Software Engineering and Testing

OBJECTIVES:

- To understand the concepts of analysis, design and implementation of a software product.
- To have general understanding about object-oriented software engineering.
- To make students to get experience and be ready for the large scale projects in IT Industry.

Unit I Introduction:- Evolution – From an Art form on Engineering Discipline: Evolution of an Art into an Engineering Discipline. – Software Development of Projects: Program versus Product – Emergence of Software Engineering: Early Computer Programming – High Level Language Programming – Control Flow-based Design – Data Structure Oriented Design – Object Oriented Design. **Software Life Cycle Models:-** A few Basic Concepts – Waterfall Model and its Extension: Classical Waterfall Model – Iterative Waterfall Model – Prototyping Model – Evolutionary Model. – Rapid Application Development (RAD): Working of RAD. – Spiral Model. (12L)

Unit II Software Project Management:- Responsibilities of a Software Project Manager – Project Planning- Project Estimation Techniques-Risk Management. **Requirements Analysis and Specification:-** Requirements Gathering and Analysis – Software Requirements Specifications (SRS):Users of SRS Document – Characteristics of a Good SRS Document – Important Categories of Customer Requirements – Functional Requirements – How to Identify the Functional Requirements? – Organisation of the SRS Document. (12L)

Unit III Software Design:- Overview of the Design Process: Outcome of the Design Process – Classification of Design Activities. – How to Characterize a good Software Design? **Function-Oriented Software Design:-** Overview of SA/SD Methodology – Structured Analysis – Developing the DFD Model of a System: Context Diagram – Structured Design – Detailed Design. (12L) Page **29** of **57**

Unit IV User Interface Design:- Characteristics of a good User Interface - Basic Concepts – Types of User Interfaces – Fundamentals of Components based GUI Development: Window System. **Coding and Testing:-** Coding – Software Documentation – Testing: Basic Concepts and Terminologies – Testing Activities. – Unit Testing – Black-box Testing: Equivalence Class Partitioning – Boundary Value Analysis. – White-box Testing. (12L)

Unit V Software Reliability and Quality Management:- Software Reliability: Hardware versus Software Reliability. – Software Quality – Software Quality Management System – ISO 9000: What is ISO 9000 Certification? – ISO 9000 for Software Industry – Shortcomings of ISO 9000 Certification. – SEI Capability Maturity Model: Level 1 to Level 5. **Software Maintenance:-** Characteristics of Software Maintenance: Characteristics of Software Evolution – Software Reverse Engineering. (12L)

Text Book: Fundamentals of Software Engineering Fourth Edition by Rajib Mall – PHI Learning Private Limited 2015.

UNIT-1

SOFTWARE ENGINEERING

Software engineering discusses systematic and cost-effective techniques for software development. These techniques help develop software using an engineering approach.

Is software engineering a science or an art?

- Writing good quality programs is an art.
- Past experiences have been systematically organized and wherever possible theoretical basis to the empirical observations have been provided.
- An appropriate solution is chosen out of the candidate solutions based on various trade-offs that need to be made on account of issues of cost, maintainability, and usability.
- Engineering disciplines such as software engineering make use of only well-understood and well-documented principles

1. EVOLUTION—FROM AN ART FORM TO AN ENGINEERING DISCIPLINE

- Early programmers used an *ad hoc* programming style. This style of program development is now variously being referred to as *exploratory*, *build and fix*, and *code and fix* styles.
- The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to adhere to—every programmer himself evolves his own software development techniques.

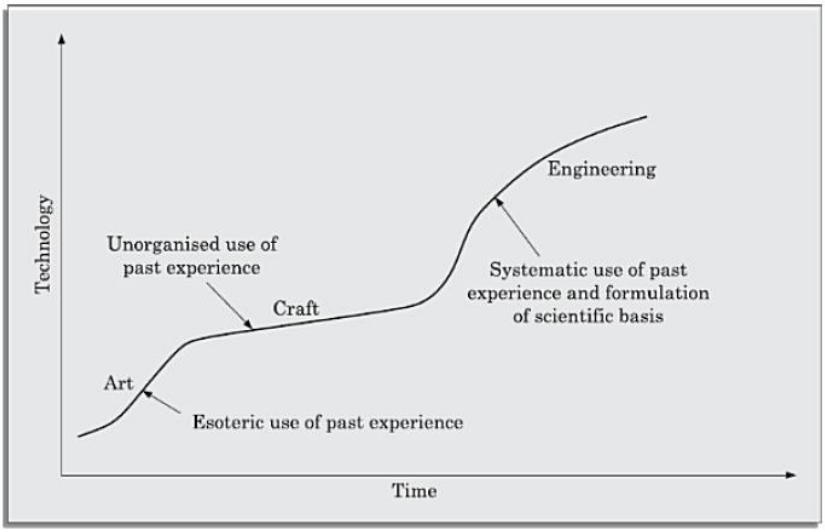


Figure 1.1: Evolution of technology with time.

- Organisations are spending increasingly larger portions of their budget on software as compared to that on hardware.

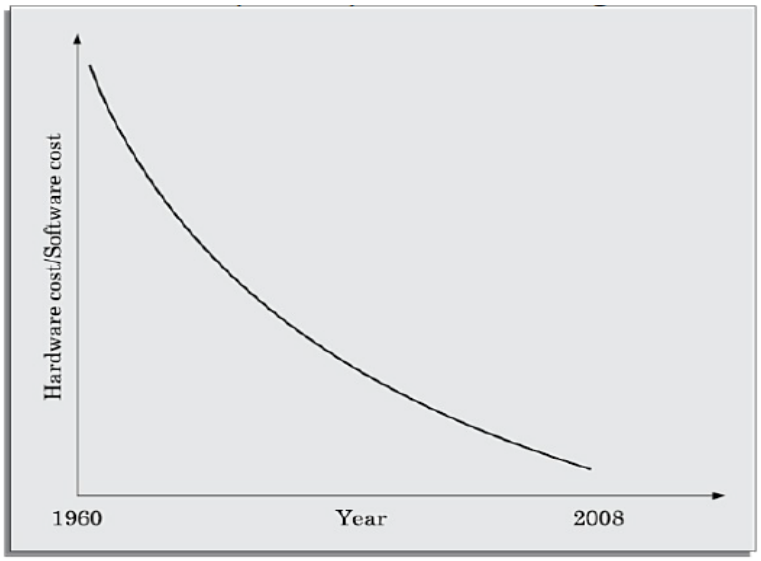


Figure 1.2: Relative changes of hardware and software costs over time.

2. SOFTWARE DEVELOPMENT PROJECTS

A Professional software is developed by a group of software developers working together in a team.

a) Types of Software Development Projects

1. Software products

- Microsoft's Windows and the Office suite, Oracle DBMS, software accompanying a camcorder or a laser printer, etc. These software are available off-the-shelf for purchase and are used by a diverse range of customers. These are called ***generic software products***

2. Software services

- Customized ***software*** is developed according to the specification drawn up by one or at most a few customers.
- Another type of software service i s ***outsourced software***. Sometimes, it can make good commercial sense for a company developing a large project to outsource some parts of its development work to other companies

b) Software Projects Being Undertaken by Indian Companies

- Indian software companies have excelled in executing software services projects and have made a name for themselves all over the world.

