UNIT 1 SOFTWARE ENGINEERING AND ITS MODELS

Structure **Page Nos.**

1.0 Introduction 5
1.1 Objectives 6
1.2 Evolution of Software Engineering 6
1.3 Software Development Models 7
1.3.1 Importance of Software Engineering
1.3.2 Various Development Models
1.4 Capability Maturity Models 12
1.4.1 Maturity Levels
1.4.2 Key Process Areas
1.5 Software Process Technology 15
1.6 Summary 20
1.7 Solutions/Answers 20
1.8 Further Readings 21

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.

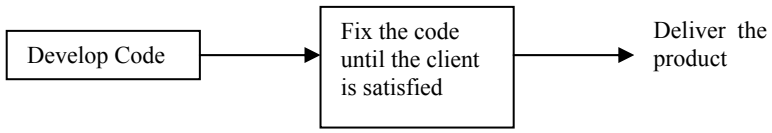


Figure 1.1 : Build and fix model

(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.

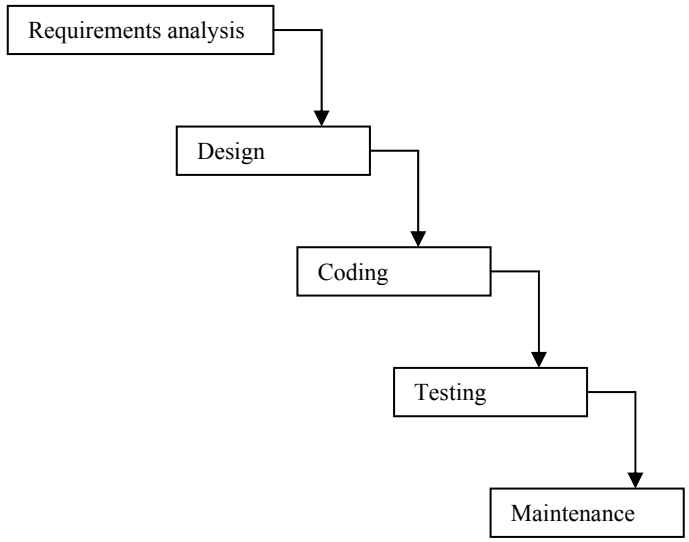


Figure 1.2 : Water fall model

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.

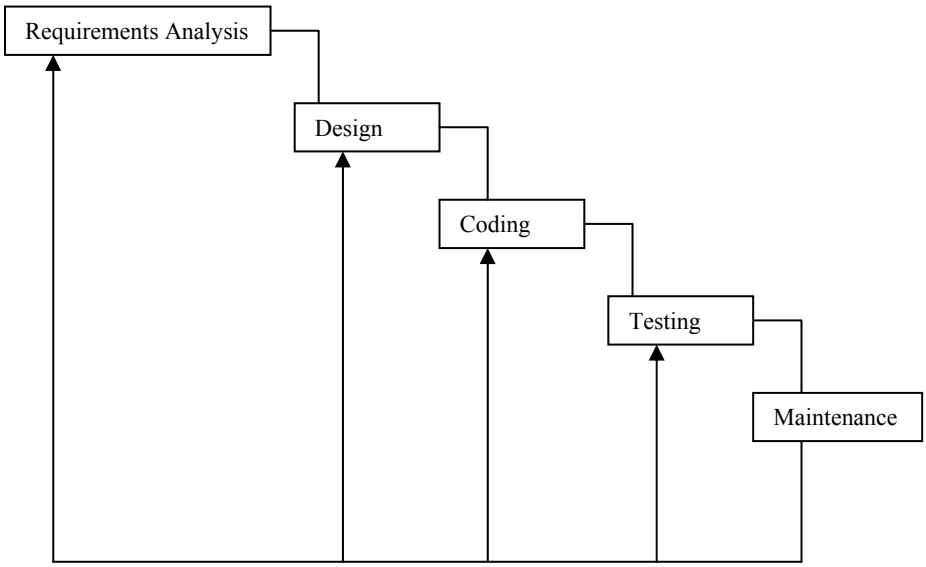


Figure 1.3 : Water fall model with feedback

(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.

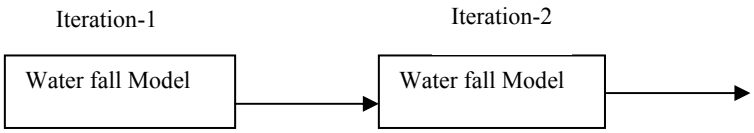


Figure 1.4 : Iterative enhancement model

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.

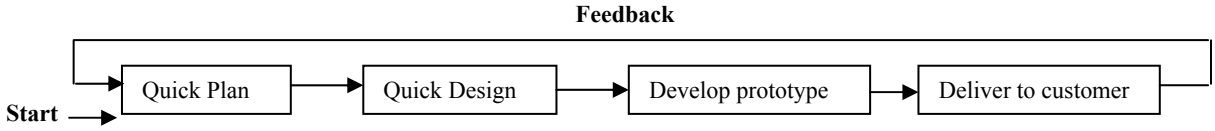


Figure 1.5: Prototyping model

(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 22
2.1 Objectives 23
2.2 Engineering the Product 23
- 2.2.1 Requirements Engineering
- 2.2.2 Types of Requirements
- 2.2.3 Software Requirements Specification (SRS)
- 2.2.4 Problems in SRS
- 2.2.5 Requirements Gathering Tools

2.3 Modeling the System Architecture 26
- 2.3.1 Elementary Modeling Techniques
- 2.3.2 Data Flow Diagrams
- 2.3.3 Rules for Making DFD
- 2.3.4 Data Dictionary
- 2.3.5 E-R Diagram
- 2.3.6 Structured Requirements Definition

2.4 Software Prototyping and Specification 34
- 2.4.1 Types of Prototype
- 2.4.2 Problems of Prototyping
- 2.4.3 Advantages of Prototyping

2.5 Software Metrics 35
2.6 Summary 36
2.7 Solutions/Answers 37
2.8 Further Readings 38

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*.

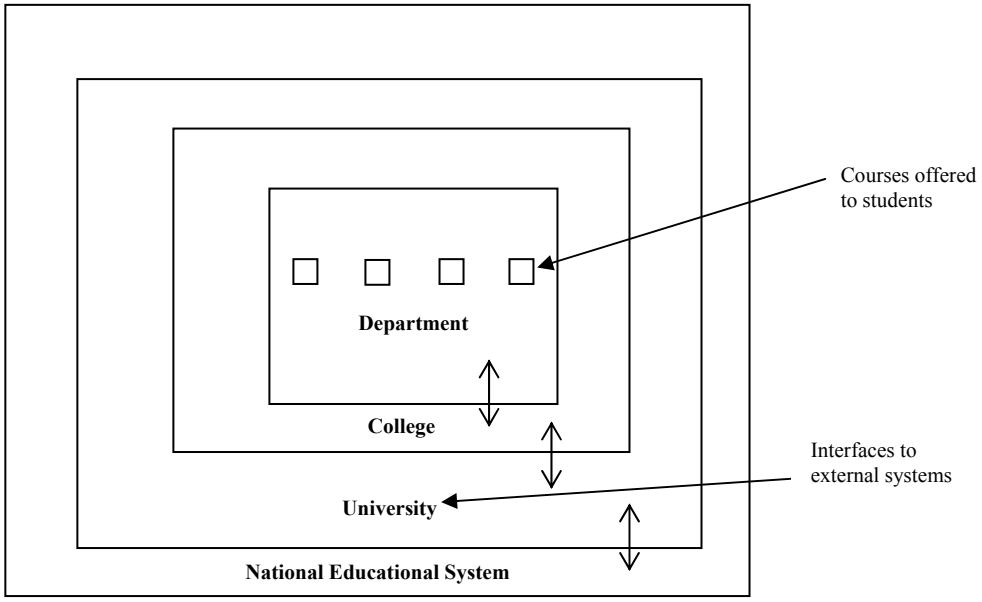


Figure 2.1 : Environmental model of educational system

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.*

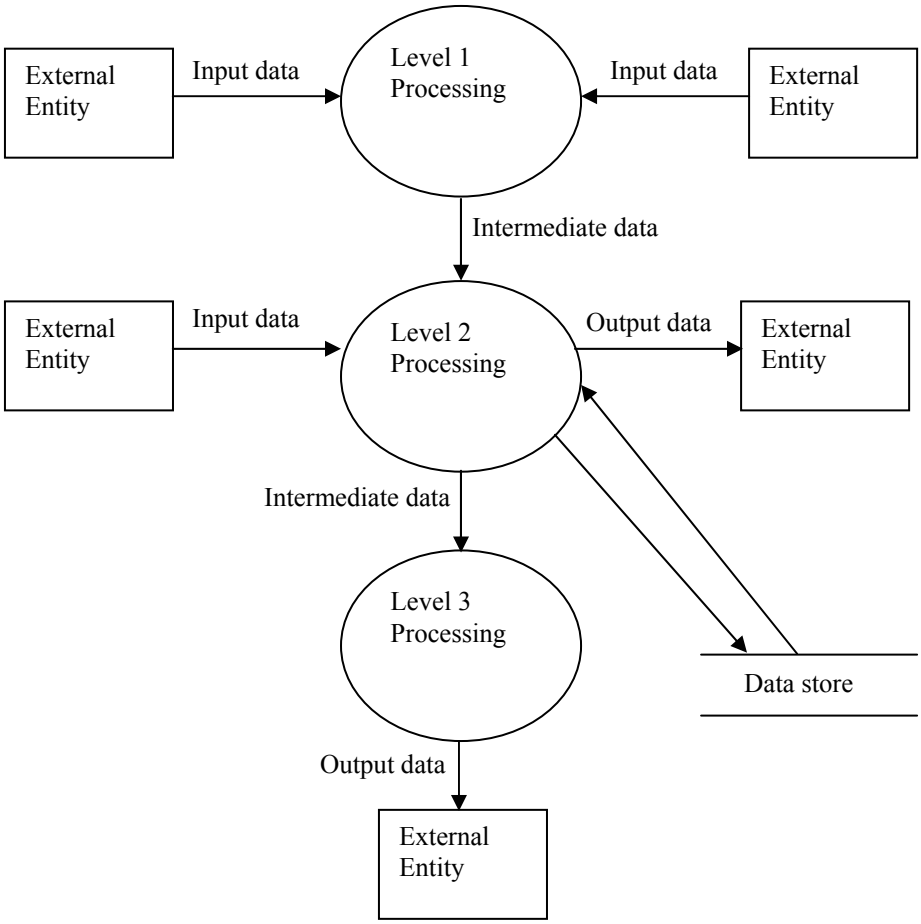


Figure 2.2: Data flow diagram

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*).

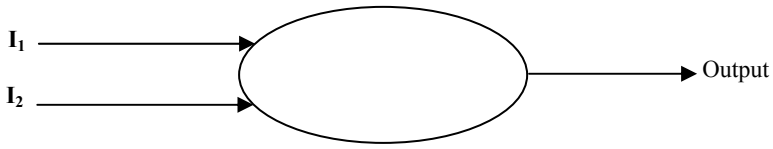


Figure 2.3: Layer 1 depiction of process A

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.*

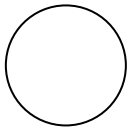
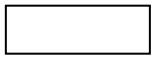
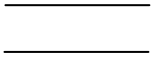
Symbol	Name	Description
	Data Flow	Represents the connectivity between various processes
	Process	Performs some processing of input data
	External Entity	Defines source or destination of system data. The entity which receives or supplies information.
	Data Store	Repository of data

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 0^{th} and 1^{st} 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)

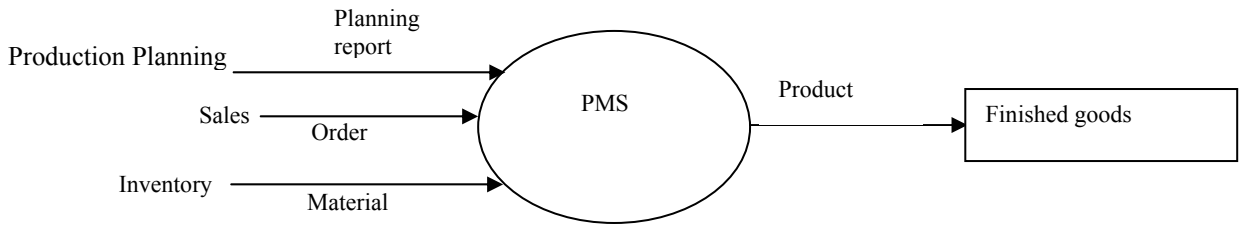


Figure 2.5 (a) : Level 0 DFD of PMS

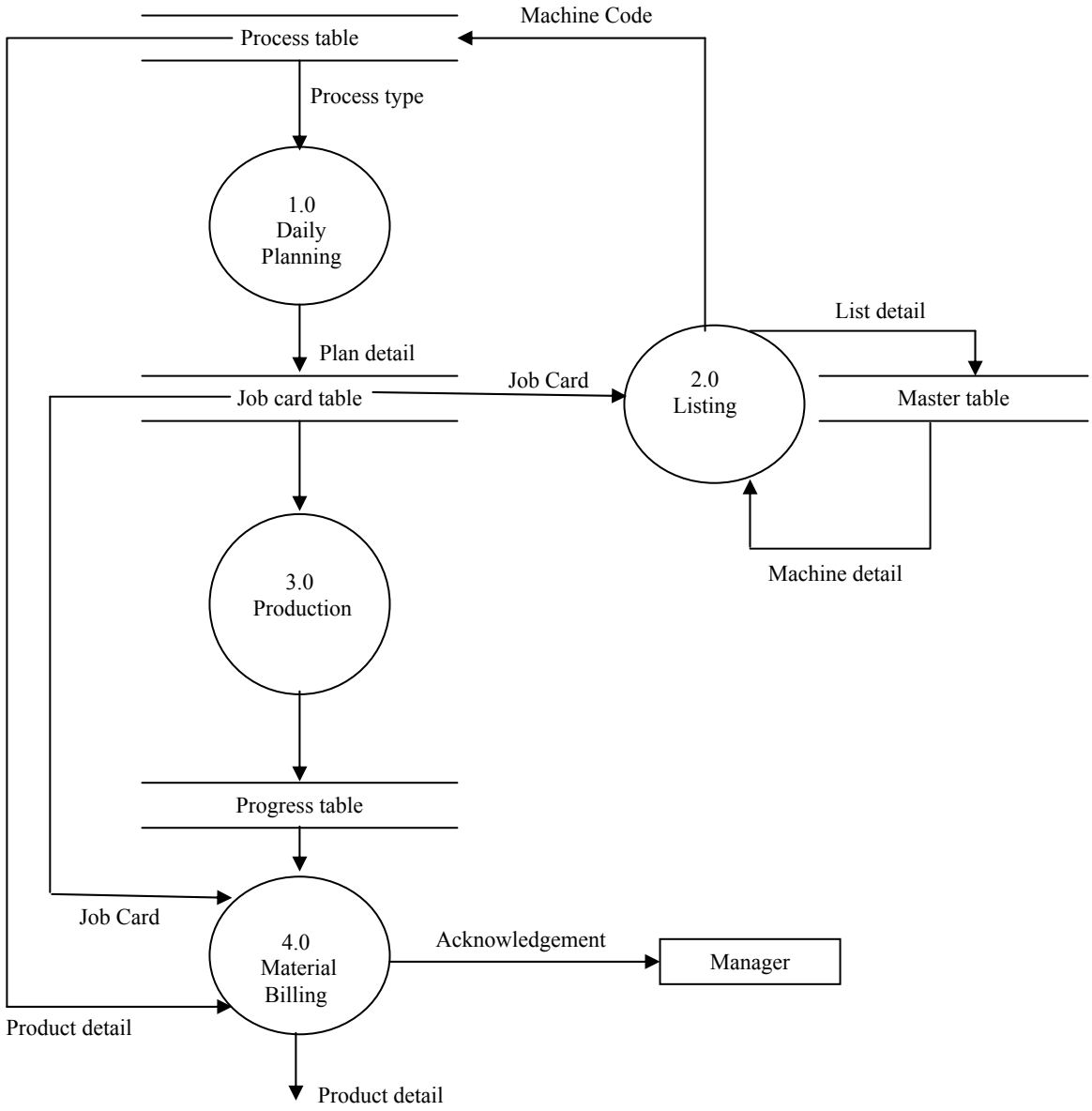


Figure 2.5 (b): Level 1 DFD of PMS

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.*

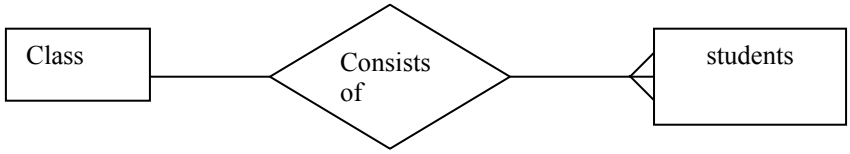


Figure 2.6: One to Many cardinality relationship

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.*

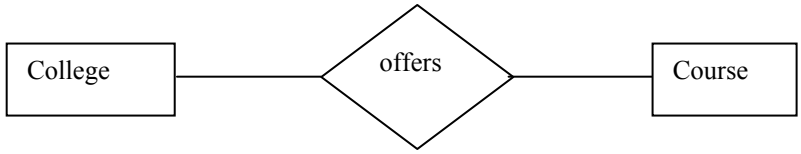


Figure 2.7: Minimum cardinality relationship

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.*

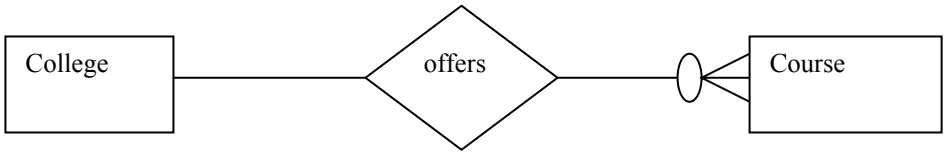


Figure 2.8: Modified ER diagram representing cardinalities

The relationship cardinalities are shown in *Figure 2.9.*

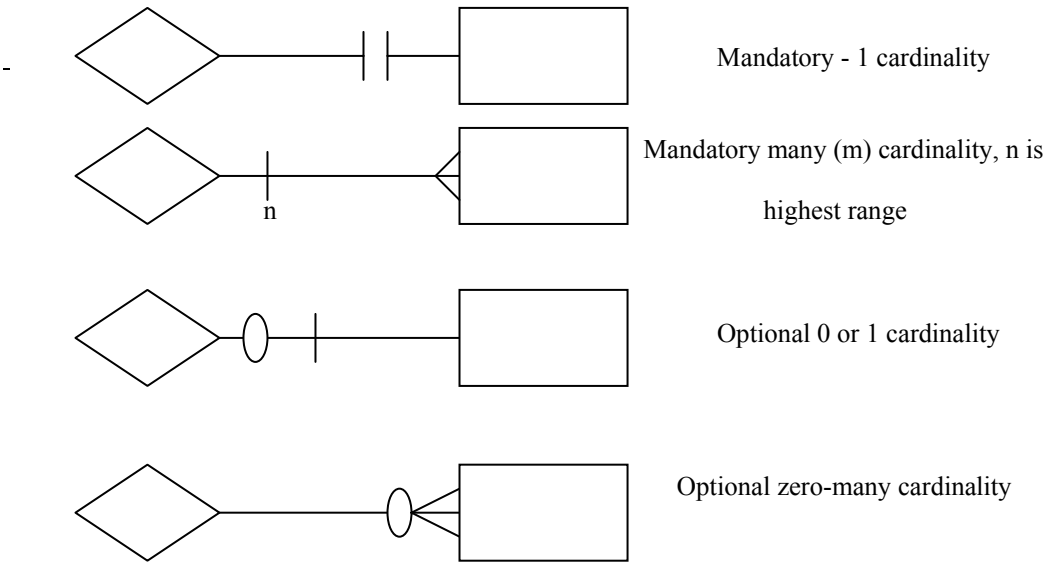


Figure 2.9: Relationship cardinalities

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.*

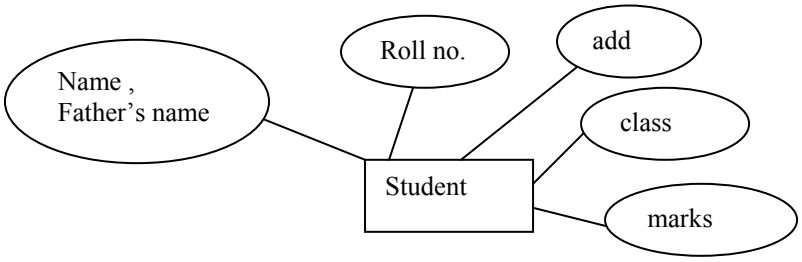


Figure 2.10: Attributes of entity *student*

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.*

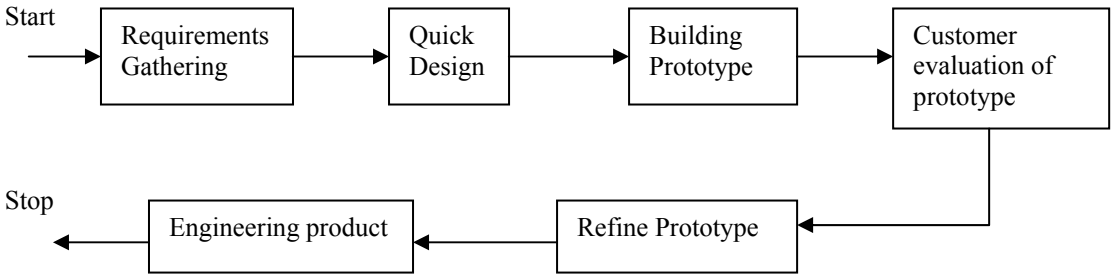


Figure 2.11: Sequence of prototyping

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.

……

……

……

2.6 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	39
3.1	Objectives	39
3.2	Data Design	39
3.3	Architectural Design	42
3.4	Modular Design	43
3.5	Interface Design	46
3.6	Design of Human Computer Interface	49
3.7	Summary	51
3.8	Solutions/ Answers	51
3.9	Further Readings	52

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.

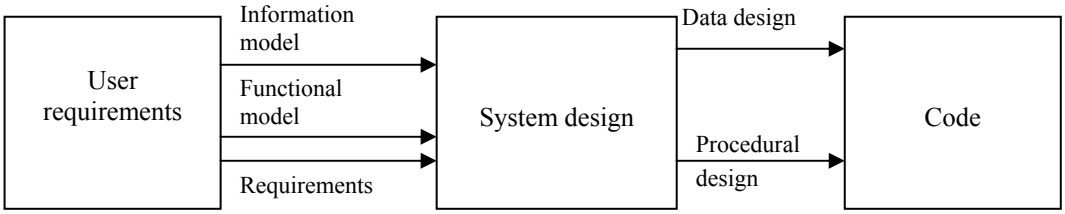


Figure 3.1 : Process of software Design

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.

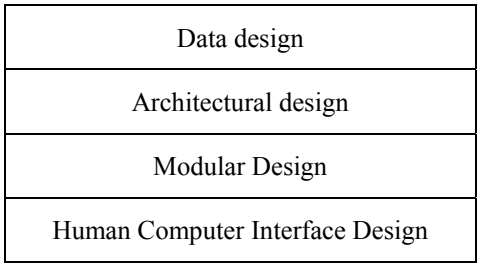


Figure 3.2 : Technical aspects of design

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*).

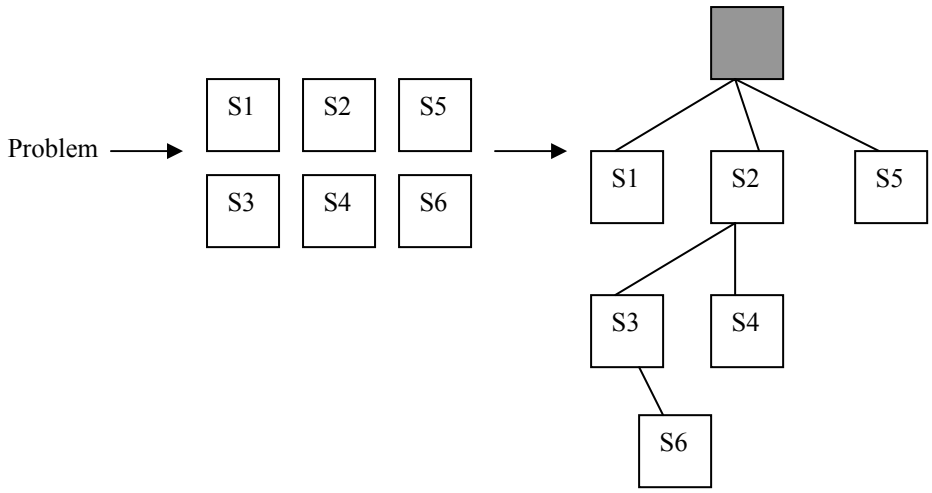


Figure 3.3: Problem, Solutions and Architecture

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.

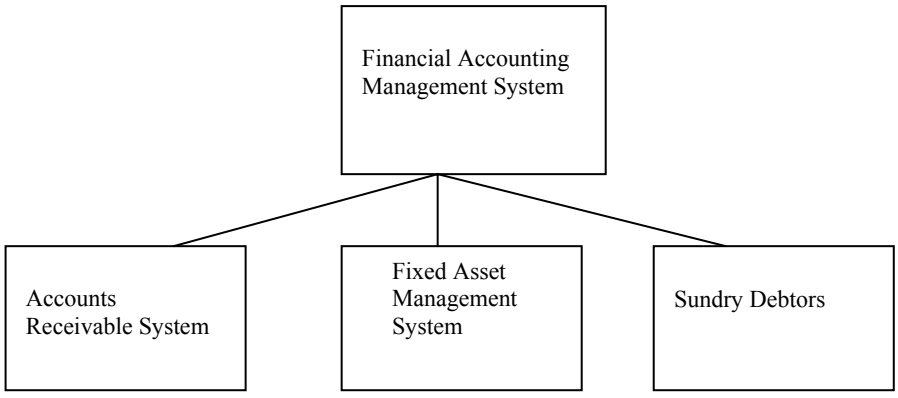


Figure 3.4: Architecture of a financial accounting system

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.

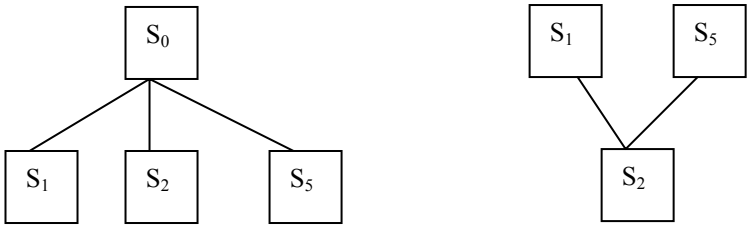


Figure 3.5: Fan-in and Fan-out

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*).

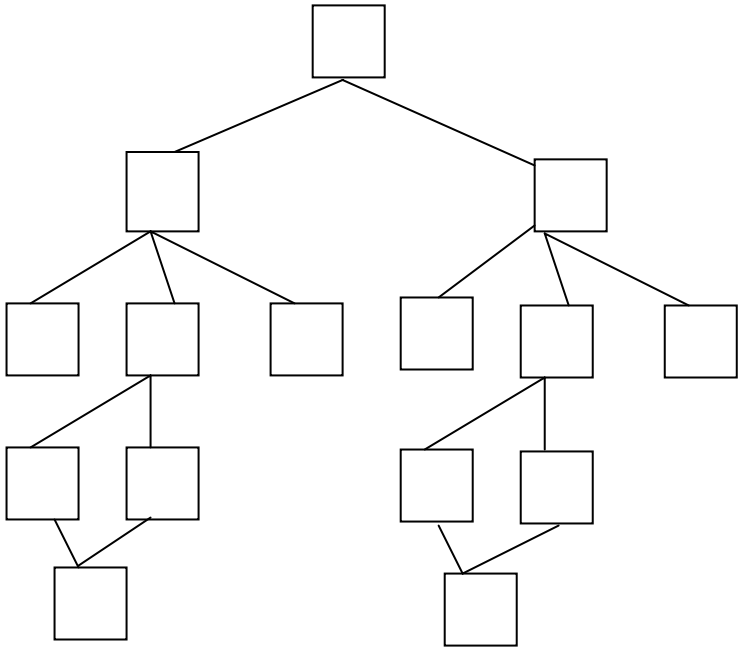


Figure 3.6 : Typical architecture of a system

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.

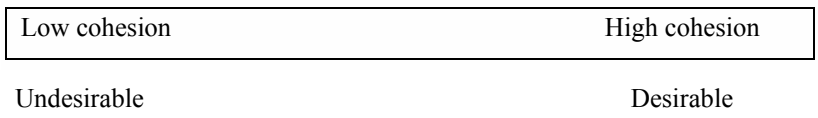


Figure 3.7 : Cohesion

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.

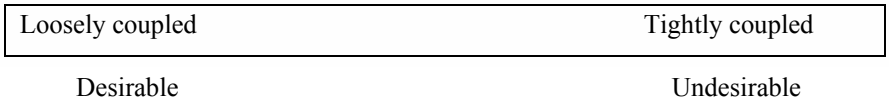


Figure 3.8 : Coupling

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.

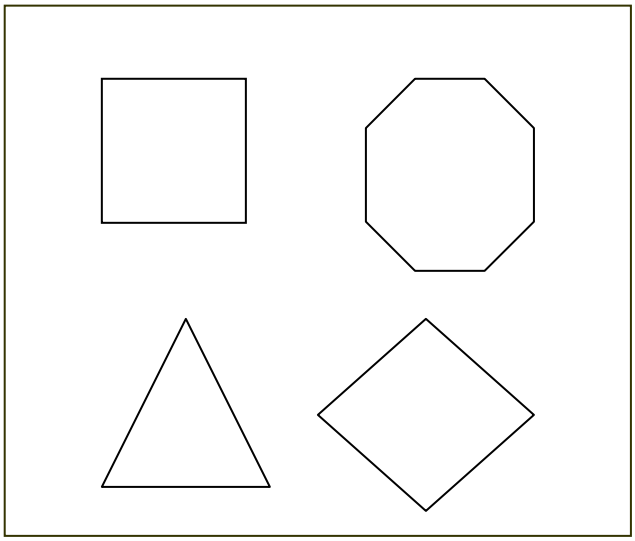


Figure 3.9 : High cohesive and loosely coupled system (desirable)

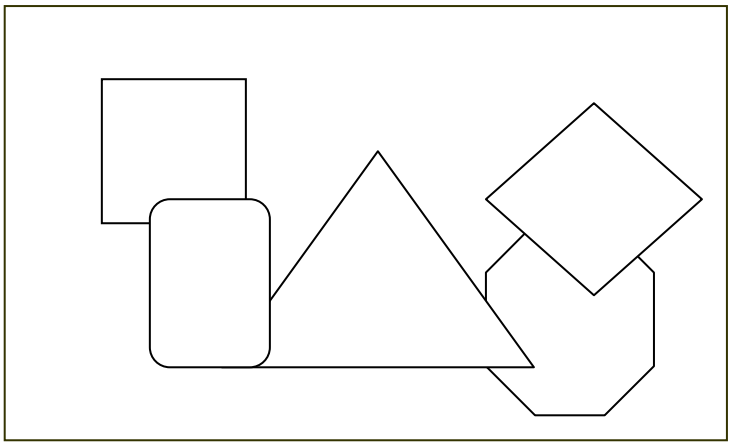


Figure 3.10 : Low cohesive and tightly coupled system (undesirable)

☞ 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).

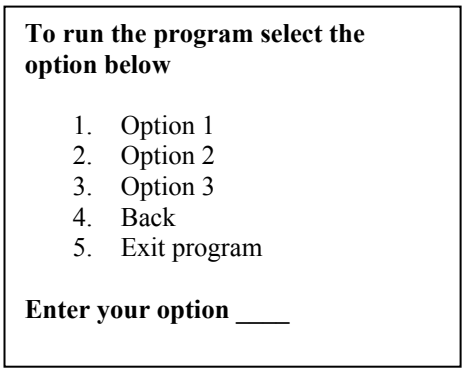


Figure 3.11 : A menu based interface

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

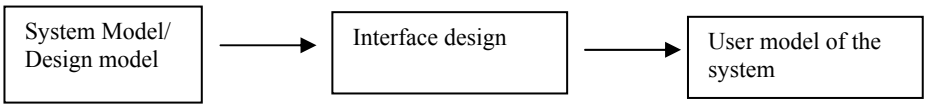


Figure 3.12: The process of interface design

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 :**
    - 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:**
    - 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 :**
    - The feedback should be informative and clear.
4. **Error prevention and handling common errors :**
    - 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
    - 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 :**
    - 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	53
4.1	Objectives	54
4.2	Basic Terms used in Testing	54
	4.2.1 Input Domain	
	4.2.2 Black Box and White Box testing Strategies	
	4.2.3 Cyclomatic Complexity	
4.3	Testing Activities	64
4.4	Debugging	65
4.5	Testing Tools	67
4.6	Summary	68
4.7	Solutions/Answers	69
4.8	Further Readings	69

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/ triangle
2	50	3	50	Isosceles triangle
3	40	50	40	Equilatera/ triangle
4	3	4	5	Right-angled triangles
5	10	10	10	Equilatera/ 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 be*" 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 *'cover*' 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.

