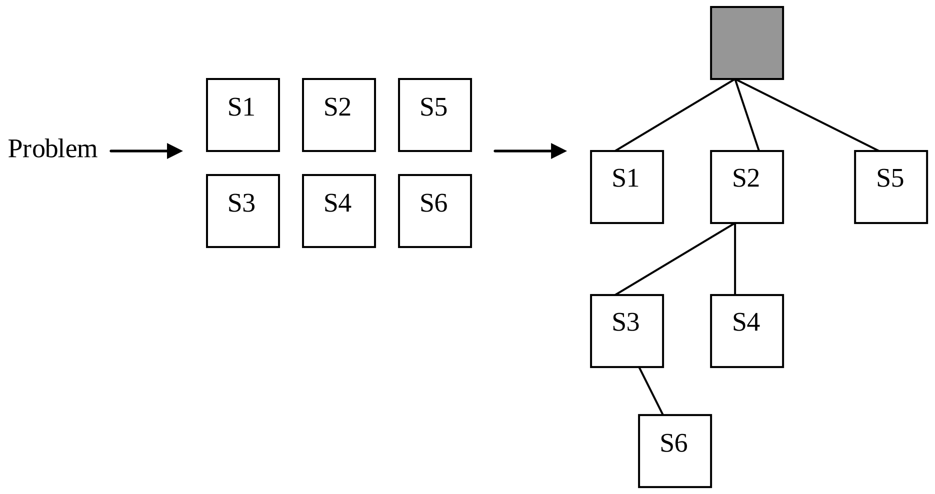

Figure 3.3: Problem, Solutions and Architecture

Architectural design defines organisation of program components. It does not provide the details of each components and its implementation. *Figure 3.4* depicts the architecture of a Financial Accounting System.

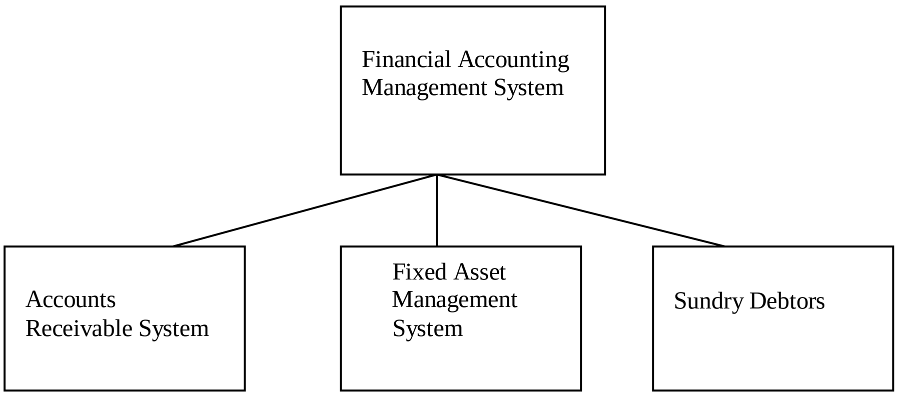
Financial Accounting Management System
Accounts Receivable System
Fixed Asset Management System
Sundry Debtors

Figure 3.4: Architecture of a financial accounting system

The objective of architectural design is also to control relationship between modules. One module may control another module or may be controlled by another module. These characteristics are defined by the fan-in and fan-out of a particular module. The organisation of module can be represented through a tree like structure.

Let us consider the following architecture of a software system.

The number of level of component in the structure is called depth and the number of component across the horizontal section is called width. The number of components which controls a said component is called fan-in i.e., the number of incoming edges to a component. The number of components that are controlled by the module is called fan-out i.e., the number of outgoing edges.

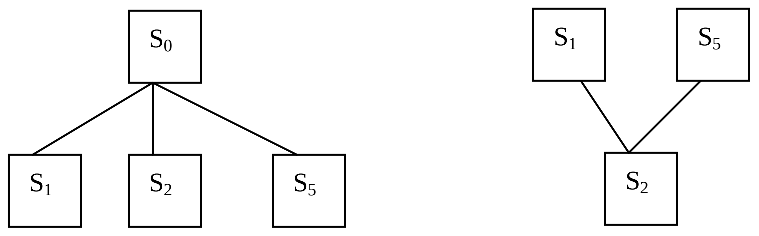

Figure 3.5: Fan-in and Fan-out

S_0 controls three components, hence the fan-out is 3. S_2 is controlled by two components, namely, S_1 and S_2, hence the fan-in is 2 (refer to *Figure 3.5*).

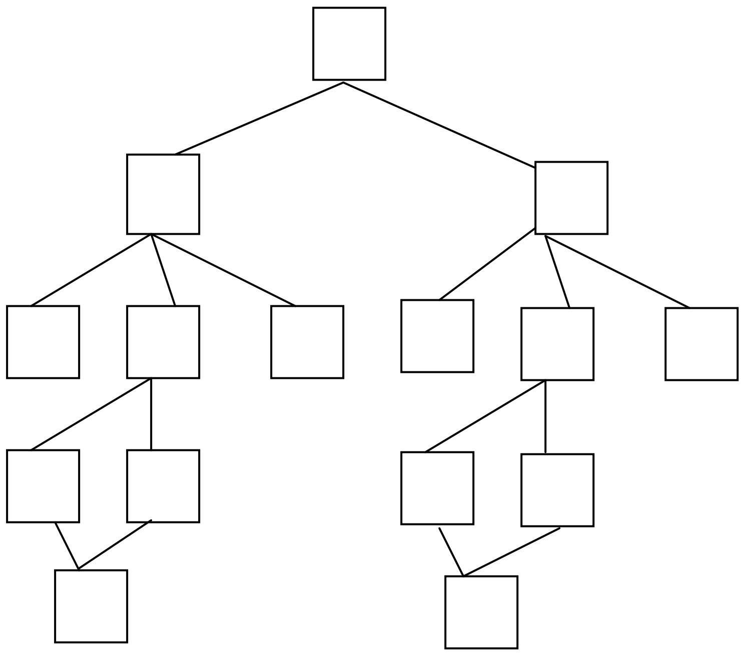

Figure 3.6 : Typical architecture of a system

The architectural design provides holistic picture of the software architecture and connectivity between different program components (refer to *Figure 3.6*).

3.4 MODULAR DESIGN

Modular design facilitates future maintenance of software. Modular design have become well accepted approach to built software product.

Software can be divided in to relatively independent, named and addressable component called a module. Modularity is the only attribute of a software product that makes it manageable and maintainable. The concept of modular approach has been derived from the fundamental concept of "divide and conquer". Dividing the software to modules helps software developer to work on independent modules that can be later integrated to build the final product. In a sense, it encourages parallel development effort thus saving time and cost.

During designing of software one must keep a balance between number of modules and integration cost. Although, more numbers of modules make the software product more manageable and maintainable at the same time, as the number of modules increases, the cost of integrating the modules also increases.

Modules are generally activated by a reference or through a external system interrupt. Activation starts life of an module. A module can be classified three types depending activation mechanism.

- An incremental module is activated by an interruption and can be interrupted by another interrupt during the execution prior to completion.

- A sequential module is a module that is referenced by another module and without interruption of any external software.

- Parallel module are executed in parallel with another module

Sequential module is most common type of software module. While modularity is good for software quality, independence between various module are even better for software quality and manageability. Independence is a measured by two parameters called cohesion and coupling.

Cohesion is a measure of functional strength of a software module. This is the degree of interaction between statements in a module. Highly cohesive system requires little interaction with external module.