3. EMERGENCE OF SOFTWARE ENGINEERING

Software engineering techniques evolution is the result of a series of innovations and accumulation of experience about writing good quality programs. They are

a) Early Computer Programming

b) High-level Language Programming

c) Control Flow-based Design
d) Data Structure-oriented Design
e) Data Flow-oriented Design
f) Object-oriented Design
g) Web based Design

a) Early Computer Programming

- Every programmer developed his own individualistic style of writing programs according to his intuition and used this style ***ad hoc*** while writing different programs.
- Then designated this style of programming as the *build and fix* (or the *exploratory programming*) style.

b) High-level Language Programming

- In 1960s high-level languages such as FORTRAN, ALGOL, and COBOL were introduced.
- Writing each high-level programming construct in effect enables the programmer to write several machine instructions. Also, the machine details (registers, flags, etc.) are abstracted from the programmer.

c) Control Flow-based Design

- experienced programmers advised other programmers to pay particular attention to the design of a program's *control flow structure*.
- A program's control flow structure indicates the sequence in which the program's instructions are executed.

```
1       if(customer_savings_balance>withdrawal_request) {
2   100:    issue_money=TRUE;
3           GOTO 110;
        }
4       else if(privileged_customer==TRUE)
5           GOTO 100;
6       else GOTO 120;
7   110: activate_cash_dispenser(withdrawal_request);
8       GOTO 130;
9   120:    print(error);
10  130:    end-transaction();
```

(a) An example unstructured program

```
1   if(privileged_customer||(customer_savings_balance>withdrawal_request)){
2           activate_cash_dispenser(withdrawal_request);
    }
3   else print(error);
4   end-transaction();
```

(b) Corresponding structured program

Figure 1.8: An example of (a) Unstructured program (b) Corresponding structured program.

- If the flow chart representation is simple, then the corresponding code should be simple.

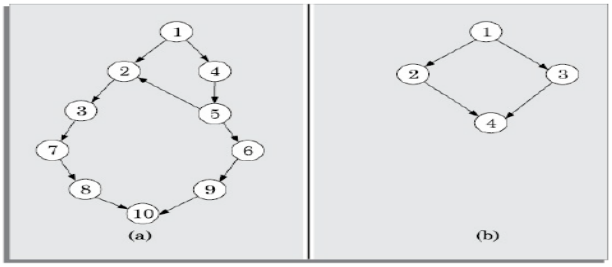


Figure 1.9: Control flow graphs of the programs of Figures 1.8(a) and (b).

- For example, for the program of Fig 1.9(a) you would have to understand the execution of the program along the paths 1-2-3-7-8-10, 1-4-5-6-9-10, and 1-4-5-2-3-7-8-10. A program having a messy control flow (i.e. flow chart) structure, would have a large number of execution paths.

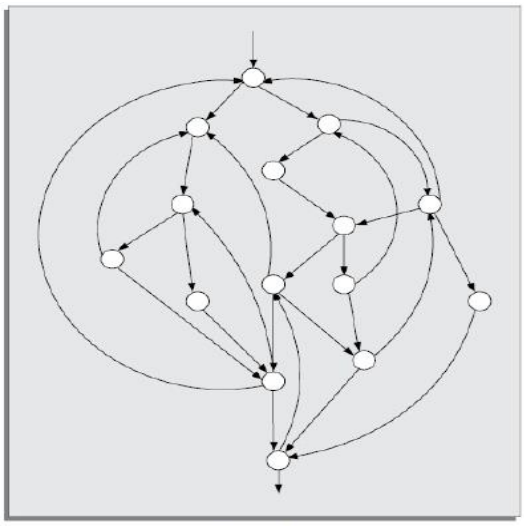


Figure 1.10: CFG of a program having too many GO TO statements.

- **Are GO TO statements the culprits?**
  Dijkstra [1968] published his (now famous) article "GO TO Statements Considered Harmful". GO TO statements alter the flow of control arbitrarily, resulting in too many paths.
- **Structured programming**
  - A program is called structured when it uses only the sequence, selection, and iteration types of constructs and is modular.
  - Very soon several languages such as PASCAL, MODULA, C, etc., became available which were specifically designed to support structured programming.

d) Data Structure-oriented Design

- It is much more important to pay attention to the design of the important data structures of the program than to the design of its control structure.
- Design techniques based on this principle are called *data structure- oriented* design techniques.
- Using data structure-oriented design techniques, first a program's data structures are designed. The code structure is designed based on the data structure.

e) Data Flow-oriented Design

- The data flow-oriented techniques advocate that the major data items handled by a system must be identified and the processing required on these data items to produce the desired outputs should be determined.

- The functions (also called as *processes*) and the data items that are exchanged between the different functions are represented in a diagram known as a *data flow diagram* (DFD).

f) Object-oriented Design

- Data flow-oriented techniques evolved into object-oriented design (OOD) techniques in the late seventies.
- Natural objects relevant to a problem are first identified and then the relationships among the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a *data hiding* (also known as *data abstraction*) entity.

g) Web based design

- Many of the present day software are required to work in a client-server environment through a web browser-based access (called *web-based software*).

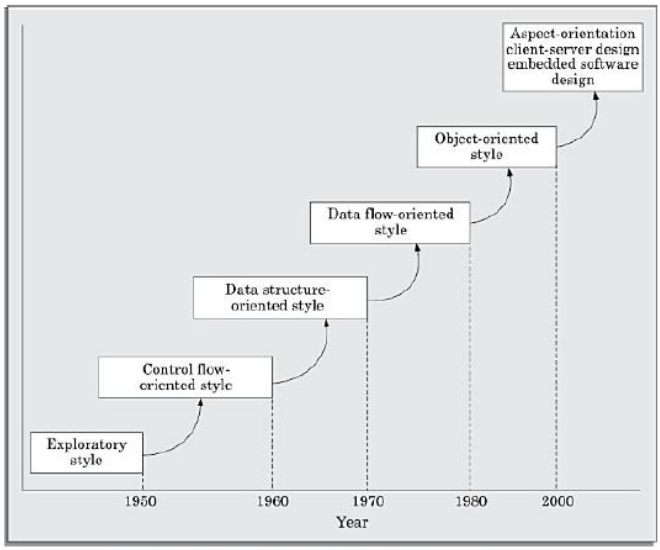


Figure 1.12: Evolution of software design techniques.

4. SOFTWARE LIFE CYCLE MODELS

In the software engineering approaches emphasize software development through a well-defined and ordered set of activities. These activities are graphically modeled (represented) as well as textually described and are variously called a s **software life cycle model**, software development life cycle (SDLC) model, and software development process model.

.

a) A few basic concepts of Software life cycle

- The software life cycle has been defined to imply the different stages (or phases) over which a software evolves from an initial customer request for it, to a fully developed software, and finally to a stage where it is no longer useful to any user, and then it is discarded.
- The life cycle of every software starts with a request for it by one or more customers.
- This stage where the customer feels a need for the software and forms rough ideas about the required features is known as the **inception stage**.

- A software evolves through a series of identifiable stages (also called phases) on account of the development activities carried out by the developers, until it is fully developed and is released to the customers.
- Once installed and made available for use, the users start to use the software. This signals the start of the operation (also called **maintenance**) phase.
- The maintenance phase usually involves continually making changes to the software to accommodate the bug-fix and change requests from the user.
- The **operation phase** is usually the longest of all phases and constitutes the useful life of a software.
- Finally the software is retired
- The life cycle of a software represents the series of identifiable stages through which it evolves during its life time.

Software development life cycle (SDLC) model

- A **software development life cycle** (SDLC) model (also called software life cycle model and software development process model) describes the different activities that need to be carried out for the software to evolve in its life cycle.
- The terms **software development life cycle (SDLC)** and software development process are interchangeable.
- An SDLC graphically depicts the different phases through which a software evolves. It is usually accompanied by a textual description of the different activities that need to be carried out during each phase.
- Process and methodology are at time used interchangeably, there is a subtle difference between the two. First, the term **process** has a broader scope and addresses either all the activities taking place during software development, or certain coarse grained activities such as design (e.g. design process), testing (test process), etc. Further, a software process not only identifies the specific activities that need to be carried out, but may also prescribe certain **methodology** for carrying out each activity.

A software development process has a much broader scope as compared to a **software development methodology**. A **process** usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity. A **methodology**, in contrast, describes the steps to carry out only a single or at best a few individual activities.

Software development organisations have realized that adherence to a suitable life cycle model helps to produce good quality software and that helps minimize the chances of time and cost overruns.

- **Programming-in-the-small** refers to development of a toy program by a single programmer.

- While development of a software of the former type could succeed even while an individual programmer uses a **build and fix** style of development,
- **Programming-in-the-large** refers to development of a professional software through team effort.
- Use of a suitable SDLC is essential for a professional software development project involving team effort to succeed.
- A **documented development process** forms a common understanding of the activities to be carried out among the software developers and helps them to develop software in a systematic and disciplined manner.
- A **documented development process model**, besides preventing the misinterpretations that might occur when the development process is not adequately documented, also helps to identify inconsistencies, redundancies, and omissions in the development process.