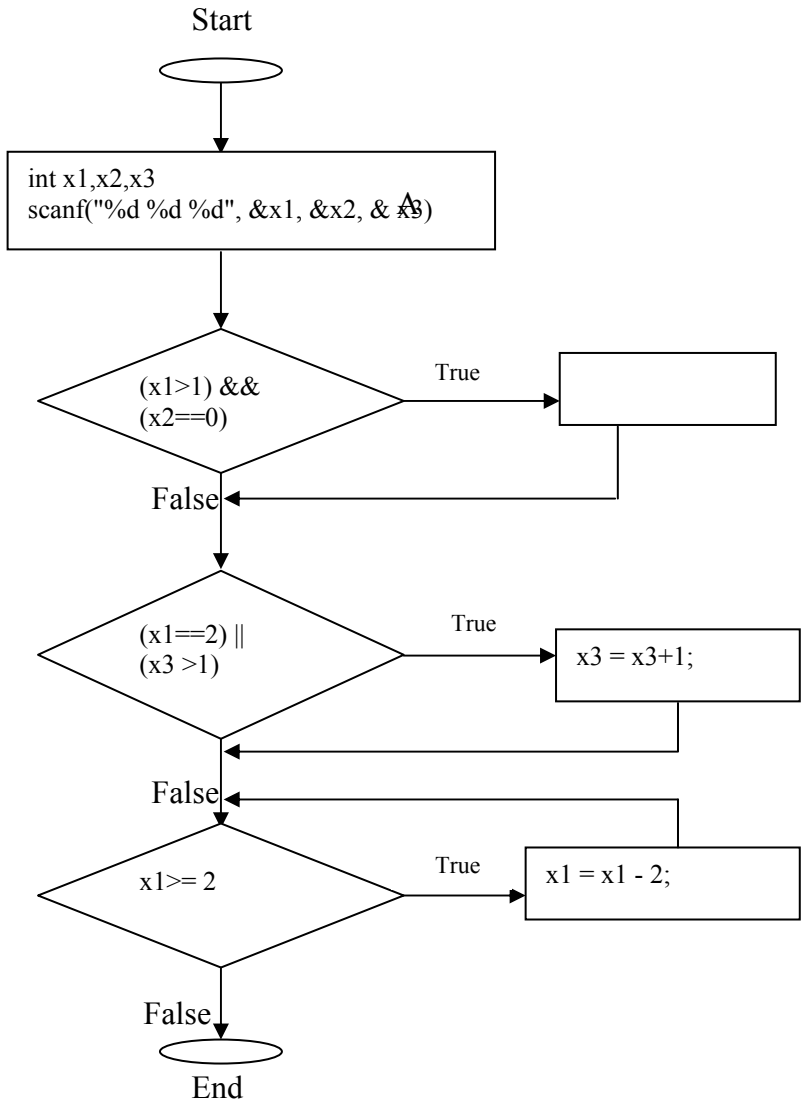


Figure 4.1: The flow chart for the program 4.1

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

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

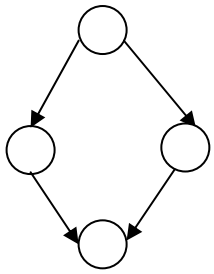


Figure 4.3 : if -else

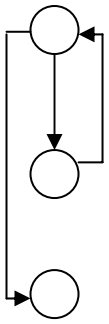


Figure 4.4 : while-loop

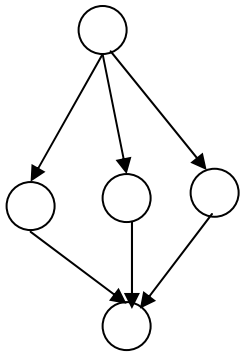


Figure 4.5: case

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*).

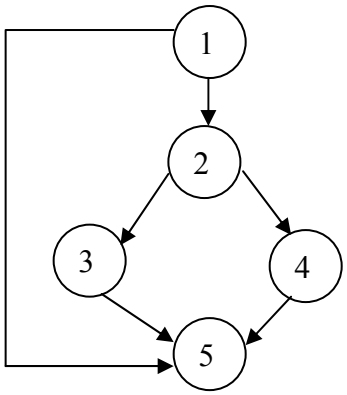


Figure 4.6: CFG for program 4.2

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:

$$\frac{\#\text{ killed Mutants}}{} \times 100$$

total mutants - # equivalent mutants

☞ 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.

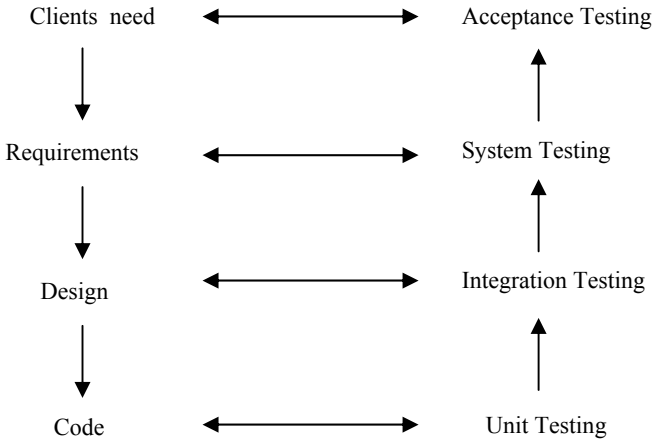


Figure 4.7: Testing levels

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

UNIT 1 SOFTWARE PROJECT PLANNING

Structure **Page Nos.**

1.0 Introduction 5
1.1 Objectives 5
1.2 Different Types of Project Metrics 5
1.3 Software Project Estimation 9
1.3.1 Estimating the Size
1.3.2 Estimating Effort
1.3.3 Estimating Schedule
1.3.4 Estimating Cost
1.4 Models for Estimation 13
1.4.1 COCOMO Model
1.4.2 Putnam's Model
1.4.3 Statistical Model
1.4.4 Function Points
1.5 Automated Tools for Estimation 15
1.6 Summary 17
1.7 Solutions/Answers 17
1.8 Further Readings 17

1.0 INTRODUCTION

Historically, software projects have dubious distinction of overshooting project schedule and cost. Estimating duration and cost continues to be a weak link in software project management. The aim of this unit is to give an overview of different project planning techniques and tools used by modern day software project managers.

It is the responsibility of the project manager to make as far as possible accurate estimations of effort and cost. This is particularly what is desired by the management of an organisation in a competitive world. This is specially true of projects subject to competition in the market where bidding too high compared with competitors would result in losing the business and a bidding too low could result in financial loss to the organisation. This makes software project estimation crucial for project managers.

1.1 OBJECTIVES

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

- understand different software estimation techniques;
- understand types of metrics used for software project estimation;
- learn about models used for software estimation, and
- learn about different automated tools used for estimation.

1.2 DIFFERENT TYPES OF PROJECT METRICS

It is said that if you cannot measure, then, it is not engineering. Effective management of software development process requires effective measurement of software development process. Often, from the input given by project leaders on the estimation of a software project, the management decides whether to proceed with the project or not.

The process of software estimation can be defined as the set of techniques and procedures that an organisation uses to arrive at an estimate. An important aspect of software projects is to know the cost, time, effort, size etc.

Need for Project metrics : Historically, the process of software development has been witnessing inaccurate estimations of schedule and cost, overshooting delivery target and productivity of software engineers in not commensurate with the growth of demand. Software development projects are quite complex and there was no scientific method of measuring the software process. Thus effective measurement of the process was virtually absent. The following phrase is aptly describing the need for measurement:

If you can not measure it, then, you can not improve it.

This is why measurement is very important to software projects. Without the process of measurement, software engineering cannot be called engineering in the true sense.

Definition of metrics : Metrics deal with measurement of the software process and the software product. Metrics quantify the characteristics of a process or a product. Merics are often used to estimate project cost and project schedule.

Examples of metrics : Lines of code(LOC), pages of documentation, number of man-months, number of test cases, number of input forms.

Types of Project metrics

Metrics can be broadly divided into two categories namely, product metrics and process metrics.

Product metrics provides a measure of a software product during the process of development which can be in terms of lines of code (either source code or object code), pages of documentation etc.

Process metrics is a measure of the software development process such as time, effort etc.

Another way of classification of metrics are primitive metrics and derived metrics.

Primitive metrics are directly observable quantities like lines of code (LOC), number of man-hours etc.

Derived metrics are derived from one or more of primitive metrics like lines of code per man-hour, errors per thousand lines of code.

Now, let us briefly discuss different types of product metrics and process metrics.

Product metrics

Lines of Code(LOC) : LOC metric is possibly the most extensively used for measurement of size of a program. The reason is that LOC can be precisely defined. LOC may include executable source code and non-executable program code like comments etc.

Looking at the following table, we can know the size of a module.

Module	**Effort in man-months**	**LOC**
Module 1	3	24,000
Module 2	4	25,000

Looking at the data above we have a direct measure of the size of the module in terms of LOC. We can derive a productivity metrics from the above primitive metrics i.e., LOC.

Productivity of a person = LOC / man-month

Quality = No. of defects / LOC

It is evident that the productivity of the developer engaged in Module 1 is more than the productivity of the developer engaged in Module 2. It is important to note here how derived metrics are very handy to project managers to measure various aspects of the projects.

Although, LOC provides a direct measure of program size, at the same time, these metrics are not universally accepted by project managers. Looking at the data in the table below, it can be easily observed that LOC is not an absolute measure of program size and largely depends on the computer language and tools used for development activity.

Consider the following table:

Module	**Effort in man-months**	**LOC in COBOL**	**LOC in Assembly**	**LOC in 4 GL**
Module- 1	3	24,000	800,000	400
Module- 2	4	25,000	100,000	500

The LOC of same module varies with the programming language used. Hence, just LOC cannot be an indicator of program size. The data given in the above table is only assumed and does not correspond to any module(s).

There are other attributes of software which are not directly reflected in Lines of Code (LOC), as the complexity of the program is not taken into account in LOC and it penalises well designed shorter program. Another disadvantage of LOC is that the project manager is supposed to estimate LOC before the analysis and design is complete.

Function point : Function point metrics instead of LOC measures the functionality of the program. Function point analysis was first developed by Allan J. Albrecht in the 1970s. It was one of the initiatives taken to overcome the problems associated with LOC.

In a Function point analysis, the following features are considered:

- **External inputs** : A process by which data crosses the boundary of the system. Data may be used to update one or more logical files. It may be noted that data here means either business or control information.

- **External outputs** : A process by which data crosses the boundary of the system to outside of the system. It can be a user report or a system log report.

- **External user inquires** : A count of the process in which both input and output results in data retrieval from the system. These are basically system inquiry processes.

- **Internal logical files** : A group of logically related data files that resides entirely within the boundary of the application software and is maintained through external input as described above.

- **External interface files** : A group of logically related data files that are used by the system for reference purposes only. These data files remain completely outside the application boundary and are maintained by external applications.

As we see, function points, unlike LOC, measure a system from a functional perspective independent of technology, computer language and development method. The number of function points of a system will remain the same irrespective of the language and technology used.

For transactions like external input, external output and user inquiry, the ranking of high, low and medium will be based on number of file updated for external inputs or number of files referenced for external input and external inquiries. The complexity will also depend on the number of data elements.

Consider the following classification for external inquiry:

No. of Data elements	**Number of file references**		
	0 to 2	**3 to 4**	**5 and above**
1 to 5	Low	Low	Low
6 to 10	Low	Medium	Medium
10 to 20	Medium	Medium	High
More than 20	Medium	High	High

Also, External Inquiry, External Input and External output based on complexity can be assigned numerical values like rating.

Component type	**Values**		
	Low	**Medium**	**High**
External Output	4	6	8
External Input	2	4	6
External Inquiry	3	5	7

Similarly, external logical files and external interface files are assigned numerical values depending on element type and number of data elements.

Component type	**Values**		
	Low	**Medium**	**High**
External Logical files	6	8	10
External Interface	5	7	9

Organisations may develop their own strategy to assign values to various function points. Once the number of function points have been identified and their significance has been arrived at, the total function point can be calculated as follows.

Type of Component	**Complexity of component**			
	Low	**Medium**	**High**	**Total**
Number of External Output	X 4 =	X 6 =	X 8 =	
Number of External Input	X 2 =	X 4 =	X 6 =	
Number of External Inquiry	X3 =	X 5 =	X 7 =	
Number of Logical files	X 6 =	X 8 =	X 10 =	
Number of Interface file	X 5 =	X 7 =	X 9 =	
			Total of function point	

Total of function points is calculated based on the above table. Once, total of function points is calculated, other derived metrics can be calculated as follows:

Productivity of Person = Total of Function point / man-month
Quality = No. of defects / Total of Function points.

Benefits of Using Function Points

- Function points can be used to estimate the size of a software application correctly irrespective of technology, language and development methodology.
- User understands the basis on which the size of software is calculated as these are derived directly from user required functionalities.
- Function points can be used to track and monitor projects.
- Function points can be calculated at various stages of software development process and can be compared.

Other types of metrics used for various purposes are quality metrics which include the following:

- **Defect metrics** : It measures the number of defects in a software product. This may include the number of design changes required, number of errors detected in the test, etc.

- **Reliability metrics** : These metrics measure mean time to failure. This can be done by collecting data over a period of time.

☞ **Check Your Progress 1**

1) Lines of Code (LOC) is a product metric. True ☐ False ☐

2) _____ are examples of process metrics.

1.3 SOFTWARE PROJECT ESTIMATION

Software project estimation is the process of estimating various resources required for the completion of a project. Effective software project estimation is an important activity in any software development project. Underestimating software project and under staffing it often leads to low quality deliverables, and the project misses the target deadline leading to customer dissatisfaction and loss of credibility to the company. On the other hand, overstaffing a project without proper control will increase the cost of the project and reduce the competitiveness of the company.

Software project estimation mainly encompasses the following steps:

- Estimating the size of project. There are many procedures available for estimating the size of a project which are based on quantitative approaches like estimating Lines of Code or estimating the functionality requirements of the project called Function point.

- Estimating efforts based on man-month or man-hour. Man-month is an estimate of personal resources required for the project.

- Estimating schedule in calendar days/month/year based on total man-month required and manpower allocated to the project

  Duration in calendar month = Total man-months / Total manpower allocated.

- Estimating total cost of the project depending on the above and other resources.

In a commercial and competitive environment, Software project estimation is crucial for managerial decision making. The following *Table* give the relationship between various management functions and software metrics/indicators. Project estimation and tracking help to plan and predict future projects and provide baseline support for project management and supports decision making.

Consider the following table:

Activity	Tasks involved
Planning	Cost estimation, planning for training of manpower, project scheduling and budgeting the project
Controlling	Size metrics and schedule metrics help the manager to keep control of the project during execution
Monitoring/improving	Metrics are used to monitor progress of the project and wherever possible sufficient resources are allocated to improve.

Figure 1.1 depicts the software project estimation.

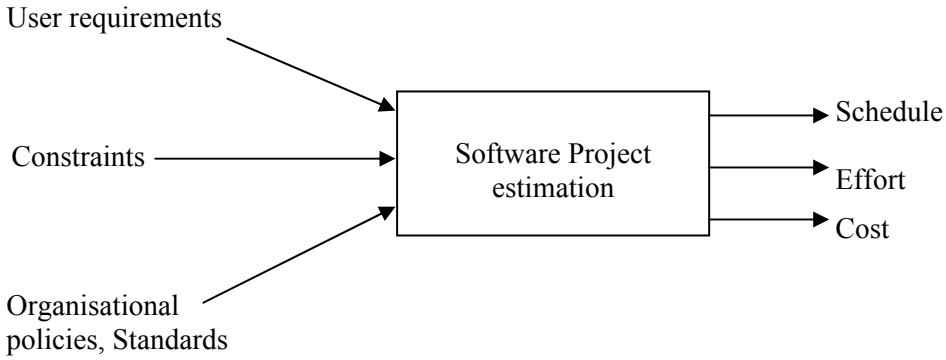


Figure 1.1: Software project estimation

1.3.1 Estimating the size

Estimating the size of the software to be developed is the very first step to make an effective estimation of the project. Customer's requirements and system specification forms a baseline for estimating the size of a software. At a later stage of the project, system design document can provide additional details for estimating the overall size of a software.

- The ways to estimate project size can be through past data from an earlier developed system. This is called estimation by analogy.
- The other way of estimation is through product feature/functionality. The system is divided into several subsystems depending on functionality, and size of each subsystem is calculated.

1.3.2 Estimating effort

Once the size of software is estimated, the next step is to estimate the effort based on the size. The estimation of effort can be made from the organisational specifics of software development life cycle. The development of any application software system is more than just coding of the system. Depending on deliverable requirements, the estimation of effort for project will vary. Efforts are estimated in number of man-months.

- The best way to estimate effort is based on the organisation's own historical data of development process. Organizations follow similar development life cycle for developing various applications.
- If the project is of a different nature which requires the organisation to adopt a different strategy for development then different models based on algorithmic approach can be devised to estimate effort.

1.3.3 Estimating Schedule

The next step in estimation process is estimating the project schedule from the effort estimated. The schedule for a project will generally depend on human resources involved in a process. Efforts in man-months are translated to calendar months.

Schedule estimation in calendar-month can be calculated using the following model [McConnell]:

Schedule in calendar months = 3.0 * (man-months) $^{1/3}$

The parameter 3.0 is variable, used depending on the situation which works best for the organisation.

1.3.4 Estimating Cost

Cost estimation is the next step for projects. The cost of a project is derived not only from the estimates of effort and size but from other parameters such as hardware, travel expenses, telecommunication costs, training cost etc. should also be taken into account.

Figure 1.2 depicts the cost estimation process.

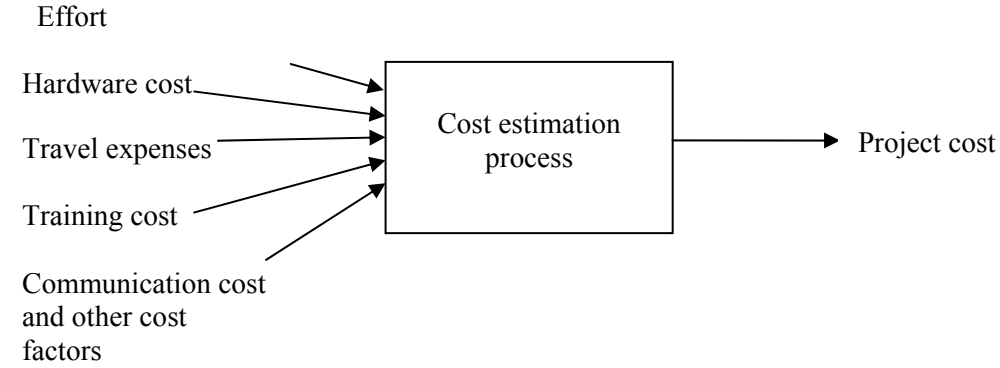


Figure 1.2 : Cost estimation process

Figure 1.3 depicts project estimation process.

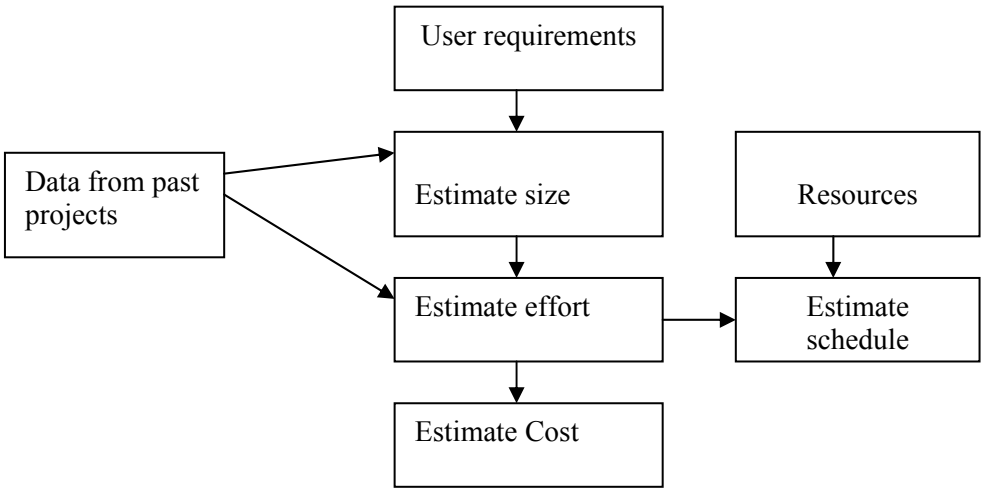


Figure 1.3: Project estimation process

Now, once the estimation is complete, we may be interested to know how accurate the estimates are to reality. The answer to this is "we do not know until the project is complete". There is always some uncertainty associated with all estimation techniques. The accuracy of project estimation will depend on the following:

- Accuracy of historical data used to project the estimation
- Accuracy of input data to various estimates
- Maturity of organisation's software development process.

The following are some of the reasons which make the task of cost estimation difficult:

- Software cost estimation requires a significant amount of effort. Sufficient time is not allocated for planning.
- Software cost estimation is often done hurriedly, without an appreciation for the actual effort required and is far from real issues.
- Lack of experience for developing estimates, especially for large projects.
- An estimator uses the extrapolation technique to estimate, ignoring the non-linear aspects of software development process

The following are some of the reasons for poor and inaccurate estimation:

- Requirements are imprecise. Also, requirements change frequently.
- The project is new and is different from past projects handled.
- Non-availability of enough information about past projects.
- Estimates are forced to be based on available resources.

Cost and time tradeoffs

If we elongate the project, we can reduce overall cost. Usually, long project durations are not liked by customers and managements. There is always shortest possible duration for a project, but it comes at a cost.

The following are some of the problems with estimates:

- Estimating size is often skipped and a schedule is estimated which is of more relevance to the management.
- Estimating size is perhaps the most difficult step which has a bearing on all other estimates.
- Let us not forget that even good estimates are only projections and subject to various risks.
- Organisations often give less importance to collection and analysis of historical data of past development projects. Historical data is the best input to estimate a new project.
- Project managers often underestimate the schedule because management and customers often hesitate to accept a prudent realistic schedule.

Project estimation guidelines

- Preserve and document data pertaining to organisation's past projects.
- Allow sufficient time for project estimation especially for bigger projects.
- Prepare realistic developer-based estimate. Associate people who will work on the project to reach at a realistic and more accurate estimate.
- Use software estimation tools.
- Re-estimate the project during the life cycle of development process.
- Analyse past mistakes in the estimation of projects.

☞ **Check Your Progress 2**

1) What is the first step in software project estimation?

……

2) What are the major inputs for software project estimation?

……

1.4 MODELS FOR ESTIMATION

Estimation based on models allows us to estimate projects ignoring less significant parameters and concentrating on crucial parameters that drive the project estimate. Models are analytic and empirical in nature. The estimation models are based on the following relationship:

$$E = f(v_i)$$

E = different project estimates like effort, cost, schedule etc.

v_i = directly observable parameter like LOC, function points

1.4.1 COCOMO Model

COCOMO stands for Constructive Cost Model. It was introduced by Barry Boehm. It is perhaps the best known and most thoroughly documented of all software cost estimation models. It provides the following three level of models:

- **Basic COCOMO** : A single-value model that computes software development cost as a function of estimate of LOC.
- **Intermediate COCOMO** : This model computes development cost and effort as a function of program size (LOC) and a set of cost drivers.
- **Detailed COCOMO** : This model computes development effort and cost which incorporates all characteristics of intermediate level with assessment of cost implication on each step of development (analysis, design, testing etc.).

This model may be applied to three classes of software projects as given below:

- Organic : Small size project. A simple software project where the development team has good experience of the application
- Semi-detached : An intermediate size project and project is based on rigid and semi-rigid requirements.
- Embedded : The project developed under hardware, software and operational constraints. Examples are embedded software, flight control software.

In the COCOMO model, the development effort equation assumes the following form:

$$E = aS^b m$$

where **a** and **b** are constraints that are determined for each model.

E = Effort
S = Value of source in LOC
m = multiplier that is determined from a set of 15 cost driver's attributes.

The following are few examples of the above cost drivers:

- Size of the application database
- Complexity of the project
- Reliability requirements for the software
- Performance constraints in run-time
- Capability of software engineer
- Schedule constraints.

Barry Boehm suggested that a detailed model would provide a cost estimate to the accuracy of ± 20 % of actual value

1.4.2 Putnam's model

L. H. Putnam developed a dynamic multivariate model of the software development process based on the assumption that distribution of effort over the life of software development is described by the Rayleigh-Norden curve.

$$P = Kt \exp(t^2/2T^2) / T^2$$

P = No. of persons on the project at time 't'
K = The area under Rayleigh curve which is equal to total life cycle effort
T = Development time

The Rayleigh-Norden curve is used to derive an equation that relates lines of code delivered to other parameters like development time and effort at any time during the project.

$$S = C_k K^{1/3} T^{4/3}$$

S = Number of delivered lines of source code (LOC)
C_k = State-of-technology constraints
K = The life cycle effort in man-years
T = Development time.

1.4.3 Statistical Model

From the data of a number of completed software projects, C.E. Walston and C.P. Felix developed a simple empirical model of software development effort with respect to number of lines of code. In this model, LOC is assumed to be directly related to development effort as given below:

$$E = a\ L^b$$

Where L = Number of Lines of Code (LOC)

E = total effort required

a and **b** are parameters obtained from regression analysis of data. The final equation is of the following form:

$$E = 5.2\ L^{0.91}$$

The productivity of programming effort can be calculated as

$$P = L/E$$

Where P = Productivity Index

1.4.4 Function Points

It may be noted that COCOMO, Putnam and statistical models are based on LOC. A number of estimation models are based on function points instead of LOC. However, there is very little published information on estimation models based on function points.

1.5 AUTOMATED TOOLS FOR ESTIMATION

After looking at the above models for software project estimation, we have reason to think of software that implements these models. This is what exactly the automated estimation tools do. These estimation tools, which estimate cost and effort, allow the project managers to perform "What if analysis". Estimation tools may only support size estimation or conversion of size to effort and schedule to cost.

There are more than dozens of estimation tools available. But, all of them have the following common characteristics:

- Quantitative estimates of project size (e.g., LOC).
- Estimates such as project schedule, cost.
- Most of them use different models for estimation.

Figure 1.4 depicts a typical structure of estimation tools.

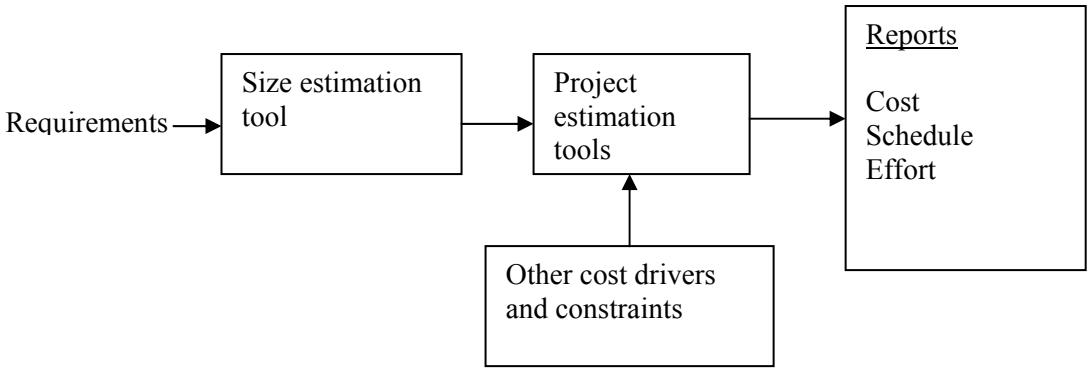


Figure 1.4: Typical structure of estimation tools

No estimation tool is the solution to all estimation problems. One must understand that the tools are just to help the estimation process.

Problems with Models

Most models require an estimate of software product size. However, software size is difficult to predict early in the development lifecycle. Many models use LOC for sizing, which is not measurable during requirements analysis or project planning. Although, function points and object points can be used earlier in the lifecycle, these measures are extremely subjective.

Size estimates can also be very inaccurate. Methods of estimation and data collection must be consistent to ensure an accurate prediction of product size. Unless the size metrics used in the model are the same as those used in practice, the model will not yield accurate results (Fenton, 1997).

The following *Table* gives some estimation tools:

Automated Estimation Tools

Tool	Tool vendor site	Functionality/Remark
EstimatorPal	http://www.effortestimator.com/	EstimatorPal© is a software tool that assists software developers estimate the effort required to be spent on various activities. This tool facilitates use of the following Estimation techniques – Effort estimation tools which supports Function Point Analysis Technique., Objects Points Technique, Use Case Points Technique and Task-Based Estimation Technique
Estimate Easy Use Case	Duvessa Software http://www.duvessa.com/	Effort estimation tool based on use cases
USC COCOMO II	USC Center for Software Engineering http://sunset.usc.edu/research/COCOMOII/index.html	Based on COCOMO
ESTIMACS	Computer Associates International Inc. http://www.cai.com/products/estimacs.htm	Provides estimates of the effort, duration, cost and personnel requirements for maintenance and new application development projects.
Checkpoint	Software Productivity Research Inc. http://www.spr.com/	Guides the user through the development of a software project estimate and plan.
Function Point Workbench	Software Productivity Research Inc. http://www.spr.com/	Automated software estimation tools, implementing function point sizing techniques, linking project estimation to project management initiatives, and collecting historical project data to improve future estimation efforts
ESTIMATE Professional	Software Productivity Center http://www.spc.com	Based on Putnam, COCOMO II
SEER-SEM	Galorath http://www.galorath.com/	Predicts, measures and analyzes resources, staffing, schedules, risk and cost for software projects
ePM.Ensemble	InventX http://www.inventx.com/	Support effort estimation, among other things.
CostXpert	Marotz, Inc. http://www.costxpert.com/	Based on COCOMO

1.6 SUMMARY

Estimation is an integral part of the software development process and should not be taken lightly. A well planned and well estimated project is likely to be completed in time. Incomplete and inaccurate documentation may pose serious hurdles to the success of a software project during development and implementation. Software cost estimation is an important part of the software development process. Metrics are important tools to measure software product and process. Metrics are to be selected carefully so that they provide a measure for the intended process/product. Models are used to represent the relationship between effort and a primary cost factor such as software product size. Cost drivers are used to adjust the preliminary estimate provided by the primary cost factor. Models have been developed to predict software cost based on empirical data available, but many suffer from some common problems. The structure of most models is based on empirical results rather than theory. Models are often complex and rely heavily on size estimation. Despite problems, models are still important to the software development process. A model can be used most effectively to supplement and corroborate other methods of estimation.

1.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1. True
2. Time, Schedule

Check Your Progress 2

1. Estimating size of the Project
2. User requirements, and project constraints.

1.8 FURTHER READINGS

1) *Software Engineering,* Ian Sommerville; *Sixth Edition, 2001*, 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://www.ncst.ernet.in

UNIT 2 RISK MANAGEMENT AND PROJECT SCHEDULING

Structure **Page Nos.**

2.0 Introduction 18
2.1 Objectives 18
2.2 Identification of Software Risks 18
2.3 Monitoring of Risks 20
2.4 Management of Risks 20
2.4.1 Risk Management
2.4.2 Risk Avoidance
2.4.3 Risk Detection
2.5 Risk Control 22
2.6 Risk Recovery 23
2.7 Formulating a Task Set for the Project 24
2.8 Choosing the Tasks of Software Engineering 24
2.9 Scheduling Methods 25
2.10 The Software Project Plan 27
2.11 Summary 28
2.12 Solutions/Answers 28
2.13 Further Readings 30

2.0 INTRODUCTION

As human beings, we would like life to be free from dangers, difficulties and any risks of any type. In case a risk arises, we would take proper measures to recover as soon as possible. Similarly, in software engineering, risk management plays an important role for successful deployment of the software product. Risk management involves monitoring of risks, taking necessary actions in case risk arises by applying risk recovery methods.

2.1 OBJECTIVES

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

- know the meaning of risk;
- identify risks; and
- manage the risks

2.2 IDENTIFICATION OF SOFTWARE RISKS

A risk may be defined as a potential problem. It may or may not occur. But, it should always be assumed that it may occur and necessary steps are to be taken.
Risks can arise from various factors like improper technical knowledge or lack of communication between team members, lack of knowledge about software products, market status, hardware resources, competing software companies, etc.

Basis for Different Types of Software risks

- **Skills or Knowledge**: The persons involved in activities of problem analysis, design, coding and testing have to be fully aware of the activities and various

techniques at each phase of the software development cycle. In case, they have partial knowledge or lacks adequate skill, the products may face many risks at the current stage of development or at later stages.

- **Interface modules**: Complete software contains various modules and each module sends and receives information to other modules and their concerned data types have to match.

- **Poor knowledge of tools**: If the team or individual members have poor knowledge of tools used in the software product, then the final product will have many risks, since it is not thoroughly tested.

- **Programming Skills:** The code developed has to be efficient, thereby, occupying less memory space and less CPU cycles to compute given task. The software product should be able to implement various object oriented techniques and be able to catch exceptions in case of errors. Various data values have to be checked and in case of improper values, appropriate messages have to be displayed. If this is not done, then it leads to risk, thereby creating panic in the software computations.

- **Management Issues** : The management of the organisation should give proper training to the project staff, arrange some recreation activities, give bonus and promotions and interact with all members of the project and try to solve their necessities at the best. It should take care that team members and the project manager have healthy coordination, and in case there are some differences they should solve or make minor shuffles.

- **Updates in the hardware resources:** The team should be aware of the latest updates in the hardware resources, such as latest CPU (Intel P4, Motorola series, etc.), peripherals, etc. In case the developer makes a product, and later in the market, a new product is released, the product should support minimum features. Otherwise, it is considered a risk, and may lead to the failure of the project.

- **Extra support:** The software should be able to support a set of a few extra features in the vicinity of the product to be developed.

- **Customer Risks:** Customer should have proper knowledge of the product needed, and should not be in a hurry to get the work done. He should take care that all the features are implemented and tested. He should take the help of a few external personnel as needed to test the product and should arrange for demonstrations with a set of technical and managerial persons from his office.

- **External Risks:** The software should have backup in CD, tapes, etc., fully encrypted with full licence facilities. The software can be stored at various important locations to avoid any external calamities like floods, earthquakes, etc. Encryption is maintained such that no external persons from the team can tap the source code.

- **Commercial Risks:** The organisation should be aware of various competing vendors in the market and various risks involved if their product is not delivered on time. They should have statistics of projects and risks involved from their previous experience and should have skilled personnel.

2.3 MONITORING OF RISKS

Various risks are identified and a risk monitor table with attributes like risk name, module name, team members involved, lines of code, codes affecting this risk, hardware resources, etc., is maintained. If the project is continued further to 2-3 weeks, and then further the risk table is also updated. It is seen whether there is a ripple effect in the table, due to the continuity of old risks. Risk monitors can change the ordering of risks to make the table easy for computation. *Table 2.1* depicts a risk table monitor. It depicts the risks that are being monitored.

Table 2.1: Risk table monitor

Sl. No.	Risk Name	Week 1	Week 2	Remarks
1.	Module compute()	Line 5, 8, 20	Line 5,25	Priority 3
2.	More memory and peripherals	Module f1(), f5() affected	Module f2() affected	Priority 1
.....	.....	.....	.....	.....

The above risk table monitor has a risk in module compute () where there is a risk in line 5, 8 and 20 in week 1. In week 2, risks are present in lines 5 and 25. Risks are reduced in week 2. The priority 3 is set. Similarly, in the second row, risk is due to more memory and peripherals, affecting module f1 (), f5 () in week-1. After some modifications in week 2, module f2 () is affected and the priority is set to 1.