Coupling is a measure of interdependence between/among modules. This is a measure of degree of interaction between module i.e., their inter relationship.

Hence, highly cohesive modules are desirable. But, highly coupled modules are undesirable (refer to *Figure 3.7* and *Figure 3.8*).

Cohesion

Cohesiveness measure functional relationship of elements in a module. An element could be a instruction, a group of instructions, data definitions or reference to another module.

Cohesion tells us how efficiently we have positioned our system to modules. It may be noted that modules with good cohesion requires minimum coupling with other module.

Low cohesion	High cohesion
Undesirable	Desirable

Figure 3.7 : Cohesion

There are several types of cohesion arranged from bad to best.

- **Coincidental :** This is worst form of cohesion, where a module performs a number of unrelated task.
- **Logical :** Modules perform series of action, but are selected by a calling module.

- **Procedural :** Modules perform a series of steps. The elements in the module must takeup single control sequence and must be executed in a specific order.

- **Communicational :** All elements in the module is executed on the same data set and produces same output data set.

- **Sequential :** Output from one element is input to some other element in the module. Such modules operates on same data structure.

- **Functional :** Modules contain elements that perform exactly one function.

  The following are the disadvantages of low cohesion:

  - Difficult to maintain
  - Tends to depend on other module to perform certain tasks
  - Difficult to understand.

Coupling

In computer science, coupling is defined as degree to which a module interacts and communicates with another module to perform certain task. If one module relies on another the coupling is said to be high. Low level of coupling means a module doses not have to get concerned with the internal details of another module and interact with another module only with a suitable interface.

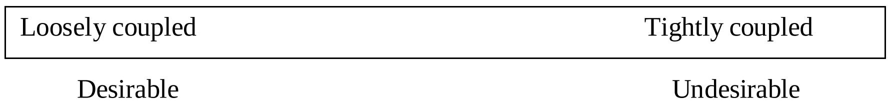

Figure 3.8 : Coupling

The types of coupling from best (lowest level of coupling) to worst (high level of coupling) are described below:

- **Data coupling:** Modules interact through parameters.
  Module X passes parameter A to module Y

- **Stamp coupling:** Modules shares composite data structure.

- **Control coupling:** One module control logic flow of another module. For example, passing a flag to another module which determines the sequence of action to be performed in the other module depending on the value of flag such as true or false.

- **External coupling:** Modules shares external data format. Mostly used in communication protocols and device interfaces

- **Common coupling:** Modules shares the same global data.

- **Content coupling:** One module modifies the data of another module.

Coupling and cohesion are in contrast to each other. High cohesion often correlates with low coupling and vice versa.

In computer science, we strive for low-coupling and high cohesive modules.
The following are the disadvantages of high coupling:

- Ripple effect.
- Difficult to reuse. Dependent modules must be included.
- Difficult to understand the function of a module in isolation.

Figure 3.9 depicts highly cohesive and loosely coupled system. *Figure 3.10* depicts a low cohesive and tightly coupled system.

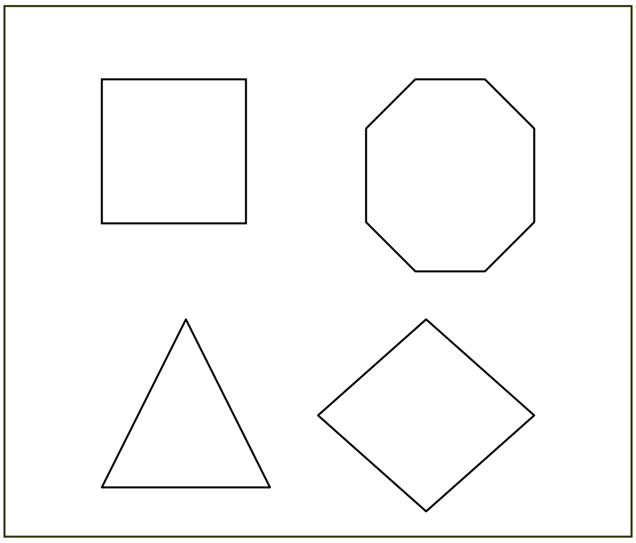

Figure 3.9 : High cohesive and loosely coupled system (desirable)

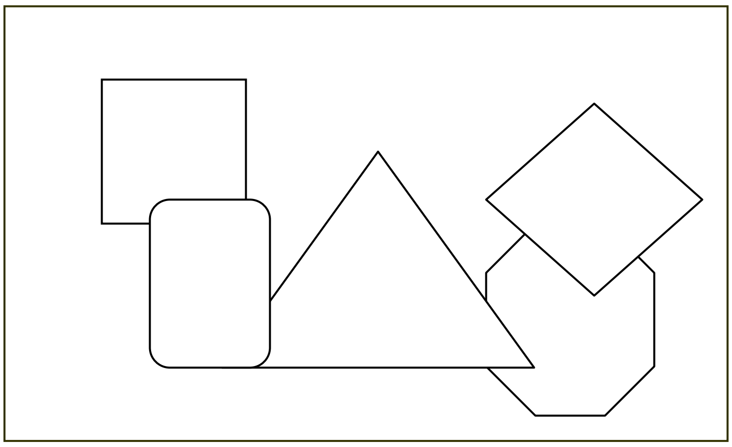

Figure 3.10 : Low cohesive and tightly coupled system (undesirable)

☞ **Check Your Progress 2**

1) Content coupling is a low level coupling   True ☐ False ☐

2) Modules sharing global data refer to ____________.

3) Which is the best form of cohesion?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3.5 INTERFACE DESIGN

Interface design deals with the personnel issues of the software. Interface design is one of the most important part of software design. It is crucial in a sense that user interaction with the system takes place through various interfaces provided by the software product.

Think of the days of text based system where user had to type command on the command line to execute a simple task.

Example of a command line interface:

- run prog1.exe /i=2 message=on

The above command line interface executes a program prog1.exe with a input i=2 with message during execution set to on. Although such command line interface gives liberty to the user to run a program with a concise command. It is difficult for a novice user and is error prone. This also requires the user to remember the command for executing various commands with various details of options as shown above.

Example of Menu with option being asked from the user (refer to Figure 3.11).

To run the program select the option below

1. Option 1
2. Option 2
3. Option 3
4. Back
5. Exit program

Enter your option ____

Figure 3.11 : A menu based interface

This simple menu allow the user to execute the program with option available as a selection and further have option for exiting the program and going back to previous screen. Although it provide grater flexibility than command line option and does not need the user to remember the command still user can't navigate to the desired option from this screen. At best user can go back to the previous screen to select a different option.

Modern graphical user interface provides tools for easy navigation and interactivity to the user to perform different tasks.

The following are the advantages of a Graphical User Interface (GUI):

- Various information can be display and allow user to switch to different task directly from the present screen.
- Useful graphical icons and pull down menu reduces typing effort by the user.
- Provides key-board shortcut to perform frequently performed tasks.
- Simultaneous operations of various task without loosing the present context.

Any interface design is targeted to users of different categories.

- Expert user with adequate knowledge of the system and application
- Average user with reasonable knowledge
- Novice user with little or no knowledge.

The following are the elements of good interface design:

- Goal and the intension of task must be identified.