Phase entry and exit criteria

- A good SDLC besides clearly identifying the different phases in the life cycle, should unambiguously define the entry and exit criteria for each phase.
- The phase entry (or exit) criteria is usually expressed as a set of conditions that needs to be be satisfied for the phase to start (or to complete).
- As an example, the phase exit criteria for the software requirements specification phase, can be that the *software requirements specification* (SRS) document is ready, has been reviewed internally, and also has been reviewed and approved by the customer.
- Only after these criteria are satisfied, the next phase can start.

5. WATERFALL MODEL AND ITS EXTENSIONS

The waterfall model and its derivatives were extremely popular in the 1970s.

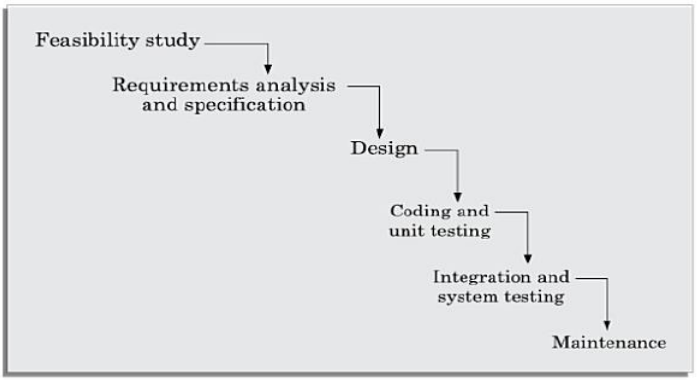
Classical Waterfall Model

- Classical waterfall model is intuitively the most obvious way to develop software. The classical waterfall model divides the life cycle into a set of phases as shown in figure below.

Phases of the classical waterfall model

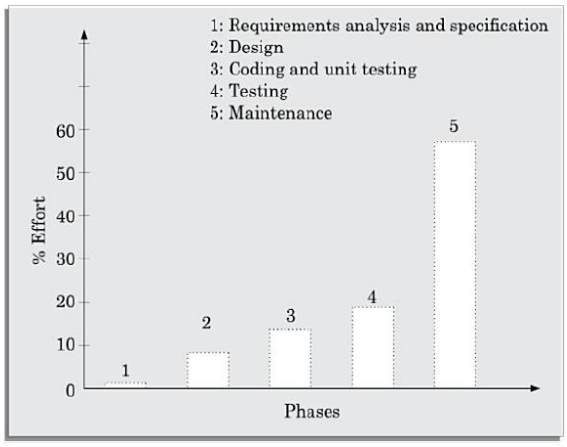
- The different phases are feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance.
- The phases starting from the feasibility study to the integration and system testing phase are known as the development phases.
- A software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer.

Fig. Phases of the classical water fall model



- After the delivery of software, customers start to use the software signaling the commencement of the **operation** phase.
- As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken.
- Therefore, the last phase is also known as the **maintenance**
- An activity that spans all phases of software development is **project management**. Since it spans the entire project duration, no specific phase is named after it.
- In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team.
- On the average, about 60 per cent of the total effort put in by the development team in the entire life cycle is spent on the maintenance activities alone.

Fig. Relative effort distribution among different phases.



- However, among the development phases, the integration and system testing phase requires the maximum effort in a typical development project.

1. Feasibility study

- The main focus of the feasibility study stage is to determine whether it would be *financially* and *technically feasible* to develop the software.

- Feasibility study involves carrying out several activities such as
    - collection of basic information relating to the software such as the different data items that would be input to the system,
    - the processing required to be carried out on these data,
    - the output data required to be produced by the system, as well as various constraints on the development.
- These collected data are analyzed to perform at the following:
    - Development of an overall understanding of the problem
    - Formulation of the various possible strategies for solving the problem
    - Evaluation of the different solution strategies.

2. Requirements analysis and specification

- The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly.
- This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification.

a) Requirements gathering and analysis:

- The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements.
- For this, first requirements are gathered from the customer and then the gathered requirements are analyzed.

b)Requirements specification:

- After the requirement gathering and analysis activities are complete, the identified requirements are documented.
- This is called a *software requirements specification* (SRS) document.
- The SRS document is written using end-user terminology.
- This makes the SRS document understandable to the customer.

3. Design

- The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language.
- During the design phase the ***software architecture*** is derived from the SRS document.

a) Procedural design approach:

- The traditional design approach is in use in many software development projects at the present time.
- This traditional design technique is based on the data flow-oriented design approach.
- It consists of two important activities; first ***structured analysis*** of the requirements specification is carried out where the detailed structure of the problem is examined.
- This is followed by a ***structured design*** step where the results of structured analysis are transformed into the software design.

Structured design consists of two main activities—**architectural design** (also called *high-level design*) and **detailed design** (also called *Low-level design*).

- High-level design involves decomposing the system into modules, and representing the interfaces and the invocation relationships among the modules.
- A high-level software design is sometimes referred to as the ***software architecture***.
- During the **detailed design** activity, internals of the individual modules such as the data structures and algorithms of the modules are designed and documented

b) Object-oriented design approach:

- In this technique, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified.
- The object structure is further refined to obtain the detailed design.
- The OOD approach is credited to have several benefits such as lower development time and effort, and better maintainability of the software.

4. Coding and unit testing

- The purpose of the coding and unit testing phase is to translate a software design into source code and to ensure that individually each function is working correctly.
- The coding phase is also sometimes called the **implementation phase**, since the design is implemented into a workable solution in this phase.
- Each component of the design is implemented as a **program module**.
- The end-product of this phase is a set of program modules that have been individually unit tested.
- The main objective of unit testing is to determine the correct working of the individual modules. The specific activities carried out during unit testing include designing test cases, testing, debugging to fix problems, and management of test cases.

5. Integration and System Testing

- Integration of different modules is undertaken soon after they have been coded and unit tested.
- During the integration and system testing phase, the different modules are integrated in a planned manner.

a) Integration testing is carried out to verify that the interfaces among different units are working satisfactorily.

- On the other hand, the goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

b) System testing usually consists of three different kinds of testing activities:

- **alpha-testing:** testing is the system testing performed by the development team.
- **beta-testing:** This is the system testing performed by a friendly set of customers.
- **Acceptance testing:** After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

6. **Maintenance**

- The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself.
- Many studies carried out in the past confirm this and indicate that the ratio of relative effort of developing a typical software product and the total effort spent on its maintenance is roughly 40:60.
- Maintenance is required in the following three types of situations:

a) Corrective maintenance: This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
b) Perfective maintenance: This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.

c) Adaptive maintenance: Adaptive maintenance is usually required for porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system.

Shortcomings of the classical waterfall model

The classical waterfall model is a very simple and intuitive model. However, it suffers from several shortcomings.

1. No feedback paths: Once a defect is detected at a later time, the developers need to redo some of the work done during that phase and also redo the work of later phases that are affected by the rework.

2. Difficult to accommodate change requests: The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.

3. Inefficient error corrections:

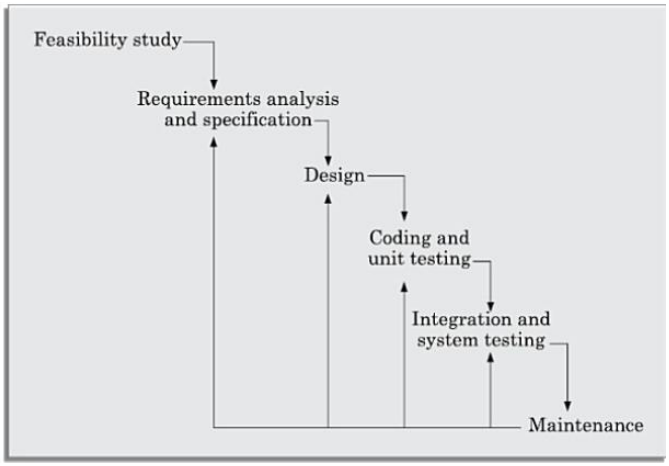
This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

4. No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes.

6. ITERATIVE WATER FALL MODEL:

- The main change brought about by the iterative waterfall model to the classical waterfall model is in the form of providing feedback paths from every phase to its preceding phases.
- The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase.
- For example, if during the testing phase a design error is identified, then the feedback path allows the design to be reworked and the changes to be reflected in the design documents and all other subsequent documents.
- There is no feedback path to the feasibility stage.