☞ Check Your Progress 1

1) Define the term risk and how it is related to software engineering.

……

……

2) List at least two risks involved with the management of team members.

……

……

3) What are commercial risks?

……

……

4) What do you mean by monitoring of risks and describe risk table.

……

……

5) Mention any two ways to prioritise risks.

……

……

2.4 MANAGEMENT OF RISKS

Risk management plays an important role in ensuring that the software product is error free. Firstly, risk management takes care that the risk is avoided, and if it not avoidable, then the risk is detected, controlled and finally recovered.

The flow of risk management is as follows:

2.4.1 Risk Management

A priority is given to risk and the highest priority risk is handled first. Various factors of the risk are who are the involved team members, what hardware and software items are needed, where, when and why are resolved during risk management. The risk manager does scheduling of risks. Risk management can be further categorised as follows:

1. Risk Avoidance
   a. Risk anticipation
   b. Risk tools
2. Risk Detection
   a. Risk analysis
   b. Risk category
   c. Risk prioritisation
3. Risk Control
   a. Risk pending
   b. Risk resolution
   c. Risk not solvable
4. Risk Recovery
   a. Full
   b. Partial
   c. Extra/alternate features

Figure 2.1 depicts a risk manager tool.

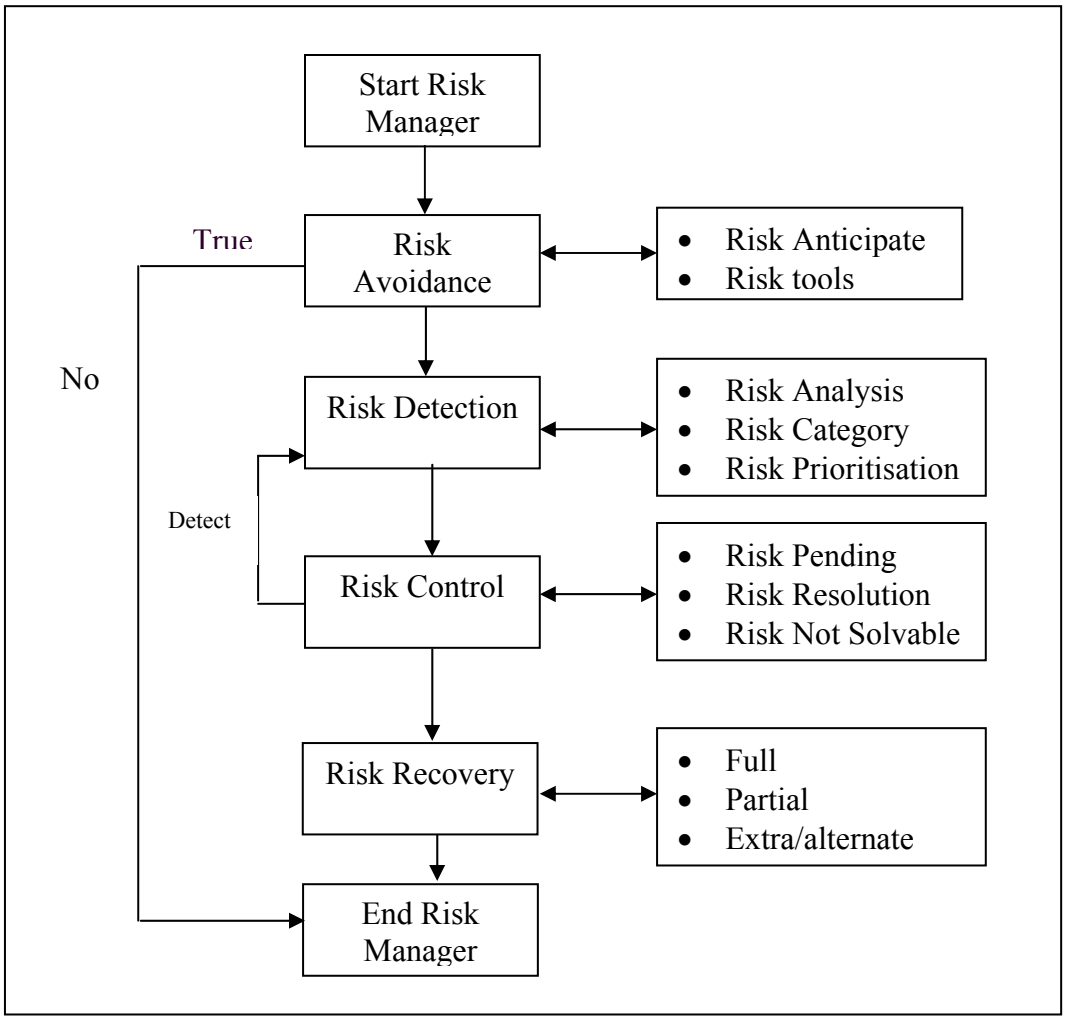


Figure 2.1 : Risk Manager Tool

From the *Figure 2.1*, it is clear that the first phase is to avoid risk by anticipating and using tools from previous project history. In case there is no risk, risk manager halts. In case there is risk, detection is done using various risk analysis techniques and further prioritising risks. In the next phase, risk is controlled by pending risks, resolving risks and in the worst case (if risk is not solved) lowering the priority. Lastly, risk recovery is done fully, partially or an alternate solution is found.

2.4.2 Risk Avoidance

Risk Anticipation: Various risk anticipation rules are listed according to standards from previous projects' experience, and also as mentioned by the project manager.

Risk tools: Risk tools are used to test whether the software is risk free. The tools have built-in data base of available risk areas and can be updated depending upon the type of project.

2.4.3 Risk Detection

The risk detection algorithm detects a risk and it can be categorically stated as :

Risk Analysis: In this phase, the risk is analyzed with various hardware and software parameters as probabilistic occurrence (*pr*), weight factor (*wf)* (hardware resources, lines of code, persons), risk exposure (*pr * wf*).

Table 2.1 depicts a risk analysis table.

Table 2.1: Risk analysis table

Sl.No.	Risk Name	Probability of occurrence (pr)	Weight factor (wf)	Risk exposure (pr * wf)
1.	Stack overflow	5	15	75
2.	No Password forgot option	7	20	140
....	.....	.....	.....	.....

Maximum value of risk exposure indicates that the problem has to solved as soon as possible and be given high priority. A risk analysis table is maintained as shown above.

Risk Category: Risk identification can be from various factors like persons involved in the team, management issues, customer specification and feedback, environment, commercial, technology, etc. Once proper category is identified, priority is given depending upon the urgency of the product.

Risk Prioritisation: Depending upon the entries of the risk analysis table, the maximum risk exposure is given high priority and has to be solved first.

2.5 RISK CONTROL

Once the prioritisation is done, the next step is to control various risks as follows:

- **Risk Pending***:* According to the priority, low priority risks are pushed at the end of the queue with a view of various resources (hardware, man power, software) and in case it takes more time their priority is made higher.

- **Risk Resolution**: Risk manager makes a strong resolve how to solve the risk.
- **Risk elimination**: This action leads to serious error in software.
- **Risk transfer***:* If the risk is transferred to some part of the module, then risk analysis table entries get modified. Thereby, again risk manager will control high priority risk.
- **Disclosures***:* Announce the risk of less priority to the customer or display message box as a warning. And thereby the risk is left out to the user, such that he should take proper steps during data entry, etc.
- **Risk not solvable***:* If a risk takes more time and more resources, then it is dealt in its totality in the business aspect of the organisation and thereby it is notified to the customer, and the team member proposes an alternate solution. There is a slight variation in the customer specification after consultation.

2.6 RISK RECOVERY

Full : The risk analysis table is scanned and if the risk is fully solved, then corresponding entry is deleted from the table.

Partial : The risk analysis table is scanned and due to partially solved risks, the entries in the table are updated and thereby priorities are also updated.

Extra/alternate features : Sometimes it is difficult to remove some risks, and in that case, we can add a few extra features, which solves the problem. Therefore, a bit of coding is done to get away from the risk. This is later documented or notified to the customer.

☞ Check Your Progress 2

1) Define the term risk management

……

……

2) What are various phases of risk manager?

……

……

3) What are the attributes mentioned in risk analysis table?

……

……

4) What is meant by risk resolution?

……

……

5) Why do we add extra features to recover from risks?

……

……

2.7 FORMULATING A TASK SET FOR THE PROJECT

The objective of this section is to get an insight into project scheduling by defining various task sets dependent on the project and choosing proper tasks for software engineering.

Various static and dynamic scheduling methods are also discussed for proper implementation of the project.

Factors affecting the task set for the project

- **Technical staff expertise**: All staff members should have sufficient technical expertise for timely implementation of the project. Meetings have to be conducted, weekly and status reports are to be generated.
- **Customer satisfaction** : Customer has to be given timely information regarding the status of the project. If not, there might be a communication gap between the customer and the organisation.
- **Technology update** : Latest tools and existing tested modules have to be used for fast and efficient implementation of the project.
- **Full or partial implementation of the project** : In case, the project is very large and to meet the market requirements, the organisation has to satisfy the customer with at least a few modules. The remaining modules can be delivered at a later stage.
- **Time allocation** : The project has to be divided into various phases and time for each phase has to be given in terms of person-months, module-months, etc.
- **Module binding** : Module has to bind to various technical staff for design, implementation and testing phases. Their necessary inter-dependencies have to be mentioned in a flow chart.
- **Milestones** : The outcome for each phase has to be mentioned in terms of quality, specifications implemented, limitations of the module and latest updates that can be implemented (according to the market strategy).
- **Validation and Verification** : The number of modules verified according to customer specification and the number of modules validated according to customer's expectations are to be specified.

2.8 CHOOSING THE TASKS OF SOFTWARE ENGINEERING

Once the task set has been defined, the next step is to choose the tasks for software project. Depending upon the software process model like linear sequential, iterative, evolutionary model etc., the corresponding task is selected. From the above task set, let us consider how to choose tasks for project development (as an example) as follows:

- **Scope** : Overall scope of the project.

- **Scheduling and planning** : Scheduling of various modules and their milestones, preparation of weekly reports, etc.
- **Technology used :** Latest hardware and software used.
- **Customer interaction** : Obtaining feedback from the customer.
- **Constraints and limitations** : Various constraints in the project and how they can be solved. Limitations in the modules and how they can be implemented in the next phase of the project, etc.
- **Risk Assessment** : Risk involved in the project with respect to limitations in the technology and resources.

2.9 SCHEDULING METHODS

Scheduling of a software project can be correlated to prioritising various tasks (jobs) with respect to their cost, time and duration. Scheduling can be done with resource constraint or time constraint in mind. Depending upon the project, scheduling methods can be static or dynamic in implementation.

Scheduling Techniques

The following are various types of scheduling techniques in software engineering are:

- **Work Breakdown Structure** : The project is scheduled in various phases following a bottom-up or top-down approach. A tree-like structure is followed without any loops. At each phase or step, milestone and deliverables are mentioned with respect to requirements. The work breakdown structure shows the overall breakup flow of the project and does not indicate any parallel flow. *Figure 2.2* depicts an example of a work breakdown structure.

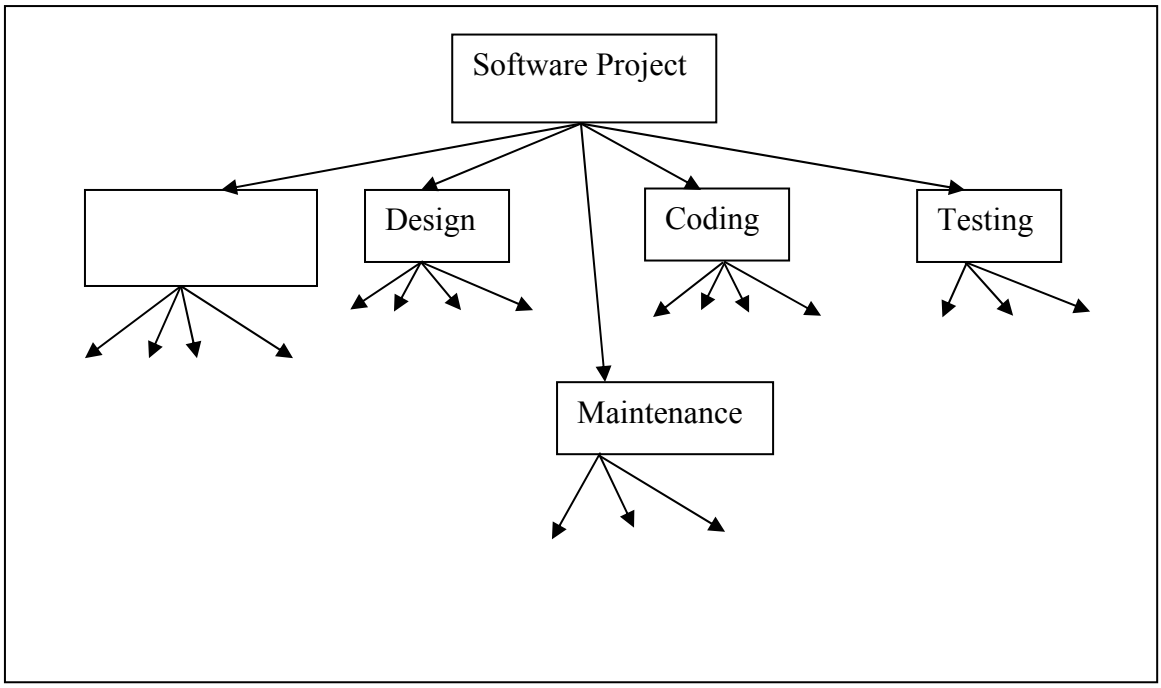
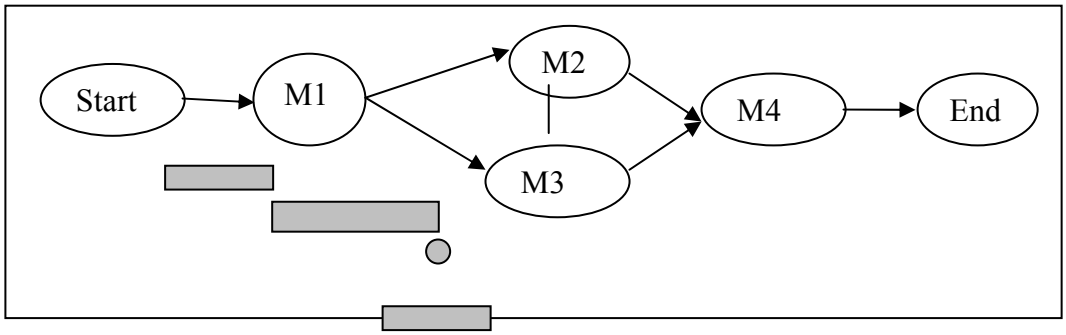


Figure 2.2: An example WBS

The project is split into requirement and analysis, design, coding, testing and maintenance phase. Further, requirement and analysis is divided into R1,R2 .. Rn; design is divided into D1,D2..Dn; coding is divided into C1,C2..Cn; testing is divided into T1,T2.. Tn; and maintenance is divided into M1, M2.. Mn. If the project

is complex, then further sub division is done. Upon the completion of each stage, integration is done.

- **Flow Graph :** Various modules are represented as nodes with edges connecting nodes. Dependency between nodes is shown by flow of data between nodes. Nodes indicate milestones and deliverables with the corresponding module implemented. Cycles are not allowed in the graph. Start and end nodes indicate the source and terminating nodes of the flow. *Figure 2.3* depicts a flow graph.



.3 : Flow Graph

M1 is the starting module and the data flows to M2 and M3. The combined data from M2 and M3 flow to M4 and finally the project terminates. In certain projects, time schedule is also associated with each module. The arrows indicate the flow of information between modules.

- **Gantt Chart or Time Line Charts :** A Gantt chart can be developed for the entire project or a separate chart can be developed for each function. A tabular form is maintained where rows indicate the tasks with milestones and columns indicate duration (weeks/months) . The horizontal bars that spans across columns indicate duration of the task. *Figure 2.4* depicts a Gantt Chart. The circles indicate the milestones.

Tasks	Week1				Week2				Week3				Week4			
A i a1 ii a2 Milestone																
B i b1 ii b2 Milestone																
C i c1 Milestone																
D i d1 ii d2 Milestone																

Figure 2.4: Gantt Chart

- **Program Evaluation Review Technique :** Mainly used for high-risk projects with various estimation parameters. For each module in a project, duration is estimated as follows:

  1. Time taken to complete a project or module under normal conditions, *tnormal.*

2. Time taken to complete a project or module with minimum time (all resources available), *tmin* .

3. Time taken to complete a project or module with maximum time (resource constraints), *tmax.*

4. Time taken to complete a project from previous related history, *Thistory.*

An average of tnormal, tmin, tmax and thistory is taken depending upon the project. Sometimes, various weights are added as 4*tnormal, 5*tmin, 0.9*tmax and 2*thistory to estimate the time for a project or module. Parameter fixing is done by the project manager.

2.10 THE SOFTWARE PROJECT PLAN

Planning is very important in every aspect of development work. Good managers carefully monitor developments at various phases. Improper planning leads to failure of the project. Software project plan can be viewed as the following :

1. Within the organisation: How the project is to be implemented? What are various constraints(time, cost, staff) ? What is market strategy?

2. With respect to the customer: Weekly or timely meetings with the customer with presentations on status reports. Customer feedback is also taken and further modifications and developments are done. Project milestones and deliverables are also presented to the customer.

For a successful software project, the following steps can be followed:

- Select a project
  - Identifying project's aims and objectives
  - Understanding requirements and specification
  - Methods of analysis, design and implementation
  - Testing techniques
  - Documentation

- Project milestones and deliverables

- Budget allocation
  - Exceeding limits within control

- Project Estimates
  - Cost
  - Time
  - Size of code
  - Duration

- Resource Allocation
  - Hardware
  - Software
  - Previous relevant project information
  - Digital Library

- Risk Management
  - Risk Avoidance
  - Risk Detection

- Risk Control
- Risk Recovery

- Scheduling techniques
  - Work Breakdown Structure
  - Activity Graph
  - Critical path method
  - Gantt Chart
  - Program Evaluation Review Technique

- People
  - Staff Recruitment
  - Team management
  - Customer interaction

- Quality control and standard

All of the above methods/techniques are not covered in this unit. The student is advised to study references for necessary information.

☞ **Check Your Progress 3**

1) Mention at least two factors to formulate a task set for a software project.

……

……

2) What are the drawbacks of work breakdown structure?

……

……

3) What are the advantages of Gantt chart?

……

……..

4) What is the purpose of software project plan?

……

……

2.11 SUMMARY

This unit describes various risk management and risk monitoring techniques. In case, major risks are identified, they are resolved and finally risk recovery is done. Risk manager takes care of all the phases of risk management. Various task sets are defined for a project from the customer point of view, the developer's point of view, the market strategy view, future trends, etc. For the implementation of a project, a proper task set is chosen and various attributes are defined. For successful implementation of a project, proper scheduling (with various techniques) and proper planning are done.

2.12 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Any problem that occurs during customer specification, design, coding, implementation and testing can be termed as a risk. If they are ignored, then they propagate further down and it is termed ripple effect. Risk management deals with avoidance and detection of risk at every phase of the software development cycle.

2) Two risks involved with team members are as follows:

   - Improper training of the technical staff.
   - Lack of proper communication between the developers.

3) The organisation should be aware of various competing vendors in the market, and various risks involved if the product is not delivered on time. It should have statistics of projects and risks involved from their previous experience or should have expertise personnel. The organisation should take care that the product will be able to support upcoming changes in hardware and software platforms.

4) Monitoring of risks means identifying problems in software functions, hardware deficiencies (lack of memory , peripherals, fast cpu and so on), etc. Risk table has entries for all the risk types and their timely weekly solution . Priorities of various risks are maintained.

5) Risks can be prioritised upon their dependencies on other modules and external factors. If a module is having many dependencies then its priority is given higher value compared to independent modules. If a module often causes security failure in the system, its priority can be set to a higher value.

Check Your Progress 2

1) Risk management means taking preventive measures for a software project to be free from various types of risks such as technical, customer, commercial, etc.

2) Various phases of risk management are risk avoidance, risk detection, risk analysis, risk monitoring, risk control and risk recovery.

3) Attributes mentioned in the risk analysis table are risk name, probability of occurrence of risk, weight factor and risk exposure.

4) Risk resolution means taking final steps to free the module or system from risk. Risk resolution involves risk elimination, risk transfer and disclosure of risk to the customer.

5) Some times, it is difficult to recover from the risk and it is better to add extra features or an alternate solutions keeping in view of customer specification with slight modifications in order to match future trends in hardware and software markets.

Check Your Progress 3

1) The two factors to formulate a task set for a software project are as follows:

   - Customer satisfaction
   - Full or partial implementation of the project

2) Work breakdown structure does not allow parallel flow design.

3) Gantt chart or time line chart indicates timely approach and milestone for each task and their relevant sub tasks.

4) Software project plan indicates scope of the project, milestones and deliverables, project estimates, resource allocation, risk management, scheduling techniques and quality control and standard .

2.13 FURTHER READINGS

1) *Software Engineering,* Ian Sommerville; *Sixth Edition, 2001*, 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://www.ncst.ernet.in

UNIT 3 SOFTWARE QUALITY ASSURANCE

Structure **Page Nos.**

3.0	Introduction	31
3.1	Objectives	31
3.2	Software Quality	31
3.3	Formal Technical Review	35
3.4	Software Reliability	40
3.5	Software Quality Standards	41
3.6	Summary	43
3.7	Solutions/Answers	43
3.8	Further Readings	44

3.0 INTRODUCTION

Software quality assurance is a series of activities undertaken throughout the software engineering process. It comprises the entire gamut of software engineering life cycle. The objective of the software quality assurance process is to produce high quality software that meets customer requirements through a process of applying various procedures and standards.

3. 1 OBJECTIVES

The purpose of this unit is to give an insight as to how software quality assurance activity is undertaken in the software engineering process.

After studying this unit, you should be able to understand the:

- concepts of Software quality and quality assurance;
- process of a formal technical review in a quality assurance process, and
- concepts of software reliability and software quality standards.

3.2 SOFTWARE QUALITY

Quality is defined as conformance to the stated and implied needs of customer. Quality also refers to the measurable characteristics of a software product and these can be compared based on a given set of standards. In the same way, software quality can be defined as conformance to explicitly stated and implicitly stated functional requirements. Here, the explicitly stated functional requirement can be derived from the requirements stated by the customer which are generally documented in some form. Implicit requirements are requirements which are not stated explicitly but are intended. Implicit functional requirements are standards which a software development company adheres to during the development process. Implicit functional requirements also include the requirements of good maintainability.

Quality software is reasonably bug-free, delivered on time and within budget, meets requirements and is maintainable. However, as discussed above, quality is a subjective term. It will depend on who the ‘customer’ is and their overall influence in the scheme of things. Each type of ‘customer’ will have their own slant on ‘quality’. The end-user might define quality as some thing which is user-friendly and bug-free.

Good quality software satisfies both explicit and implicit requirements. Software quality is a complex mix of characteristics and varies from application to application and the customer who requests for it.

.

Attributes of Quality

The following are some of the attributes of quality:

Auditability : The ability of software being tested against conformance to standard.

Compatibility : The ability of two or more systems or components to perform their required functions while sharing the same hardware or software environment.

Completeness: The degree to which all of the software's required functions and design constraints are present and fully developed in the requirements specification, design document and code.

Consistency: The degree of uniformity, standardisation, and freedom from contradiction among the documents or parts of a system or component.

Correctness: The degree to which a system or component is free from faults in its specification, design, and implementation. The degree to which software, documentation, or other items meet specified requirements.

Feasibility: The degree to which the requirements, design, or plans for a system or component can be implemented under existing constraints.

Modularity : The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

Predictability: The degree to which the functionality and performance of the software are determinable for a specified set of inputs.

Robustness: The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Structuredness : The degree to which the SDD (System Design Document) and code possess a definite pattern in their interdependent parts. This implies that the design has proceeded in an orderly and systematic manner (e.g., top-down, bottom-up). The modules are cohesive and the software has minimised coupling between modules.

Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Traceability: The degree to which a relationship can be established between two or more products of the development process. The degree to which each element in a software development product establishes its reason for existing (e.g., the degree to which each element in a bubble chart references the requirement that it satisfies). For example, the system's functionality must be traceable to user requirements.

Understandability: The degree to which the meaning of the SRS, SDD, and code are clear and understandable to the reader.

Verifiability : The degree to which the SRS, SDD, and code have been written to facilitate verification and testing.

Causes of error in Software

- Misinterpretation of customers' requirements/communication
- Incomplete/erroneous system specification
- Error in logic
- Not following programming/software standards
- Incomplete testing
- Inaccurate documentation/no documentation
- Deviation from specification
- Error in data modeling and representation.

Measurement of Software Quality (Quality metrics)

Software quality is a set of characteristics that can be measured in all phases of software development.

Defect metrics

- Number of design changes required
- Number of errors in the code
- Number of bugs during different stages of testing
- Reliability metrics
- It measures the mean time to failure (MTTF), that may be defined as probability of failure during a particular interval of time. This will be discussed in software reliability.

Maintainability metrics

- Complexity metrics are used to determine the maintainability of software. The complexity of software can be measured from its control flow.

Consider the graph of *Figure 3.1.* Each node represents one program segment and edges represent the control flow. The complexity of the software module represented by the graph can be given by simple formulae of graph theory as follows:

$$V(G) = e - n + 2 \text{ where}$$

V(G) : is called Cyclomatic complexity of the program
e = number of edges
n = number of nodes

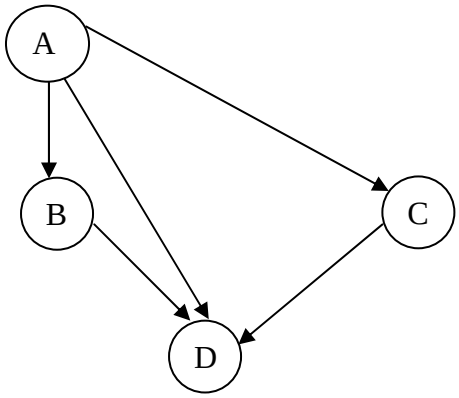


Figure 3.1: A software module

Applying the above equation the complexity V(G) of the graph is found to be 3.

The cyclomatic complexity has been related to programming effort, maintenance effort and debugging effort. Although cyclomatic complexity measures program complexity, it fails to measure the complexity of a program without multiple conditions.

The information flow within a program can provide a measure for program complexity.

Important parameters for measurement of Software Quality

- To the extent it satisfies user requirements; they form the foundation to measure software quality.
- Use of specific standards for building the software product. Standards could be organisation's own standards or standards referred in a contractual agreement.
- Implicit requirements which are not stated by the user but are essential for quality software.

Software Quality Assurance

The aim of the Software Quality Assurance process is to develop high-quality software product.

The purpose of Software Quality Assurance is to provide management with appropriate visibility into the process of the software project and of the products being built. Software Quality Assurance involves reviewing and auditing the software products throughout the development lifecycle to verify that they conform to explicit requirements and implicit requirements such as applicable procedures and standards. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits.

Software Quality Assurance (SQA) is a planned, coordinated and systematic actions necessary to provide adequate confidence that a software product conforms to established technical requirements.

Software Quality Assurance is a set of activities designed to evaluate the process by which software is developed and/or maintained.

The process of Software Quality Assurance

1. Defines the requirements for software controlled system fault/failure detection, isolation, and recovery;

2. Reviews the software development processes and products for software error prevention and/ or controlled change to reduced functionality states; and

3. Defines the process for measuring and analysing defects as well as reliability and maintainability factors.

Software engineers, project managers, customers and Software Quality Assurance groups are involved in software quality assurance activity. The role of various groups in software quality assurance are as follows:

- **Software engineers**: They ensure that appropriate methods are applied to develop the software, perform testing of the software product and participate in formal technical reviews.

- **SQA group**: They assist the software engineer in developing high quality product. They plan quality assurance activities and report the results of review.

☞ **Check Your Progress 1**

1) What is auditability?

……
…

2) Traceability is the ability of traceacing design back to _____

3) Software quality is designed in to software and can't be infused after the product is released. Explain?

……

…

……

…

3.3 FORMAL TECHNICAL REVIEW

What is software Review ? Software review can't be defined as a filter for the software engineering process. The purpose of any review is to discover errors in analysis, design, coding, testing and implementation phase of software development cycle. The other purpose of review is to see whether procedures are applied uniformly and in a manageable manner.

Reviews are basically of two types, informal technical review and formal technical review.

- Informal Technical Review : Informal meeting and informal desk checking.
- Formal Technical Review: A formal software quality assurance activity through various approaches such as structured walkthroughs, inspection, etc.

What is Formal Technical Review: Formal technical review is a software quality assurance activity performed by software engineering practitioners to improve software product quality. The product is scrutinised for completeness, correctness, consistency, technical feasibility, efficiency, and adherence to established standards and guidelines by the client organisation.

Structured walkthrough is a review of the formal deliverables produced by the project team. Participants of this review typically include end-users and management of the client organization, management of the development organisation, and sometimes auditors, as well as members of the project team. As such, these reviews are more formal in nature with a predefined agenda, which may include presentations, overheads, etc.

An inspection is more formalised than a 'Structured walkthrough', typically with 3-8 people. The subject of the inspection is typically a document such as a requirements specification or a test plan, and the purpose is to find problems and see what's missing, not to suggest rectification or fixing. The result of the inspection meeting should be a written report.

Verification **:** Verification generally involves reviews to evaluate whether correct methodologies have been followed by checking documents, plans, code, requirements, and specifications. This can be done with checklists, walkthroughs, etc.

Validation **:** Validation typically involves actual testing and takes place after verifications are completed.

Objectives of Formal Technical Review

- To uncover errors in logic or implementation
- To ensure that the software has been represented accruing to predefined standards
- To ensure that software under review meets the requirements
- To make the project more manageable.

Each Formal Technical Review (FTR) is conducted as a meeting and requires well coordinated planning and commitments.

For the success of formal technical review, the following are expected:

- The schedule of the meeting and its agenda reach the members well in advance
- Members review the material and its distribution
- The reviewer must review the material in advance.

The meeting should consist of two to five people and should be restricted to not more than 2 hours (preferably). The aim of the review is to review the product/work and the performance of the people. When the product is ready, the producer (developer) informs the project leader about the completion of the product and requests for review. The project leader contacts the review leader for the review. The review leader asks the reviewer to perform an independent review of the product/work before the scheduled FTR.

Result of FTR

- Meeting decision
  - Whether to accept the product/work without any modification
  - Accept the product/work with certain changes
  - Reject the product/work due to error

- Review summary report
  - What was reviewed?
  - Who reviewed it?
  - Findings of the review
  - Conclusion

Checklist - Typical activities for review at each phase are described below:

Software Concept and Initiation Phase

Involvement of SQA team in both writing and reviewing the project management plan in order to assure that the processes, procedures, and standards identified in the plan are appropriate, clear, specific, and auditable.

- Have design constraints been taken in to account?
- Whether best alternatives have been selected.

Software Requirements Analysis Phase

During the software requirements phase, review assures that software requirements are complete, testable, and properly expressed as functional, performance, and interface requirements. The output of a software requirement analysis phase is Software Requirements Specification (SRS). SRS forms the major input for review.

Compatibility

- Does the interface enables compatibility with external interfaces?
- Whether the specified models, algorithms, and numerical techniques are compatible?

Completeness

- Does it include all requirements relating to functionality, performance, system constraints, etc.?
- Does SRS include all user requirements?

- Are the time-critical functions defined and identified?
- Are the requirements consistent with available resources and budget?
- Does the SRS include requirements for anticipated future changes?

Consistency

- Are the requirements consistent with each other? Is the SRS free of contradictions?
- Whether SRS uses standard terminology and definitions throughout.
- Has the impact of operational environment on the software been specified in the SRS?

Correctness

- Does the SRS conform to SRS standards?
- Does the SRS define the required responses to all expected types of errors and failure? hazard analysis?
- Were the functional requirements analysed to check if all abnormal situations are covered by system functions?
- Does SRS reference development standards for preparation?
- Does the SRS identify external interfaces in terms of input and output mathematical variables?
- Has the rationale for each requirement been defined?
- Are the requirements adequate and correctly indicated?
- Is there justification for the design/implementation constraints?

Traceability

- Are the requirements traceable to top level?
- Is there traceability from the next higher level of specification?
- Is SRS traceable forward through successive software development phases like the design, coding and testing ?

Verifiability and Testability

- Are validation criteria complete?
- Is there a verification procedure defined for each requirement in the SRS?
- Are the requirements verifiable?

Software Design Phase

Reviewers should be able to determine whether or not all design features are consistent with the requirements. And, the program should meet the requirements. The output of the software design phase is a system design document (SDD) and forms an input for a Formal Technical Review.

Completeness

- Whether all the requirements have been addressed in the SDD?
- Have the software requirements been properly reflected in software architecture?
- Are algorithms adequate, accurate, and complete?
- Does the design implement the required program behavior with respect to each program interface?
- Does the design take into consideration all expected conditions?
- Does the design specify appropriate behaviour in case of an unexpected or improper input and other abnormal conditions?

Consistency

- Are standard terminology and definitions used throughout the SDD? Is the style of presentation and the level of detail consistent throughout the document.
- Does the design configuration ensure integrity of changes?
- Is there compatibility of the interfaces?
- Is the test documentation compatible with the test requirements of the SRS?
- Are the models, algorithms, and numerical techniques that are specified mathematically compatible?
- Are input and output formats consistent to the extent possible?
- Are the designs for similar or related functions are consistent?
- Are the accuracies and units of inputs, database elements, and outputs that are used together in computations or logical decisions compatible?

Correctness

- Does the SDD conform to design documentation standards?
- Does the design perform only that which is specified in the SRS?
- Whether additional functionality is justified?
- Is the test documentation current and technically accurate?
- Is the design logic sound i.e., will the program do what is intended?
- Is the design consistent with documented descriptions?
- Does the design correctly accommodate all inputs, outputs, and database elements ?

Modifiability

- The modules are organised such that changes in the requirements only require changes to a small number of modules.
- Functionality is partitioned into programs to maximize the internal cohesion of programs and to minimize program coupling.
- Is the design structured so that it comprises relatively small, hierarchically related programs or sets of programs, each performing a particular unique function?
- Does the design use a logical hierarchical control structure?

Traceability

- Is the SDD traceable to requirements in SRS?
- Does the SDD show mapping and complete coverage of all requirements and design constraints in the SRS?
- Whether the functions in the SDD which are outside the scope of SRS are defined and identified?