- The important thing about designing interfaces is all about maintaining consistency. Use of consistent color scheme, message and terminologies helps.
- Develop standards for good interface design and stick to it.
- Use icons where ever possible to provide appropriate message.
- Allow user to undo the current command. This helps in undoing mistake committed by the user.
- Provide context sensitive help to guide the user.
- Use proper navigational scheme for easy navigation within the application.
- Discuss with the current user to improve the interface.
- Think from user prospective.
- The text appearing on the screen are primary source of information exchange between the user and the system. Avoid using abbreviation. Be very specific in communicating the mistake to the user. If possible provide the reason for error.
- Navigation within the screen is important and is specially useful for data entry screen where keyboard is used intensively to input data.
- Use of color should be of secondary importance. It may be kept in mind about user accessing application in a monochrome screen.
- Expect the user to make mistake and provide appropriate measure to handle such errors through proper interface design.
- Grouping of data element is important. Group related data items accordingly.
- Justify the data items.
- Avoid high density screen layout. Keep significant amount of screen blank.
- Make sure an accidental double click instead of a single click may does some thing unexpected.
- Provide file browser. Do not expect the user to remember the path of the required file.
- Provide key-board shortcut for frequently done tasks. This saves time.
- Provide on-line manual to help user in operating the software.
- Always allow a way out (i.e., cancellation of action already completed).
- Warn user about critical task, like deletion of file, updating of critical information.
- Programmers are not always good interface designer. Take help of expert professional who understands human perception better than programmers.
- Include all possible features in the application even if the feature is available in the operating system.

- Word the message carefully in a user understandable manner.
- Develop navigational procedure prior to developing the user interface.

3.6 DESIGN OF HUMAN COMPUTER INTERFACE

Human Computer Interface (HCI) design is topic that is gaining significance as the use of computers is growing. Let us look at a personal desktop PC that has following interface for humans to interact.

- A key board
- A Computer Mouse
- A Touch Screen (if equipped with)
- A program on your Windows machine that includes a trash can, icons of various programs, disk drives, and folders

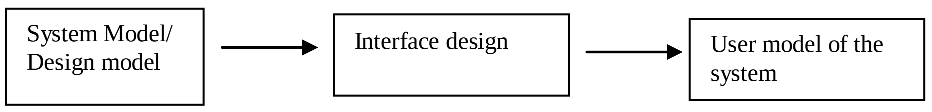

Figure 3.12: The process of interface design

What do these things have in common? These are all human-computer interfaces which were designed to make it easier to accomplish things with a computer. If we recall the early days computer, some one had to remember long cryptic strings of commands in accomplish simplest to simplest thing with a computer like coping a file from one folder to another or deleting a file etc. It is due to the evolution of human-computer interface and human-computer interface designers that's now being accomplished with the click of a mouse.

The overall process of design leads to the translation of design models to user model. Human-Computer Interface design goal is to discover the most efficient way to design interfaces for human interaction and understandable electronic messages. A lot of research is taking place in this area that even helps physically disabled person to operate computer as easily as a normal person. *Figure 3.12* depicts the process of interface design.

A branch of computer science is devoted to this area, with recommendations for the proper design of menus, icons, forms, messages, dialogue as well as data display. The user friendliness of the program we are using is a result of interface design. The buttons, pull down menu, context sensitive help have been designed to make it easy for you to access the program.

The process of HCI design begins with the a creation of a model of system function as perceived by the end user. These models are designed by delineating them from design issues and program structure. HCI establishes a user model of the overall system.

The following are some of the principles of Good Human computer interface design:

Diversity: Consider the types of user frequently use the system. The designer must consider the types of users ranging from novice user, knowledgeable but intermittent user and expert frequent user. Accommodating expectation of all types of user is important. Each type of user expects the screen layout to accommodate their desires, novices needing extensive help where as expert user want to accomplish the task in quickest possible time.

For example providing a command such as ^P (control P) to print a specific report as well as a printer icon to do the same job for the novice user.

Rules for Human Computer Interface Design:

1. **Consistency :**
    - o Interface is deigned so as to ensure consistent sequences of actions for similar situations. Terminology should be used in prompts, menus, and help screens should be identical. Color scheme, layout and fonts should be consistently applied throughout the system.
2. **Enable expert users to use shortcuts:**
    - o Use of short cut increases productivity. Short cut to increase the pace of interaction with use of, special keys and hidden commands.
3. **Informative feedback :**
    - o The feedback should be informative and clear.
4. **Error prevention and handling common errors :**
    - o Screen design should be such that users are unlikely to make a serious error. Highlight only actions relevant to current context. Allowing user to select options rather than filling up details. Don't allow alphabetic characters in numeric fields
    - o In case of error, it should allow user to undo and offer simple, constructive, and specific instructions for recovery.

5 **Allow reversal of action :** Allow user to reverse action committed. Allow user to migrate to previous screen.

6. **Reduce effort of memorisation by the user :**
    - o Do not expect user to remember information. A human mind can remember little information in short term memory. Reduce short term memory load by designing screens by providing options clearly using pull-down menus and icons
7. **Relevance of information :** The information displayed should be relevant to the present context of performing certain task.
8. **Screen size:** Consideration for screen size available to display the information. Try to accommodate selected information in case of limited size of the window.
9. **Minimize data input action :** Wherever possible, provide predefined selectable data inputs.
10. **Help :** Provide help for all input actions explaining details about the type of input expected by the system with example.

About Errors: Some errors are preventable. Prevent errors whenever possible. Steps can be taken so that errors are less likely to occur, using methods such as organising screens and menus functionally, designing screens to be distinctive and making it difficult for users to commit irreversible actions. Expect users to make errors, try to anticipate where they will go wrong and design with those actions in mind.

Norman's Research

Norman D. A. has contributed extensively to the field of human-computer interface design. This psychologist has taken insights from the field of industrial product design and applied them to the design of user interfaces.

According to Norman, design should use both knowledge in the world and knowledge in the head. Knowledge in the world is overt–we don't have to overload our short term memory by having to remember too many things (icons, buttons and menus provide us with knowledge in the world. The following are the mappings to be done:

- Mapping between actions and the resulting effect.
- Mapping between the information that is visible and the interpretation of the system state. For example, it should be obvious what the function of a button or menu is. Do not try to built your own meaning to the commonly used icons instead use conventions already established for it. Never use a search icon to print a page.
- Hyperlink should be used to migrate from one page to connected page.

☞ **Check Your Progress 3**

1) Keyboard short-cut is used to perform __________.

2) What are the types of user errors you may anticipate while designing a user interface. Explain?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………

3.7 SUMMARY

Design is a process of translating analysis model to design models that are further refined to produce detailed design models. The process of refinement is the process of elaboration to provides necessary details to the programmer. Data design deals with data structure selection and design. Modularity of program increases maintainability and encourages parallel development. The aim of good modular design is to produce highly cohesive and loosely coupled modules. Independence among modules is central to modularity. Good user interface design helps software to interact effectively to external environment. Tips for good interface design helps designer to achieve effective user interface.

Quality is built into software during the design of software. The final word is: Design process should be given due weightage before rushing for coding.

3.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) In building a design, where the quality of structural strength and other aspects are determined by the design of the building. In software design, the design process goes through a series of refinement process and reviews, to see whether the user requirements and other quality parameters are properly not reflected in the design model. A good design is more likely to deliver good software. The process of getting it right starts from the software design stage.

   - Data design affects the efficiency of algorithms.
   - Modularity enables the software component to perform distinct independent tasks with little chance of propagating errors to other module.
   - Good interface design minimizes user errors and increases user friendliness of the system.

2) True.