Figure. Iterative waterfall model



Phase containment of errors: It is advantageous to detect these errors in the same phase in which they take place, since early detection of bugs reduces the effort and time required for correcting those.

- In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the testing activities, thereby incurring higher cost.
- The principle of detecting errors as close to their points of commitment as possible is known as ***phase containment of errors***.
- After all, the end product of many phases is text or graphical documents, e.g. SRS document, design document, test plan document, etc.

Phase overlap

- In spite of the best effort to detect errors in the same phase in which they are committed, some errors escape detection and are detected in a later phase. An important reason for phase overlap is that usually the work required to be carried out in a phase is divided among the team members.
- Some members may complete their part of the work earlier than other members. If strict phase transitions are maintained, then the team members who complete their work early would idle waiting for the phase to be complete, and are said to be in a ***blocking state***.

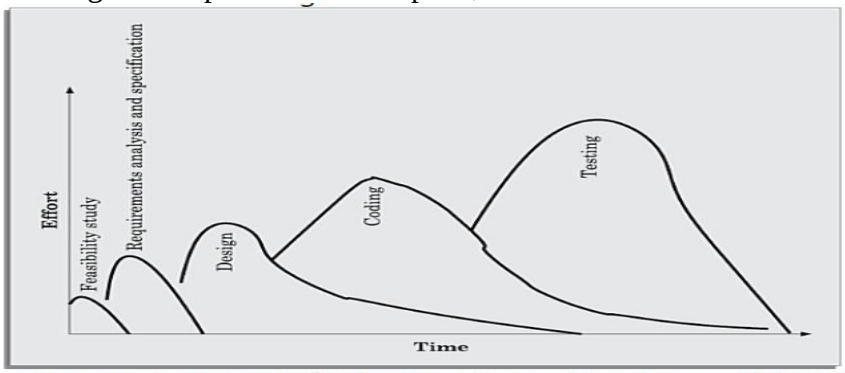


Figure 2.4: Distribution of effort for various phases in the iterative waterfall model.

Shortcomings of the iterative waterfall model

1. Difficult to accommodate change requests: Based on the frozen requirements, detailed plans are made for the activities to be carried out during the design, coding, and testing phases. Since activities are planned for the entire duration, substantial effort and resources are invested in the activities as developing the complete requirements specification, design for the complete functionality and so on. Once requirements have been frozen, the waterfall model provides no scope for any modifications to the requirements.

2. Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before it is delivered to the customer. There is no provision for any intermediate deliveries to occur.

3. Phase overlap not supported: For most real life projects, it becomes difficult to follow the rigid phase sequence prescribed by the waterfall model. By the term *a rigid phase sequence*, we mean that a phase can start only after the previous phase is complete in all respects. As already discussed, strict adherence to the waterfall model creates *blocking states*.

4. Error correction unduly expensive: In waterfall model, validation is delayed till the complete development of the software. As a result, the defects that are noticed at the time of validation incur expensive rework and result in cost escalation and delayed delivery.

5. Limited customer interactions: This model supports very limited customer interactions. It is generally accepted that software developed in isolation from the customer is the cause of many problems. In fact, interactions occur only at the start of the project and at project completion.

6. No support for risk handling and code reuse: It becomes difficult to use the waterfall model in projects that are susceptible to various types of risks, or those involving significant reuse of existing development artifacts. Please recollect that software services types of projects usually involve significant reuse.

7. PROTOTYPING MODEL

- The prototype model suggests building a working *prototype* of the system, before development of the actual software.
- A prototype is a toy and crude implementation of a system.
- It has limited functional capabilities, low reliability, or inefficient performance as compared to the actual software.
- A prototype can be built very quickly by using several shortcuts.
- The shortcuts usually involve developing inefficient, inaccurate, or dummy functions.
- Normally the term *rapid prototyping* is used when software tools are used for prototype construction.
- For example, tools based on *fourth generation languages* (4GL) may be used to construct the prototype for the GUI parts.

Necessity of the prototyping model

- The prototyping model is advantageous to use for specific types of projects.

- It is advantageous to use the prototyping model for development of the *graphical user interface* (GUI) part of an application.
- Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer.
- The prototyping model is especially useful when the exact technical solutions are unclear to the development team.
- Often, major design decisions depend on issues such as the response time of a hardware controller, or the efficiency of a sorting algorithm, etc.
- In such circumstances, a prototype is often the best way to resolve the technical issues.
- The prototyping model is considered to be useful for the development of not only the GUI parts of a software, but also for a software project for which certain technical issues are not clear to the development team.

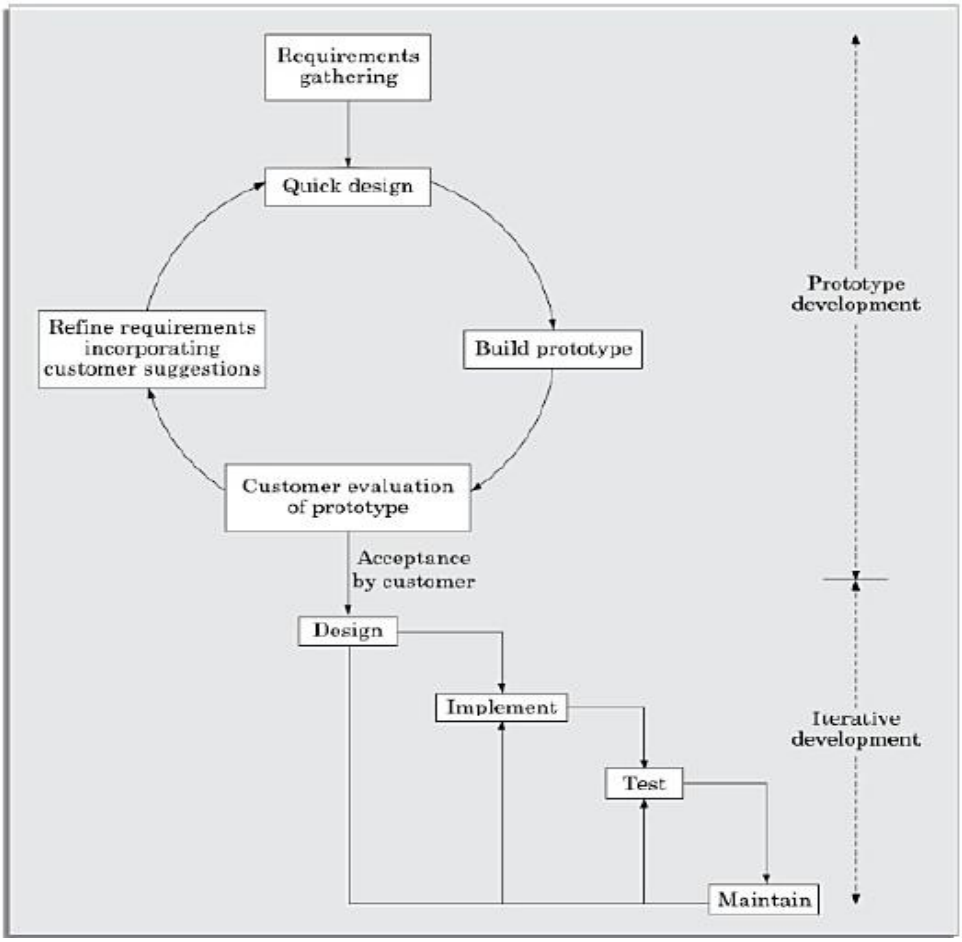


Figure 2.6: Prototyping model of software development.

Life cycle activities of prototyping model

- The prototyping model software is developed through **two** major activities—prototype construction and iterative waterfall-based software development.

Prototype development: Prototype development starts with an initial requirements gathering phase.