Verifiability

- Does the SDD describe each function using well-defined notation so that the SDD can be verified against the SRS
- Can the code be verified against the SDD?
- Are conditions, constraints identified so that test cases can be designed?

Review of Source Code

The following checklist contains the kinds of questions a reviewer may take up during source code review based on various standards

Completeness

- Is the code complete and precise in implementation?
- Is the code as per the design documented in the SDD?
- Are there any unreferenced or undefined variables, constants, or data types?
- Whether the code follows any coding standard that is referenced?

Consistency

- Is the code consistent with the SDD?
- Are the nomenclature of variables, functions uniform throughout?

Correctness

- Does the code conform to specified coding standards?
- Are all variables properly specified and used?
- Does the code avoid recursion?
- Does the code protect against detectable runtime errors?
- Are all comments accurate and sufficient?
- Is the code free of unintended infinite loops?

Modifiability

- Is the program documentation sufficient to facilitate future changes?
- Does the code refer to constants symbolically so as to facilitate change?
- Is the code written in a language with well-defined syntax and semantics?
- Are the references or data dictionaries included to show variable and constant access by the program?
- Does the code avoid relying on defaults provided by the programming language?

Traceability

- Is the code traceable to design document?
- Does the code identify each program uniquely?
- Is there a cross-reference through which the code can be easily and directly traced to the SDD?
- Does the code contain, or has reference to, a revision history of all code modifications done by different authors?
- Whether the reasons for such modification are specified and authorized?

Understandability

- Do the comment statements in the code adequately describe each routine?
- Whether proper indent and formatting has been used to enhance clarity?
- Has the formatting been used consistently?
- Does naming patterns of the variables properly reflect the type of variable?
- Is the valid range of each variable defined?

Software Testing and Implementation Phase

- Whether all deliverable items are tested.
- Whether test plans are consistent and sufficient to test the functionality of the systems.
- Whether non-conformance reporting and corrective action has been initiated?.
- Whether boundary value is being tested.
- Whether all tests are run according to test plans and procedures and any non-conformances are reported and resolved?
- Are the test reports complete and correct?
- Has it been certified that testing is complete and software including documentation are ready for delivery?

Installation phase

Completeness

- Are the components necessary for installation of a program in this installation medium ?
- Whether adequate information for installing the program, including the system requirements for running the program available.
- Is there more than one operating environment?
- Are installation procedures for different running environments available?
- Are the installation procedures clearly understandable?

☞ Check Your Progress 2

1) The purpose of review is to uncover errors and non conformity during testing phase. Yes ☐ No ☐

2) The result of a review does not include
   a) The items that are to be reviewed
   b) The person who reviews it
   c) Findings of the review
   d) Solution to a problem

3) What could become an input for FTR in design phase?
   ……
   ……
   ……

3.4 SOFTWARE RELIABILITY

Software Reliability: Unlike reliability of the hardware device, the term software reliability is difficult to measure. In the software engineering environment, software reliability is defined as the probability that software will provide failure-free operation in a fixed environment for a fixed interval of time.

Software reliability is typically measured per a unit of time, whereas probability of failure is generally time independent. These two measures can be easily related if you know the frequency with which inputs are executed per unit of time. Mean-time-to-failure (MTTF) is the average interval of time between failures; this is also sometimes referred to as Mean-time-before-failure.

Possibly the greatest problem in the field of software reliability estimation has to do with the accuracy of operational profiles. Without accurate profiles, estimates will almost certainly be wrong. An operational profile is the probability density function (over the entire input space) that best represents how the inputs would be selected during the life-time of the software. There is nothing fancy about operational profiles; they are really just “guesses” about what inputs will occur in the future.

Definitions of Software reliability

- IEEE Definition: The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system in the software. The inputs to the system determine whether existing faults, if any, are encountered.

- The ability of a program to perform its required functions accurately and reproducibly under stated conditions for a specified period of time.

Software Reliability Models

A number of models have been developed to determine software defects/failures. All these models describe the occurrence of defects/failures as a function of time. This

allows us to define reliability. These models are based on certain assumptions which can be described as below:

- The failures are independent of each other, i.e., one failure has no impact on other failure(s).
- The inputs are random samples.
- Failure intervals are independent and all software failures are observed.
- Time between failures is exponentially distributed.

The following formula gives the cumulative number of defects observed at a time 't'.

$$D(t) = T_d (1 - e^{-bct})$$

D(t) = Cumulative number of defects observed at a time t

T_d = Total number of defects

'b' and 'c' are constants and depend on historical data of similar software for which the model is being applied

We may find the mean time to failure (MMFT) as below:

$$MTTF(t) = e^{bct} / c\, T_d$$

3.5 SOFTWARE QUALITY STANDARDS

Software standards help an organization to adopt a uniform approach in designing, developing and maintaining software. There are a number of standards for software quality and software quality assurance. Once an organisation decides to establish a software quality assurance process, standards may be followed to establish and operate different software development activities and support activities. A number of organisations have developed standards on quality in general and software quality in specific.

Software Engineering Institute (SEI) has developed what is called a '*Capability Maturity Model'* (CMM) now called the *Capability Maturity Model Integration* (CMMI) to help organisations to improve software development processes.

It is a model of 5 levels of process maturity that determine the effectiveness of an organisation in delivering quality software. Organizations can receive CMMI ratings by undergoing assessments by qualified auditors. The organizations are rated as CMM Level 1, CMM Level 2 etc. by evaluating their organisational process maturity.

SEI-CMM Level 1: Characterised by un organised, chaos, periodic panics, and heroic efforts required by individuals to successfully complete projects. Successes depend on individuals and may not be repeatable.

SEI-CMM Level 2 : Software project tracking, requirements management, realistic planning, and configuration management processes are in place; successful practices can be repeated.

SEI-CMM Level 3: Standard software development and maintenance processes are integrated throughout an organisation; a Software Engineering Process Group is in place to oversee software processes, and training programs are used to ensure understanding and compliance.

SEI-CMM Level 4: Metrics are used to track productivity, processes, and products. Project performance is predictable, and quality is consistently high.

SEI-CMM Level 5: The focus is on continuous process improvement. The impact of new processes and technologies can be predicted and effectively implemented when required.

The International Organisation for Standardisation (ISO) developed the ISO 9001:2000 standard (which replaces the previous set of three standards of 1994) that helps the organisation to establish, operate, maintain and review a quality management system that is assessed by outside auditors. The standard is generic in nature and can be applied to any organisation involved in production, manufacturing service including an organisation providing software services.

It covers documentation, design, development, production, testing, installation, servicing, and other processes. It may be noted that ISO certification does not necessarily indicate quality products. It only indicates that the organisation follows a well documented established process. *Table 3.1* gives a list of standards along with the corresponding titles.

Table 3.1 : List of standards

Standards	Title
ISO/IEC 90003	Software Engineering. Guidelines for the Application of ISO 9001:2000 to Computer Software
ISO 9001:2000	Quality Management Systems - Requirements
ISO/IEC 14598-1	Information Technology-Evaluation of Software Products-General Guide
ISO/IEC 9126-1	Software Engineering - Product Quality - Part 1: Quality Model
ISO/IEC 12119	Information Technology-Software Packages-Quality Requirements and Testing
ISO/IEC 12207	Information Technology-Software Life Cycle Processes
ISO/IEC 14102	Guideline For the Evaluation and Selection of CASE Tools
IEC 60880	Software for Computers in the Safety Systems of Nuclear Power Stations
IEEE 730	Software Quality Assurance Plans
IEEE 730.1	Guide for Software Assurance Planning
IEEE 982.1	Standard Dictionary of Measures to produce reliable software
IEEE 1061	Standard for a Software Quality Metrics Methodology
IEEE 1540	Software Life Cycle Processes - Risk Management
IEEE 1028	Software review and audits
IEEE 1012	Software verification and validation plans

☞ Check Your Progress 3

1) Success of a project depends on individual effort. At what stage of maturity does the organisation's software process can be rated?

   ……..

   ……...

2) Why ISO 9001 : 2000 is called generic standard?

   ……..

   ……..

3) What is the difference between SEI CMM standards and ISO 9000 : 2000 standards?

   ……

   …

   ……

   …

3.6 SUMMARY

Software quality assurance is applied at every stages of system development through a process of review to ensure that proper methodology and procedure has been followed to produce the software product. It covers the entire gamut of software development activity. Software quality is conformance with explicitly defined requirements which may come from a customer requirement or organisation may define their own standards. Software review also called a Formal Technical Review is one of the most important activity of software quality assurance process. Software reliability provides measurement of software dependability and probability of failure is generally time independent. The aim of software quality assurance is to improve both software product and process through which the software is built. The help of various software standards taken to establish and operate software quality assurance process.

3.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Auditability is that attribute of software which determines the degree to which a software can be tested for conformance against a standard.

2) Requirements.

3) Software quality assurance activity is an umbrella activity comprising activities like application of standards, technical methods, review, software testing, change control, measurement, control & reporting during the process of software development life cycle. High quality product can come from high quality design specification. Unlike quality, testing quality assurance can't be achieved at the end of the product completion phase.

Check Your Progress 2

1) No. The purpose of any review is to discover errors in the analysis, design, coding, testing and maintenance phases of software development life cycle.

2) Solution to a problem.

3) Output of the software design phase is a system design document (SDD) and it is an input for the Formal Technical Review.

Check Your Progress 3

1) SEI CMM level 1.

2) The standards are called generic in nature as they can be applied to any organisation involved in the production, manufacturing service including organisations providing software services which involve no manufacturing process.

3) SEI CMM standards are developed to rate the maturity of organisation's software development process. The maturity is rated from level 1 to level 5, i.e., from immature (no formal process) to predictable matured process with focus on continuous improvement. In case of ISO 9001 : 2000, the organisation's process (quality system) is tested against conformance to the requirements of ISO 9000 : 2000 standard.

3.8 FURTHER READINGS

1) *Software Quality, 1997*, Mordechai Ben-Menachem and Garry S.Marliss; Thomson Learning.

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

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

Reference websites

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

UNIT 4 SOFTWARE CHANGE MANAGEMENT

Structure **Page Nos.**

4.0 Introduction 45
4.1 Objectives 45
4.2 Baselines 45
4.3 Version Control 48
4.4 Change Control 51
4.5 Auditing and Reporting 54
4.6 Summary 56
4.7 Solutions/Answers 56
4.8 Further Readings 56

4.0 INTRODUCTION

Software change management is an umbrella activity that aims at maintaining the integrity of software products and items. Change is a fact of life but uncontrolled change may lead to havoc and may affect the integrity of the base product. Software development has become an increasingly complex and dynamic activity. Software change management is a challenging task faced by modern project managers, especially in a environment where software development is spread across a wide geographic area with a number of software developers in a distributed environment. Enforcement of regulatory requirements and standards demand a robust change management. The aim of change management is to facilitate justifiable changes in the software product.

4.1 OBJECTIVES

After studying this unit, you should be able to:

- define baselines;
- know the concept of version control and change control, and
- audit and report software change management activity.

4.2 BASELINES

Baseline is a term frequently used in the context of software change management. Before we understand the concept of baseline, let us define a term which is called software configuration item. A software configuration item is any part of development and /or deliverable system which may include software, hardware, firmware, drawings, inventories, project plans or documents. These items are independently tested, stored, reviewed and changed. A software configuration item can be one or more of the following:

- System specification
- Source code
- Object code
- Drawing
- Software design

- Design data
- Database schema and file structure
- Test plan and test cases
- Product specific documents
- Project plan
- Standards procedures
- Process description

Definition of Baseline: A baseline is an approved software configuration item that has been reviewed and finalised. An example of a baseline is an approved design document that is consistent with the requirements. The process of review and approval forms part of the formal technical review. The baseline serves as a reference for any change. Once the changes to a reference baseline is reviewed and approved, it acts as a baseline for the next change(s).

A baseline is a set of configuration items (hardware, documents and software components) that have been formally reviewed and agreed upon, thereafter serve as the basis for future development, and that can be changed only through formal change control procedures.

Figure 4.1 depicts a baseline for design specification.

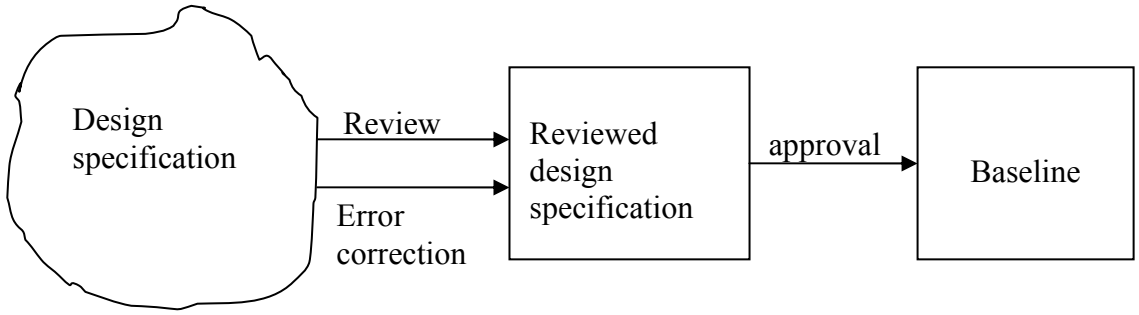


Figure 4.1 : A baseline for design specification

Figure 4.2 depicts the evolution of a baseline.

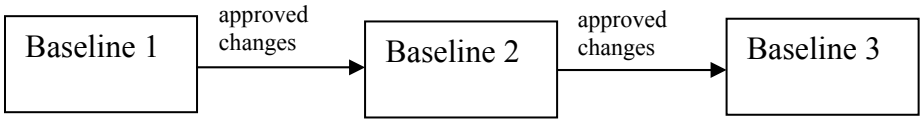


Figure 4.2: Evolution of a baseline

A baseline is functionally complete, i.e., it has a defined functionality. The features of these functionalities are documented for reference for further changes. The baseline has a defined quality which has undergone a formal round of testing and reviews before being termed as a baseline. Any baseline can be recreated at any point of time.

The process of change management

The domain of software change management process defines how to control and manage changes.

A formal process of change management is acutely felt in the current scenario when the software is developed in a very complex distributed environment with many versions of a software existing at the same time, many developers involved in the development process using different technologies. The ultimate bottomline is to maintain the integrity of the software product while incorporating changes.

The following are the objectives of software change management process:

1. **Configuration identification**: The source code, documents, test plans, etc. The process of identification involves identifying each component name, giving them a version name (a unique number for identification) and a configuration identification.
2. **Configuration control**: Cntrolling changes to a product. Controlling release of a product and changes that ensure that the software is consistent on the basis of a baseline product.
3. **Review**: Reviewing the process to ensure consistency among different configuration items.
4. **Status accounting** : Recording and reporting the changes and status of the components.
5. **Auditing and reporting**: Validating the product and maintaining consistency of the product throughout the software life cycle.

Process of changes: As we have discussed, baseline forms the reference for any change. Whenever a change is identified, the baseline which is available in project database is copied by the change agent (the software developer) to his private area. Once the modification is underway the baseline is locked for any further modification which may lead to inconsistency. The records of all changes are tracked and recorded in a status accounting file. After the changes are completed and the changes go through a change control procedure, it becomes a approved item for updating the original baseline in the project database.

Figure 4.3 depicts the process of changes to a baseline.

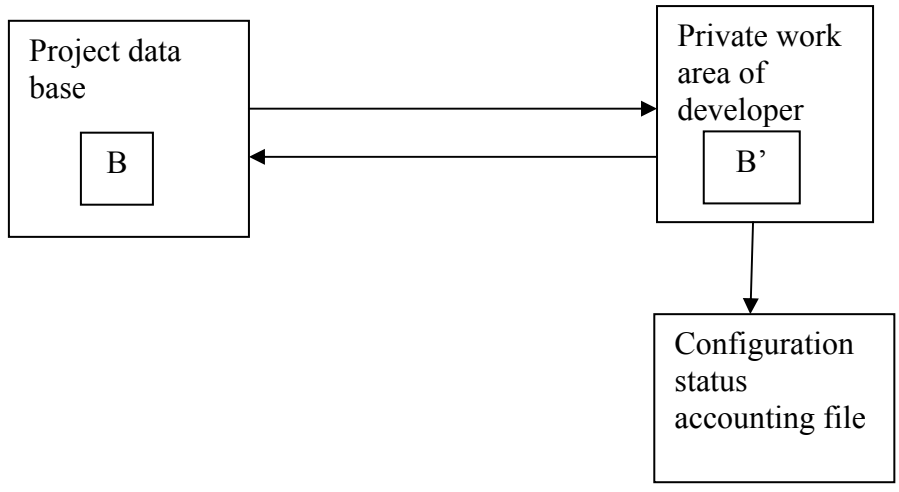


Figure 4.3: Process of changes to baseline

All the changes during the process of modification are recorded in the configuration status accounting file. It records all changes made to the previous baseline B to reach the new baseline B'. The status accounting file is used for configuration authentication which assures that the new baseline B' has all the required planned and approved changes incorporated. This is also known as auditing.

☞ **Check Your Progress 1**

1) _____ serves as reference for any change.

2) What is the aim of software change management process?

……...

……...

4.3 VERSION CONTROL

Version control is the management of multiple revisions of the same unit of item during the software development process. For example, a system requirement specification (SRS) is produced after taking into account the user requirements which change with time into account. Once a SRS is finalized, documented and approved, it is given a document number, with a unique identification number. The name of the items may follow a hierarchical pattern which may consist of the following:

- Project identifier
- Configuration item (or simply item, e.g. SRS, program, data model)
- Change number or version number

The identification of the configuration item must be able to provide the relationship between items whenever such relationship exists.

The identification process should be such that it uniquely identifies the configuration item throughout the development life cycle, such that all such changes are traceable to the previous configuration. An evolutionary graph graphically reflects the history of all such changes. The aim of these controls is to facilitate the return to any previous state of configuration item in case of any unresolved issue in the current unapproved version.

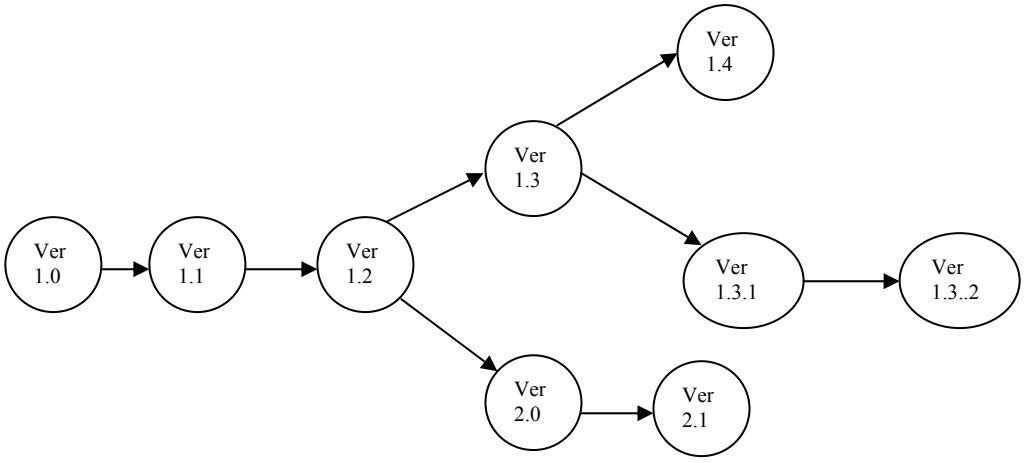


Figure 4.4 : An evolutionary graph for a different version of an item

The above evolutionary graph(*Figure 4.4*) depicts the evolution of a configuration item during the development life cycle. The initial version of the item is given version number Ver 1.0. Subsequent changes to the item which could be mostly fixing bugs or adding minor functionality is given as Ver 1.1 and Ver 1.2. After that, a major modification to Ver 1.2 is given a number Ver 2.0 at the same time, a parallel version of the same item without the major modification is maintained and given a version number 1.3.

Depending on the volume and extent of changes, the version numbers are given by the version control manager to uniquely identify an item through the software development lifecycle. It may be noted that most of the versions of the items are released during the software maintenance phase.

Software engineers use this version control mechanism to track the source code, documentation and other configuration items. In practice, many tools are available to store and number these configuration items automatically. As software is developed and deployed, it is common to expect that multiple versions of the same software are deployed or maintained for various reasons. Many of these versions are used by developers to privately work to update the software.

It is also sometimes desirable to develop two parallel versions of the same product where one version is used to fix a bug in the earlier version and other one is used to develop new functionality and features in the software. Traditionally, software developers maintained multiple versions of the same software and named them uniquely by a number. But, this numbering system has certain disadvantages like it does not give any idea about a nearly identical versions of the same software which may exist.

The project database maintains all copies of the different versions of the software and other items. It is quite possible that without each other's knowledge, two developers may copy the same version of the item to their private area and start working on it. Updating to the central project database after completing changes will lead to overwriting of each other's work. Most version control systems provide a solution to this kind of problem by locking the version for further modification.

Commercial tools are available for version control which performs one or more of following tasks;

- Source code control
- Revision control
- Concurrent version control

There are many commercial tools like Rational ClearCase, Microsoft Visual SourceSafe and a number of other commercial tools to help version control.

Managing change is an important part of computing. The programmer fixes bugs while producing a new version based on the feedback of the user. System administrator manages various changes like porting database, migrating to a new platform and application environment without interrupting the day to day operations. Revisions to documents are carried out while improving application.

An example of revision control

Let us consider the following simple HTML file in a web based application (welcome.htm)

```
<html>
<head>
<Title> A simple HTML Page</title>
</head>
<body>
<h1> Welcome to HTML Concepts</h1>
</body>
</html>
```

Once the code is tested and finalized, the first step is to register the program to he project database. The revision is numbered and this file is marked read-only to prevent any further undesirable changes. This forms the building block of source control. Each time the file is modified, a new version is created and a new revision number is given.

The first version of the file is numbered as version 1.0. Any further modification is possible only in the developer's private area by copying the file from the project

database. The process of copying the configuration object (the baseline version) is called check-out.

Figure 4.5 depicts the changes to a baselined file in project database.

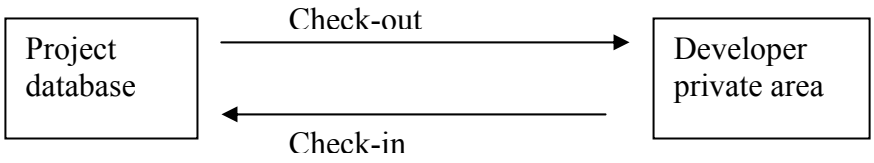


Figure 4.5: Changes to a baselined file in project database

The version (revision) control process starts with registering the initial versions of the file. This essentially enforces a check on the changes which ensure that the file can't be changed unless it is checked-out from the project database.

When a change is required to be made, allowing an email address to be added to the above html file, the developer will extract the latest version of the file welcome.htm from the project database. Once it is checked-out from the project database for modification, the file is locked for further modifications by any other developer. Other developers can check-out the file for read-only purpose.

Now, the following lines are added to the file welcome.htm.

<hr>
a href=mailto:webmaster@xyz.com> webmaster
<hr>

The revised version of the file welcome.htm become

```
<html>
<head>
<Title> A simple HTML Page</title>
</head>
<body>
<h1> Welcome to HTML Concepts</h1>
<hr>
a href=mailto:webmaster@xyz.com> webmaster</a>
<hr>
</body>
</html>
```

Then the developer check-in's the revised version of the file to the project database with a new version (revision) number version 1.1 i.e. the first revision along with the details of the modification done.

Suppose another modification is done by adding a graphic to the html file welcome.htm. This becomes version 1.2. The version tree after two modifications looks as shown below (*Figure 4.6*).

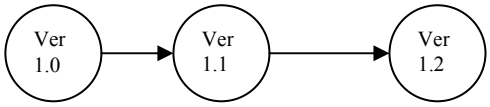


Figure 4.6: Version tree of welcome.htm

Suppose further modification is required for text-based browser as graphic will not be supported by text-based browser. Then the version 1.1 will be selected from the project database. This shows the necessity of storing all versions of the file in the project database.

☞ **Check Your Progress 2**

1) _____ is an example of source code control tool.

2) Version control mechanism allows multiple versions of an item to be maintained at the same time. (Yes/No)

3) How do version control systems ensure that two software developers do not attempt the same change at the same time?

……

……...

4.4 CHANGE CONTROL

Change is a fact of life, and the same applies to software development. Although, all changes requested by the user are not justified changes, but most of them are. The real challenge of change manager and project leader is to accept and accommodate all justifiable changes without affecting the integrity of product or without any side effect. The central to change management process is change control which deals with the formal process of change control.

The adoption and evolution of changes are carried out in a disciplined manner. In a large software environment where, as changes are done by a number of software developers, uncontrolled and un-coordinated changes may lead to havoc grossly diverting from the basic features and requirements of the system. For this, a formal change control process is developed.

Change control is a management process and is to some extent automated to provide a systematic mechanism for change control. Changes can be initiated by the user or other stake holder during the maintenance phase, although a change request may even come up during the development phase of the software.

A change request starts as a beginning of any change control process. The change request is evaluated for merits and demerits, and the potential side effects are evaluated. The overall impact on the system is assessed by the technical group consisting of the developer and project manager. A change control report is generated by the technical team listing the extent of changes and potential side effects. A designated team called change control authority makes the final decision, based on the change control report, whether to accept or reject the change request.

A change order called engineering change order is generated after the approval of the change request by the change control authority. The engineering change order forms the starting point of effecting a change in the component. If the change requested is

not approved by the change control authority, then the decision is conveyed to the user or the change request generator.

Once, change order is received by the developers, the required configuration items are identified which require changes. The baseline version of configuration items are copied from the project data base as discussed earlier.

The changes are then incorporated in the copied version of the item. The changes are subject to review (called audit) by a designated team before testing and other quality assurance activity is carried out. Once the changes are approved, a new version is generated for distribution.

The change control mechanisms are applied to the items which have become baselines. For other items which are yet to attain the stage of baseline, informal change control may be applied. For non- baseline items, the developer may make required changes as he feels appropriate to satisfy the technical requirement as long as it does not have an impact on the overall system.

The role of the change control authority is vital for any item which has become a baseline item. All changes to the baseline item must follow a formal change control process.

As discussed, change request, change report and engineering change order (change order) are generated as part of the change control activity within the software change management process. These documents are often represented inprinted or electronic forms. The typical content of these documents is given below:

Software Change Request Format

1.0 Change request Identification

1.1 Name, identification and description of software configuration item(s):
The name, version numbers of the software configuration is provided. Also, a brief description of the configuration item is provided.

1.2 Requester and contact details: The name of the person requesting the change and contact details

1.3 Date, location, and time when the change is requested

2.0 Description of the change

2.1 Description : This section specifies a detailed description of the change request.

- 2.1.1 Background Information, Background information of the request.
- 2.1.2 Examples: Supporting information, examples, error report, and screen shoots
- 2.1.3 The change : A detailed discussion of the change requested.

2.2 Justification for the change : Detailed justification for the request.

2.3 Priority : The priority of the change depending on critical effect on system functionalities.

Software Change Report Format

1.0 Change report Identification

1.1 Name, identification and description of software configuration item(s): The name, version numbers of the software configuration item and a brief description of it.

1.2 Requester: The name and contact details of the person requesting the change.

1.3 Evaluator : The name of the person or team who evaluated the change request.

1.4 Date and time : When change report was generated.

2.0 Overview of changes required to accommodate request

2.1 Description of software configuration item that will be affected

2.2 Change categorization : Type of change, in a generic sense

2.3 Scope of the change : The evaluator's assessment of the change.

2.3.1 Technical work required including tools required etc. A description of the work required to accomplish the change including required tools or other special resources are specified here

2.3.2 Technical risks : The risks associated with making the change are described.

3.0 Cost Assessment : Cost assessment of the requested change including an estimate of time required.

4.0 Recommendation

4.1 Evaluator’s recommendation : This section presents the evaluator's recommendation regarding the change

4.2 Internal priority: How important is this change in the light of the business operation and priority assigned by the evaluator.

Engineering Change Order Format

1.0 Change order Identification

1.1 Name, identification and description of software configuration item(s) : The name, version numbers including a brief description of software configuration items is provided.

1.2 Name of Requester

1.3 Name of Evaluator

2.0 Description of the change to be made

2.1 Description of software configuration(s) that is affected

2.2 Scope of the change required

The evaluator's assessment of scope of the change in the configuration item(s).

2.2.1 Technical work and tools required : A description of the work and tools required to accomplish the change.

2.3 Technical risks: The risks associated with making the change are described in this section.

3.0 Testing and Validation requirements

A description of the testing and review approach required to ensure that the change has been made without any undesirable side effects.

3.1 Review plan : Description of reviews that will be conducted.

3.2 Test plan

Description of the test plans and new tests that are required.

Benefits of change control management

The existence of a formal process of change management helps the developer to identify the responsibility of code for which a developer is responsible. An idea is achieved about the changes that affect the main product. The existence of such mechanism provides a road map to the development process and encourages the developers to be more involved in their work.

Version control mechanism helps the software tester to track the previous version of the product, thereby giving emphasis on testing of the changes made since the last approved changes. It helps the developer and tester to simultaneously work on multiple versions of the same product and still avoid any conflict and overlapping of activity.

The software change management process is used by the managers to keep a control on the changes to the product thereby tracking and monitoring every change. The existence of a formal process reassures the management. It provides a professional approach to control software changes.

It also provides confidence to the customer regarding the quality of the product.

4.5 AUDITING AND REPORTING

Auditing

Auditing and Reporting helps change management process to ensure whether the changes have been properly implemented or not, whether it has any undesired impact on other components. A formal technical review and software configuration audit helps in ensuring that the changes have been implemented properly during the change process.

A Formal Technical Review generally concentrates on technical correctness of the changes to the configuration item whereas software configuration audit complements it by checking the parameters which are not checked in a Formal Technical Review.

- Whether a formal technical review is carried out to check the technical accuracy of the changes made?
- Whether the changes as identified and reported in the change order have been incorporated?
- Have the changes been properly documented in the configuration items?
- Whether standards have been followed.
- Whether the procedure for identifying, recording and reporting changes has been followed.

As it is a formal process, it is desirable to conduct the audit by a separate team other than the team responsible for incorporating the changes.

Reporting: Status reporting is also called status accounting. It records all changes that lead to each new version of the item. Status reporting is the bookkeeping of each release. The process involves tracking the change in each version that leads the latest(new) version.

The report includes the following:

- The changes incorporated
- The person responsible for the change
- The date and time of changes
- The effect of the change
- The reason for such changes (if it is a bug fixing)

Every time a change is incorporated it is assigned a unique number to identify it from the previous version. Status reporting is of vital importance in a scenario where a large number of developers work on the same product at the same time and have little idea about the work of other developers.

For example, in source code, reporting the changes may be as below:

****
****
****

Title : Sub routine Insert to Employee Data

Version : Ver 1.1.3

Purpose : To insert employee data in the master file

Author : John Wright

Date : 23/10/2001

Auditor : J Waltson

Modification History:

12/12/2002 : by D K N
To fix bugs discovered in the first release

4/5/2003 : by S K G
to allow validation in date of birth data

6/6/2004 : by S S P
To add error checking module as requested by the customer

☞ Check Your Progress 3

1) Who decides the acceptance of a change request?.

……

……

2) How auditing is different from a Formal Technical Review (FTR)?

……

……

4.6 SUMMARY

Software change management is an activity that is applied throughout the software development life cycle. The process of change management includes configuration identification, configuration control, status reporting and status auditing. Configuration identification involves identification of all configurations that require changes. Configuration or change control is the process of controlling the release of the product and the changes. Status accounting or reporting is the process of recording the status of each component. Review involves auditing consistency and completeness of the component.

4.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Baseline.
2) The domain of software change management process defines how to control and manage changes. The ultimate aim is to maintain the integrity of the software product while incorporating changes.

Check Your Progress 2

1) Microsoft Visual SourceSafe
2) Yes
3) Version control system locks the configuration item once it is copied from project database for modification by a developer.

Check Your Progress 3

1) Change control authority
2) Formal Technical Review is a formal process to evaluate the technical accuracy of any process or changes. Whereas software change or audit is carried out by a separate team to ensure that proper change management procedure has been followed to incorporate the changes.

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.

Reference websites

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

UNIT 1 WEB SOFTWARE ENGINEERING

Structure **Page Nos.**

1.0 Introduction 5
1.1 Objectives 6
1.2 Different Characteristics 6
1.2.1 Characteristics of a Web Application
1.2.2 Development, Testing and Deployment
1.2.3 Usage of Web Applications
1.2.4 Maintaining Web Applications
1.2.5 Designing Web Applications
1.3 Issues of Management of Web Based Projects 9
1.3.1 Review of Good Software Development Practices
1.3.2 Organisation of Web Application Teams
1.3.3 Development and Maintenance Issues
1.4 Metrics 16
1.5 Analysis 17
1.6 Design and Construction 18
1.7 Reviews and Testing 19
1.8 Summary 20
1.9 Solutions/Answers 21
1.10 Further Reading 21

1.0 INTRODUCTION

Today's Internet age has brought about a large demand for services that can be delivered over the network. These services rely in large measure on web software applications that can be accessed concurrently by many users. Increasing competition has meant that vendors want shorter and shorter cycle times for producing new products in any sphere, but in the arena of web based services, the situation is extreme. Instead of years or even months, new applications have to be readied and deployed in weeks, before someone else does so. Such pressures are compounded by the need to have high quality software as no one will visit a buggy website for the second time.

Besides technical challenges of architecture and design, web software brings in difficulties caused by the peculiarly distributed nature of the whole endeavour. Not only is the architecture such that the different tiers could be working out of potentially different machines located widely apart, the users are likely to be geographically separated by vast distances and could come from different cultural backgrounds. And, as likely as not, the development team could be spread out in space and these greatly separated team members being responsible for the software produced.

Web software could be –

- Meant for the Internet, with users coming from the public in a geographical region that could perhaps be the whole world.
- Meant for a closed group such as a corporation, in which case, it would be running over an Intranet.
- Meant for an extended, but still closed groups such as corporations, its customers and suppliers, where upon it is an Extranet.

Irrespective of the type of usage intended, the users could be separated by large distances and the application needs to behave correctly for all of them. In this unit, we will concentrate on web applications meant for the Internet.

We will also explore the difficulties associated with developing and managing a web based software project. Also examined will be some approaches and general rules

that could help us to do such a project well. Please be aware that the subject is vast, and in this short unit we can only touch upon the complexities of the topic.