3) Software design is the first step of translating user requirement to technical domain of software development. Without designing there is always a risk of designing an unstable system. Without design we can't review the requirements of the product until it is delivered to the user. Design helps in future maintenance of the system.

Check Your Progress 2

1) False
2) Common coupling
3) Functional.

Check your Progress 3

1) Frequently performed tasks.

2) Anticipation user error provides vital input for user interface design. Prevent errors wherever possible and steps can be taken to design interface such that errors are less likely to occur. The methods that may be adopted may include well organisation of screen and menus functionally. Using user understandable language, designing screens to be distinctive and making it difficult for users to commit irreversible actions. Provide user confirmation to critical actions. Anticipate where user can go wrong and design the interface keeping this in mind.

3.9 FURTHER READINGS

1) *Software Engineering, Sixth Edition, 2001*, Ian Sommerville; Pearson Education.
2) *Software Engineering – A Practitioner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.
3) *Software Design : From Programming to Architecture, First Eition, 2003, Eric J. Braude; Wiley*

Reference Websites

http://www.ieee.org
http://www.rspa.com
http://sdg.csail.mit.edu

UNIT 4 SOFTWARE TESTING

Structure **Page Nos.**

4.0 INTRODUCTION

Testing means executing a program in order to understand its behaviour, that is, whether or not the program exhibits a failure, its response time or throughput for certain data sets, its mean time to failure, or the speed and accuracy with which users complete their designated tasks. In other words, it is a 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 can also be described as part of the process of Validation and Verification.

Validation is the process of evaluating a system or component during or at the end of the development process to determine if it satisfies the requirements of the system, or, in other words, are we building the correct system?

Verification is the process of evaluating a system or component at the end of a phase to determine if it satisfies the conditions imposed at the start of that phase, or, in other words, are we building the system correctly?

Software testing gives an important set of methods that can be used to evaluate and assure that a program or system meets its non-functional requirements.

To be more specific, software testing means that executing a program or its components in order to assure:

- The correctness of software with respect to requirements or intent;
- The performance of software under various conditions;
- The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;
- The usability of software under various conditions;
- The reliability, availability, survivability or other dependability measures of software; or
- Installability and other facets of a software release.

The purpose of testing is to show that the program has errors. The aim of most testing methods is to systematically and actively locate faults in the program and repair them. Debugging is the next stage of testing. Debugging is the activity of:

- Determining the exact nature and location of the suspected error within the program and
- Fixing the error. Usually, debugging begins with some indication of the existence of an error.

The purpose of debugging is to locate errors and fix them.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- know the basic terms using in testing terminology;
- black box and White box testing techniques;
- other testing techniques; and
- some testing tools.

4.2 BASIC TERMS USED IN TESTING

Failure: A failure occurs when there is a deviation of the observed behavior of a program, or system, from its specification. A failure can also occur if the observed behaviour of a system, or program, deviates from its intended behavior.

Fault: A fault is an incorrect step, process, or data definition in a computer program. Faults are the source of failures. In normal language, software faults are usually referred to as “bugs”.

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Test Cases: Ultimately, testing comes down to selecting and executing test cases. A test case for a specific component consists of three essential pieces of information:

- A set of test inputs;
- The expected results when the inputs are executed; and
- The execution conditions or environments in which the inputs are to be executed.

Some Testing Laws

- Testing can only be used to show the presence of errors, but never the absence or errors.
- A combination of different verification & validation (V&V) methods outprform any single method alone.
- Developers are unsuited to test their own code.
- Approximately 80% of the errors are found in 20% of the code.
- Partition testing, that is, methods that partition the input domain or the program and test according to those partitions. This is better than random testing.
- The adequacy of a test suite for coverage criterion can only be defined intuitively.

4.2.1 Input Domain

To conduct an analysis of the input, the sets of values making up the input domain are required. There are essentially two sources for the input domain. They are:

1. The software requirements specification in the case of black box testing method; and
2. The design and externally accessible program variables in the case of white box testing.

In the case of white box testing, input domain can be constructed from the following sources.

- Inputs passed in as parameters; Variables that are inputs to function under test can be: (i) Structured data such as linked lists, files or trees, as well as atomic data such as integers and floating point numbers;

(ii) A reference or a value parameter as in the *C* function declaration

```
int P(int *power, int base) {
 ...}
```

- Inputs entered by the user via the program interface;
- Inputs that are read in from files;
- Inputs that are constants and precomputed values; Constants declared in an enclosing scope of function under test, for example,

```
#define PI 3.14159
double circumference(double radius)
{
return 2*PI*radius;
}
```

In general, the inputs to a program or a function are stored in program variables. A program variable may be:

- A variable declared in a program as in the *C* declarations

For example: int base; char s[];

- Resulting from a read statement or similar interaction with the environment, For example: scanf(‘‘%d\n’’, &x);

4.2.2 Black Box and White Box Test Case Selection Strategies

- **Black box Testing**: In this method, where test cases are derived from the functional specification of the system; and
- **White box Testing**: In this method, where test cases are derived from the internal design specifications or actual code (Sometimes referred to as Glass-box).

Black box test case selection can be done without any reference to the program design or the program code. Test case selection is only concerned with the functionality and features of the system but not with its internal operations.

- The real advantage of black box test case selection is that it can be done *before* the design or coding of a program. Black box test cases can also help to get the design and coding correct with respect to the specification. Black box testing methods are good at testing for missing functions or program behavior that deviates from the specification. Black box testing is ideal for evaluating products that you intend to use in your systems.

- The main disadvantage of black box testing is that black box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that need to be safe (additional code may interfere with the safety of the system) or secure (additional code may be used to break security).

White box test cases are selected using the specification, design and code of the program or functions under test. This means that the testing team needs access to the internal designs or code for the program.

- The chief advantage of white box testing is that it tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. White box testing can check additional functions or code that has been implemented, but not specified.

- The main disadvantage of white box testing is that you must wait until after design and coding of the programs of functions under test have been completed in order to select test cases.

Methods for Black box testing strategies

A number of test case selection methods exist within the broad classification of black box and white box testing.

For Black box testing strategies, the following are the methods:

- Boundary-value Analysis;
- Equivalence Partitioning.

We will also study State Based Testing, which can be classified as opaque box selection strategies that is somewhere between black box and white box selection strategies.

Boundary-value-analysis

The basic concept used in Boundary-value-analysis is that if the specific test cases are designed to check the boundaries of the input domain then the probability of detecting an error will increase. If we want to test a program written as a function F with two input variables x and y., then these input variables are defined with some boundaries like $a1 \leq x \leq a2$ and $b1 \leq y \leq b2$. It means that inputs x and y are bounded by two intervals [a1, a2] and [b1, b2].

Test Case Selection Guidelines for Boundary Value Analysis

The following set of guidelines is for the selection of test cases according to the principles of boundary value analysis. The guidelines do not constitute a firm set of rules for every case. You will need to develop some judgement in applying these guidelines.

1. If an *input* condition specifies a range of values, then construct valid test cases for the ends of the range, and invalid input test cases for input points just beyond the ends of the range.
2. If an *input* condition specifies a number of values, construct test cases for the minimum and maximum values; and one beneath and beyond these values.
3. If an *output* condition specifies a range of values, then construct valid test cases for the ends of the output range, and invalid input test cases for situations just beyond the ends of the output range.
4. If an *output* condition specifies a number of values, construct test cases for the minimum and maximum values; and one beneath and beyond these values.
5. If the input or output of a program is an ordered set (e.g., a sequential file, linear list, table), focus attention on the first and last elements of the set.