- A quick design is carried out and a prototype is built.
- The developed prototype is submitted to the customer for evaluation.
- Based on the customer feedback, the requirements are refined and the prototype is suitably modified.
- This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach.

- In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases.

Strengths of the prototyping model

- This model is the most appropriate for projects that suffer from technical and requirements risks. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

- The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks.
- Even when a project is susceptible to risks, the prototyping model is effective only for those projects for which the risks can be identified upfront before the development starts.
- Since the prototype is constructed only at the start of the project, the prototyping model is ineffective for risks identified later during the development cycle.

8. EVOLUTIONARY MODEL

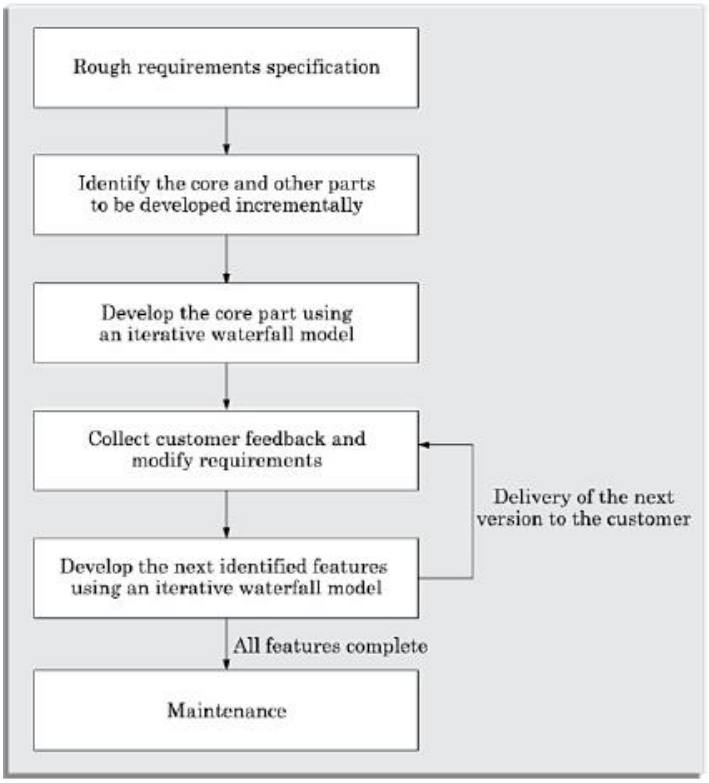
- The software is developed over a number of increments. At each increment, a concept is implemented and is deployed at the client site.
- The software is successively refined and feature-enriched until the full software is realized.
- In evolutionary model, the requirements, plan, estimates and solution evolve over the iterations rather than fully defined and frozen in a major up front specification effort before the development iterations begin.
- The evolutionary model is sometimes referred to as design a little, build a little, test a little, deploy a little model.
- After the requirements have been specified, the design, build, test and deployment activities are iterated.

Advantages:

- 1. **Effective elicitation of actual customer requirements**: The user gets a chance to experiment with a partially developed software much before the complete requirements are developed. SO the change request after delivery of the complete software gets substantially reduced.
- 2. **Easy handling change requests**: Handling change requests is easier as no long term plans are made. Reworks required are much smaller.

Disadvantages:

1. **Feature division into incremental parts can be non-trivial:** For small sized projects it is difficult to divide the required features into several parts for incrementally implemented and delivered. For larger problems, features are intertwined that expert would need considerable effort to plan the incremental deliveries.



2. **Adhoc design**: Design for only the current increment is done, the design can become adhoc without specific attention being paid to maintainability and optimality.

Applicability of evolutionary model:

- The evolutionary model is well suited to use in object oriented software development projects.

9. RAPID APPLICATION DEVELOPMENT (RAD):

- This model has the features of both prototyping and evolutionary models.
- In this model prototypes are constructed and incrementally the features are developed and delivered to the customer.
- But the prototypes are not thrown away but are enhanced and used in the software construction.
- The major goals of the RAD model are as follows:

1. To decrease the time taken and the cost incurred to develop software systems.
2. TO limit the costs of accommodating change requests.
3. TO reduce the communication gap between the customer and the developers.

Main motivation: The RAD model tries to overcome the problems of customer expectations, change requests of the customer, taking long time to have a good solution by inviting and incorporating customer feedback on successively developed and refined prototypes.

Working of RAD:

- In the RAD model, development takes place in a series of short cycles or iterations.
- At any time, the development team focuses on the present iteration only and therefore plans are made for one increment at a time. The time planned for each iteration is called a time box.

- Each iteration is planned to enhance the implemented functionality of the application by only a small amount.
- During each time box, a quick and duty prototype style software for some functionaly is developed.
- The customer evaluates the prototype and gives feedback on the specific improvements.
- The development team consists of about five to six members including a customer representative.

How does RAD facilitate accommodation of change requests:

- Features are delivered in small increments, incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

How does RAD facilitate Faster development:

- The decrease in development time and cost and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two ways:
  1. Minimal use of planning and heavy reuse of any existing code through rapid prototyping.
  2. The lack of long term an and detailed planning gives the flexibility to accommodate later requirements changes.

Applicability of RAD Model:

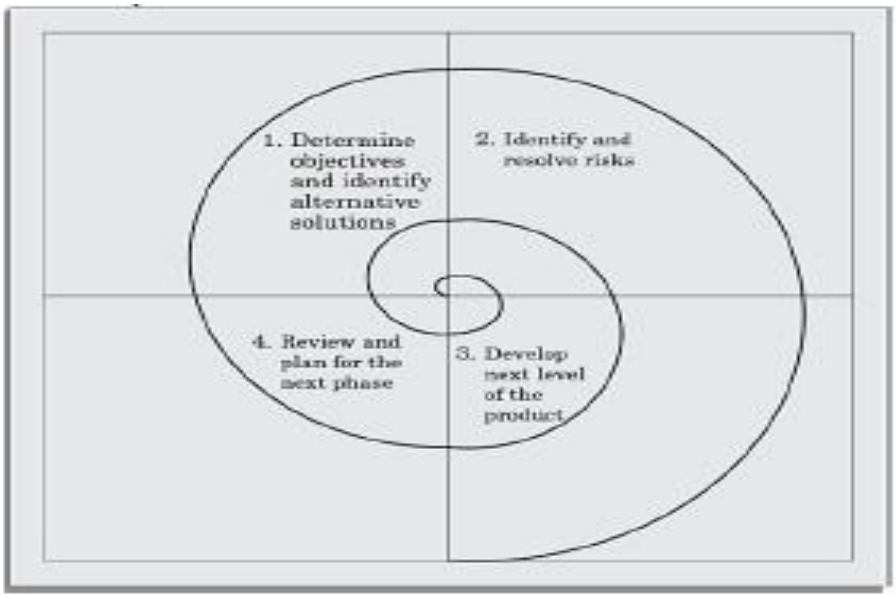
1. **Customized software:** Automated package , tailored, educational software.
2. **Non-critical software:** Developed product is usually far from being optimal in performance and reliability.
3. **Highly constrained project schedule:** RAD aims to reduce development time at the expense of good documentation, performance and reliability.
4. **Large software**: software supporting many features can incremental development and delivery.

10. SPIRAL MODEL

- The model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops.
- The exact number of loops of the spiral is not fixed and can vary from project to projects.
- Each loop of the spiral is called a phase of the software process.
- The exact number of phases through with the product is developed can be varied by the project manager depending upon the project risks.
- Over each loop one or more features of the product are elaborated and analyzed and the risks at that point of time are identified and are resolved through prototyping.

Risk handling in spiral model:

- A risk is essentially any adverse circumstance that hamper the successful completion of a software project.



- The risk can be resolved by building a prototype of the subsystem and experimenting with the exact fault.

Phases of the Spiral Model:

- Each phase in the model is split into four sectors(quadrants).
- In the first quadrant , a few features of the software are identified to be taken up for immediate development based on how crucial it is to the software development

Quadrant 1: The objectives are investigated, elaborated and analyzed. The risks involved in the phase objectives are identified. Alternative solutions are proposed.