1.1 OBJECTIVES

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

- identify the characteristics of a web based software application;
- describe the architecture of such an application;
- identify the challenges of managing a web based software project;
- think up relevant metrics to gauge the health of such a project;
- ensure proper understanding of the requirements in such a project so that development can be pursued, and
- facilitate quality control of a web based project.

1.2 DIFFERENT CHARACTERISTICS

While web based software applications can vary a lot in size and complexity, all of them share common characteristics, some of which are peculiar to them and are not prominent in other applications. We can look at these characteristics in terms of–

- The way the applications are developed, tested and deployed
- The way the applications are used
- The way they are maintained
- The design of these applications.

1.2.1 Characteristics of a Web Application

In this section, we will look at the overall architecture and deployment context of web based applications and see how this leads to their peculiar management challenges. Unlike conventional applications that can be monolithic, web applications are by their very nature amenable to layering. One cannot have a monolithic web application as the client and the rest of the application have to be of necessity separated. In principle, one could have a thick client application with a lot of intelligence embedded at the client end, but that would defeat the very purpose of delivering an application over the web. The idea is to have a client that is common to all users so that anybody can access the application without having to do anything special. This client is typically a browser of some sort that can render a HTML page on the user's screen.

Most of the processing is done at the other end of the application, that is, at the server. Here again there can be separation between the application and the data storage in the database. These two layers can be on separate machines that are themselves separated over the network.

The layering approach can bring about much greater flexibility and simplicity in design, maintenance and usage. In a monolithic application, everything is coupled together and in a layered application, we can change one layer without affecting the behaviour of others. For example, the business logic at the application layer can be changed without affecting the user interface or the database design. We can change the database management system from one vendor's offering to another without changing the application code or the user interface.

There can be many kinds of web applications such as –

- Those with static text
- Content that is changed frequently
- Interactive websites that act on user input

- Portals that are merely gateways to different kinds of websites
- Commercial sites that allow transactions
- Those that allow searches.

1.2.2 Development, Testing and Deployment

Although many basic processes and methodologies do not change, there are some features of the development cycle of web applications that are different. While some of these will be looked at in more detail later in the unit, we need to understand these features to be able to effectively develop a web application project.

Development

The development of such an application has to take into account the fact that there is now one more element in the whole behaviour of the application – the network. The element of communication could be ignored in conventional software, but it plays a pivotal role in web based projects. Different users can be using networks that are slow or fast and everything in between. Networks are not fully reliable and communication lines do drop or fluctuate in their speed. While network protocols will take care of the issues of data reliability, this comes at the cost of uniform speed. Designers of web applications need to take care of situations like race conditions, lost connections, timeouts and all the other tiresome realities of wide area networks.

Web applications, if used over the Internet, can potentially have a very large number of users. So we need to be careful about matters like machine configurations and sizing. It is also frequently necessary to have a scalable architecture that can be upgraded as application loads rise with the growing popularity of the site. And if the site becomes really popular, this need can be very frequent indeed!

Testing

The testing of web applications has its own difficulties, caused mainly by the variable speed with which inputs reach the server and responses reach the user over the World Wide Web. It is hard to simulate real Internet like conditions in test environments that are typically based on a local area network.

Testing tools do exist that allow inputs to be sent from machines located in different parts of the world to the application. While comparatively expensive, they do allow for realistic testing of the applications. There are other ways of testing the applications by trying to simulate delays and other conditions of the real Internet. To be able to obtain best, average and worst case response times, it is necessary to run tests many times. Automated testing is important for such load, performance and stress tests. They cannot be realistically performed manually.

Although interoperability and consistency are assumed to be the good features of standardized browsers and databases, in practice there are several problems that can arise because of the differing behaviour of browsers caused by–

- Different operating systems such as Linux, Windows versions or Unix systems.
- Different versions of the operating systems.
- Differences in other components such as the Java Virtual Machine.
- Differences in behaviour of the browsers themselves because of extensions or incomplete adherence to standards.
- Bugs in certain versions of the browsers on certain operating systems.

It is often impractical to test the application on all possible combinations of browsers, operating systems, databases and their various versions. In such cases the combinations for which tests have been performed are sometimes indicated and the

appropriate fixes have to be made to the application as bugs are discovered and reported during field use.

Deployment

In most conventional application rollouts, there is often a fair degree of control available to the developers because of the possibility of limiting access. Limited releases to a select test group can be made so that the most obvious and common errors and annoyances are taken care of before the regular release. But in a web application that works over the Internet, it can be difficult to do such tests. True, access can be limited by not publicising the URL at which the application is available, or by having access control in some form that is known only to the select band of users in the test group. Still, since the application is to be available publicly, there is a conflict between the small group which gets early access and the possibility of realistic testing. The very characteristic of a web application is having a large body of concurrent users.

We have already mentioned the need for scalability in the hardware. This is particularly important in terms of storage. We need to be careful that we do not quickly run out of space, for example, in an e-mail or storage site. Besides, it might be important to have the capability to beef up the bandwidth available to the server that is hosting the application, lest it get swamped by user requests.

1.2.3 Usage of Web Applications

Applications hosted on the web are meant to be used by lay users who may have little knowledge of computers. Conventional applications also have to deal with usage by non- technical people, but there we can often organise training or help desks to take care of any problems that may arise. That possibility is not available for a web based application. Because, we cannot reach and train all of our target audience, even if it were possible to identify them individually. As any developer with a little experience knows, users can “exercise” software in ways that were never anticipated.

Normally, if a user comes across difficulties while using the application, she could be expected to report the problem to the appropriate group in the organisation or to the vendor responsible for maintaining and deploying the software. But in the case of an application on the web, the user may simply give up and just not come back to the site if she is not able to work the page, and that too in the first few seconds. Or, her network connection might break while trying to use the site, and that could well be her last visit. So, the tolerance level of users is exceptionally low in such applications.

Users can also be expected to want a fast response and if some operation is likely to take more than a few seconds, it will be best to keep them informed. Feedback on the state of a request is also necessary, as also instructions on what to do if they are disconnected in the middle of a long operation.

An application on the web is much more likely to be internationalised as it might have to cater to the needs of users from different parts of the world. You may be shutting out a large number of potential visitors to your application if you restrict yourself to any particular language. You should also expect to see little data entry and as much as possible of features such as auto completion, drop down menus, look up tables and the like.

1.2.4 Maintaining Web Applications

The maintenance of web applications has its own set of challenges. Frequent changes to the presentation, or look and feel are expected. Many of the static text pages may need updating ever so often and some of the content could be changing daily or hourly. So the task of maintenance has to expand to take care of changing content and this can be a major exercise in some sites. Business rules could also be volatile, with new schemes and packages being frequently invented to entice the users. It means that the maintainers have to take care of frequently changing the business logic with little change to the other layers.

People are sensitive about sites that are perceived as not trendy enough and expectations are high as regards features, good looks and easily usable interfaces. One might have to work hard at changing and upgrading these to keep up with the latest advances. If a site is not changed for more than a few months, it will not be looked upon as live or up to date. So, taking care of a web application is often much more than just fixing bugs or updating the software to implement changes to business rules. There is a fair amount of effort that has to be put in to improve usability.

1.2.5 Designing Web Applications

Since web applications are layered, we get a simplified design that can be easily maintained and altered. However, that also produces the corresponding need to design the different components to be generalised enough so that they can be useful to a larger audience. Applications could be running under the control of an application server for which the designers have to create and implement the requisite beans. For some purposes, beans can be purchased from other software vendors or they could have to be written from scratch for the application.

These days, some functionality in the application could be coming out of web services, components on the web that can run with your application and that provide defined behavior. An application could look for an available web service that can do some needed action. Web services advertise their available functionality that needy applications can tap. This distributed nature of the application greatly increases the amount of reusability, but brings about problems of security and reliability.

The user interface in a web application is very important indeed and can often be the tipping point in the success of a website. Ease of use and ergonomic considerations are important. The application has to be able to cater to users of disparate skill levels and in some countries may be required to provide access to people with disabilities.

All the above characteristics need to be kept in mind when planning and otherwise managing a web based application project.

Check Your Progress 1

1) The _____ can bring about much greater flexibility and simplicity in design, maintenance and usage of web applications.

2) One of the significant characteristics of a Web Application is to have a large number of _____ users.

1.3 ISSUES OF MANAGEMENT OF WEB BASED PROJECTS

Like any other software application project, we need to use good software development practices when faced with working on a web application. Otherwise the project would not remain in control and we would face problems with timeliness, budgets and quality. Before going on to the special issues, it would be useful to review the general good practices that we need to follow.

1.3.1 Review of Good Software Development Practices

There is by now a wealth of literature and experience on how to manage software projects. While two decades ago the software crisis was a big stumbling block and failed or overdue projects were commonplace, the efforts of quality gurus have resulted in much better techniques of management that address these problems. Many projects still have trouble, but that is not because the techniques were not known but because they were not followed.

There are several quality models available to organisations that can be followed, such as the ISO 9001:2000 quality system model or the Capability Maturity Model Integration (CMMI) of the Software Engineering Institute of the Carnegie Mellon

University, Pittsburgh, USA. Let us look now at some of the basic, common principles of software management practice.

Managing Requirements

It is very important in any project to have a clear understanding of the requirements shared with the customer. Very often the developers have no notion of how the software will be used or what would be nice for the user to have. While a requirements document might be prepared and approved by the customer, there are many issues that are hard to capture in any document, despite having detailed discussion with customer. To ensure that the viewpoints of the developers and customer do not diverge, regular communication is a must. Finer points that could not be captured in the documentation have to be understood by the project team through proper coordination with the customer. Here the customer could be an external party or could be another group within the same organisation, in which case it is an internal customer.

Even if we achieve a good understanding of the requirements, software projects are notorious for changes. This means that we must have a good system of tracking requirements as they change. The rest of the artifacts in the project must reflect the changes that have occurred in the requirements.

Managing the Project

The essence of managing the project is proper planning for the project and then executing the project according to the plan. If the project deviates from the plan, we need to take corrective action or change the plan. We must always be proactive and keep looking for the risks in the project, rather than react to problems after they have arisen. We should try to anticipate what can go wrong and take action accordingly. Here a plan needs to keep something apart from the schedule of the project.

The project plan is the basic document used to execute the project and has all the required information about it. It should be the guiding document for the project and hence must be kept up to date. Some of the items of information in the plan could be–

- Background information and context
- Customer and other stakeholder information
- Estimates of cost, effort and duration
- Dependencies on outside groups
- Resource requirements – equipment and people
- Methodology to be used to do the project
- How the project will be monitored.
- Schedule.

Besides the project management plan, there are various subordinate plans that need to be made for different aspects of the project, such as,

- Configuration Management Plan
- Quality Assurance Plan that covers audits and process compliance
- Quality Control Plan that covers technical reviews and testing
- Risk Management Plan
- Training Plan
- Metrics Plan
- Defect Prevention Plan

There should be regular communication between the team members themselves to ensure that there is a shared vision and sense of purpose, and that perceptions remain aligned to the project goals.

As the project is being executed, there has to be regular monitoring of the progress and of the different parameters of the project, such as effort, cost and quality. If any of these parameters is not in line with what was planned, appropriate investigation, analysis and corrective action needs to be taken. Sometimes circumstances change such that it is no longer possible to bring back the project on track with the original plan, whereupon the plan needs to be reviewed and revised.

Configuration Management

This is vital to retaining control of the artifacts produced by the project. Given that changes will occur, we need to be able to track and know at all times which version of an artifact is current and which ones are obsolete. We do not discard obsolete versions because it may sometimes be necessary to backtrack or to branch off in another direction from a common base.

Whenever an artifact is produced it should be reviewed so that we can have confidence in the veracity and appropriateness of its content. After this it should be placed in a baseline area and any further changes to it should be done only according to formal change procedures. We have to plan for which artifacts will be so controlled.

Access to baselined artifacts should be limited to a designated configuration controller. Any member of the team who needs to alter a baselined artifact needs to approach the configuration controller who will then check out the artifact and allow the team member to make the change. Simultaneous checking out by more than one team member is discouraged, but if done, any conflicting changes that may be made by them need to be taken care of.

Measurements

It is important to measure our software work so that we can keep control of the proceedings. This is the key to being able to manage the project. Without measurements we will not be able to objectively assess the state of the progress and other parameters of the project, and we would be reduced to relying on our intuition and feel for where the project stands.

Software engineers have been notorious for not bothering to measure their work, or for insisting that their work cannot be measured. It is an engineering discipline unique in that measurements have not been given. However, in the 1990s, great progress was made in spreading awareness and appreciation of the need for software measurements. Still, even today, many organisations do not have a good metrics program going.

Risk Management

An important constituent of project management is managing the risks of the project. A risk is an event that can have a deleterious impact on the project, but which may or may not occur. Thinking of the possible risks is the first step in trying to take care of them. While we cannot get rid of risks, we need to keep thinking of ways to reduce their effects. Better still, we should try to take steps that will prevent the risk from becoming a reality.

Not all risks need to be looked at. When we analyse risks, there are some that are more probable and also have a significant adverse effect on the project. Others may have a severe effect if they occur, but are not too likely to happen. Still others may not be too harmful, even if they do come to pass. There can be a large number of risks in a project, but thinking about all of them is not practical. We need to concentrate on the most important risks and direct our energies to managing them.

Over time, the risk parameters can change. We, therefore need to keep re-looking at our risks and alter our risk plan to take cognizance of the changed situation as the project progresses. We should keep carrying out the actions that make the risk less likely. Whenever a risk does occur, we should put into action our plan for reducing its harmful effects.

Thus risk management allows us to look at a project actively, rather than passively reacting to events after they have occurred.

1.3.2 Organisation of Web Application Teams

Web applications tend to need much more ongoing support, maintenance and enhancement than others. After the initial application has been rolled out, comes the stage of maintenance. This can amount to weekly or even daily releases in the first instance while the software stabilises. This endeavour requires a team structure that is a bit different from others. We start from the maintenance end to emphasise its importance, and the fact that design and development are done comparatively quickly. Sometimes the need for project management during development itself is questioned but, as in any other project, good management is critical to success in a web application project as well.

Webmaster

This role is not unique to web applications, but is usually otherwise referred to as the administrator. It entails taking care of the site on a day to day basis and keeping it in good health. It involves close interaction with the support team to ensure that problems are resolved and the appropriate changes made. Some of her activities include –

- Gathering user feedback, both on bugs (such as broken links) and also on suggestions and questions from the public. These need to be passed on to the support team who are engaged in adapting the site in tune with this information.
- Ensuring proper access control and security for the site. This could include authentication, taking care of the machines that house the server and so on.
- Obtaining statistics and other usage metrics on the site. This could include, among many others –
    - Number of hits, distributed by IP address
    - Number of registered visitors
    - Frequency distribution of the different pages visited
    - Bandwidth utilised for upload and download
    - Number and type of errors, particularly of service denials.
- Helping to ensure that change control procedures are followed.
- Archiving old content while keeping newer content in a more prominent and easy to find location.

Application Support Team

In conventional software projects, while maintenance is important, it may or may not be done by the organisation that developed the project. In web applications, the organisation that developed the site is quite likely to be given the responsibility of its maintenance. This is because web applications tend to keep evolving and what corresponds to the development phase in a conventional application is here quite brief and frenetic. A large part of the evolution of the software tends to happen subsequently, based on user feedback and continuing refinement of the concept by those who conceived of the application. The activities here can consist of–

- Removing bugs and taking care of cosmetic irritants.
- Changing the user interface from time to time for novelty, to accommodate user feedback and to make it more contemporary.
- Altering the business rules of the application as required.
- Introducing new features and schemes, while disabling or removing older ones.

The support team could consist of designers including graphic artists, programmers, database specialists and testers.

Content Development Team

In conventional business application software, there are usually changes and modifications required to master tables in databases, such as altering price lists, available items and so forth. Web applications frequently require much more ongoing, sustained effort to retain user interest because of the need to change the content in the site. In the case of a news site, this updation can be every few minutes. The actual news could be coming from a wire feed from some news agencies, or from the organisation's own sources.

Though not all sites need to be so current, it is often required to keep adding articles and papers of interest to the user community. These stories or articles are written by a team that has expertise in the domain concerned. A site could be serving out a wide variety of information and other stories from different interest areas. Content development teams could be researchers who seek out useful or interesting facts, authors, experts in different areas, and so on. The content may be edited before being served out.

The members of this team are usually not knowledgeable from the software point of view, though they might be well respected in their own domains. They may be full time members of the organisation or casual, freelance contributors. So though called a team, some of them might really be individual providers. The content they produce is typically in the form of a word processor file that might have images, graphs, tables and other non-text ingredients.

Other forms of content are now quite commonplace, such as audio files, slide presentations or video clippings of events, places or speakers. With increasing availability of high speed network connections for even individual users, such content is now becoming increasingly popular because of the impact it can produce.

Web Publisher

This is an important role that connects the content development team to the actual website. The raw material created by the writers has to be converted into a form that is suitable for serving out of a webserver. It means formatting the content according to the requirements of a markup language such as HTML. Though tools can help perform much of this, there still might be human effort that is needed.

In the case of automated news feeds, such publishing needs to be as tool driven as possible. The web publisher must have a good understanding of the technical aspects of web servers and the markup language.

1.3.3 Development and Maintenance Issues

Let us now look at some of the management issues that we will face while developing and maintaining a web application. We have already seen that web applications tend to evolve, and there is little distinction between what in a more conventional project we would refer to as development or maintenance. It is hard to say when the development is complete and maintenance starts in a web project. In practice it is a continual cycle of delivering something and then of fixing and correcting and enhancing it.

Configuration Management

During maintenance of a web application, because of the frequency of changes, configuration management assumes considerable importance. The change control process must be up to the demands placed on it of keeping control over the software configuration inspite of the many changes that will happen. So it must not be overly elaborate or hard to follow. That would mean it might not get followed and the consequences will be disturbing. Once one loses track of what is happening in the software it can become very difficult to get back. It will then become impossible to

predict the effect of changes to the software and one would have to grapple with mysterious bugs whose cause will be hard to find.

Besides the software itself, content also needs to be subjected to configuration management. This could include items such as information on schemes or other service or product offerings. Lack of discipline here could mean horrors like –

- Postings on past offerings that are no longer available,
- Incorrect prices or other terms being visible to users,
- Postings of content that is still work in progress and was not ready for publication apart from lesser embarrassments such as archival content reappearing on the site.

A very important aspect is that of inadequately tested code finding its way into the active configuration. It could mean that visitors to the site have to put up with features that do not work as expected or simply crash on them. Worse, it can result in security holes that compromise the security and integrity of the site. Sometimes, the privacy of data of registered users could be in question. In other cases it could expose other systems of the organisation itself to unauthorised intruders.

Since contributions to an organisation's website can come from diverse sources and departments, it can become difficult to establish accountability and responsibility for the web application. In conventional business software applications it is clearly a particular division or group in the company that is served and that would have an interest in the software working properly. This interest is often diffused and spread out in a web application. It cannot be the technical team or the software wing, as this is often an outsourced vendor. In any case the ownership of the application needs to be assumed by the business and not a technical group. While there can be many claimants for ownership and authority, the same may not hold when it comes to owning responsibility.

To determine who is responsible for the website, some of the relevant questions can be –

- Who pays for the website (this can be a charge to an internal budget)?
- Who is concerned with the accuracy of the content that is published?
- Who gives the requirements to the development or maintenance team?
- Who ensures quality control checks and does the acceptance check?

It is this owner of the website who should be ensuring that proper configuration management practices are followed. Even if this question is resolved, we have to grapple with how exactly to do configuration management. The identification of configuration items can be difficult. This is because there are many components to a web application, and some items such as news content can be very short-lived. Other content can last longer. To what extent does one look at configurations for transient content? Perhaps just recording when it was published and when it was removed is sufficient. For other content that may appear for days or weeks, we might have to deal with normal, conventional changes and updates. What constitutes a configuration unit is also not easy to define, as hyperlinks from within content can complicate the matter.

Conventional configuration management tools are also not really suited to deal with this kind of constant evolution. The usage of these tools is also made difficult because many in the maintenance team, such as content developers, may not be familiar with software, let alone the tools. Also, web application support teams may not be geographically located in the same place and people could be working in different places. So the tools used need to be such that they can be operated over the network.

Project Management

If configuration management has its challenges as described above, they pale into insignificance compared to the difficulties we may have while managing the web

application project itself. When we look at the elements of project management the reasons will become apparent.

1. We have already seen that evolution and hazy, fluid requirement specifications are the norm here. How then does one perform estimation? That presupposes a clear knowledge of the scope and specifications.
2. More than in any other kind of software, schedules are dictated by the users. Quite often, the software is the means to seize upon a business opportunity that may be available for a small window in time. So the developers or maintainers may not have much influence on the time they can get. Whatever the estimate, the software has to be delivered by the date the user needs it.
3. Coordination and human issues can be more daunting than ever. The team may be geographically dispersed, may come from disparate backgrounds and may have conflicting interests or vision.
4. The question of ownership we have already seen in the section on configuration management. Even though stakeholders may not be hard to identify, overall accountability and responsibility could be.
5. The metrics to use to gauge success have to be identified carefully, keeping in mind the unique characteristics of a web application.
6. The actual engineering aspects, particularly analysis, design and testing have to be tuned to the life cycle of such a project. Agile programming methodologies may be of help here.
7. Quality assurance is made more difficult by the frenzied pace of activity that may induce the harried project team to jettison process driven working.

While none of these challenges permit of simple solutions, we can try to keep some things in mind and work towards keeping the project under control by –

- Recognising that the software will follow an iterative life cycle model for development, with each iteration being quite short. We may not be able to work out a grand plan for the whole project as it will be at the end, simply because we cannot predict what shape it will take eventually. So we could concentrate on the current iteration and be clear about what we will deliver in that one. Other items on the user wish list should be taken care of in the next delivery. Save in exceptional circumstances, do not alter the objectives or scope of the current iteration.
- The question of schedule is a difficult one. Managing expectations of the users is imperative here. If what the user's desire is clearly unattainable, we need to negotiate and reduce the scope for the current iteration. Also with more experience in getting a web application developed, the users will hopefully have a better appreciation of what is possible.
- With current day groupware and other tools, we can make it simpler to work with distributed teams. But there should be adequate attention paid to communication among team members so that the objectives of the project are clear. As in a conventional project, we should have team meetings regularly, even if those meetings are done over the telephone or using network meeting tools.
- As always, the processes to be followed in the project should be appropriate to the situation. They should not be cumbersome or difficult to do, so that the project team will be able to follow them. We need to remember that if the processes are complex and are not followed, the impact on the project is much worse, as we potentially could lose control of what is happening.

In subsequent sections we will be reviewing the strategies to follow to get a grip on the other issues such as life cycle activities and measurements to be made. The other, usual elements of project management such as managing risks, analysing defects and

trying to prevent them or monitoring project parameters such as cost, effort, schedule and quality, remain as valid as in any other software project.

Check Your Progress 2

1) The essence of managing the project is proper _____ for the project and then executing the project according to the plan.

2) The _____ process must be up to the demands placed on it of keeping control over the software configuration in spite of the many changes that will happen.

1.4 METRICS

Management of any kind is difficult, if not impractical, without measurements being made. Those values tell us where we stand in our endeavour, whatever it may be. Software engineers have been especially tardy in understanding the need for measurement, preferring to rely on intuition and subjective assessment. However, realisation does seem to have set in and many software projects do measure at least the basic parameters like schedule, effort and defects. When it comes to cost, practices seem to vary across organisations. Some are secretive about the costs of projects and only senior management is privy to such information. Others place the responsibility for the budget for the project on the shoulders of the project manager.

In web applications, these core metrics need to be measured and analysed just as in any other software application. Schedules could be made fine grained so that we are tracking things on small, quick milestones rather than only at the "end" of the project, which is difficult to define here. That could mean that it may not be very meaningful to measure them in terms of slippages (a one-day delay in a task lasting a week is 14%). One possibility could be to look at the percentage of schedules that could not be kept.

Effort metrics can be gathered through a time sheet and we can measure the total effort in different kinds of activities to build up our metrics database for future guidance. However, here we need to be sure to record the actual effort and not try to confine it to a conventional workday of 8 hours. Metrics data on effort is valuable in building up a body of knowledge for future estimation. This data should be recorded under time spent on analysis, design, construction, project management and so forth. Past data is also helpful in negotiating with users for a reasonable amount of time to perform a task

Another core attribute of a software project is its size. This should preferably be measured in an implementation independent way, such as function points. The International Function Point Users Group (IFPUG) has come up with guidelines for measuring the size of a web application from the user point of view. If the organisation has good reason to feel that this measure is not suitable, then the creation of one's own size measure could be considered. If a good size measure can be built, then it can be used to estimate the effort required for future projects, provided the productivity factor that converts the size to effort is available from data on past projects.

The quality of the application can be measured in terms of defects that are found during internal testing or reviews as well as external defects that are reported by visitors who come to the site. But all defects are not equal, and we need to analyse defect data carefully. Some attributes that can be used to classify defects for further analysis are:

- Severity on a scale of 3 levels, such as high, medium and low. More levels up to 5 can also be considered. The severity is in terms of impact on the usage of the software by the user.

- Phase in the life cycle where they were introduced, such as analysis, design or coding.
- Whether they are internal defects, detected by the development or testing team, or whether they were found by the end users.
- Whether they were detected during reviews or during testing.
- Effort required to fix the defect, as this might not have any correlation to the perceived severity of the defect.

Such an analysis would then form the basis for the formulation of a defect prevention plan for the software project. We would strive to prevent the kind of defects that are of high severity or those that take a lot of effort to remedy.

We can also look at other measures like –

- Percentage human resource utilisation in the project. Roles that are idle for significant periods could be allocated to another project as well.
- Percentage of code that is reused from the organisation's library of such components.
- Quantum of changes to the user requirements as the project progresses. An overly high value could indicate that more effort needs to be directed towards scoping out the project and eliciting the requirements.
- Percentage of attrition in the project. An unusually high value for this could point towards problems in managing the human angle.

Besides metrics to give insight into the development aspects of the project and the general state of the team, there can be a large number of statistics and measures that can be gathered about the usage of the website once it has been put into active use. Some of these have been already touched upon in earlier section under the role of Webmaster. Most of them rely on traffic analysis to draw conclusions about the users and their interests. This can give an insight into what changes should be made to the site or how it can be improved.

We can also look at some metrics from the point of view of maintenance. These could be the average time taken to fix bugs of different severity levels, together with the best and worst case times. Other measures can include –

- User satisfaction index
- Mean time between failures (bugs detected or reported)
- Number and percentage of repeat visitors.

Gathering and calculating metrics is only one part of the work involved here. The crux of the matter is how well the knowledge of metrics is used to shape the functionality and features of the project. The metrics should be aligned to the goals and objectives of the project and the lower level measures should be derived from the top level ones, so that we measure whatever is important for us and do not gather extraneous data. From a measurement deficient situation it is quite possible to go overboard and drown oneself in a mass of low level data that is not meaningful or useful.

1.5 ANALYSIS

The usual principles of analysis continue to apply to a web software application as well. So the requirements have to be, as always, elicited, represented and validated. These are then the basis for further work in the project. Let us look at some of the characteristics that we need to bear in mind while analysing a web application project.

It is particularly important to be clear about the process owner and the stakeholders here. We must not end up getting wish lists and requirements from someone who, it later turns out, is not the one authorised to provide them. While always a point of concern, the issues of ownership of the web application accentuate the problem. From the scope that would have been talked about while setting up the project, the detailed system analysis would need to be done to come up with the requirements.

Besides the normal considerations of what the application has to do, we need to look at how the user will interact with the application even more closely than in a conventional application for the reasons we have already looked at in the beginning of this unit. The user interface has to be simple to use, visually appealing and forgiving of a lay visitor who may not be familiar with computers or software. The actual functionality of the application is something that has to be ensured the same way as it would be for any other software, being the very reason for going to all the trouble. Where things are different is in the importance of content to a web application. We have to work out what will be shown to the user, such as a product brochure, an offering for a tour package or anything else. This has to be placed before the user in a manner that will be eye catching but not garish, and the user has to be able to obtain some benefit from the offering without having to do anything elaborate.

More than in conventional applications, it is hard to distinguish entirely between analysis and design in a web application. The kind of actual content that has to be shown needs to be studied to decide on the best way of displaying it on the site. Whether the user will like to show it as text, graphics, audio or video is something that will depend on the resources available for hosting the site as well as on considerations that have to do with the resources available with the group of people that can be expected to visit the site. It is here that the study of the target audience becomes crucial. This should be done on considerations that have to do with –

- Geographical region,
- Language, beliefs and culture,
- Social strata,
- Age group and gender.

and other similar characteristics. All of these demand that we change our design to appeal to our typical visitor.

1.6 DESIGN AND CONSTRUCTION

In designing the functional aspects of a web application, we work much as we would for a conventional application. Because of the compressed time scales and volatility, we should consciously look to reusing as many things as possible. Reuse is always desirable but becomes imperative in the case of a web application. While code and object libraries come first to mind when we think of reuse, there are elements that can be reused for the design as well. We should, for instance, be seeking out any available design patterns that can be reused.

The advent of application servers and other component-based technologies has facilitated the task. The application server gives the framework under which the application can run and takes care of many mundane details that would otherwise be the burden of the programmer. Security and persistence are two such tasks. The functionality of the application is provided by stringing together beans, many of which may have been already written or can be purchased from third parties. There are also other architectural models from other vendors to help write such applications.

Where we have to lay great emphasis is on the user interface. There are several aspects to creating a good one, some of which are:

1. **Visual appeal and aesthetics:** This is the domain of a graphic artist. Things such as cute sound effects, blinking and moving images that can appear interesting or attractive for the first couple of times can irritate after a while. So a pleasant, clean visual design is perhaps a safe bet to hold continuing interest on repeat visits. However, if there is ever a choice between graphical appeal and functionality, we must plumb for functionality every time.
2. **Ease of entry of data:** Given the fact that most users may be unfamiliar with computers or software, data entry should be as simple as can be made. Actual entry using the keyboard should be kept to the minimum. Allow for using the mouse for selection from lists. It should be easy to ask for information of the application.
3. **Ease of obtaining the application's response:** Whenever we query the application, the results should be easy to see and interpret. If the processing is going to take a while, give visual feedback to the user on how long she can expect to wait. At no point should a user feel lost, not knowing where s/he is in the site or what she should do next. Context sensitive help should be available throughout the site.
4. **Intuitive, easy navigation around the site:** As a visitor navigates around the site, there should be some aids to helping her find her bearings. Things like a site map and your location on it are useful. It should be possible to come back to a reference point like the home page in a single click from anywhere deep in the site. It is our application that should provide all navigation facilities and we must not expect the user to use browser buttons to go around.
5. **Robust and forgiving of errors:** If there is any erroneous input, the consequences should not be too inconvenient. We should expect a discreet and graceful error message, not a disconnection from the server. It should not happen that we land up in some error page that does not give us any explanation of what went wrong.

Website users would not like to have to scroll and so we must try to see that this is not required for any screen resolution. A horizontal scroll is much more irritating than a vertical scroll to most users.

1.7 REVIEWS AND TESTING

In this section we will look at some of the considerations for reviewing testing web applications. The content of a website is the item of interest to the visitor and so we must take care to review all content to make sure that there are no errors of fact or language therein. Besides, the design of the site should be reviewed to catch errors like difficulties in navigation or data entry.

It may not be very useful to merely check out the components of a website on a standalone basis as part of unit testing. While in a conventional application a program is the smallest unit that we test, in web applications it is more likely to be a single web page. We should check out the content, the navigation and the links to other pages as part of testing out a page. There should not be any broken links that leave the user bemused. If there is some functionality being provided by that page, that part should work correctly. This part of the software is close to a conventional application and the actual processing that happens can be unit tested as usual. Where the processing is done by using bought out components, it may not be necessary to unit test those components. We would then only have to concentrate on checking out the full functionality.

Once all the pages of the web application are ready we can perform integration testing. We check out how the pages work together. Whenever an error is caught and fixed we should repeat the tests to ensure that solving a problem has not caused

another problem elsewhere. It is possible that individual pages work well but not together.

Unlike a conventional application, here we do not have much control (except our instructions to users) on all parts of the working system. This is because we do not know what browser, what version of it and what operating system the user will be working with. While it may not be practical to test out with all possible combinations, we should ensure that the application works with most sets of browsers and operating systems. We should also see that the software works at different screen resolutions. The network bandwidth expected should also be clearly decided and we should test things out at all network speeds up to the lowest.

The actual implementation of the application has to be done carefully. In the deployment environment, the configuration has to be checked out to be in tune with what the application was designed and constructed for. Thereafter we may release the software to a limited test group of users to gather their feedback before making it public.

☛ Check Your Progress 3

1) The _____ gives the framework under which the application can run and takes care of many mundane details that would otherwise be the burden of the programmer.

2) In the _____ environment, the configuration has to be checked out to be in tune with what the application was designed and constructed for.

1.8 SUMMARY

This unit has been a very brief introduction to different aspects of engineering web software applications. While a lot more could have been written about each topic, and other topics could have been added here, we have been able to concentrate on only a few important points here that are with in the scope.

- Web applications have many points in common with conventional software applications, but have their own unique characteristics.
- There are differences in the ways web applications are developed, deployed, tested and maintained.
    - Web applications often do not have a clear dividing line between development and maintenance and usually continue to evolve.
    - They have to be developed and put into use quickly and so try to make use of reusable components as much as possible.
    - The user interface is very important because they are used by lay visitors.
    - Testing them can be difficult
- Following good management practices like project management and configuration management of web applications is not straightforward because of their characteristics.
- We need to think through the metrics we will use to evaluate project progress and product quality of web applications.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Layering approach
2) Concurrent

Check Your Progress 2

1) Planning
2) Change control

Check Your Progress 3

1) Application server
2) Deployment

1.10 FURTHER READING

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

Reference Websites

- http://sei.cmu.edu/cmmi
- http://standards.ieee.org

UNIT 2 MOBILE SOFTWARE ENGINEERING

Structure **Page Nos.**

2.0 Introduction 22
2.1 Objectives 22
2.2 Introduction to GSM 22
2.3 Wireless Application Development using J2ME 23
2.4 Introduction to Java Device Test Suite 28
2.5 Summary 28
2.6 Solutions/Answers 29
2.7 Further Readings 29
2.8 Glossary 29

2.0 INTRODUCTION

This unit introduces learner to the subject of development of mobile applications. With the increased usage of mobile devices across the world, more and more applications are being targeted at them. If an application that is developed for computers can also run on the mobile devices, then the entire paradigm of application usage changes as it enables the user to run application from wherever he is.