Example 1: Boundary Value Analysis for the Triangle Program

Consider a simple program to classify a triangle. Its input consists of three positive integers (say x, y, z) and the data types for input parameters ensures that these will be integers greater than zero and less than or equal to 100. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled.

Solution: Following possible boundary conditions are formed:

1. Given sides (A; B; C) for a scalene triangle, the sum of any two sides is greater than the third and so, we have boundary conditions $A + B > C$, $B + C > A$ and $A + C > B$.
2. Given sides (A; B; C) for an isosceles triangle two sides must be equal and so we have boundary conditions $A = B$, $B = C$ or $A = C$.
3. Continuing in the same way for an equilateral triangle the sides must all be of equal length and we have only one boundary where $A = B = C$.
4. For right-angled triangles, we must have $A^2+B^2 = C^2$.

On the basis of the above boundary conditions, test cases are designed as follows (*Table 4.1*):

Table 4.1: Test cases for Example-1

Test case	x	y	z	Expected Output
1	100	100	100	Equilatera*l* triangle
2	50	3	50	Isosceles triangle
3	40	50	40	Equilatera*l* triangle
4	3	4	5	Right-angled triangles
5	10	10	10	Equilatera*l* triangle
6	2	2	5	Isosceles triangle
7	100	50	100	Scalene triangle
8	1	2	3	Non-triangles
9	2	3	4	Scalene triangle
10	1	3	1	Isosceles triangle

Equivalence Partitioning

Equivalence Partitioning is a method for selecting test cases based on a partitioning of the input domain. The aim of equivalence partitioning is to divide the input domain of the program or module into classes (sets) of test cases that have a similar effect on the program. The classes are called Equivalence classes.

Equivalence Classes

An Equivalence *Class* is a set of inputs that the program treats identically when the program is tested. In other words, a test input taken from an equivalence class is representative of all of the test inputs taken from that class. Equivalence classes are determined from the specification of a program or module. Each equivalence class is used to represent certain conditions (or predicates) on the input domain. For equivalence partitioning it is usual to also consider valid and invalid inputs. The terms input condition, valid and invalid inputs, are not used consistently. But, the following definition spells out how we will use them in this subject. An input condition on the input domain is a predicate over the values of the input domain. A *Valid* input to a program or module is an element of the input domain that is expected to return a non-error value. An *Invalid* input is an input that is expected to return an error value. Equivalence partitioning is then a systematic method for identifying interesting input conditions to be tested. An input condition can be applied to a set of values of a specific input variable, or a set of input variables
as well.

A Method for Choosing Equivalence Classes

The aim is to minimize the number of test cases required to cover all of the identified equivalence classes. The following are two distinct steps in achieving this goal:

Step 1: Identify the equivalence classes

If an input condition specifies a range of values, then identify one valid equivalence class and two invalid equivalence classes.
For example, if an input condition specifies a range of values from 1 to 99, then, three equivalence classes can be identified:

- One valid equivalence class: $1 < X < 99$
- Two invalid equivalence classes $X < 1$ and $X > 99$

Step 2: Choose test cases

The next step is to generate test cases using the equivalence classes identified in the previous step. The guideline is to choose test cases on the boundaries of partitions and test cases close to the midpoint of the partition. In general, the idea is to select at least one element from each equivalence class.

Example 2: Selecting Test Cases for the Triangle Program

In this example, we will select a set of test cases for the following triangle program based on its specification. Consider the following informal specification for the Triangle Classification Program. The program reads three integer values from the standard input. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled. The specification of the triangle classification program lists a number of inputs for the program as well as the form of output. Further, we require that each of the inputs *"must b*e" a positive integer. Now, we can determine valid and invalid equivalence classes for the input conditions. Here, we have a range of values. If the three integers we have called x, y and z are all greater than zero, then, they are valid and we have the equivalence class.

ECvalid = f(x,y, z) $x > 0$ and $y > 0$ and $z > 0$.

For the invalid classes, we need to consider the case where each of the three variables in turn can be negative and so we have the following equivalence classes:

ECInvalid1 = f(x, y, z) $x < 0$ and $y > 0$ and $z > 0$
ECInvalid2 = f(x, y, z) $x > 0$ and $y <0$ and $z > 0$
ECInvalid3 = f(x, y, z) $x > 0$ and $y > 0$ and $z < 0$

Note that we can combine the valid equivalence classes. But, we are not allowed to combine the invalid equivalence classes. The output domain consists of the text 'strings' 'isosceles', 'scalene', 'equilateral' and 'right-angled'. Now, different values in the input domain map to different elements of the output domain to get the equivalence classes in *Table 4.2*. According to the equivalence partitioning method we only need to choose one element from each of the classes above in order to test the triangle program.

Table 4.2: The equivalence classes for the triangle program

Equivalence class	**Test Inputs**	**Expected Outputs**
ECscalene	f(3, 5, 7), . . . g	"Scalene"
ECisosceles	f(2, 3, 3), . . . g f(2, 3, 3), . . g	"Isosceles"
ECequilateral	f(7, 7, 7), . . . g f(7, 7, 7), . . . g	"Equilateral"
ECright angled	f(3, 4, 5), . . .	"Right Angled"
ECnon _ triangle	f(1, 1, 3), . . .	"Not a Triangle"
ECinvalid1	f(-1, 2, 3), (0, 1, 3), . . .	"Error Value"
ECinavlid2	f(1, -2, 3), (1, 0, 3), . . .	"Error Value"
ECinavlid3	f(1, 2, -3), (1, 2, 0),. . .	"Error Value"

Methods for White box testing strategies

In this approach, complete knowledge about the internal structure of the source code is required. For *White-box testing strategies,* the methods are:

1. Coverage Based Testing
2. Cyclomatic Complexity
3. Mutation Testing

Coverage based testing

The aim of coverage based testing methods is to *'cove*r' the program with test cases that satisfy some fixed coverage criteria. Put another way, we choose test cases to exercise as much of the program as possible according to some criteria.

Coverage Based Testing Criteria

Coverage based testing works by choosing test cases according to well-defined 'coverage' criteria. The more common coverage criteria are the following.

- **Statement Coverage or Node Coverage:** Every statement of the program should be exercised at least once.
- **Branch Coverage or Decision Coverage:** Every possible alternative in a branch or decision of the program should be exercised at least once. For *if* statements, this means that the branch must be made to take on the values *true* or *false*.
- **Decision/Condition Coverage**: Each condition in a branch is made to evaluate to both true and false and each branch is made to evaluate to both true and false.
- **Multiple condition coverage**: All possible combinations of condition outcomes within each branch should be exercised at least once.
- **Path coverage**: Every execution path of the program should be exercised at least once.

In this section, we will use the control flow graph to choose white box test cases according to the criteria above. To motivate the selection of test cases, consider the simple program given in Program 4.1.

Example 3:

```
void main(void)
{
int x1, x2, x3;
scanf("%d %d %d", &x1, &x2, &x3);
if ((x1 > 1) && (x2 == 0))
x3 = x3 / x1;
if ((x1 == 2) || (x3 > 1))
x3 = x3 + 1;
while (x1 >= 2)
x1 = x1 - 2;
printf("%d %d %d", x1, x2, x3);
}
```

Program 4.1: A simple program for white box testing

The first step in the analysis is to generate the flow chart, which is given in Figure 4.1. Now what is needed for statement coverage? If all of the branches are true, at least once, we will have executed every statement in the flow chart. Put another way to execute every statement at least once, we must execute the path ABCDEFGF. Now, looking at the conditions inside each of the three *branches*, we derive a set of constraints on the values of x1, x2 and x3 such that all the three branches are extended. A test case of the form (x1; x2; x3) = (2; 0; 3) will execute all of the statements in the program.