Quadrant 2: The alternative solutions are evaluated to select the best possible solution. The solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities consist of developing and verifying the next level of the software, The identified features have been implemented and next version of the software is available.

Quadrant 4: Reviewing the results of the stages traversed so far with the customer and planning the next iteration of the spiral.

- The radius of the spiral at any point represents the cost incurred in the project so far.
- To the developers of a project the spiral model usually appears as a complex model to follow since it is risk driven and is more complicated phase structure than other models.
- For projects having many unknown risks that show up as the development proceeds, the spiral model would be the most appropriate development model to follow.

Unit II Software Project Management:- Responsibilities of a Software Project Manager – Project Planning- Project Estimation Techniques-Risk Management. **Requirements Analysis and Specification:-** Requirements Gathering and Analysis – Software Requirements Specifications (SRS):Users of SRS Document – Characteristics of a Good SRS Document – Important Categories of Customer Requirements – Functional Requirements – How to Identify the Functional Requirements? – Organisation of the SRS Document. (12L)

UNIT-2
SOFTWARE PROJECT MANAGEMENT

- The main goal of software project management is to enable a group of developers to work effectively towards the successful completion of a project.
- Project management involves use of a set of techniques and skills to steer a project to success.

1.RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER:

a) Job responsibilities for managing software projects:

- A software project manager takes the overall responsibility of steering a project to success.
- Most managers takes the responsibilities of project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, managerial report writing and presentation and interfacing with clients.
- The activities can be broadly classified into two major types
    1. project planning
    2. project monitoring and control.

1. **Project planning**: Project planning is done immediately after the feasibility study and before the starting of the requirements analysis and specification phase.

Project planning involves estimating several characteristics of a project and then planning the project activities based on these estimates made.

2. **Project monitoring and control**: This is undertaken once the development activities start. The focus of project monitoring and control activities is to ensure that the software development proceeds as per plan.

b) Skills necessary for managing software projects:

- Effective software project management calls for good qualitative judgment and decision taking capabilities.
- Also good grasp of latest software project management techniques like cost estimation, risk management and configuration management, good communication skills and the ability to get work done.
- **Three** skills that are most critical to successful project management are the following:
    1. knowledge of project management techniques.
    2. Decision taking capabilities
    3. Previous experience in managing similar projects.

2.PROJECT PLANNING:

- Once a project has been found to be feasible, software project managers undertake project planning.
- Project planning is undertaken and completed before any development activity starts.
- Project planning requires utmost care and attention as schedule delays can cause customer dissatisfaction.
- During project planning, the project manager performs the following activities:

a) **Estimation**: the following attributes are estimated.

Cost: How much is it going to cost to develop the software product?
Duration: How long is it going to take to develop the product?
Effort: How much effort would be necessary to develop the product?

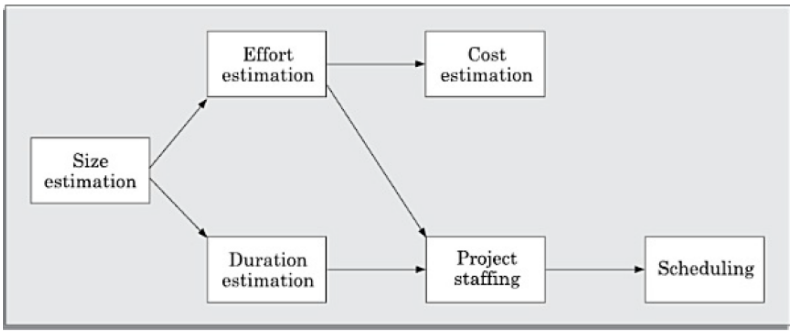
b) **Scheduling**: After all the project parameters are estimated, the schedules of manpower and other resources are developed.

c) **Staffing**: staff organisation and staffing plans are made.

d) **Risk Management**: This includes risk identification, analysis and abatement planning.

e) **Miscellaneous plans**: Plans like quality assurance plan and configuration management plan etc.

Size is the most fundamental parameter based on which all other estimations and project plans are made.
Figure shows the precedence ordering among planning activities.



1. Sliding window planning:

- Project managers undertake project planning over several stages.
- Planning a project over a number of stages protects managers from making big commitments at the start of the project.
- This technique of staggered planning is known as sliding window planning.
- In the sliding window planning technique starting with an initial plan, the project is planned more accurately over a number of stages.

2. SPMP Document of project planning

Organisation of the software project management plan (SPMP) document

1. **Introduction**
   (a) Objectives
   (b) Major Functions
   (c) Performance Issues
   (d) Management and Technical Constraints
2. **Project estimates**
   (a) Historical Data Used
   (b) Estimation Techniques Used
   (c) Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule**
   (a) Work Breakdown Structure
   (b) Task Network Representation
   (c) Gantt Chart Representation
   (d) PERT Chart Representation
4. **Project resources**
   (a) People
   (b) Hardware and Software
   (c) Special Resources
5. **Staff organisation**
   (a) Team Structure
   (b) Management Reporting
6. **Risk management plan**
   (a) Risk Analysis
   (b) Risk Identification
   (c) Risk Estimation
   (d) Risk Abatement Procedures
7. **Project tracking and control plan**
   (a) Metrics to be tracked
   (b) Tracking plan
   (c) Control plan
8. **Miscellaneous plans**
   (a) Process Tailoring
   (b) Quality Assurance Plan
   (c) Configuration Management Plan
   (d) Validation and Verification
   (e) System Testing Plan
   (f) Delivery, Installation, and Maintenance Plan

3. PROJECT ESTIMATION TECHNIQUES:

Estimation of various project parameters is an important project planning activity. The different parameters of a project that need to be estimated include—project size, effort required to complete the project, project duration, and cost. Accurate estimation of these parameters is important.

Project estimation techniques an broadly be classified into three main categories:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

1. Empirical Estimation Techniques

Empirical estimation techniques are essentially based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful.

Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalised to a large extent.

2. Heuristic Techniques

Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression. Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

- **Single variable estimation models** assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterised by an expression of the following form:

  Estimated Parameter = $c1 * e^{d1}$

In the above expression, e represents a characteristic of the software that has already been estimated (independent variable). Estimated Parameter is the dependent parameter (to be estimated). The dependent parameter to be estimated could be effort, project duration, staff size, etc., c1 and d1 are constants. The values of the constants c1 and d1 are usually determined using data collected from past projects (historical data). The COCOMO model is an example of a single variable cost estimation model.

- A **multivariable cost estimation model** assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:

  Estimated Resource = c1 * p1d1 + c2 * p2d2 + ...

where, p1, p2, ... are the basic (independent) characteristics of the software already estimated, and c1, c2, d1, d2, ... are constants.

- **Multivariable estimation models** are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters.

The independent parameters influence the dependent parameter to different extents. This is modelled by the different sets of constants c1 d1 , c2 , d2 , ... Values of these constants are usually determined from an analysis of historical data.

3. Analytical Estimation Techniques

- Analytical estimation techniques derive the required results starting with certain basic assumptions regarding a project. Unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis. Halstead's software science derives some interesting results.
- Halstead's software science is especially useful for estimating software maintenance efforts.
- In fact, it outperforms both empirical and heuristic techniques as far as estimating software maintenance efforts is concerned.

4. RISK MANAGEMENT

- Every project is susceptible to a large number of risks.
- A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway.
- It is necessary for the project manager to anticipate and identify different risks that a project is susceptible to get.
- Risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real,
- Risk management consists of three essential activities- risk identification, risk assessment and risk mitigation.

a) Risk identification:

- The project manager needs to anticipate the risks in a project as early as possible.
- When risk is identified, effective risk management plans are made and the possible impacts of the risks is minimized.
- To identify the important risks it is necessary to categorize risks from each class are relevant to the project.
- Three main categories of risks include project risks, technical risks and business risks.

1. **Project risks**: includes budgetary, schedule, personnel, resource and customer related problems. Schedule slippage, software is intangible, difficult to monitor and control a software project. The invisibility of the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

2. **Technical risks**: includes potential design, implementation, interfacing, testing and maintenance problems. Ambiguous specification, incomplete specification, changing specification, technical uncertainty, development teams insufficient knowledge and technical obsolescence.