There were different standards on which the mobile networks are based. One of the popular standard is GSM. In this unit, we introduce the GSM architecture. There were a number of environments under which mobile applications can be developed. One of the popular environments is Java. We introduce J2ME (Java 2 Micro Edition) in this unit in conjunction with wireless application development with it. Any application should be tested before being put to full-fledged use. In this unit, Java Device Test Suite is introduced with the help of which wireless applications can be tested.

2.1 OBJECTIVES

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

- know the GSM architecture;
- know about wireless application development using J2ME, and
- know about Java Device Test Suite.

2.2 INTRODUCTION TO GSM

Mobile devices are used by a number of people across many countries. In this unit, mobile devices refer to mobile phones. Every country is having a set of standards on which the mobile devices in their country are based upon. GSM is one of the popular architectures on which mobile devices are based on. GSM stands for Global System for Mobile Communications.

GSM is a digital wireless network standard. All mobile devices that are based on GSM standard across the world will have similar capabilities.

The following are some of the features of GSM:

- If a mobile device is based on GSM, then it can be used in all those countries where this particular standard is prevailing.

- Almost all the services that are existent in a wireline network are provided by GSM to all the users of mobile devices which are based on it.
- Though the quality of voice telephony is not excellent, it is not inferior to the systems that are analog based.
- It also provides good security as there is an option to encrypt the information that is being exchanged using this standard.
- There is no need for significant modification of wireline networks due to the establishment of networks based on GSM standard.

Architecture of GSM

There are three different parts in a mobile device. They are SIM card, Mobile equipment and Terminal equipment. SIM stands for Subscriber Identity Module. A mobile device can be loaded by any SIM. The number of the mobile device is associated with the SIM. The size of a SIM is similar to that of a smart card. Every SIM is also having a PIN (Personal Identity Number) associated with it. After loading the SIM into the mobile device, the user is asked to enter the PIN. Only on entering the correct PIN, the user will be able to start using the services offered by the mobile operator. Initially, PIN is loaded by the operator. The PIN can be changed by the user. If the user is unable to enter the SIM correctly for the first time, then s/he is given an opportunity to re-enter it for a fixed number of times. In case, s/he is not able to enter it correctly and exhausted all the tries, then the PIN is blocked. So, that particular SIM cannot be used unless the network operator activates it. The SIM can be unblocked only on entering the correct PUK (PIN Unblocking Key). Information related to subscriber, PIN and PUK codes are present in SIM.

The architecture of GSM is composed of Mobile devices, Base Station Systems (BSS), Mobile Switching Center (MSC) etc. The communication between Mobile device and BSS is through radio interface. BSS communicates with MSC by connecting to Network and Switching Subsystem (NSS). An MSC is a telephone exchange that is specifically configured for mobile applications. Mobile devices and Public Switched Telephone Network (PSTN) interface through Base stations. BSS consists of a Base Transceiver Station (BTS) and Base Station Controller (BSC). Equipment related to transmission, reception and signalling is part of BTS and this equipment is used by it to contact Mobile devices. BSC deals with the allocation of radio channel and its release apart from hand off management. Using ISDN protocols, BSC communicates with BTS.

Apart from GSM, there are other standards such as CDMA etc.

2.3 WIRELESS APPLICATION DEVELOPMENT USING J2ME

Java supports mobile application development using its J2ME. J2ME stands for Java 2 Platform, Micro Edition. J2ME provides an environment under which application development can be done for mobile phone, personal digital assistants and other embedded devices. As like any other Java platform, J2ME includes API's (Application Programming Interface) and Java Virtual Machines. It includes a range of user interfaces, provides security and supports a large number of network protocols. J2ME also supports the concept of *write once, run any where* concept. Initially, an application can be developed using J2ME targeting a specific type of devices. Then, the same application can be used for different types of devices. Also, it is possible to use the native capabilities of these devices. J2ME is one of the popular platforms that is being used across the world for a number of mobile devices, embedded devices etc. There is a huge market for applications that target wireless world. Some of the

wireless applications are Games, Applications that access databases, Location based services etc.

The architecture of J2ME consists of a number of components that can be used to construct a suitable Java Runtime Environment (JRE) for a set of mobile devices. When right components are selected, it will lead to a good memory, processing strength and I/O capabilities for the set of devices for which JRE is being constructed.

Currently, two configurations are supported by J2ME. They are Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC). Each configuration consists of a virtual machine and a set of class libraries. Each configuration provides functionality of a particular level to the set of devices that have similar features. One of the features is network connectivity. To provide a full set of functionalities, a profile and an additional set of APIs should be added to the configuration. These additional APIs will enable the usage of native properties of devices as well as define suitable user interface. Each profile will support a smaller set of devices among the devices that are supported by the chosen configuration.

In the case of J2ME, often CLDC is combined with MIDP. MIDP stands for Mobile Information Device Profile. This combination provides a Java based environment for cell phones in which applications can be developed. It is always possible to add optional packages to the configuration that was already selected to support an additional set of services. An optional package consists of a set of standard APIs that help us to use latest technologies such as connecting to databases from cell phones, blue tooth, multimedia etc. It is never mandatory to use optional packages. In fact, it is to the developer to select those optional packages which provide the additional functionality as per his desire.

CDC is the larger of the two configurations supported by J2ME. CLDC is designed for devices that have less memory, slow processor and unstable network connections. CLDC is suitable for cell phones. Such devices have a memory that ranges from 128KB to 512KB. Java has to be implemented within this memory. CDC is designed for devices that have larger memory, faster processors and a significant network bandwidth. TV set top boxes, high range PDAs (Personal Digital Assistant) are examples of the devices that are suitable for CDC. CDC consists of a JVM (Java Virtual Machine) and Java software that includes a significant portion of J2SE (Java 2 Standard Edition) platform. Usually, devices that are suitable CDC at least possess 2MB of memory in which Java software can be loaded.

MIDP is designed for cell phones and other lower level PDAs. User interface, network connectivity, local data storage and application management are offered by MIDP for applications that are based on mobile devices. The combination of MIDP and CLDC provides a full-fledged Java runtime environment for the mobile devices. The environment provided by their combination will add to the features of the mobile devices. This will lead to a better power consumption and efficiency in memory utilisation. The lowest level profile for CDC is known as Foundation Profile (FP). FP provides embedded devices which does not have a user interface with network capability. There are two more profiles, namely, Personal Basis Profile (PBP) and Personal Profile (PP). FP can be combined with these profiles to provide GUI (Graphical User Interface) for the devices that require it. All the profiles of CDC are in a layered manner. This will facilitate adding as well as removing profiles as per the need of features.

PP is useful for all the devices that require a GUI, Internet support etc. Usually, devices such as higher level PDAs, Communicators etc. need such features. Java AWT (Abstract Window Toolkit) is contained in this profile. It also offers web fidelity, applets etc. PBP is a subset of PP. PBP supports application environment to those devices in the network that support at least a basic graphical representation. PP as well as PBP are layered to the top of CDC and FP.

There are several reasons for the usage of Java technology for wireless application development. Some of them are given below:

- Java platform is secure. It is safe. It always works within the boundaries of Java Virtual Machine. Hence, if something goes wrong, then only JVM is corrupted. The device is never damaged.
- Automatic garbage collection is provided by Java. In the absence of automatic garbage collection, it is to the developer to search for the memory leaks.
- Java offers exception handling mechanism. Such a mechanism facilitates the creation of robust applications.
- Java is portable. Suppose that you develop an application using MIDP. This application can be executed on any mobile device that implements MIDP specification. Due to the feature of portability, it is possible to move applications to specific devices over the air.

You can use J2ME wireless toolkit for development of wireless applications using J2ME. It is possible to set up a development environment by using the following software:

- J2SE (Java 2 Standard Edition) SDK (Software Development Kit) of v1.4.2 or upwards.
- J2ME (Java 2 Micro Edition) Wireless toolkit. Using this toolkit, MIDlets can be developed.
- Any Text editor.

MIDlets are programs that use the Mobile Information Device Profile (MIDP).
The first step is to install J2SE SDK. J2ME wireless tool kit runs on the Java platform provided by J2SE. J2SE needs to be installed because it consists of Java compiler as well as other requisite software which is used by J2ME wireless tool kit to build programs.

The second step is to install J2ME wireless toolkit. It is possible to develop MIDP applications using J2ME wireless toolkit. The J2ME wireless toolkit uses the concept of projects rather than files. Once J2ME wireless toolkit is installed and run, we can start creating projects. At the end of the creation of the project, you have on MIDP suite. Once the project is created, its properties can be changed. Also, the project can be build and executed in the device emulator. Let us create a new project. Let the title of the project be project1. Along with the title of the project, MIDlet class name should also be given. Let it be projectlet1.

Once these names are confirmed, the process of creating the project project1 commences. As the result of creation, a directory titled project1 will be created in which the following subdirectories will be present:

- bin
- lib
- res
- src

A compiled MIDlet suite (a .jar file) and the MIDlet suite descriptor (a .jad file) are created in the bin directory. Extra JAR that are to be used in the project may be placed in the lib directory. Any resource files such as images, text files etc. may be placed in the RES directory. The source code created by you will be in SRC directory. Apart from the above mentioned subdirectories, the following subdirectories are also created:

- tmpclasses
- tmplib

But, these subdirectories are not used by the developer explicitly. They are basically used internally.

The details of code with which MIDlets can be developed is a larger topic and references may be studied for it.

Once the MIDlet projectlet1 is developed, it should be saved as projectlet1.java file in the SRC directory. The following is the exact location of this file:

- D:\WTK22\apps\project1\src\projectlet1.java

Now, build the project. After successful build, run it. Upon running, a mobile phone emulator will pop up. In the emulator, the title of the project will be displayed along with a button labelled Launch. On clicking it, the MIDlet developed will be executed and the result is displayed on the screen. There is a button labelled Exit at the bottom of the display on which the MIDlet is executed. If the Exit button is clicked, then, we exit the MIDlet that is executed. Now, there are two ways to exit the emulator. Either the emulator window can be closed or ESC key can be pressed.

There are a number of emulators provided by J2ME wireless toolkit. One of them can be selected from the KT (ToolKit) tool bar. After selection, we need to execute the project again.

When the project is build, the toolkit will compile all the .java files in the src directory. The compilation is performed in the MIDP environment. There were different APIs in the MIDP and J2SE environment. When the project is being compiled in the MIDP environment, then the APIs from MIDP should be considered for the classes that are used in MIDlets. The corresponding APIs in J2SE are not used. All MIDP classes are verified in two phases before they are loaded and executed. At build time, the first verification is done by J2ME wireless toolkit. When classes are loaded, the second verification is done by device's runtime system. Finally, all MIDlets are packaged into MIDlet suites. These suites are distributed to actual devices.

Along with the J2ME wireless toolkit, information about the following is also provided:

- Application development cycle
- Attributes of MIDlet
- Files in the installed directories
- Device types
- Portability
- Guidelines on the process of configuring an emulator
- Usage of J2ME wireless toolkit.

Until now, we focused on the development of MIDlet. It is possible to do a bit of extra work and make these MIDlets work across networks. For this to happen, we may need to develop servlets to whom MIDlets can establish network connections.

For developing servlets, we need server. Tomcat is the server that can be used for this purpose. There are a number of servers available and every server is having its own advantages and disadvantages.

The first step is to install and run Tomcat. The latest version can be downloaded from the website of Apache Jakarta Project. It comes as a ZIP archive. After downloading, it can be installed into any directory. Tomcat is written in Java. Tomcat needs the location of J2SE to run. Hence, the location of J2SE should be set in the

JAVA_HOME environment variable. Once, all these steps are performed, then, open the command window. Change to the bin directory of Tomcat. Type *startup* at the prompt. That's it. Tomcat starts running. Don't worry about the things that are displayed on the screen once you start it.

It is possible to check whether Tomcat is really running or not. For that, just open the browser window. Try to open http://localhost:8080/. If a default page from Tomcat with links to servlet and JSP examples is displayed, then, it means that the Tomcat is running. To shutdown the Tomcat, open the command window. Change to bin directory of Tomcat and then type the command *shutdown*. That's it. Tomcat starts shutting down.

Now, suppose that a servlet called *counter* is developed. The name of the file will be counter.java. It should be saved to the root directory of Tomcat. Once it is saved to the root directory, it can be compiled. To compile servlets, there is need for servlet API. This should be added to the CLASSPATH before compiling servlets. The servlet API is present in common/lib/servlet.jar under the Tomcat directory. Now, the counter.java servlet can be compiled using javac as follows:

D:\>javac counter.java

Let us place this servlet in a web application. The servlet can be saved to a directory in webapps directory of Tomcat. Let this directory be abcd. Tomcat will know about the new web application through its configuration files. The necessary changes to the configuration files needs to be done by the developer. This can be done by opening the server.xml file from conf directory and adding a context entry.

The reason for performing all these steps is to handle the incoming http requests. If the request begins with /abcd, then the request will be sent to the web application that is located at webapps/abcd. The most important file in any web application is web.xml. It is always stored in WEB-INF directory. This is the time when we start thinking about making the servlet counter.java accessible to the entire world. Suppose that counter.java is placed in a directory called count.

Now, the following is the full path to the servlet:

http://localhost:8080/abcd/counts

Now, necessary changes are to be done to web.xml file so that Tomcat will know that counter servlet should be mapped to abcd/counts.

As we have seen, both servlets and MIDlets can connect to the world via http. Hence, connecting MIDlet to servlet is not a complicated issue.

The following are different steps in the process of connecting MIDlet to a servlet:

- Start Ktoolbar. This is part of J2ME wireless toolkit.
- Open the project1.
- Write the code for the MIDlet (let it be *countmidlet.java*) that connects to counter servlet.
- The screen of countmidlet.java (after executing it) consists of two commands namely Exit and Connect. Clicking on Connect will lead to the invocation of connect( ) method that will establish a network connection. It will also transport the result.
- The countmidlet.java should be saved to apps/project1/src directory under J2ME wireless toolkit directory.
- Now, J2ME wireless toolkit should know that a new MIDlet is added to it. This can be done by going to Settings→ MIDlets→Add. Enter countmidlet for both the MIDlet name and class name. Click on OK.
- Go to Settings→ User Defined. Add the property name as countmidlet.URL. This URL will invoke counter servlet.

☞ **Check Your Progress 1**

1) J2ME never needs J2SE. True ☐ False ☐

2) Tomcat can be used for developing _____.

3) MIDP stands for _____.

2.4 INTRODUCTION TO JAVA DEVICE TEST SUITE

Using Java Device Test Suite (JDTS), it is possible to test the J2ME applications. The implementations of CLDC and MIDP can be evaluated using this test suite. They can also be validated and verified.

The following are different types of tests can be conducted using JDTS:

- Functional tests
- Stress tests
- Performance tests
- Security tests.

During functional testing, the application is tested whether it behaves as intended or not. It's behaviour with both internal and external components is checked. During stress tests, the application will be run under the upper limits of the memory and the processing power. During performance tests, response time etc. are evaluated. Under security tests, the MIDlets are checked for their security model which also includes access permissions.

It is possible to include additional test suites to JDTS depending on the requirement. This will enable the developer to execute a select set of test cases. It is also possible to automate the entire testing process. But, in this case, test results does not include the cases where problems may arise due to interaction with application. In JDTS, there is a Test Console as well as a Test Manager which help developers to test their J2ME applications.

The following are some of the features of JDTS:

- More than one test suite can be executed at once.
- Test Manager can accept connections from multiple test consoles concurrently. Each test console might be testing several devices with the same configuration.
- Test reports are generated in HTML format.

☞ **Check Your Progress 2**

1) Using Java Device Test Suite, _____ applications can be tested.

2) Only one test suite can be executed at a time in JDTS. True ☐ False ☐

2.5 SUMMARY

The most basic issue that governs mobile devices is the standard on which they are based on. GSM, CDMA etc. are some of such standards. In this unit, GSM is introduced. Its architecture is also discussed. One of the most popular environments under which wireless applications can be developed is Java. J2ME is the platform under which wireless applications can be developed. The entire process of wireless application development using J2ME is discussed in this unit. Finally, Java Device

Test Suite, which is used to test wireless applications developed using Java is discussed. Different types of tests that can be conducted using are also explained.

2.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1) False

2) Servlets

3) Mobile Information Device Profile.

Check Your Progress 2

1) J2ME

2) False.

2.7 FURTHER READINGS

1) *Programming Wireless Devices with the Java 2 Platform*, Micro Edition, 2nd Edition, Riggs, Roger/ Taivalsaari, Antero/ Van Peursem, Jim/ Huopaniemi, Jyri/ Patel, Mark/ Uotila, Aleksi; Addison Wesley.

2) *Wireless and Mobile Network Architectures*, Yi-Bing Lin and Imrich Chlamtac; Wiley

Reference Websites

- http://java.sun.com
- http://www.rspa.com

2.8 GLOSSARY

- **API** stands for Application Programming Interface. It is a set of classes that can be used in application.
- **CDMA** stands for Code-Division Multiple Access. It is one of the cellular technologies. There are 3 CDMA standards, namely CDMA One, CDMA2000 and W-CDMA.
- **CDC** stands for Connected Device Configuration. It is a specification for J2ME configuration.
- **CLDC** stands for Connected Limited Device Configuration. It is a specification for J2ME configuration.
- **CVM** stands for Compact Virtual Machine (CVM). It is used by CDC and is an optimised Java Virtual Machine (JVM).
- **J2ME** stands for Java 2 Micro Edition. It is a group of specifications and technologies related to Java on small devices.
- **MIDP** stands for Mobile Information Device Profile. It is a specification for J2ME.

- **MIDlet** is an application written for MIDP. They are derived form javax.microedition.midlet.MIDlet class.
- **MSC** stands for Mobile Switching Center. It is a part of cellular phone network which coordinates and switches calls in a given cell.
- **Optional package** is a set of J2ME APIs which provide specific functions such as database access etc.

UNIT 3 CASE TOOLS

Structure **Page Nos.**

3.0 Introduction 31
3.1 Objectives 31
3.2 What are CASE Tools? 31
3.2.1 Categories of CASE Tools
3.2.2 Need of CASE Tools
3.2.3 Factors that affect deployment of CASE Tools in an organisation
3.2.4 Characteristics of a successful CASE Tool
3.3 CASE Software Development Environment 35
3.4 CASE Tools and Requirement Engineering 36
3.5 CASE Tools and Design and Implementation 39
3.6 Software Testing 42
3.7 Software Quality and CASE Tools 43
3.8 Software Configuration Management 44
3.9 Software Project Management and CASE Tools 45
3.10 Summary 46
3.11 Solutions/Answers 47
3.12 Further Readings 48

3.0 INTRODUCTION

Software Engineering is broadly associated with the development of quality software with increasing use of software preparation standards and guidelines. Software is the single most expensive item in a computer system as the cost of software during the life time of a machine is equivalent to more than 95% of the total cost (including hardware). Software Engineering requires a lot of data collection and information generation. Since the computer itself is a very useful device as the information processor, it may be a good idea to automate software engineering tasks. Computer Aided Software Engineering (CASE) tools instill many software engineering tasks with the help of information created using computer. CASE tools support software engineering tasks and are available for different tasks of the Software Development Life Cycle (SDLC). You have been introduced to CASE tools in Unit 10 of MCS-014. This unit covers various aspects of these CASE tools and their functionality for various phases of software development.

3.1 OBJECTIVES

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

- define different kinds of CASE tools and the needs of the CASE tools;
- describe the features of analysis, design, tools use in Software Engineering; and
- identify software configuration management, and project management tool.

3.2 WHAT ARE CASE TOOLS?

Computer Aided Software Engineering (CASE) tools are gradually becoming popular for the development of software as they are improving in the capabilities and functionalities and are proving to be beneficial for the development of quality software. But, what are the CASE tools? And how do they support the process of development of software?

CASE tools are the software engineering tools that permit collaborative software development and maintenance. CASE tools support almost all the phases of the

software development life cycle such as analysis, design, etc., including umbrella activities such as project management, configuration management etc. The CASE tools in general, support standard software development methods such as Jackson Structure programming or structured system analysis and design method. The CASE tools follow a typical process for the development of the system, for example, for developing data base application, CASE tools may support the following development steps:

- Creation of data flow and entity models
- Establishing a relationship between requirements and models
- Development of top-level design
- Development of functional and process description
- Development of test cases.

The CASE tools on the basis of the above specifications can help in automatically generating data base tables, forms and reports, and user documentation.

Thus, the CASE tools –

- support contemporary development of software systems, they may improve the quality of the software
- help in automating the software development life cycles by use of certain standard methods
- create an organsiation wide environment that minimizes repetitive work
- help developers to concentrate more on top level and more creative problem-solving tasks
- support and improve the quality of documentation (Completeness and non-ambiguity), testing process (provides automated checking), project management and software maintenance.

Most of the CASE tools include one or more of the following types of tools:

- Analysis tools
- Repository to store all diagrams, forms, models and report definitions etc.
- Diagramming tools
- Screen and report generators
- Code generators
- Documentation generators
- Reverse Engineering tools (that take source code as input and produce graphical and textual representations of program design-level information)
- Re-engineering tools (that take source code as the input analyse it and interactively alters an existing system to improve quality and/or performance).

Some necessary features that must be supported by CASE tools in addition to the above are:

- It should have Security of information. The information may be visible/ changeable by authorised users only.
- Version Control for various products
- A utility to Import/Export information from various external resources in a compatible fashion
- The process of Backup and Recovery as it contains very precious data.

Figure 3.1 shows an environment having CASE.

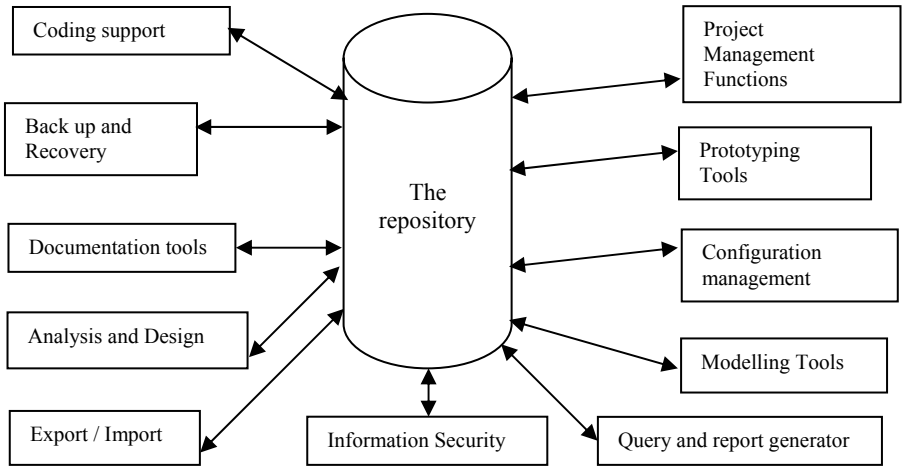


Figure 3.1: CASE Tools

3.2.1 Categories of CASE Tools

On the basis of their activities, sometimes CASE tools are classified into the following categories:

1. Upper CASE tools
2. Lower CASE tools
3. Integrated CASE tools.

Upper CASE: Upper CASE tools mainly focus on the analysis and design phases of software development. They include tools for analysis modeling, reports and forms generation.

Lower CASE: Lower CASE tools support implementation of system development. They include tools for coding, configuration management, etc.

Integrated CASE Tools: Integrated CASE tools help in providing linkages between the lower and upper CASE tools. Thus creating a cohesive environment for software development where programming by lower CASE tools may automatically be generated for the design that has been developed in an upper CASE tool.

Figure 3.2 shows the positioning of CASE tools in a Software Application development.

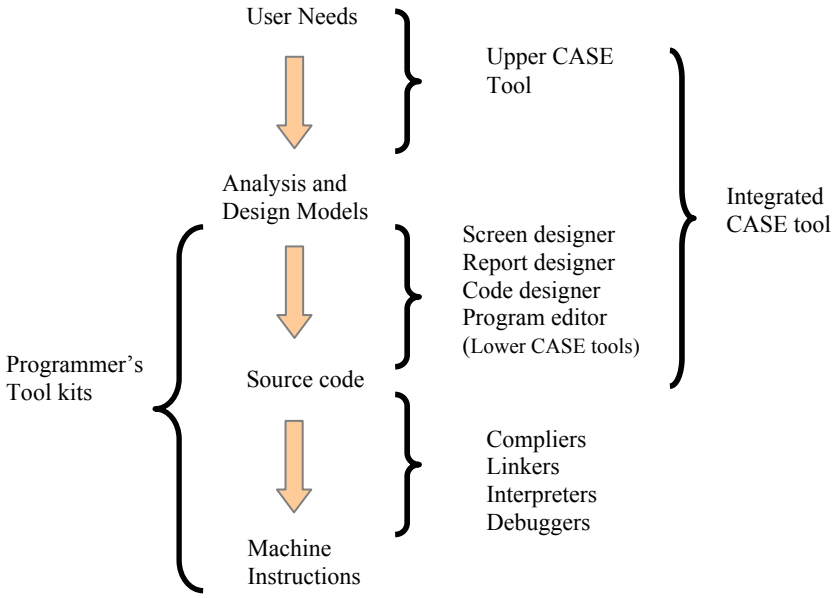


Figure 3.2: The CASE tools and Application Development

3.2.2 Need of CASE Tools

The software development process is expensive and as the projects become more complex in nature, the project implementations become more demanding and expensive. The CASE tools provide the integrated homogenous environment for the development of complex projects. They allow creating a shared repository of information that can be utilised to minimise the software development time. The CASE tools also provide the environment for monitoring and controlling projects such that team leaders are able to manage the complex projects. Specifically, the CASE tools are normally deployed to –

- Reduce the cost as they automate many repetitive manual tasks.
- Reduce development time of the project as they support standardisation and avoid repetition and reuse.
- Develop better quality complex projects as they provide greater consistency and coordination.
- Create good quality documentation
- Create systems that are maintainable because of proper control of configuration item that support traceability requirements.

But please note that CASE tools cannot do the following:

- Automatic development of functionally relevant system
- Force system analysts to follow a prescribed methodology
- Change the system analysis and design process.

There are certain disadvantages of CASE tools. These are:

- Complex functionality
- Many project management problems are not amenable to automation. Hence, CASE tools cannot be used in such cases.

3.2.3 Factors that affect deployment of CASE Tools in an organisation

A successful CASE implementation requires the following considerations in an organisation:

1. Training all the users in typical CASE environment that is being deployed, also giving benefits of CASE tools.
2. Compulsory use of CASE initially by the developers.
3. Closeness of CASE tools methodology to the Software Development Life Cycle
4. Compatibility of CASE tools with other development platforms that are being used in an organisation.
5. Timely support of vendor with respect to various issues relating to CASE tools:
    - Low cost support.
    - Easy to use and learn CASE tools having low complexity and online help.
    - Good graphic support and multiple users support.
6. Reverse Engineering support by the CASE tools: It is important that a CASE tool supports complicated nature of reverse engineering.

3.2.4 Characteristics of a successful CASE Tools

A CASE tool must have the following characteristics in order to be used efficiently:

- **A standard methodology:** A CASE tool must support a standard software development methodology and standard modeling techniques. In the present scenario most of the CASE tools are moving towards UML.

- **Flexibility**: Flexibility in use of editors and other tools. The CASE tool must offer flexibility and the choice for the user of editors' development environments.
- **Strong Integration**: The CASE tools should be integrated to support all the stages. This implies that if a change is made at any stage, for example, in the model, it should get reflected in the code documentation and all related design and other documents, thus providing a cohesive environment for software development.
- **Integration with testing software**: The CASE tools must provide interfaces for automatic testing tools that take care of regression and other kinds of testing software under the changing requirements.
- **Support for reverse engineering**: A CASE tools must be able to generate complex models from already generated code.
- **On-line help**: The CASE tools provide an online tutorial.

Now let us discuss some of the important characteristics for various types of CASE tools in the subsequent sections.

☞ Check Your Progress 1

1) What is the need of CASE tools?

……..

……..

……

2) What are the important characteristics of CASE tools?

……..

……..

……

3.3 CASE SOFTWARE DEVELOPMENT ENVIRONMENT

CASE tools support extensive activities during the software development process. Some of the functional features that are provided by CASE tools for the software development process are:

1. Creating software requirements specifications
2. Creation of design specifications
3. Creation of cross references.
4. Verifying/Analysing the relationship between requirement and design
5. Performing project and configuration management
6. Building system prototypes
7. Containing code and accompanying documents.
8. Validation and verification, interfacing with external environment.

Some of the major features that should be supported by CASE development environment are:

- a strong visual support
- prediction and reporting of errors
- generation of content repository
- support for structured methodology
- integration of various life cycle stages
- consistent information transfer across SDLC stages
- automating coding/prototype generation.

Present CASE tools support Unified Model Language (UML).

We will elaborate on the features of CASE tools for various stages of software development process in coming sub-sections.

CASE and Web Engineering

CASE Tools are also very useful in the design, development and implementation of web site development.

Web Engineering requires tools in many categories. They are:

- Site content management tools
- Site version control tools
- Server management tool
- Site optimisation tools
- Web authoring and deployment tools
- Site testing tools that include load and performance testing
- Link checkers
- Program checkers
- Web security test tools.

A detailed discussion on these tools is beyond the scope of this unit. However, various stages of development of a web project also follows the normal SDLC. These are discussed in the subsequent sections.

3.4 CASE TOOLS AND REQUIREMENT ENGINEERING

Let us first answer the question:

Which is the most Common Source of Risk during the process of software development?

One of the major risk factors that affect project schedule, budget and quality can be defined as the ability to successfully elicit requirements to get a solution.

Statistically it has been seen that about 80% of rework in projects is due to requirement defects.

How can a CASE tools help in effective Requirements Engineering (RE)

A good and effective requirements engineering tool needs to incorporate the best practices of requirements definition and management.

The requirements Engineering approach should be highly iterative with the goal of establishing managed and effective communication and collaboration.

Thus, a CASE tool must have the following features from the requirements engineering viewpoint:

- a dynamic, rich editing environment for team members to capture and manage requirements
- to create a centralised repository
- to create task-driven workflow to do change management, and defect tracking.

But, what is a good process of Requirements Engineering for the CASE?

A simple four-step process for requirements engineering is given in *Figure 3.3*.

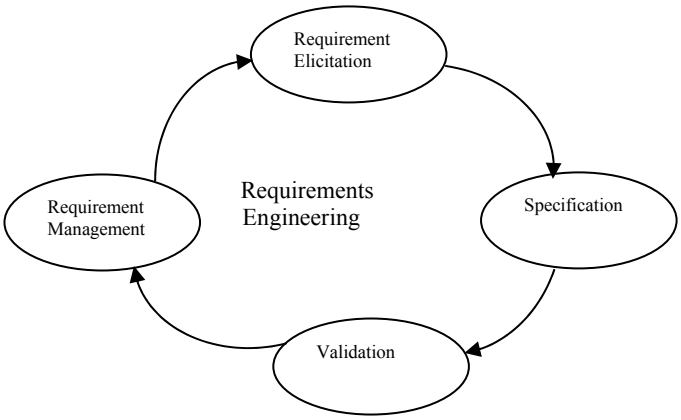


Figure 3.3: The four-step requirement engineering process

Requirement Elicitation

A simple technique for requirements elicitation is to ask "WHY".

CASE tools support a dynamic, yet intuitive, requirements capture and management environment that supports content and its editing. Some of the features available for requirement elicitation are:

- Reusable requirements and design templates for various types of system
- Keeping track of important system attributes like performance and security
- It may also support a common vocabulary of user-defined terms that can be automatically highlighted to be part of glossary.
- Provide feature for the assessment of the quality of requirements
- Separate glossary for ambiguous terms that can be flagged for additional clarification.

What do we expect from the tool?

- It should have rich support for documentation that is easily understandable to stakeholders and development teams.
- It should have the capability of tracking of requirements during various SDLC systems.
- It should help in the development of standard technical and business terminology for end clients.

Software Analysis and Specification

One of the major reasons of documenting requirements is to remove the ambiguity of information. A good requirement specification is testable for each requirement.

One of the major features supported by CASE tools for specification is that the design and implementation should be traceable to requirements. A good way to do so is to support a label or a tag to the requirements. In addition it should have the following features:

- Must have features for storing and documenting of requirements.
- Enable creation of models that are critical to the development of functional requirements.
- Allow development of test cases that enable the verification of requirements and their associated dependencies.
- Test cases help in troubleshooting the correlation between business requirements and existing system constraints.

What do we expect from the tool?

- It should allow development of a labeled requirements document that helps in traceability of requirements.
- It should allow both functional and non-functional requirements with related quality attributes to be made.
- We should be able to develop the associated models.