Note that we need not make every branch evaluate to true and false, nor have we make every condition evaluate to true and false, nor we traverse every path in the program.

To make the first branch true, we have test input (2; 0; 3) that will make all of the branches true. We need a test input that will now make each one false. Again looking at all of the conditions, the test input (1; 1; 1) will make all of the branches false.

For any of the criteria involving condition coverage, we need to look at each of the five conditions in the program: C1 = (x1>1), C2 = (x2 == 0), C3 = (x1 == 2), C4 = (x3>1) and C5 = (x1 >= 2). The test input (1; 0; 3) will make C 1 false, C2 true, C3 false, C4 true and C5 false.

Examples of sets of test inputs and the criteria that they meet are given in Table 4.3. The set of test cases meeting the multiple condition criteria is given in Table 4.4. In the table, we let the branches B1 = C1&&C2, B2 = C3||C4 and B3 = C5.

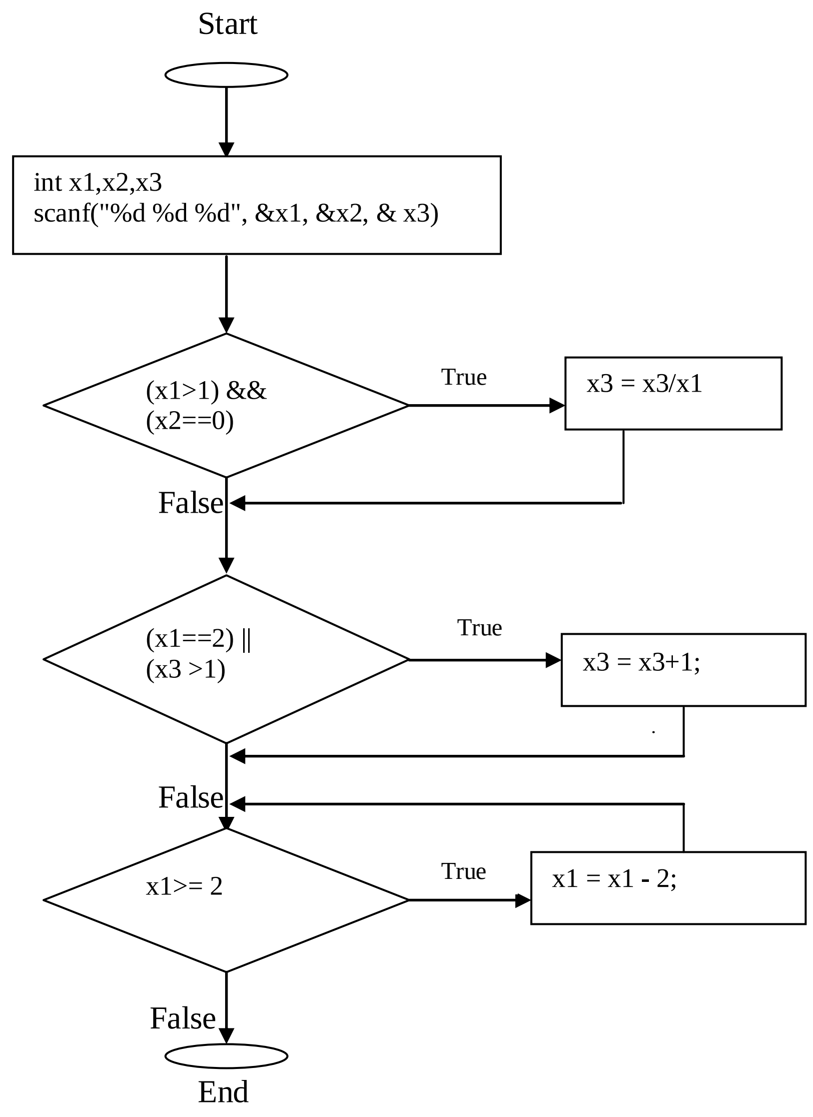

Figure 4.1: The flow chart for the program 4.1

Table 4.3: Test cases for the various coverage criteria for the program 4.1

Coverage Criteria	**Test Inputs (x1, x2, x3)**	**Execution Paths**
Statement	(2, 0, 3)	ABCDEFGF
Branch	(2, 0, 3), (1, 1, 1)	ABCDEFGF ABDF
Condition	(1, 0, 3), (2, 1, 1)	ABDEF ABDFGF
Decision/ Condition	(2, 0, 4), (1, 1, 1)	ABCDEFGF ABDF
Multiple Condition	(2, 0, 4), (2, 1, 1), (1, 0, 2), (1, 1, 1)	ABCDEFGF ABDEFGF ABDEF ABDF
Path	(2, 0, 4), (2, 1, 1), (1, 0, 2), (4, 0, 0),	ABCDEFGF ABDFGF ABDEF ABCDFGFGF

Table 4.4: Multiple condition coverage for the program in Figure 4.1

Test cases	C1 x1 > 1	C2 x2==0	B1	C3 x1==2	C4 x3 > 1	B2	B3 C5 x1 ≥ 2
(1,0,3)	F	T	F	F	T	T	F
(2,1,1)	T	F	F	T	F	F	T
(2,0,4)	T	T	T	T	T	T	T
(1,1,1)	F	F	F	F	F	F	F
(2,0,4)	T	T	T	T	T	T	T
(2,1,1)	T	F	F	T	F	T	T
(1,0,2)	F	T	F	F	T	T	F
(1,1,1)	F	F	F	F	F	F	F

4.2.3 Cyclomatic Complexity

Control flow graph (CFG)

A control flow graph describes the sequence in which different instructions of a program get executed. It also describes how the flow of control passes through the program. In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph. An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. Following structured programming constructs are represented as CFG:

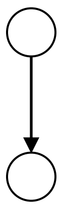
Figure 4.2 : sequence

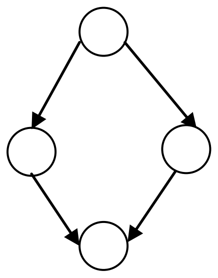
Figure 4.3 : if -else

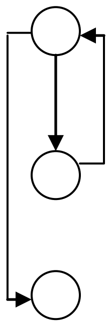
Figure 4.4 : while-loop

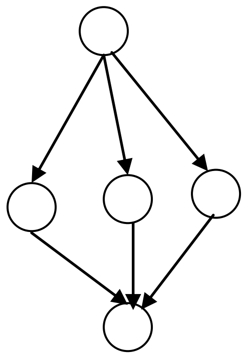
Figure 4.5: case

Example 4.4: Draw CFG for the program given below.

```
int sample (a,b)
int a,b;
{
1       while (a!= b) {
2       if (a > b)
3       a = a-b;
4       else b = b-a;}
5       return a;
}
```

Program 4.2: A program

In the above program, two control constructs are used, namely, while-loop and if-then-else. A complete CFG for the program of Program 4.2 is given below: (*Figure 4.6*).