3. **Business risks**: Includes risk of building an excellent product that no one wants, losing budgetary commitments etc.

b) Risk Assessment:

- The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:
    - The likelihood of a risk becoming real
    - The consequence of the problems associated with that risk.
- The priority of each risk can be computed as follows p=r*s
- Where p is the priority with which the risk must be handled, r is the probability of the risk becoming real and s is the severity of damage caused due to the risk becoming real.

c) Risk Mitigation:

- After all the identified risks of a project have been assessed plans are made to contain the most damaging and the most likely risks first.
- Different types of risks require different containment procedures.
- There are three main strategies for risk containment.

1. **Avoid the risk**: Risks can be avoided in several ways. Risks arise due to project constrains and can be avoided by suitably modifying the constraints.

The different categories of constraints give rise to risks are

Process related risk: arise due to aggressive work schedule, budget and resource utilization.

Product related risk: arise due to commitment to challenging product features, quality, reliability etc.

Technology related risk: arise due to commitment to use certain technology.4

2. **Transfer the risk**: involves getting the risky components developed by a third party, buying insurance cover.

3. **Risk reduction**: involves planning ways to contain the damage due to a risk. Most important risk reduction techniques is to build a prototype.

Risk leverage is the difference in risk procedure divided by the cost of reducing the risk.

$$risk\,leverage = \frac{risk\,exposure\,before\,reduction - risk\,exposure\,after\,reduction}{cost\,of\,reduction}$$

REQUIREMENTS ANALYSIS AND SPECIFICATION

5. Requirements gathering and Analysis:

- The requirements have to be gathered by the analyst from several sources in bits and pieces.
- Conceptually divide the requirements gathering and analysis activity into two separate tasks:
  - ✓ Requirements gathering
  - ✓ Requirements analysis

1. Requirements gathering:

- ➢ Also known as requirements elicitation.
- ➢ The primary objective of the requirements gathering task is to collect the requirements from the stakeholders.
- ➢ A stakeholder is a source of the requirements and is usually a person, or a group of persons who either directly or indirectly are concerned with the software.
- ➢ Good analysts share their experience and expertise with the customer and give his suggestions to define certain functionalities more comprehensively make the functionalities more general and more complete.
- ➢ The important ways to gather requirements:

1. Studying existing documentation:

- ✓ the analyst study all the available documents regarding the system.
- ✓ Customers provide Statement of Purpose(SoP) document to the developers.
- ✓ Documents discuss issues in which the software is required, basic purpose, the stakeholders.

2. Interview:

- ✓ Analyst have to identify the different categories of users and then determine the requirements of each.
- ✓ Eg. In library automation software, library members, librarians and accountants.
- ✓ To systemize the requirements gathering, Delphi technique can be followed.

3. Task Analysis:

- ➢ Users have a black box view of a software and consider the software that provides a set of services. A service supported by a software is called a task.
- ➢ Analyst tries to identify and understand the different steps to realize the required functionality in consultation with the user.
- ➢ Task analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.

Scenario analysis:

- ➢ A task have many scenarios of operation.
- ➢ For different types of scenarios of a task, the behavior of the software may be different.
- ➢ Eg. Book is issued successfully to the member and the book issue slip is printed.

Form Analysis:

- It is an important and effective requirements gathering activity that is undertaken by the analyst.
- In form analysis the existing forms and the formats of the notifications produced are analyzed to determine the data input to the system and the data that are output from the system.

2. Requirements Analysis:

- After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in gathered requirements.
- The main purpose of the requirements analysis activity is to analyze the gathered requirements to remove all ambiguities, incompleteness and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.
- The basic questions to the project should be clearly understood by the analyst before carrying out analysis.
    - What is the problem?
    - Why is it important to solve the problem?
    - What exactly are the data input and data output of the system?
    - What are the possible procedures to solve the problem?
    - What are the likely complexities to solve the problem?
    - Is there any external software or hardware?
- The analyst proceeds to identify and resolve the various problems that he detects in the gathered requirements.
- The analyst needs to identify and resolve three main types of problems in the requirements:
    - ✓ Anomaly
    - ✓ Inconsistency
    - ✓ Incompleteness
- **Anomaly**: is an ambiguity in a requirement. Any anomaly in any of the requirements lead to the development of an incorrect system, since an anomalous requirement can be interpreted in the several ways during development.
- **Inconsistency**: Two requirements are said to be inconsistent if one of the requirement contradicts the other. Eg. The furnace should be switched off when the temperature of the furnace rises above 500 degree Celsius.
- **Incompleteness**: An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features should be felt by the customer much later, while using the software.

6. SOFTWARE REQUIREMENTS SPECIFICATION(SRS)

After removing all incompleteness, inconsistencies and anomalies from the specification, analyst organize the requirements in the form of an SRS document.
SRS document is probably the most important document and is the toughest to write.

1. USERS OF SRS DOCUMENT:

Some of the important categories of users of SRS document and their needs for use are as follows:

a) Users, Customers and Marketing Personnel: Stakeholders refer to SRS document to ensure that the system in the document will meet their needs.
For generic products, marketing personnel need to understand the requirements.
b) Software developers: Software developers refer to SRS document to make sure that they are developing exactly what is required by the customer.
c)Test Engineers: Test engineers use the SRS document to understand the functionalities and based on this write the test cases to validate its working. The required functionality, input and output data should be identified precisely.
d)User documentation writers: need to read the SRS document to ensure that they understand the features of the product well enough to be able to write the users manual.
e) Project managers: refer to SRS document to ensure that they can estimate the cost of the project easily and it contains all the information required to plan the project.
f) Maintenance engineers: The SRS document helps the maintenance engineers to understand the functionalities supported by the system.

SRS document can be used as a legal document to settle disputes between the customers and developers in a court of law.

2. CHARACTERISTICS OF A GOOD SRS DOCUMENT:

The skill of writing a good SRS document comes from the experience gained from writing SRS documents for many projects.
Some of the identified desirable qualities of an SRS document are the following:
a) **Concise**: SRS document should be concise and at the same time unambiguous, consistent and complete. Verbose and irrelevant descriptions reduce readability and also increase the possibilities of errors in the document.
b) Implementation-independent: SRS should be free of design and implementation decisions that reflect actual requirements.
SRS document should specify the externally visible behavior of the system and not implementation issues.
The SRS document should describe the system to be developed as a black box and should specify only the externally visible behavior of the system.
c) **Traceable**: IT should be possible to trace a specific requirement to the design elements that implement it and vice versa.
Traceability is also important to verify the results of a phase with respect to the previous phase and to analyze the impact of changing a requirement on the design elements and the code.
d) **Modifiable**: Customers frequently change the requirements during the software development due to variety of reasons.
SRS document undergoes several revisions during software development.
SRS document should be easily modifiable.
e) **Identification of response to undesired events**: SRS document discuss the system responses to various undesired events and exceptional conditions that may arise
f) **Verifiable**: All requirements as documents in the SRS document should be verifiable.
To design test cases based on the description of the functionality as to whether or not requirements have been met in an implementation.
Any feature of the required system that is not verifiable should be listed separately in the goals of the implementation section of the SRS document.

UNIT III

SOFTWARE DESIGN

Overview of the Design Process: Outcome of the Design Process – Classification of Design Activities. – How to Characterize a good Software Design? **Function-Oriented Software Design:-** Overview of SA/SD Methodology – Structured Analysis – Developing the DFD Model of a System: Context Diagram – Structured Design – Detailed Design.

SOFTWARE DESIGN:

The activities carried out during the design phase (called as design process) transform the SRS document into the design document.

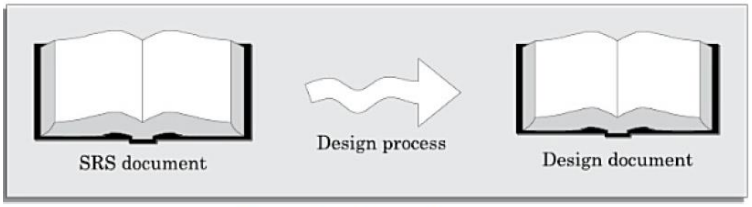


Figure 5.1: The design process.