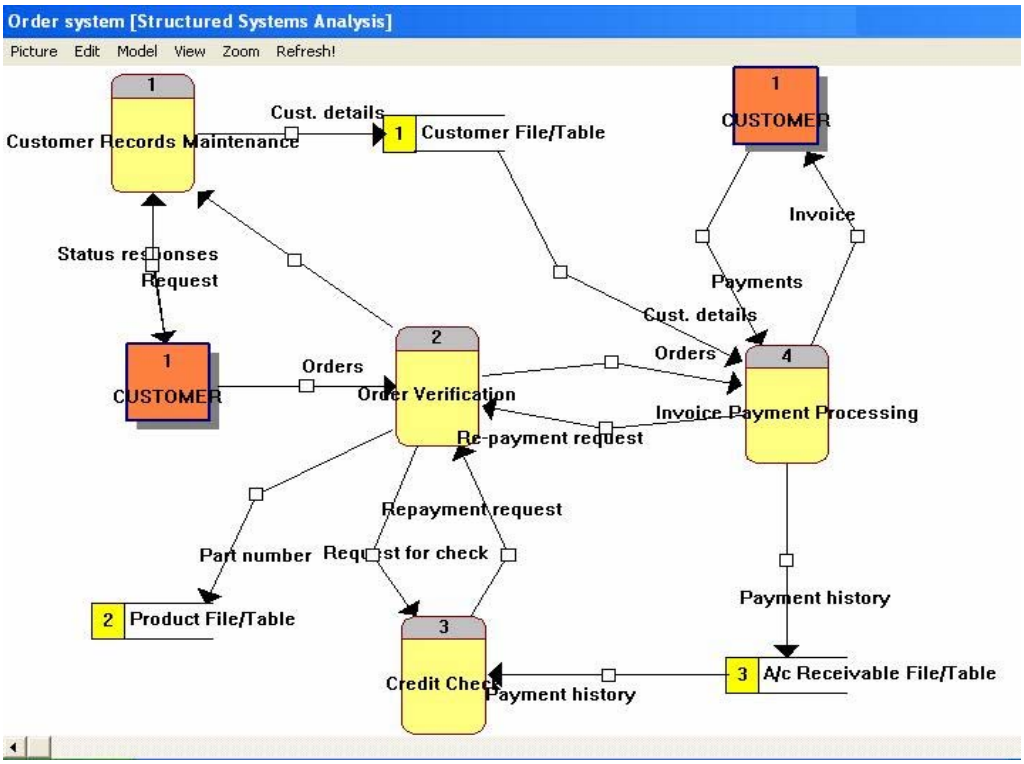


Figure 3.4: A sample DFD using CASE Tools

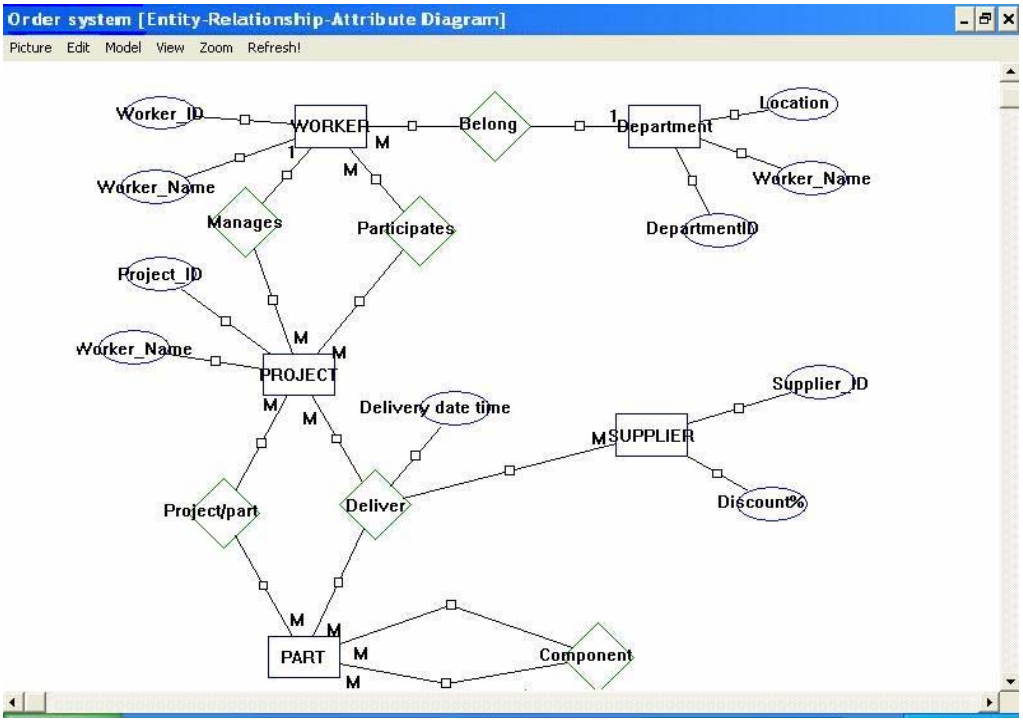


Figure 3.5: A Sample ER Diagram using CASE Tools

Figures 3.4 and 3.5 show some of the models developed with the help of a sample CASE Tool.

"The automated approach of software development needs to have a spirit of collaboration, bringing together world-renowned principal investigators, business analysts, and software development teams."

Validation of Requirements

A very important feature in this regard is to allow collaboration yet customizable workflows for the software development team members. Also facilitating approvals and electronic signatures to facilitate audit trails. Assigning owner of requirement may be helpful if any quality attributes may need changes. Thus, a prioritised validated documented and approved requirements can be obtained.

Managing Requirements

The requirements document should have visibility and help in controlling the software delivery process. Some such features available in CASE tools are:

- estimation of efforts and cost
- specification of project schedule such as deadline, staff requirements and other constraints
- specification of quality parameters.

Software Change Management

An incomplete or ambiguous requirement if detected early in the analysis phase can be changed easily with minimum cost. However, once they are converted to baselines after requirements validations the change should be controlled.

One of the major requirements of change management is:

Any change to these requirements should follow a process which is defined as the ability to track software requirements and their associated models/documents, etc. (e.g., designs, DFDs and ERDs, etc.). This helps in determining the components that will be impacted due to change. This also helps tracking and managing change. The whole process starts with labeling the requirements properly.

Most CASE Tools store requirement baselines, including type, status, priority and change history of a software item. Such traceability may be bi-directional in nature.

Static word processing documents or spreadsheet recedes communication issues because those tools do not allow sharing and collaboration on up-to-date requirements. To overcome this a CASE enabled requirements management infrastructure offers the familiarity of environments such as Microsoft Word, with added communication methods, such as email. Thus, CASE is a very useful tool for requirement engineering.

What are the features of high quality requirements?

- Correctness
- Unambiguous
- Must
- Consistency
- Known constraints
- Modifiable
- Priority Assigned
- Traceable
- Verifiable

3.5 CASE TOOLS AND DESIGN AND IMPLEMENTATION

In general, some CASE tools support the analysis and design phases of software development. Some of the tools supported by the design tools are:

- Structured Chart.
- Pogram Document Language (PDL).
- Optimisation of ER and other models.
- Flow charts.
- Database design tools.
- File design tools.

Some of functions that these diagrams tool support are simple but are very communicative as far as representations of the information of the analysis and design phases are concerned. CASE Tools also support standard representation of program architecture; they also contain testing items related to the design and debugging. Automatic support for maintenance is provided in case any of the requirements or design items is modified using these diagrams. These CASE tools also support

error-checking stages. They allow checks for completeness and consistency of the required functional design types and consistency at various levels of cross referencing among the documents consistency checking for these documents during the requirements analysis and design phases.

Proper modeling helps in proper design architecture. All the CASE tools have strong support for models. They also help in resolving conflicts and ambiguity among the models and help in optimising them to create excellent design architecture and process implementation.

But why do we need to model?

Can you understand code? If you understand it, then why would you call it code?

The major advantages for modeling are:
A model enhances communication, as it is more pictorial and less code.

Even early humans have used pictures to express ideas in a better way.

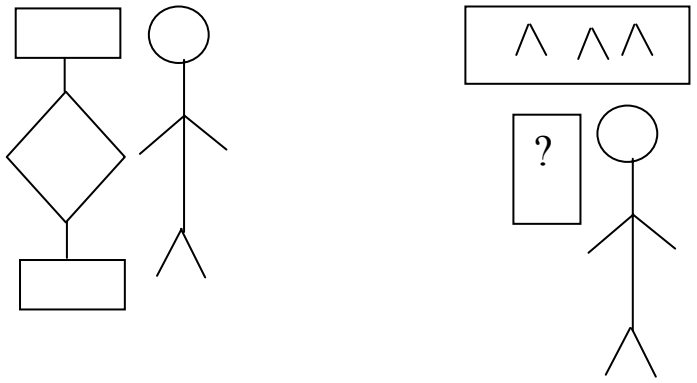


Figure 3.6: Model versus code

A model

- helps the users
- may reduce the cost of project
- can convey a broad spectrum of information.

Some standard models may need to represent:

- Overall system architecture of the software
- Software dependencies
- Flow of information through a software system
- Database organisation and structure.
- Deployment configuration, etc.

Models also help in better planning and reduction of risk as one can make top down models, thus controlling complexity.

So what is a good modeling tool?

The following are some of the characteristics of a good modeling tool:

CASE tools provide continuously synchronized models and code, thus also help in consistent understanding of information and documentation. It also helps other software developers to understand the portions and relationships to portions other team members are working on.

Help in management of source code through a more visual model.

Refactoring of the code allows a re-look and improvement of code, as modeling tools contains, thus are most continuously sychronised code and models suitable for refactoring.

Modeling can help in creating good patterns including modeling patterns, language specific patterns and basic patterns for common operations such as implementing interfaces, etc.

Models also facilitate reverse engineering.

Given the task of maintaining software that was built by a team that has moved on to a new project, there may be very little documentation available for the project.

The value of a modeling tool can be tremendous in such cases. The developer can take the existing code, create models for it to understand the software. This can save a lot of time for the developer.

A modeling tool should have been integrated with requirements tools, so that architects see consistently unambiguous information.

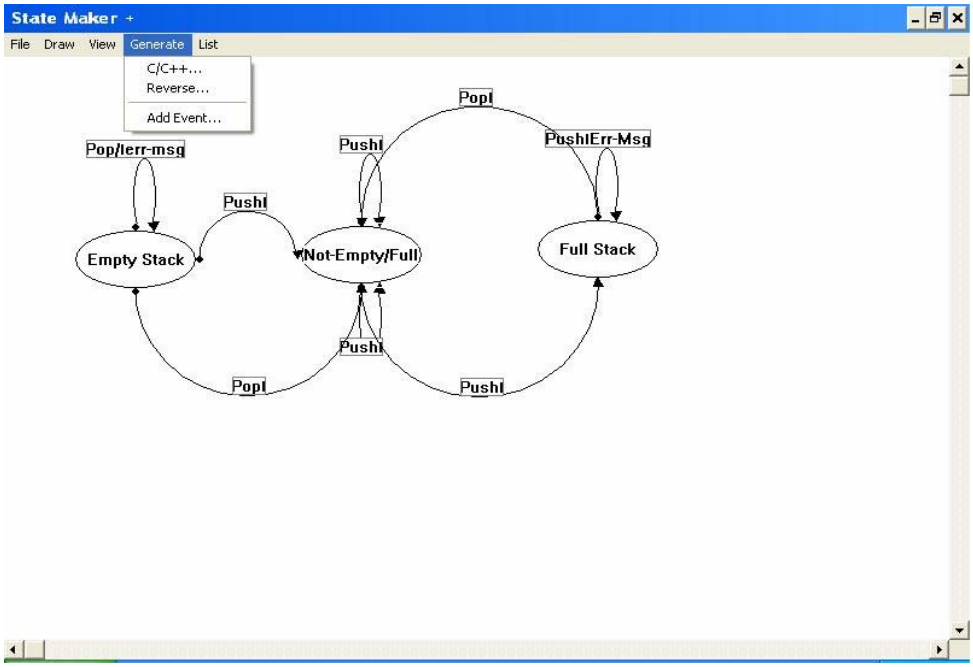


Figure 3.7: A State Model

CASE Repository

CASE Repository stores software system information. It includes analysis and design specifications and helps in analysing these requirements and converting them into program code and documentation. The following is the content of CASE repository:

- Data
- Process
- Models
- Rules/ Constraints.

Data: Information and entities/object relationship attributes, etc.

Process: Support structured methodology, link to external entities, document structured software development process standards.

Models: Flow models, state models, data models, UML document etc.

Rules/Constraints: Business rules, consistency constraints, legal constraints.

The CASE repository has two primary segments.

1. Information repository
2. Data dictionary.

Information Repository includes information about an organisation's business information and its applications.

The CASE tools manage and control access to repository. Such information data can also be stored in corporate database.

Data dictionary contains all the data definitions for all organisational applications, along with cross-referencing if any.

Its entries have a standard definition, viz., element name and alias; textual description of the element; related elements; element type and format; range of acceptable values and other information unique to the proper processing of the element.

CASE Repository has additional advantages such that it assists the project management tasks; aids in software reusability by enabling software modules in a manner so that they can be used again.

Implementation tools and CASE

CASE tools provide the following features for implementation:

- Diagramming tools enable visual representation of a system and its components
- Allow representing process flows.
- Help in implementing the data structures and program structures
- Support automatic creation of system forms and reports.
- Ready prototype generation.
- Create of both technical and user documentation.
- Create master templates used to verify documentation and its conformance to all stages of the Software Development Life Cycle (SDLC).
- Enable the automatic generation of program and database from the design documents, diagrams, forms and reports stored in the repository.

3.6 SOFTWARE TESTING

CASE tools may also support software testing. One of the basic issues of testing is management, execution and reporting. Testing tools can help in automated unit testing, functional regression testing, and performance testing. A testing tool ensures the high visibility of test information, types of common defects, and application readiness.

Features Needed for Testing

The tool must support all testing phases, viz., plan, manage and execute all types of tests viz., functional, performance, integration, regression, testing from the same interface.

It should integrate with other third party testing tools.

It should support local and remote test execution.

It should help and establish and manage traceability by linking requirements to test cases. This also helps when requirements change; in such as case the test cases traced to the requirement are automatically flagged as possible candidates for change.

It should create a log of test run.

It should output meaningful reports on test completeness, test analysis.

It may provide automated testing.

☞ Check Your Progress 2

1) List any four tools that are used in web software engineering but are not used in general software projects.

……

……

2) Which of the requirements engineering processes is not supported by CASE tools?

……

……

3) List any four major requirements of a design tool.

……

……

4) List four important features of a testing tool.

……

……...

3.7 SOFTWARE QUALITY AND CASE TOOLS

Software quality is sacrificed by many developers for more functionality, faster development and lower cost. However, one must realise that a good quality product actually enhances the speed of software development. It reduces the cost and allows enhancement and functionality with ease as it is a better structured product. You need to pay for a poor quality in terms of more maintenance time and cost. Can the good quality be attributed to a software by enhancing the testing? The answer is NO. The high quality software development process is most important for the development of quality software product. Software quality involves functionality for software usability, reliability, performance, scalability, support and security.

Integrated CASE tools:

- help the development of quality product as they support standard methodology and process of software development
- supports an exhaustive change management process
- contains easy to use visual modeling tools incorporating continuous quality assurance.

Quality in such tools is represented in all life cycle phases viz., Analysis/ Design development, test and deployment. Quality is essential in all the life cycle phases.

Analysis: A poor understanding of analysis requirements may lead to a poor product. CASE tools help in reflecting the system requirements clearly, accurately and in a simple way. CASE tools support the requirements analysis and coverage also as we have modeling. CASE also helps in ambiguity resolution of the requirements, thus making high quality requirements.

Design: In design the prime focus of the quality starts with the testing of the architecture of the software. CASE tools help in detecting, isolating and resolving structure deficiency during the design process. On an average, a developer makes 100 to 150 errors for every thousand lines of code. Assuming only 5% of these errors are

serious, if software has ten thousand lines of code you may still have around 50 serious coding errors in your system. One of the newer software development processes called the Agile process helps in reducing such problems by asking the developer to design their test items first before the coding.

A very good approach that is supported by CASE tools specially running time development of C, C++, JAVA or .NET code is to provide a set of automatic run time Language tools for development of reliable and high performance applications.

Testing: Functionality and performance testing is an integrated part of ensuring high quality product. CASE support automated testing tools that help in testing the software, thus, helping in improving the quality of testing. CASE tools enhance the speed breadth and reliability of these design procedures. The design tools are very important specifically in case of a web based system where scalability and reliability are two major issues of design.

Deployment: After proper testing a software goes through the phase of deployment where a system is made operational. A system failure should not result in a complete failure of the software on restart. CASE tools also help in this particular place. In addition, they support configuration management to help any kind of change thus to be made in the software.

Quality is teamwork: It involves integration of workflow of various individuals. It establishes a traceability and communication of information, all that can be achieved by sharing workload documents keeping their configuration items.

3.8 SOFTWARE CONFIGURATION MANAGEMENT

Software Configuration Management (SCM) is extremely important from the view of deployment of software applications. SCM controls deployment of new software versions. Software configuration management can be integrated with an automated solution that manages distributed deployment. This helps companies to bring out new releases much more efficiently and effectively. It also reduces cost, risk and accelerates time.

A current IT department of an organisation has complex applications to manage. These applications may be deployed on many locations and are critical systems. Thus, these systems must be maintained with very high efficiency and low cost and time. The problem for IT organisations has increased tremendously as some of the organisations may need to rebuild and redeploy new software versions a week over multi-platform global networks.

In such a situation, if rebuilding of application versions is built or deployed manually using a spreadsheet, then it requires copying of rebuilt software to many software locations, which will be very time consuming and error prone. What about an approach where newer software versions are automatically built and deployed into several distributed locations through a centralised control. For example, assume that IGNOU has a data entry software version 1.0 for entering assignment marks which is deployed all the RCs. In case its version 1.1 is to be deployed, if the re-built software need to be sent and deployed manually, it would be quite troublesome. Thus, an automatic deployment tool will be of great use under the control of SCM.

What do we need?

We need an effective SCM with facilities of automatic version control, access control, automatic re-building of software, build audit, maintenance and deployment. Thus, SCM should have the following facilities:

- Creation of configuration

- This documents a software build and enables versions to be reproduced on demand
- Configuration lookup scheme that enables only the changed files to be rebuilt. Thus, entire application need not be rebuilt.
- Dependency detection features even hidden dependencies, thus ensuring correct behaviour of the software in partial rebuilding.
- Ability for team members to share existing objects, thus saving time of the team members.

Figure 3.8 shows a simple Configuration Management based rebuilding and deployment process.

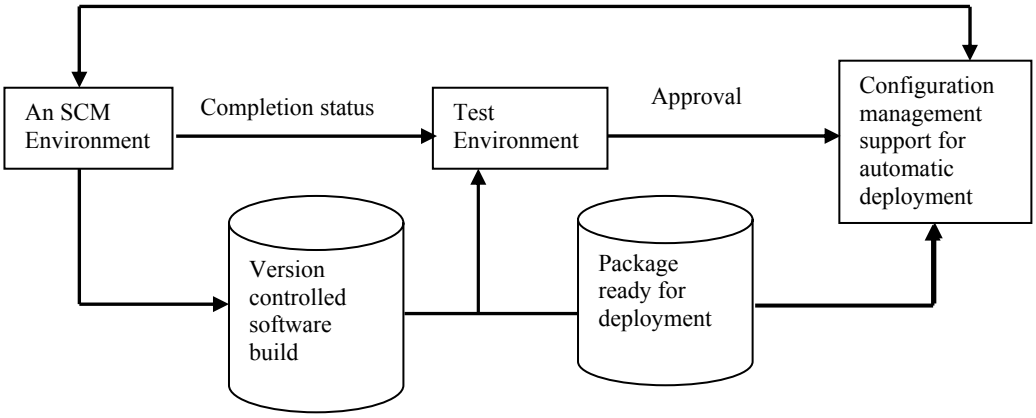


Figure 3.8: Simple configuration management environment

3.9 SOFTWARE PROJECT MANAGEMENT AND CASE TOOLS

Developing commercial software is not a single-player activity. It is invariably a collaborative team activity. The development of business-critical systems are also too risky, or are too large for a single developer to deliver the product in a stipulated time and quality frame.

A software team involves designers, developers, and testers who work together for delivering the best solution in the shortest time. Sometimes, these teams can be geographically dispersed. Managing such a team may be a major change requests, and task management.

The CASE tools help in effective management of teams and projects. Let us look into some of the features of CASE in this respect:

- Sharing and securing the project using the user name and passwords.
- Allowing reading of project related documents
- Allowing exclusive editing of documents
- Linking the documents for sharing
- Automatically communicative change requests to approver and all the persons who are sharing the document.
- You can read change requests for yourself and act on them accordingly.
- Setting of revision labels so that versioning can be done.
- Any addition or deletion of files from the repository is indicated.
- Any updating of files in repository is automatically made available to users.
- Conflicts among versions are reported and avoided
- Differences among versions can be visualized.

- The linked folder, topics, and change requests to an item can be created and these items if needed can be accessed.
- It should have reporting capabilities of information

The project management tools provide the following benefits:

- They allow control of projects through tasks so control complexity.
- They allow tracking of project events and milestones.
- The progress can be monitored using Gantt chart.
- Web based project related information can be provided.
- Automatic notifications and mails can be generated.

Some of the features that you should look into project management software are:

- It should support drawing of schedules using PERT and Gantt chart.
- It should be easy to use such that tasks can be entered easily, the links among the tasks should be easily desirable.
- Milestones and critical path should be highlighted.
- It should support editing capabilities for adding/deleting/moving tasks.
- Should map timeline information against a calendar.
- Should allow marking of durations for each task graphically.
- It should provide views tasks, resources, or resource usage by task.
- Should be useable on network and should be able to share information through network.

☞ **Check Your Progress 3**

1) How do CASE tools support quality?

……

……

2) What is the role of CASE in software configuration management?

……

……

3) Can CASE tools result in perfect project management?

……

……

3.10 SUMMARY

This unit provides an introduction to CASE tools in action. The CASE tools are primarily used to automate the stages of the Software Development Life Cycle. It automates most of the tasks of software engineering such as requirements analysis, modeling, designing, support for testing, and implementation, project management, software configuration management, software quality management, etc. CASE tools also help in traceability of requirements. They are useful tools for reverse engineering. Presently, many CASE tools which integrate the complete Software Development Life Cycle are available and can be deployed successfully in an organisation. These CASE tools support standard methodology, provide flexibility of environment, strong integration of various software development stages, reverse engineering, project management, and configuration management under one single environment. However, it is to be noted that all deployments of CASE have not succeeded.

Successful CASE tools are those which are simple to use support standard methodology, and reverse engineering and have a strong vendor support including training.

3.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) CASE tools are needed for
   - Development of cost effective software
   - Minimisation of development time
   - Standardising the software development process
   - Avoiding repetition and maximising reuse
   - Development of better quality product
   - Collaborative developments etc.

2) CASE tools
   - Follow standard methodology
   - Allow integration of information
   - Allow traceability
   - Help improving of quality
   - Reduce cost
   - Support reverse engineering
   - Provide on-line help
   - Check for inconsistency of information
   - Provide tools for configuration management and project management.

Check Your Progress 2

1) Web security test tools, link checkers, site optimization tools, server management tools

2) Although requirements engineering phases viz., requirement elicitation, specification, validation, requirements management, have some part being played by CASE tools, yet it is the stage of requirement elicitation where CASE tools are least helpful as this stage requires interaction with the users.

3) The design tools support the following:
   a. A strong consistent data dictionary
   b. Support for presenting design architecture and data design.
   c. Support for traceability and integration with test scenarios
   d. Optimisation of models and process flow charts.

4) Test planning, Integration with automatic testing tool, Traceability, Flagging the possible candidates for change.

Check Your Progress 3

1) The quality of the software is supported at all the Life Cycle stages of software development with the help of CASE tools. For example, in the analysis phase CASE tools may automatically flag requirement ambiguity in

the design phase they may point out any inconsistency, in the test phase they may point out the reliability of software, etc.

2) CASE tools control the baseline configuration items thus support a strict change control such that any changes made during any life cycle phase results in automatic flagging of the various documents of various stages of SDLC affected by that change. They also help in version control.

3) No. They are only predictors of various failures to meet the schedule of the project.

3.12 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**

UNIT 4 ADVANCED TOPICS IN SOFTWARE ENGINEERING

Structure **Page Nos.**

4.0	Introduction	49
4.1	Objectives	50
4.2	Evolution of Formal Methods	50
4.3	Use of Mathematics in Software Development	51
4.4	Formal Methods	51
	4.4.1 What Can Be Formally Specified?	
	4.4.2 Goals of Formal Specification	
4.5	Application Areas	53
4.6	Limitations of Formal Specification Using Formal Methods	54
4.7	Cleanroom Software Engineering	55
4.8	Conventional Software Engineering Models Vs. Cleanroom Software Engineering Model	56
4.9	Cleanroom Software Engineering Principles, Strategy and Process Overview	56
410	Limitations of Cleanroom Engineering	57
4.11	Similarities and Differences between Cleanroom and OO Paradigm	58
4.12	Software Reuse and its Types	58
4.13	Why Component based Software Engineering?	59
4.14	Component based Software Engineering (CBSE) Process	60
	4.14.1 Domain Engineering	
	4.14.2 Component based Development	
4.15	Component Technologies Available	62
4.16	Challenges for CBSE	62
4.17	Reengineering: An Introduction	64
4.18	Objectives of Reengineering	64
4.19	Software Reengineering Life Cycle	65
4.20	Summary	66
4.21	Solutions / Answers	67
4.22	Further Readings	68
4.23	Glossary	68

4.0 INTRODUCTION

In the first few units we have studied the various methods for conventional software engineering. Software engineering methods can be categorised on a “formality spectrum.” The analysis and design methods discussed previously would be placed at the informal to moderately rigorous end of the spectrum. In these methods, a combination of diagrams, text, tables, and simple notation is used to create analysis and design models.

At the other end of the formality spectrum are formal methods, which are completely mathematical in form. A specification and design are described using a formal syntax and semantics that specify system function and behaviour. Other specialised models being the Cleanroom software engineering, component-based software engineering, re-engineering and reverse engineering.

Harlan Mills has developed the Cleanroom software engineering methodology, which is an approach for integrating formal methods into the lifecycle. The Cleanroom approach combines formal methods and structured programming with Statistical Process Control (SPC), the spiral lifecycle and incremental releases, inspections, and software reliability modeling. It fosters attitudes, such as emphasising defect prevention over defect removal, that are associated with high quality products in non-software fields.

In most engineering disciplines, systems are designed by composition (building system out of components that have been used in other systems). Software engineering has focused on custom development of components. To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt *systematic reuse* as a design process. There are various ways to achieve this and one of the models is the Component-based software engineering.

Let us study the various specialised software engineering models like the formal method, Cleanroom engineering, component-based engineering, re-engineering and reverse engineering in this unit.

4.1 OBJECTIVES

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

- understand the pitfalls of the conventional software engineering;
- know various advanced methods of software engineering – their advantages and drawbacks;
- describe the methodology of formal methods and Cleanroom software engineering, and
- discuss the Re-engineering and Reverse engineering process.

4.2 EVOLUTION OF FORMAL METHODS

The seventies witnessed the structured programming revolution. After much debate, software engineers became convinced that better programs result from following certain precepts in program design. Recent imperative programming languages provide constructs supporting structured programming. Achieving this consensus did not end debate over programming methodology. On the contrary, a period of continuous change began, with views on the best methods of software development mutating frequently. Top-down development, modular decomposition, data abstraction, and, most recently, object oriented design are some of the jargon terms that have arisen to describe new concepts for developing large software systems. Both researchers and practitioners have found it difficult to keep up with this onslaught of new methodologies.

There is a set of core ideas that lies at the base of these changes. Formal methods have provided a unifying philosophy and central foundation upon which these methodologies have been built. Those who understand this underlying philosophy can more easily adopt these and other programming techniques.

One way to improve the quality of software is to change the way in which software is documented: at the design stage, during development, and after release. Existing methods of documentation offer large amounts of text, pictures, and diagrams, but these are often imprecise and ambiguous. Important information is hidden amongst irrelevant details, and design flaws are discovered too late, making them expensive or impossible to correct.

There is an alternative. *Formal* methods, based upon elementary mathematics, can be used to produce precise, unambiguous documentation, in which information is structured and presented at an appropriate level of abstraction. This documentation can be used to support the design process, and as a guide to subsequent development, testing, and maintenance.

It seems likely that the use of formal methods will become standard practice in software engineering. The mathematical basis is different from that of civil or mechanical engineering, but it has the same purpose: to add precision, to aid understanding, and to reason about the properties of a design. Whatever the discipline, the use of mathematics can be expensive, but it is our experience that it can actually reduce costs.

Existing applications of formal methods include: the use of the probability theory in performance modeling; the use of context-free grammars in compiler design; the use of the relational calculus in database theory. The formal method described in this book has been used in the specification and design of large software systems. It is intended for the description of state and state-based properties, and includes a theory of refinement that allows mathematics to be used at every stage of program development. Let us see the details of the Formal methods in the next section.

4.3 USE OF MATHEMATICS IN SOFTWARE DEVELOPMENT

Mathematics supports *abstraction* and therefore is a powerful medium for modeling. Because they are exact, mathematical specifications are *unambiguous* and can be validated to uncover *contradictions* and *incompleteness*. It allows a developer to *validate* a specification for functionality. It is possible to demonstrate that a design matches a specification, and that some program code is a correct reflection of a design.

How is the mathematics of formal languages applied in software development? What engineering issues have been addressed by their application? Formal methods are of global concern in software engineering. They are directly applicable during the requirements, design, and coding phases and have important consequences for testing and maintenance. They have influenced the development and standardisation of many programming languages, the programmer's most basic tool. They are important in ongoing research that may change standard practice, particularly in the areas of specifications and design methodology. They are entwined with lifecycle models that may provide an alternative to the waterfall model, namely rapid prototyping, the Cleanroom variant on the spiral model, and "transformational" paradigms. The concept of formalism in formal methods is borrowed from certain trends in 19th and 20th century mathematics. Formal methods are merely an adoption of the axiomatic method, as developed by these trends in mathematics, for software engineering. Mastery of formal methods in software requires an understanding of this mathematics background. Mathematical topics of interest include formal logic, both the propositional calculus and predicate logic, set theory, formal languages, and automata such as finite state machines.

4.4 FORMAL METHODS

A **formal method** *in software development is a method that provides a formal language for describing a software artifact (e.g., specifications, designs, source code) such that formal proofs are possible, in principle, about properties of the artifact so expressed.*

The definition is based on two essential components. First, formal methods involve the essential use of a formal language. A formal language is a set of strings over some well-defined alphabet. Rules are given for distinguishing those strings, defined over the alphabet, that belong to the language from strings that do not. Secondly, formal methods in software support formal reasoning about formulae in the language. These methods of reasoning are exemplified by formal proofs. A proof begins with a set of axioms, which are to be taken as statements postulated to be true. Inference rules state that if certain formula, known as premises, are derivable from the axioms, then another formula, known as the consequent, is also derivable. A set of inference rules must be given in each formal method. A proof consists of a sequence of well-defined formulae in the language in which each formula is either an axiom or derivable by an inference rule from previous formulae in the sequence. The last formula in the sequence is said to be proven.

The major concerns of the formal methods are as follows:

i) The correctness of the problem
   - producing software that is "correct" is famously difficult;
   - by using rigorous mathematical techniques, it may be possible to make probably correct software.

ii) Programs are mathematical objects
   - they are expressed in a **formal** language;
   - they have a **formal** semantics;
   - programs can be treated as mathematical theories.

Formal methods support precise and rigorous specifications of those aspects of a computer system capable of being expressed in the language. Since defining what a system should do, and understanding the implications of these decisions, are the most troublesome problems in software engineering, this use of formal methods has major benefits. In fact, practitioners of formal methods frequently use formal methods solely for recording precise specifications, not for formal verifications.

Formal methods were originally developed to support verifications, but higher interest currently exists in specification methods. Several methods and languages can be used for specifying the functionality of computer systems. No single language, of those now available, is equally appropriate for all methods, application domains, and aspects of a system. Thus, users of formal specification techniques need to understand the strength and weaknesses of different methods and languages before deciding on which to adopt. The distinction between a specification method and a language is fundamental. A method states what a specification must say. A language determines in detail how the concepts in a specification can be expressed. Some languages support more than one method, while most methods can be used in several specification languages.

Some of the most well-known formal methods consist of or include specification languages for recording a system's functionality. These methods include:

- Z (pronounced "Zed")
- Communicating Sequential Processes (CSP)
- Vienna Development Method (VDM)
- Larch
- Formal Development Methodology (FDM)

4.4.1 What can be Formally Specified?

Formal methods can include graphical languages. Data Flow Diagrams (DFDs) are the most well-known graphical technique for specifying the function of a system. DFDs can be considered a semi-formal method, and researchers have explored techniques for treating DFDs in a completely formal manner. Petri nets provide another well-known graphical technique, often used in distributed systems. Petri nets are a fully formal technique.

Finally, finite state machines are often presented in tabular form. This does not decrease the formalism in the use of finite state machines. So the definition of formal methods provided earlier is quite encompassing.

Software engineers produce models and define the properties of systems at several levels of abstraction. Formal methods can be employed at each level. A specification should describe what a system should do, but not how it is done. More details are provided in designs, with the source code providing the most detailed model.

For example, Abstract Data Types (ADTs) frequently are employed at intermediate levels of abstraction. ADTs, being mathematical entities, are perfect candidates for formal treatment and are often so treated in the literature.

Formal methods are not confined to the software components of large systems. System engineers frequently use formal methods. Hardware engineers also use formal methods, such as VHSIC Hardware Description Language (VHDL) descriptions, to

model integrated circuits before fabricating them. The following section lists the goals of the formal specification.

4.4.2 Goals of Formal Specification

Once a formal description of a system has been produced, what can be done with it? Usable formal methods provide a variety of techniques for reasoning about specifications and drawing implications. The completeness and consistency of a specification can be explored. Does a description imply a system should be in several states simultaneously? Do all legal inputs yield one and only one output? What surprising results, perhaps unintended, can be produced by a system? Formal methods provide reasoning techniques to explore these questions. Do lower level descriptions of a system properly implement higher level descriptions? Formal methods support formal verification, the construction of formal proofs that an implementation satisfies a specification. The possibility of constructing such formal proofs was historically the principal driver in the development of formal methods. Prominent technology for formal verification includes E Dijkstra's "weakest precondition" calculus and Harlan Mills' "functional correctness" approach which we are not going to discuss here. The following are the goals of the formal specification:

- **Removal of ambiguity:** The formal syntax of a specification language enables requirements or design to be interpreted in only one way, eliminating the ambiguity that often occurs when using natural language. Removes ambiguity and encourages greater rigour in the early stages of the software engineering process.
- **Consistency:** Facts stated in one place should not be contradicted in another. This is ensured by mathematically proving that initial facts can be formally mapped (by inference) into later statements later in the specification.
- **Completeness:** This is difficult, even when using formal methods. Some aspects of a system may be left undefined by mistake or by intention. It is impossible to consider every operational scenario in a large, complex system.

4.5 APPLICATION AREAS

Formal methods can be used to specify aspects of a system other than functionality. Software safety and security are other areas where formal methods are sometimes applied in practice. The benefits of proving that unsafe states will not arise, or security will not be violated, can justify the cost of complete formal verifications of the relevant portions of a software system. Formal methods can deal with many other areas of concern to software engineers, but have not been much used, other than in research organisations, for dealing with issues unrelated to functionality, safety, and security. Areas in which researchers are exploring formal methods include fault tolerance, response time, space efficiency, reliability, human factors, and software structure dependencies. The following are the Formal specifications application areas:

- safety critical systems
- security systems
- The definition of standards
- Hardware development
- Operating systems
- Transaction processing systems
- Anything that is hard, complex, or critical.

Let us see the drawbacks of the formal specifications in the next section.