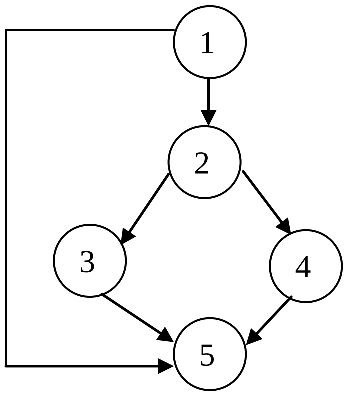

Figure 4.6: CFG for program 4.2

Cyclomatic Complexity: This technique is used to find the number of independent paths through a program. If CFG of a program is given, the Cyclomatic complexity V(G) can be computed as: V(G) = E – N + 2, where N is the number of nodes of the CFG and E is the number of edges in the CFG. For the example 4, the Cyclomatic Complexity = 6 – 5 + 2 = 3.

The following are the properties of Cyclomatic complexity:

- V(G) is the maximum number of independent paths in graph G
- Inserting and deleting functional statements to G does not affect V(G)
- G has only one path if and only if V(G) = 1.

Mutation Testing

Mutation Testing is a powerful method for finding errors in software programs. In this technique, multiple copies of programs are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program. The mutant that is detected by a test case is termed "killed" and the goal of the mutation procedure is to find a set of test cases that are able to kill groups of mutant programs. Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. It is essential that all mutants must be killed by the test cases or shown to be equivalent to the original expression. If we run a mutated program, there are two possibilities:

1. The results of the program were affected by the code change and the test suite detects it. We assumed that the test suite is perfect, which means that it must detect the change. If this happens, the mutant is called a killed mutant.

2. The results of the program are not changed and the test suite does not detect the mutation. The mutant is called an equivalent mutant.

If we take the ratio of killed mutants to all the mutants that were created, we get a number that is smaller than 1. This number gives an indication of the sensitivity of program to the changes in code. In real life, we may not have a perfect program and we may not have a perfect test suite. Hence, we can have one more scenario:

3. The results of the program are different, but the test suite does not detect it because it does not have the right test case.

Consider the following program 4.3:

```
main(argc, argv)                           /* line 1 */
int argc;                              /* line 2 */
char *argv[];                              /* line 3 */
{                                          /* line 4 */
  int c=0;                                 /* line 5 */
                                           /* line 6 */
    if(atoi(argv[1]) < 3){             /* line 7 */
      printf("Got less than 3\n");         /* line 8 */
      if(atoi(argv[2]) > 5)                /* line 9 */
        c = 2;                         /* line 10 */
    }                                      /* line 11 */
    else                                   /* line 12 */
      printf("Got more than 3\n");     /* line 13 */
   exit(0);                                /* line 14 */
}                                          /* line 15 */
```

Program 4.3: A program

The program reads its arguments and prints messages accordingly.

Now let us assume that we have the following test suite that tests the program:

Test case 1:
    Input: 2 4
    Output: Got less than 3
Test case 2:
    Input: 4 4
    Output: Got more than 3
Test case 3:
    Input: 4 6
    Output: Got more than 3
Test case 4:
    Input: 2 6
    Output: Got less than 3
Test case 5:
    Input: 4
    Output: Got more than 3

Now, let's mutate the program. We can start with the following simple changes:

Mutant 1: change line 9 to the form

```
if(atoi(argv[2]) <= 5)
```

Mutant 2: change line 7 to the form

```
if(atoi(argv[1]) >= 3)
```

Mutant 3: change line 5 to the form

```
int c=3;
```

If we take the ratio of all the killed mutants to all the mutants generated, we get a number smaller than 1 that also contains information about accuracy of the test suite. In practice, there is no way to separate the effect that is related to test suite inaccuracy and that which is related to equivalent mutants. In the absence of other possibilities, one can accept the ratio of killed mutants to all the mutants as the measure of the test suite accuracy. The manner by which a test suite is evaluated via mutation testing is as follows: For a specific test suite and a specific set of mutants, there will be three types of mutants in the code (i) killed or dead (ii) live (iii) equivalent. The score (evaluation of test suite) associated with a test suite T and mutants M is simply computed as follows:

$$\frac{\text{\# killed Mutants}}{\text{\# total mutants - \# equivalent mutants}} \times 100$$

☞ **Check Your Progress 1**

1) What is the use of Cyclomatic complexity in software development?

……………………………………………………………………………

……………………………………………………………………………

……………………………………………………………………………

4.3 TESTING ACTIVITIES

Although testing varies between organisations, there is a cycle to testing:

Requirements Analysis: Testing should begin in the requirements phase of the software devlopment life cycle (SDLC).

Design Analysis: During the design phase, testers work with developers in determining what aspects of a design are testable and under what parameters should the testers work.

Test Planning: Test Strategy, Test Plan(s).

Test Development: Test Procedures, Test Scenarios, Test Cases, Test Scripts to use in testing software.

Test Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.

Test Reporting: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.

Retesting the Defects: Defects are once again tested to find whether they got eliminated or not.

Levels of Testing:

Mainly, Software goes through three levels of testing:

- Unit testing
- Integration testing
- System testing.

Unit Testing

Unit testing is a procedure used to verify that a particular segment of source code is working properly. The idea about unit tests is to write test cases for all functions or methods. Ideally, each test case is separate from the others. This type of testing is mostly done by developers and not by end users.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Unit testing provides a strict, written contract that the piece of code must satisfy. Unit testing will not catch every error in the program. By definition, it only tests the functionality of the units themselves. Therefore, it will not catch integration errors, perfomance problems and any other system-wide issues. A unit test can only show the presence of errors; it cannot show the absence of errors.

Integration Testing

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing. Integration testing takes as its input, modules that have been checked out by unit testing, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output, the integrated system ready for system testing.The purpose of Integration testing is to verify functional, performance and reliability requirements placed on major design items.

System Testing

System testing is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. As a rule, system testing takes, as its input, all of the “integrated” software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). In system testing, the entire system can be tested as a whole against the software requirements specification (SRS). There are *rules* that describe the functionality that the vendor (developer) and a customer have agreed upon. System testing tends to be more of an *investigatory* testing phase, where the focus is to have a destructive attitude and test not only the design, but also the behavior and even the believed expectations of the customer. System testing is intended to test up to and beyond the bounds defined in the software requirements specifications.

Acceptance tests are conducted in case the software developed was a custom software and not product based. These tests are conducted by customer to check whether the software meets all requirements or not. These tests may range from a few weeks to several months.

4.4 DEBUGGING

Debugging occurs as a consequence of successful testing. Debugging refers to the process of identifying the cause for defective behavior of a system and addressing that problem. In less complex terms - fixing a bug. When a test case uncovers an error, debugging is the process that results in the removal of the error. The debugging process begins with the execution of a test case. The debugging process attempts to match symptoms with cause, thereby leading to error correction. The following are two alternative outcomes of the debugging:

1. The cause will be found and necessary action such as correction or removal will be taken.
2. The cause will not be found.

Characteristics of bugs

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Life Cycle of a Debugging Task

The following are various steps involved in debugging:

a) **Defect Identification/Confirmation**

- A problem is identified in a system and a defect report created
- Defect assigned to a software engineer
- The engineer analyzes the defect report, performing the following actions:
  - What is the expected/desired behaviour of the system?
  - What is the actual behaviour?
  - Is this really a defect in the system?
  - Can the defect be reproduced? (While many times, confirming a defect is straight forward. There will be defects that often exhibit quantum behaviour.)

b) **Defect Analysis**

Assuming that the software engineer concludes that the defect is genuine, the focus shifts to understanding the root cause of the problem. This is often the most challenging step in any debugging task, particularly when the software engineer is debugging complex software.