4.6 LIMITATIONS OF FORMAL SPECIFICATION USING FORMAL METHODS

Some problems exist, however formal specification focuses primarily on function and data. Timing, control, and behavioural aspects of a problem are more difficult to represent. In addition, there are elements of a problem (e.g., HCI, the human-machine interface) that are better specified using graphical techniques. Finally, a specification using formal methods is more difficult to learn than other analysis methods. For this reason, it is likely that formal mathematical specification techniques will be incorporated into a future generation of CASE tools. When this occurs, mathematically-based specification may be adopted by a wider segment of the software engineering community. The following are the major drawbacks:

Cost: It can be almost as costly to do a full formal specification as to do all the coding. The solution for this is, don't formally specify everything, just the subsystems you need to (hard, complex, or critical).

Complexity: Not everybody can read formal specifications (especially customers and users). The solution is to provide a very good and detailed documentation. No one wants to read pure formal specifications–formal specifications should always be interspersed with natural language (e.g. English) explanation.

Deficiencies of Less Formal Approaches: The methods of structured and object-oriented design discussed previously make heavy use of natural language and a variety of graphical notations. Although careful application of these analysis and design methods can and does lead to high–quality software, sloppiness in the application of these methods can create a variety of problems which are:

- **Contradictions**: statements that are at variance with one another.
- **Ambiguities** : statements that can be interpreted in a number of ways
- **Vagueness** : statements that lack precision, and contribute little information.
- **Incomplete statements** : a description is not functionally complete.
- **Mixed levels of abstraction** : statements with high and low levels of detail are interspersed, which can make it difficult to comprehend the functional architecture.

This method gained popularity among the software developers who must build safety-critical software like aircraft avionics and medical devices. But the applicability in the business environment has not clicked.

Let us study the Cleanroom Software Engineering in the next section.

☞ Check Your Progress 1

1) What are Formal methods?

……

……

2) What are the main parts of the Formal methods?

……

……

4.7 CLEANROOM SOFTWARE ENGINEERING

Cleanroom software engineering is an engineering and managerial process for the development of high-quality software with certified reliability. Cleanroom was originally developed by Dr. Harlan Mills. The name "Cleanroom" was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication. It reflects the same emphasis on defect

prevention rather than defect removal, as well as Certification of reliability for the intended environment of use. It combines many of the formal methods and software quality methods we have studied so far. The focus of Cleanroom involves moving from traditional software development practices to rigorous, engineering-based practices.

Cleanroom software engineering yields software that:

- Is correct by mathematically sound design
- Is certified by statistically-valid testing
- Reduces the cycle time results from an incremental development strategy and the avoidance of rework
- Is well-documented
- Detects errors as early as possible and reduces the cost of errors during development and the incidence of failures during operation; thus the overall life cycle cost of software developed under Cleanroom can be expected to be far lower than the industry average.

The following principles are the foundation for the Cleanroom-based software development:

Incremental development under statistical quality control (SQC): Incremental development as practiced in Cleanroom provides a basis for statistical quality control of the development process. Each increment is a complete iteration of the process, and measures of performance in each increment (feedback) are compared with pre-established standards to determine whether or not the process is "in control." If quality standards are not met, testing of the increment ceases and developers return to the design stage.

Software development based on mathematical principles: In Cleanroom software engineering development, the key principle is that, a computer program is an expression of a mathematical function. The Box Structure Method is used for specification and design, and functional verification is used to confirm that the design is a correct implementation of the specification. Therefore, the specification must define that function before design and functional verification can begin. Verification of program correctness is performed through team review based on correctness questions. There is no execution of code prior to its submission for independent testing.

Software testing based on statistical principles: In Cleanroom, software testing is viewed as a statistical experiment. A representative subset of all possible uses of the software is generated, and performance of the subset is used as a basis for conclusions about general operational performance. In other words, a "sample" is used to draw conclusions about a "population." Under a testing protocol that is faithful to the principles of applied statistics, a scientifically valid statement can be made about the expected operational performance of the software in terms of reliability and confidence.

4.8 CONVENTIONAL SOFTWARE ENGINEERING MODELS Vs. CLEANROOM SOFTWARE ENGINEERING MODEL

Cleanroom has been documented to be very effective in new development and reengineering like whole systems or major subunits. The following points highlight the areas where Cleanroom affects or differs from more conventional software development engineering practice:

Small team of software engineers: A Cleanroom project team is small, typically six to eight qualified professionals, and works in an organised way to ensure the intellectual control of work in progress. The composition of the Cleanroom Process Teams are as follows:

- **Specification team** develops and maintains the system specification
- **Development team** develops and verifies software
- **Certification team** develops set of statistical tests to exercise software after development and reliability growth models used to assess reliability.

Time allocation across the life cycle phases: Because one of the major objectives of Cleanroom is to prevent errors from occurring, the amount of time spent in the design phase of a Cleanroom development is likely to be greater than the amount of time traditionally devoted to design.

Existing organisational practices: Cleanroom does not preclude using other software engineering techniques as long as they are not incompatible with Cleanroom principles. Implementation of the Cleanroom method can take place in a gradual manner. A pilot project can provide an opportunity to "tune" Cleanroom practices to the local culture, and the new practices can be introduced as pilot results to build confidence among software staff.

The following are some of the examples of the projects implemented using the Cleanroom engineering method:

- IBM COBOL/SF product
- Ericsson OS-32 operating system project
- USAF Space Command and Control Architectural Infrastructure (SCAI) STARS Demonstration Project at Peterson Air Force Base in Colorado Springs, Colorado.
- US Army Cleanroom project in the Tank-automotive and Armaments Command at the U.S. Army Picatinny Arsenal.

4.9 CLEANROOM SOFTWARE ENGINEERING PRINCIPLES, STRATEGY AND PROCESS OVERVIEW

This software development is based on mathematical principles. It follows the box principle for specification and design. Formal verification is used to confirm correctness of implementation of specification. Program correctness is verified by team reviews using questionnaires. Testing is based on statistical principles. The test cases are randomly generated from the usage model –failure data is interpreted using statistical models. The following is the phase-wise strategy followed for Cleanroom software development.

Increment planning: The project plan is built around the incremental strategy.

Requirements gathering: Customer requirements are elicited and refined for each increment using traditional methods.

Box structure specification: Box structures isolate and separate the definition of behaviour, data, and procedures at each level of refinement.

Formal design: Specifications (black-boxes) are iteratively refined to become architectural designs (state-boxes) and component-level designs (clear boxes).

Correctness verification: Correctness questions are asked and answered, formal mathematical verification is used as required.

Code generation, inspection, verification: Box structures are translated into program language; inspections are used to ensure conformance of code and boxes, as well as syntactic correctness of code; followed by correctness verification of the code.

Statistical test planning: A suite of test cases is created to match the probability distribution of the projected product usage pattern.

Statistical use testing: A statistical sample of all possible test cases is used rather than exhaustive testing.

Certification: Once verification, inspection, and usage testing are complete and all defects removed, the increment is certified as ready for integration.

Figure 4.1 depicts the Cleanroom software engineering development overview:

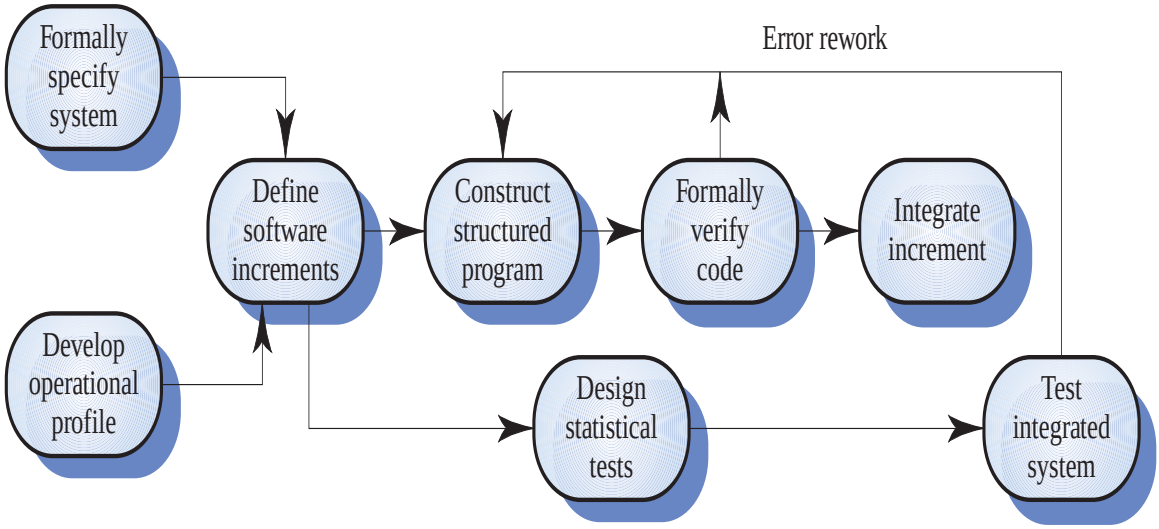


Figure 4.1: Overview of Cleanroom Software Engineering Development

4.10 LIMITATIONS OF CLEANROOM ENGINEERING

The following are some of the limitations of the Cleanroom software engineering method.

- Some people believe cleanroom techniques are too theoretical, too mathematical, and too radical for use in real software development.
- Relies on correctness verification and statistical quality control rather than unit testing (a major departure from traditional software development).
- Organisations operating at the ad hoc level of the Capability Maturity Model do not make rigorous use of the defined processes needed in all phases of the software lifecycle.

4.11 SIMILARITIES AND DIFFERENCES BETWEEN CLEANROOM AND OO PARADIGM

The following are the similarities and the differences between the Cleanroom software engineering development and OO software engineering paradigm.

Similarities

- Lifecycle - both rely on incremental development
- Usage - cleanroom usage model similar to OO use case
- State Machine Use - cleanroom state box and OO transition diagram
- Reuse - explicit objective in both process models

Key Differences

- Cleanroom relies on decomposition and OO relies on composition
- Cleanroom relies on formal methods while OO allows informal use case definition and testing
- OO inheritance hierarchy is a design resource whereas cleanroom usage hierarchy is system itself

- OO practitioners prefer graphical representations while cleanroom practitioners prefer tabular representations
- Tool support is good for most OO processes, but usually tool support is only found in cleanroom testing, not design.

Let us study the component based software engineering in the next section.

☞ Check Your Progress 2

1) What is Cleanroom Engineering and what are its objectives?

……

……

2) What is Statistical Process Control?

……

……

3) What are the benefits of Cleanroom software engineering?

……

……

……

4) What are the limitations of Cleanroom software engineering?

……

……

4.12 SOFTWARE REUSE AND ITS TYPES

In most engineering disciplines, systems are designed by composition (building system out of components that have been used in other systems). Software engineering has focused on custom development of components. To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt *systematic reuse* as a design process. The following are the types of software reuse:

- **Application System Reuse**
  - reusing an entire application by incorporation of one application inside another (COTS reuse)
  - development of application families (e.g., MS Office)
- **Component Reuse**
  - components (e.g., subsystems or single objects) of one application reused in another application
- **Function Reuse**
  - reusing software components that implement a single well-defined function.

Let us take into consideration the ***Component Reuse*** type and the development process of the components in the following sections:

"Suppose you have purchased a Television. Each component has been designed to fit a specific architectural style – connections are standardised, a communication protocol has been pre-established. Assembly is easy because you don't have to build the system from hundreds of discrete parts. Component-based software engineering (CBSE) strives to achieve the same thing. A set of pre-built, standardised software components are made available to fit a specific architectural style for some application domain. The application is then assembled using these components, rather than the "discrete parts" of a conventional programming language. Components may be

constructed with the explicit goal to allow them to be generalised and reused. Also, component reusability should strive to reflect stable domain abstractions, hide state representations, be independent (low coupling) and propagate exceptions via the component interface".

4.13 WHY COMPONENT BASED SOFTWARE ENGINEERING?

The goal of component-based software engineering is to increase the productivity, quality, and decrease time-to-market in software development. One important paradigm shift implied here is to build software systems from standard components rather than "reinventing the wheel" each time. This requires thinking in terms of system families rather than single systems.

CBSE uses Software Engineering principles to apply the same idea as OOP to the whole process of designing and constructing software systems. It focuses on *reusing* and *adapting* existing components, as opposed to just coding in a particular style. CBSE encourages the composition of software systems, as opposed to programming them. Two important aspects of the question are:

- Firstly, several underlying technologies have matured that permit building components and assembling applications from sets of those components. Object oriented and Component technology are examples – especially standards such as CORBA.
- Secondly, the business and organisational context within which applications are developed, deployed, and maintained has changed. There is an increasing need to communicate with legacy systems, as well as constantly updating current systems. This need for new functionality in current applications requires technology that will allow easy additions.

Software Component and its types

A software component is a nontrivial, independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. In some ways, this is similar to the description of an object in OOP. Components have an interface. They employ inheritance rules. Components can be characterised based on their use in the CBSE process. One category is *commercial off-the-shelf* (or COTS) components. These are components that can be purchased, pre-built, with the disadvantage that there is (usually) no source code available, and so the definition of the use of the component given by the manufacturer, and the services which it offers, must be relied upon or thoroughly tested, as they may or may not be accurate. The advantage, though, is that these types of components should (in theory) be more robust and adaptable, as they have been used and tested (and reused and retested) in many different applications.

In addition to COTS components, the CBSE process yields:

- qualified components
- adapted components
- assembled components
- updated components.

4.14 COMPONENT BASED SOFTWARE ENGINEERING (CBSE) PROCESS

CBSE is in many ways similar to conventional or object-oriented software engineering. A software team establishes requirements for the system to be built using conventional requirements elicitation techniques. An architectural design is established. Here though, the process differs. Rather than a more detailed design task,

the team now examines the requirements to determine what subset is directly amenable to *composition*, rather than *construction*.

For each requirement, we should question:

- Whether any commercial off-the-shelf (COTS) components available to implement the requirement?
- Whether internally developed reusable components available to implement the requirement?
- Whether interfaces for available components compatible within the architecture of the system to be built?

The team will attempt to modify or remove those system requirements that cannot be implemented with COTS or in-house components. This is not always possible or practical, but reduces the overall system cost and improves the time to market of the software system. It can often be useful to prioritise the requirements, or else developers may find themselves coding components that are no longer necessary as they have been eliminated from the requirements already.

The CBSE process identifies not only candidate components but also qualifies each component's interface, adapts components to remove architectural mismatches, assembles components into selected architectural style, and updates components as requirements for the system change.

Two processes occur in parallel during the CBSE process. These are:

- Domain Engineering
- Component Based Development.

4.14.1 Domain Engineering

This aims to identify, construct, catalogue, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. An application domain is like a *product family* – applications with similar functionality or intended functionality. The goal is to establish a mechanism by which software engineers can share these components in order to reuse them in future systems. As defined by Paul Clements, *domain engineering is about finding commonalities among systems to identify components that can be applied to many systems and to identify program families that are positioned to take fullest advantage of those components.*

Some examples of application domains are as follows:

- Air traffic control systems
- Defence systems
- Financial market systems.

Domain engineering begins by identifying the domain to be analysed. This is achieved by examining existing applications and by consulting experts of the type of application you are aiming to develop. A *domain model* is then realised by identifying operations and relationships that recur across the domain and therefore being candidates for reuse. This model guides the software engineer to identify and categorise components, which will be subsequently implemented.

One particular approach to domain engineering is *Structural Modeling*. This is a pattern-based approach that works under the assumption that every application domain has repeating patterns. These patterns may be in function, data, or behaviour that have reuse potential. This is similar to the pattern-based approach in OOP, where a particular style of coding is reapplied in different contexts. An example of *Structural Modeling* is in Aircraft avionics: The systems differ greatly in specifics, but all modern software in this domain has the same structural model.

4.14.2 Component Based Development

There are three stages in this process. These are:

- Qualification,
- Adaptation (also known as wrapping),
- Composition (all in the context of components).

Component qualification: examines reusable components. These are identified by characteristics in their interfaces, i.e., the services provided, and the means by which consumers access these services. This does not always provide the whole picture of whether a component will fit the requirements and the architectural style. This is a process of discovery by the Software Engineer. This ensures a candidate component will perform the function required, and whether it is compatible or adaptable to the architectural style of the system. The three important characteristics looked at are *performance, reliability and usability*.

Component Adaptation is required because very rarely will components integrate immediately with the system. Depending on the component type(e.g., COTS or in-house), different strategies are used for adaptation (also known as *wrapping*). The most common approaches are:

- **White box wrapping:** The implementation of the component is directly modified in order to resolve any incompatibilities. This is, obviously, only possible if the source code is available for a component, which is extremely unlikely in the case of COTS.
- **Grey box wrapping:** This relies on the component library providing a component extension language or API that enables conflicts to be removed or masked.
- **Black box wrapping:** This is the most common case, where access to source code is not available, and the only way the component can be adapted is by pre / post-processing at the interface level.

It is the job of the software engineer to determine whether the effort required to wrap a component adequately is justified, or whether it would be "cheaper" (from a software engineering perspective) to engineer a custom component which removes these conflicts. Also, once a component has been adapted it is necessary to check for compatibility for integration and to test for any unexpected behaviour which has emerged due to the modifications made.

Component Composition: integrated the components (whether they are qualified, adapted or engineered) into a working system. This is accomplished by way of an infrastructure which is established to bind the components into an operational system. This infrastructure is usually a library of specialised components itself. It provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

There are many mechanisms for creating an effective infrastructure, in order to achieve component composition, but in particular there is the *Data Exchange Model*, which allows users and applications to interact and transfer data (an example of which is *drag-and-drop* and *cut-and-paste*). These types of mechanisms should be defined for all reusable components. Also important is the *Underlying object model*. This allows the interoperation of components developed in different programming languages that reside on different platforms.

4.15 COMPONENT TECHNOLOGIES AVAILABLE

Common methods of achieving this is by the use of technologies such as *Corba* which allows components to communicate remotely and transparently over a network*, Java*

Beans and Microsoft COM which allows components from different vendors to work together within Windows. Sometimes it is necessary to bridge these technologies (e.g., COM/Corba) to facilitate integration of software developed by different vendors, or for integration with legacy systems.

It must be noted that as the requirements for a system change, components may need to be updated. Usually, some sort of *change control procedure* should be in place to facilitate configuration management, and to ensure updates to the components follow the architectural style of the system being developed.

4.16 CHALLENGES FOR CBSE

CBSE is facing many challenges today, some of these are summarised as follows:

- **Dependable systems and CBSE:** The use of CBD in safety-critical domains, real-time systems, and different process-control systems, in which the reliability requirements are more rigorous, is particularly challenging. A major problem with CBD is the limited possibility of ensuring the quality and other non-functional attributes of the components and thus our inability to guarantee specific system attributes.
- **Tool support:** The purpose of Software Engineering is to provide practical solutions to practical problems, and the existence of appropriate tools is essential for a successful CBSE performance. Development tools, such as Visual Basic, have proved to be extremely successful, but many other tools are yet to appear – component selection and evaluation tools, component repositories and tools for managing the repositories, component test tools, component-based design tools, run-time system analysis tools, component configuration tools, etc. The objective of CBSE is to build systems from components simply and efficiently, and this can only be achieved with extensive tool support.
- **Trusted components:** Because the trend is to deliver components in binary form and the component development process is outside the control of component users, questions related to component trustworthiness become of great importance.
- **Component certification:** One way of classifying components is to certificate them. In spite of the common belief that certification means absolute trustworthiness, it is in fact only gives the results of tests performed and a description of the environment in which the tests were performed. While certification is a standard procedure in many domains, it is not yet established in software in general and especially not for software components.
- **Composition predictability:** Even if we assume that we can specify all the relevant attributes of components, it is not known how these attributes determine the corresponding attributes of systems of which they are composed. The ideal approach, to derive system attributes from component attributes is still a subject of research. A question remains - “Is such derivation at all possible? Or should we not concentrate on the measurement of the attributes of component composites?”
- **Requirements management and component selection:** Requirements management is a complex process. A problem of requirements management is that requirements in general are incomplete, imprecise and contradictory. In an in-house development, the main objective is to implement a system which will satisfy the requirements as far as possible within a specified framework of different constraints. In component-based development, the fundamental approach is the reuse of existing components. The process of engineering requirements is much more complex as the possible candidate components are usually lacking one or more features which meet the system requirements exactly. In addition, even if some components are individually well suited to the

system, it is not necessary that they do not function optimally in combination with others in the system- or perhaps not at all. These constraints may require another approach in requirements engineering – an analysis of the feasibility of requirements in relation to the components available and the consequent modification of requirements. As there are many uncertainties in the process of component selection there is a need for a strategy for managing risks in the components selection and evolution process.

- **Long-term management of component-based systems:** As component-based systems include sub-systems and components with independent lifecycles, the problem of system evolution becomes significantly more complex. CBSE is a new approach and there is little experience as yet of the maintainability of such systems. There is a risk that many such systems will be troublesome to maintain.
- **Development models:** Although existing development models demonstrate powerful technologies, they have many ambiguous characteristics, they are incomplete, and they are difficult to use.
- **Component configurations:** Complex systems may include many components which, in turn, include other components. In many cases compositions of components will be treated as components. As soon as we begin to work with complex structures, the problems involved with structure configuration pop up.

☞ Check Your Progress 3

1) What are the benefits of reuse of software?

……

……

2) What is Commercial off the Shelf Software (COTS)?

……

……

……

4.17 REENGINEERING – AN INTRODUCTION

Over the past few years, legacy system reengineering has emerged as a business critical activity. Software technology is evolving by leaps and bounds, and in most cases, legacy software systems need to operate on new computing platforms, be enhanced with new functionality, or be adapted to meet new user requirements. The following are some of the reasons to reengineer the legacy system:

- Allow legacy software to quickly adapt to changing requirements
- Comply to new organisational standards
- Upgrade to newer technologies/platforms/paradigms
- Extend the software's life expectancy
- Identify candidates for reuse
- Improve software maintainability
- Increase productivity per maintenance programmer
- Reduce reliance on programmers who have specialised in a given software system
- Reduce maintenance errors and costs.

Reengineering applies reverse engineering to existing system code to extract design and requirements. Forward engineering is then used to develop the replacement system.

Let us elaborate the definition. Reengineering is the examination, analysis, and alteration of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form. The process typically encompasses a

combination of reverse engineering, re-documentation, restructuring, and forward engineering. The goal is to understand the existing software system components (specification, design, implementation) and then to re-do them to improve the system's functionality, performance, or implementation.

The reengineering process is depicted in *Figure 4.2*. Re-engineering starts with the code and comprehensively reverse engineers by increasing the level of abstraction as far as needed toward the conceptual level, rethinking and re-evaluating the engineering and requirements of the current code, then forward engineers using a waterfall software development life-cycle to the target system. In re-engineering, industry reports indicate that approximately 60% of the time is spent on the reverse engineering process, while 40% of the time is spent on forward engineering. Upon completion of the target system, most projects must justify their effort, showing that the necessary functionality has been maintained while quality has improved (implying improved reliability and decreased maintenance costs).

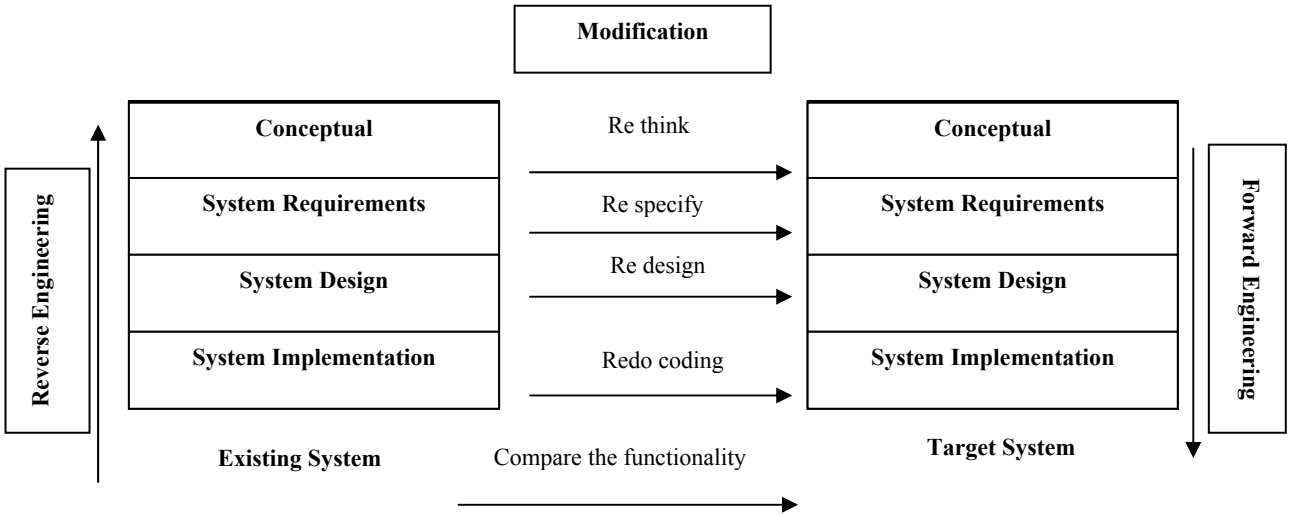


Figure 4.2: Process of Software Reengineering

4.18 OBJECTIVES OF REENGINEERING

The objectives of a specific software reengineering activity are determined by the goals of the clients and users of the system. However, there are two general reengineering objectives as shown below:

- **Improve quality:** Typically, the existing software system is of low quality, due to many modifications. User and system documentation is often out-of-date or no longer in existence. Re-engineering is intended to improve software quality and to produce current documentation. Improved quality is needed to increase reliability, to improve maintainability, to reduce the cost of maintenance, and to prepare for functional enhancement. Object-oriented technology may be applied as a means for improving maintainability and reducing costs.
- **Migration:** Old working software may still meet users' needs, but it may be based on hardware platforms, operating systems, or languages that have become obsolete and thus may need to be re-engineered, transporting the software to a newer platform or language. Migration may involve extensive redesign if the new supporting platforms and operating systems are very different from the original, such as the move from a mainframe to a network-based computing environment.

4.19 SOFTWARE REENGINEERING LIFE CYCLE

In this section, we present an evolutionary view of the software reengineering life-cycle:

Requirements analysis phase: It refers to the identification of concrete reengineering goals for a given software. The specification of the criteria should be specified and illustrated in the new reengineered system (for example, faster performance). Violations that need to be repaired are also identified in this phase.

Model analysis phase: This refers to documenting and understanding the architecture and the functionality of the legacy system being reengineered. In order to understand and to transform a legacy system, it is necessary to capture its design, its architecture and the relationships between different elements of its implementation. As a consequence, a preliminary model is required in order to document the system and the rationale behind its design. This requires reverse engineering the legacy system in order to extract design information from its code.

Source code analysis phase: This phase refers to the identification of the parts of the code that are responsible for violations of requirements originally specified in the system's analysis phase. This task encompasses the design of methods and tools to inspect, measure, rank, and visualize software structures. Detecting error prone code that deviates from its initial requirement specifications requires a way to measure where and by how much these requirements are violated. Problem detection can be based on a static analysis of the legacy system (i.e., analysing its source code or its design structure), but it can also rely on a dynamic usage analysis of the system (i.e., an investigation of how programs behave at run-time).

Remediation phase: It refers to the selection of a target software structure that aims to repair a design or a source code defect with respect to a target quality requirement. Because legacy applications have been evolved in such a way that classes, objects, and methods may heavily depend on each other, a detected problem may have to be decomposed into simpler sub-problems.

Transformation phase: This consists of physically transforming software structures according to the remediation strategies selected previously. This requires methods and tools to manipulate and edit software systems, re-organise and re-compile them automatically, debug and check their consistency, and manage different versions of the software system being reengineered.

Evaluation phase refers to the process of assessing the new system as well as establishing and integrating the revised system throughout the corporate operating environment. This might involve the need for training and possibly the need for adopting a new improved business process model.

Such a reengineering life-cycle yields a reengineering process. First, the source code is represented as an Abstract Syntax Tree. The tree is further decorated with annotations that provide linkage, scope, and type information. Once software artifacts have been understood, classified and stored during the reverse engineering phase, their behaviour can be readily available to the system during the forward engineering phase. Then, the forward engineering phase aims to produce a new version of a legacy system that operates on the target architecture and aims to address specific maintainability or performance enhancements. Finally, we use an iterative procedure to obtain the new migrant source code by selecting and applying a transformation which leads to performance or maintainability enhancements. The transformation is selected from the soft-goal interdependency graphs. The resulting migrant system is then evaluated and the step is repeated until the specific quality requirements are met.

☞ Check Your Progress 4

1) What is Software Reengineering and what are its objectives?

……

……

2) What is Reverse Engineering?

……

……

4.20 SUMMARY

In this unit, we have gone through the Formal Methods, Cleanroom software engineering, Component Based Software Engineering and the Software Reengineering process.

Formal methods promise to yield benefits in quality and productivity. They provide an exciting paradigm for understanding software and its development, as well as a set of techniques for use by software engineers. Formal methods can provide more precise specifications, better internal communication, and an ability to verify designs before executing them during test, higher quality and productivity. To get their full advantages, formal methods should be incorporated into a software organisation's standard procedures. Software development is a social process, and the techniques employed need to support that process. How to fully fit formal methods into the lifecycle is not fully understood. Perhaps there is no universal answer, but only solutions that vary from organisation to organisation.

Harlan Mills has developed the Cleanroom methodology, which is one approach for integrating formal methods into the lifecycle. The Cleanroom approach combines formal methods and structured programming with Statistical Process Control (SPC), the spiral lifecycle and incremental releases, inspections, and software reliability modeling. It fosters attitudes, such as emphasising defect prevention over defect removal, that are associated with high quality products in non-software fields.

Component Based Software Engineering introduced major changes into design and development practices, which introduces extra cost. Software engineers need to employ new processes and ways of thinking – this, too, can introduce extra cost in training and education. However, initial studies into the impact of CBSE on product quality, development quality and cost, show an overall gain, and so it seems likely that continuing these practices in the future will improve software development. It is still not clear how exactly CBSE will mature, but it is clear that it aids in the development of today's large-scale systems, and will continue to aid in the development of future systems, and is the perfect platform for addressing the requirements of modern businesses**.**

Software reengineering is a very wide-ranging term that encompasses a great many activities. As the name suggests, software reengineering is applied to existing pieces of software, in an after-the-fact fashion, via the reapplication of an engineering process.

4.21 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Formal methods is that area of computer science that is concerned with the application of mathematical techniques to the design and implementation of computer software.

2) Following are the main parts to formal methods:

   i) **Formal specification:** Using mathematics to specify the desired properties of a computer system.

   ii) **Formal verification:** Using mathematics to prove that a computer system satisfies its specification.

iii) **Automated programming:** Automating the process of program generation.

Check Your Progress 2

1) The Cleanroom methodology is an iterative, life-cycle approach focused on software quality, especially reliability. Begun by Harlan Mills, it combines formal specifications, structured programming, formal verifications, formal inspections, functional testing based on random selection of test data, software reliability measurement, and Statistical Process Control (SPC) in an integrated whole. The Cleanroom approach fosters attitudes, such as emphasising defect prevention over defect removal, that are associated with high quality products in fields other than software.

2) Statistical Process Control (SPC) is commonly used in manufacturing, involves continuous process monitoring and improvement to reduce the variance of outputs and to ensure that the process remains under control.

3) Benefits of Cleanroom software engineering include significant improvements in *correctness*, *reliability*, and *understandability*. These benefits usually translate into a reduction in field-experienced product failures, reduced cycle time, ease of maintenance, and longer product life.

4) Cleanroom techniques are too theoretical, too mathematical, and too radical for use in real software development. They rely on correctness verification and statistical quality control rather than unit testing (a major departure from traditional software development). Organisations operating at the ad hoc level of the Capability Maturity Model do not make rigorous use of the defined processes needed in all phases of the software life cycle.

Check Your Progress 3

1) Benefits of software reuse are:

- **Increased Reliability**: components already exercised in working systems
- **Reduced Process Risk**: Less uncertainty in development costs
- **Effective Use of components**: Reuse components instead of people
- **Standards Compliance**: Embed standards in reusable components
- **Accelerated Development**: Avoid custom development and speed up delivery.

2) COTS systems usually have a complete applications library and also offers an applications programming interface (API). These are helpful in building large systems by integrating COTS components is a viable development strategy for some types of systems (e.g., E-commerce or video games).

Check Your Progress 4

1) Software reengineering is a process that aims to either

   i) Improve understanding of a piece of software, or
   ii) Prepare for an improvement in the software itself (e.g., increasing its maintainability or reusability).

As the name suggests software reengineering is applied to existing pieces of software, in an after-the-fact fashion, via the reapplication of an engineering process. Software reengineering is a very wide-ranging term that encompasses a great many activities. For example, software reengineering could involve refactoring a piece of software, redocumenting it, reverse engineering it or changing its implementation language.

2) Reverse engineering is a process which is used to improve the understanding of a program. So-called program comprehension is crucial to maintenance. Approximately 50% of the development time of programmers is spent understanding the code that they are working on. Reverse engineering often involves the production of diagrams. It can be used to abstract away irrelevant

implementation and design details to recover the original requirements of the program.

4.22 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) *Component Software –Beyond Object-Oriented Programming,1998,* Szyperski C; Addison-Wesley.

4) Software Engineering, Schaum's Outlines, 2003,David Gustafson,Tata Mc Graw-Hill.

Reference websites

- **http://www.rspa.com**
- **http://www.ieee.org**

4.23 GLOSSARY

Software Reengineering: The examination and alteration of an existing subject system to reconstitute it in a new form. This process encompasses a combination of sub-processes such as reverse engineering, restructuring, redocumentation, forward engineering, and retargeting.

Reverse Engineering: The engineering process of understanding, analysing, and abstracting the system to a new form at a higher abstraction level.

Forward Engineering: Forward engineering is the set of engineering activities that consume the products and artifacts derived from legacy software and new requirements to produce a new target system.

Data Reengineering: Tools that perform all the reengineeering functions associated with source code (reverse engineering, forward engineering, translation, redocumentation, restructuring / normalisation, and retargeting) but act upon data files.

Redocumentation: The process of analysing the system to produce support documentation in various forms including users manuals and reformatting the systems' source code listings.

Restructuring: The engineering process of transforming the system from one representation form to another at the same relative abstraction level, while preserving the subject system's external functional behavior.

Retargeting: The engineering process of transforming and hosting / porting the existing system with a new configuration.

Source Code Translation: Transformation of source code from one language to another or from one version of a language to another version of the same language (e.g., going from COBOL-74 to COBOL-85).

Business Process Reengineering (BPR): The fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service, and speed.