Many engineers debug by starting a debugging tool, generally a debugger and try to understand the root cause of the problem by following the execution of the program step-by-step. This approach may eventually yield success. However, in many situations, it takes too much time, and in some cases is not feasible, due to the complex nature of the program(s).

c) **Defect Resolution**

Once the root cause of a problem is identified, the defect can then be resolved by making an appropriate change to the system, which fixes the root cause.

Debugging Approaches

Three categories for debugging approaches are:

- Brute force
- Backtracking
- Cause elimination.

Brute force is probably the most popular despite being the least successful. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" technique, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. *Backtracking* is a common debugging method that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backwards till the error is found. In *Cause elimination,* a list of possible causes of an error are identified and tests are conducted until each one is eliminated.

☞ **Check Your Progress 2**

1) What are the different levels of testing and their goals? For each level specify which of the testing approaches are most suitable.

…………………………………………………………………………………..

…………………………………………………………………………………..

2) Mention the steps involved in the process of debugging.

…………………………………………………………………………………

………...

4.5 TESTING TOOLS

The following are different categories of tools that can be used for testing:

- **Data Acquisition**: Tools that acquire data to be used during testing.
- **Static Measurement**: Tools that analyse source code without executing test cases.
- **Dynamic Measurement**: Tools that analyse source code during execution.
- **Simulation**: Tools that simulate functions of hardware or other externals.
- **Test Management**: Tools that assist in planning, development and control of testing.
- **Cross-Functional tools**: Tools that cross the bounds of preceding categories.

The following are some of the examples of commercial software testing tools:

Rational Test Real Time Unit Testing

- **Kind of Tool**

  Rational Test RealTime's Unit Testing feature automates C, C++ software component testing.
- **Organisation**

  IBM Rational Software
- **Software Description**

Rational Test RealTime Unit Testing performs black-box/functional testing, i.e., verifies that all units behave according to their specifications without regard to how that functionality is implemented. The Unit Testing feature has the flexibility to naturally fit any development process by matching and automating developers' and testers' work patterns, allowing them to focus on value-added tasks. Rational Test RealTime is integrated with native development environments (Unix and Windows) as well as with a large variety of cross-development environments.

- **Platforms**

Rational Test RealTime is available for most development and target systems including Windows and Unix.

AQtest

- **Kind of Tool**

  Automated support for functional, unit, and regression testing
- **Organisation**

  AutomatedQA Corp.
- **Software Description**

AQtest automates and manages functional tests, unit tests and regression tests, for applications written with VC++, VB, Delphi, C++Builder, Java or VS.NET. It also supports white-box testing, down to private properties or methods. External tests can be recorded or written in three scripting languages (VBScript, JScript, DelphiScript). Using AQtest as an OLE server, unit-test drivers can also run it directly from application code. AQtest automatically integrates AQtime when it is on the machine. Entirely COM-based, AQtest is easily extended through plug-ins using the complete IDL libraries supplied. Plug-ins currently support Win32 API calls, direct ADO access, direct BDE access, etc.

- **Platforms**

Windows 95, 98, NT, or 2000.

csUnit

- **Kind of Tool**

  "Complete Solution Unit Testing" for Microsoft .NET (freeware)

- **Organisation**

  csUnit.org

- **Software Description**

  csUnit is a unit testing framework for the Microsoft .NET Framework. It targets test driven development using .NET languages such as C#, Visual Basic .NET, and managed C++.

- **Platforms**

  Microsoft Windows

Sahi

http://sahi.sourceforge.net/

Software Description

Sahi is an automation and testing tool for web applications, with the facility to record and playback scripts. Developed in Java and JavaScript, it uses simple JavaScript to execute events on the browser. Features include in-browser controls, text based scripts, Ant support for playback of suites of tests, and multi-threaded playback. It supports HTTP and HTTPS. Sahi runs as a proxy server and the browser needs to use the Sahi server as its proxy. Sahi then injects JavaScript so that it can access elements in the webpage. This makes the tool independant of the website/ web application.

- **Platforms**

OS independent. Needs at least JDK1.4

4.6 SUMMARY

The importance of software testing and its impact on software is explained in this unit. Software testing is a fundamental component of software development life cycle and represents a review of specification, design and coding. The objective of testing is to have the highest likelihood of finding most of the errors within a minimum amount of time and minimal effort. A large number of test case design methods have been developed that offer a systematic approach to testing to the developer.

Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational. A strategy for software testing may be to move upwards along the spiral. Unit testing happens at the vortex of the spiral and concentrates on each unit of the software as implemented by the source code. Testing happens upwards along the spiral to integration testing, where the focus is on design and production of the software architecture. Finally, we perform system testing, where software and other system elements are tested together.

Debugging is not testing, but always happens as a response of testing. The debugging process will have one of two outcomes:

1) The cause will be found, then corrected or removed, or

2) The cause will not be found. Regardless of the approach that is used, debugging has one main aim: to determine and correct errors. In general, three kinds of debugging approaches have been put forward: Brute force, Backtracking and Cause elimination.

4.7 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Cyclomatic Complexity is asoftware metric that provides a quantitative measure of the logical complexity of a program. When it is used in the context of the basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program. It also provides an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

Check Your Progress 2

1) The basic levels of testing are: unit testing, integration testing, system testing and acceptance testing.

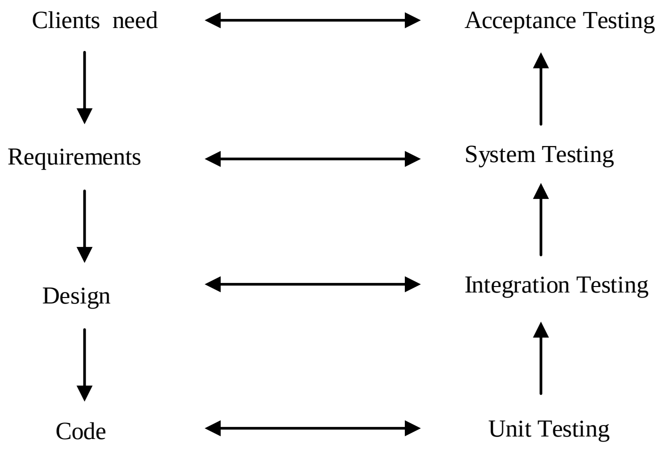

Figure 4.7: Testing levels

For unit testing, structural testing approach is best suited because the focus of testing is on testing the code. In fact, structural testing is not very suitable for large programs. It is used mostly at the unit testing level. The next level of testing is integration testing and the goal is to test interfaces between modules. With integration testing, we move slowly away from structural testing and towards functional testing. This testing activity can be considered for testing the design. The next levels are system and acceptance testing by which the entire software system is tested. These testing levels focus on the external behavior of the system. The internal logic of the program is not emphasized. Hence, mostly functional testing is performed at these levels.

2) The various steps involved in debugging are:

- Defect Identification/Confirmation
- Defect Analysis
- Defect Resolution

4.8 FURTHER READINGS

1) *Software Engineering, Sixth Edition, 2001*, Ian Sommerville; Pearson Education.

2) *Software Engineering – A Practitioner's Approach*, Roger S. Pressman; McGraw-Hill International Edition.

3) An *Integrated approach to Software Engineering*, Pankaj Jalote; Narcosis Publishing House.

Reference websites

http://www.rspa.com
http://www.ieee.org
http://standards.ieee.org
http://www.ibm.com
http://www.opensourcetesting.org