OVERVIEW OF THE DESIGN PROCESS

The design process essentially transforms the SRS document into a design document.

1. OUTCOME OF THE DESIGN PROCESS

The following items are designed and documented during the design phase.

Different modules required: The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.

Control relationships among modules: A control relationship between two modules essentially arises due to function calls across the two modules. The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules: The interfaces between two modules identify the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

Data structures of the individual modules: Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

Algorithms required to implement the individual modules: Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities. Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps. The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

2. CLASSIFICATION OF DESIGN ACTIVITIES

- A good software design is seldom realized by using a single step procedure, rather it requires iterating over a series of steps called the design activities. Depending on the order in which various design activities are performed, we can broadly classify them into **two** important stages.
  - Preliminary (or high-level) design, and
  - Detailed design.

The meaning and scope of these two stages can vary considerably from one design methodology to another. However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level design as follows:

- Through high-level design, a problem is decomposed into a set of **modules**. The **control** relationships among the modules are identified, and also the **interfaces** among various modules are identified.
- The outcome of high-level design is called the **program structure** or the software architecture. High-level design is a crucial step in the overall design of a software.

- When the high-level design is complete, the problem should have been decomposed into many small functionally independent **modules** that are cohesive, have low coupling among themselves, and are arranged in a hierarchy.
- Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the **structure chart**. Another popular design representation techniques called **UML** that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.
- Once the high-level design is complete, **detailed design** is undertaken.
- During detailed design each module is examined carefully to design its **data structures** and the **algorithms**.
- The outcome of the detailed design stage is usually documented in the form of a module specification (**MSPEC**) document.
- After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding.

3. HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

- Coming up with an accurate characterization of a good software design that would hold across diverse problem domains is certainly not easy.
- In fact, the definition of a "good" software design can vary depending on the exact application being designed.
- For example, "memory size used up by a program" may be an important issue to Characterize a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations.
- For embedded applications, factors such as design comprehensibility may take a back seat while judging the goodness of design. Thus for embedded applications, one may sacrifice design comprehensibility to achieve code compactness.
- Most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess.

These characteristics are listed below:

Correctness: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency: A good design solution should adequately address resource, time, and cost optimization issues.

Maintainability: A good design should be easy to change.

Understandability of a Design: A Major Concern

- ✓ While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one.
- ✓ Obviously all incorrect designs have to be discarded first.
- ✓ Out of the correct design solutions, how can we identify the best one?
- ✓ Given that we are choosing from only correct design solutions, understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

An understandable design is modular and layered

- To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess.
- A design solution should have the following characteristics to be easily understandable:
  - ✓ It should assign consistent and meaningful names to various design components.
  - ✓ It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
  - ✓ A design solution should be modular and layered to be understandable.

Modularity

- A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution.
- A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.

- Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle.
- If different modules have either no interactions or little interactions with each other, then each module can be understood separately.
- This reduces the perceived complexity of the design solution greatly.
- To understand why this is so, remember that it may be very difficult to break a bunch of sticks which have been tied together, but very easy to break the sticks individually.
- A design solution is said to be highly modular, if the different modules in the solution have high cohesion and their inter-module couplings are low.
- A software design with high cohesion and low coupling among modules is the effective problem decomposition bringing down the perceived problem complexity.

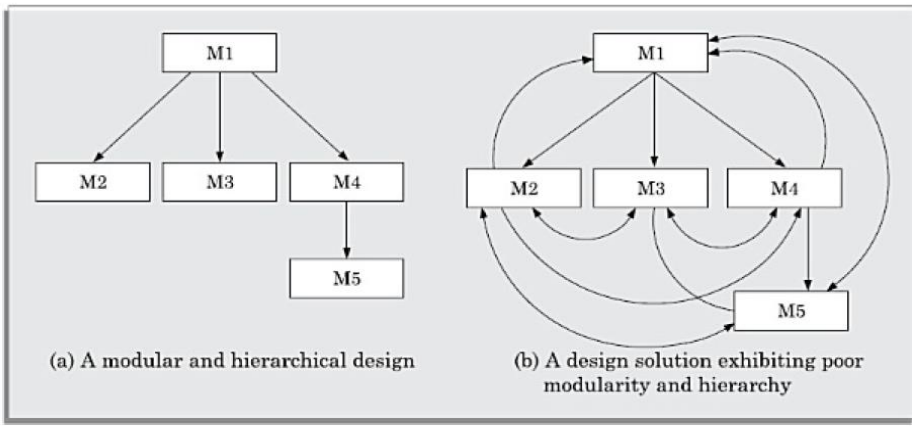


Figure 5.2: Two design solutions to the same problem.

- Based on this classification, we would be able to easily judge the cohesion and coupling existing in a design solution.
- From a knowledge of the cohesion and coupling in a design, the modularity of the design solution can be achieved.

Layered design

- A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering.
- In a layered design solution, the modules are arranged in a hierarchy of layers.
- A module can only invoke functions of the modules in the layer immediately below it.
- The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.

- A layered design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of (about how to call) the higher layer modules.

4. FUNCTION-ORIENTED SOFTWARE DESIGN:-

The term top-down decomposition is often used to denote the successive decomposition of a set of high-level functions into more detailed functions.

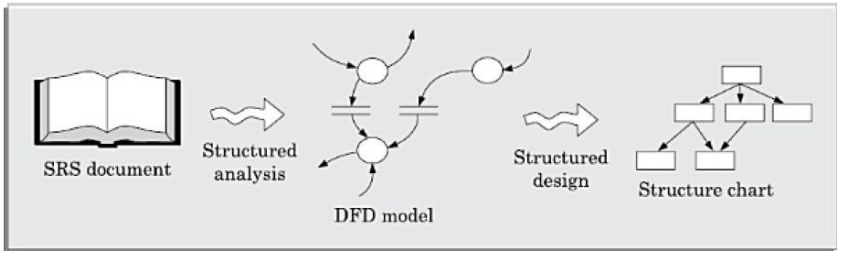
OVERVIEW OF SA/SD METHODOLOGY

SA/SD methodology involves carrying out **two** distinct activities:

- Structured analysis (SA)
- Structured design (SD)

During **structured analysis**, the SRS document is transformed into a data flow diagram (DFD) model.

During **structured design**, the DFD model is transformed into a structure chart.



The structured analysis activity transforms the SRS document into a graphic model called the **DFD model**. During structured analysis, **functional decomposition** of the system is achieved. It is important to understand that the purpose of structured analysis is to capture the **detailed structure** of the system as perceived by the user, whereas the purpose of structured design is to define the structure of the solution that is suitable for **implementation** in some programming language.

5. STRUCTURED ANALYSIS

The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using data flow diagrams (DFDs).

A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.

Primitive symbols used for constructing DFDs

There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meaning of these symbols are explained as follows:

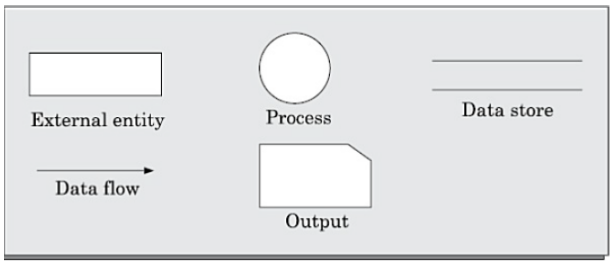


Figure 6.2: Symbols used for designing DFDs.

Function symbol: A function is represented using a circle.

External entity symbol: An external entity such as a librarian, a library member, etc. is represented by a rectangle.

A **data flow symb**ol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow.

Data store symbol: A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk

Output symbol: The output symbol i s as shown in Figure 6.2. The output symbol is used when a hard copy is produced.

A **data dictionary** lists the purpose of all data items and the definition of all composite data items in terms of their component data items.

6. DEVELOPING THE DFD MODEL OF A SYSTEM

A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.

The DFD model of a problem consists of many of DFDs and a single data dictionary.

Example. Trading-house Automation System (TAS)) A trading house wants to develop a computerized system that would automate various bookkeeping activities associated with its business.

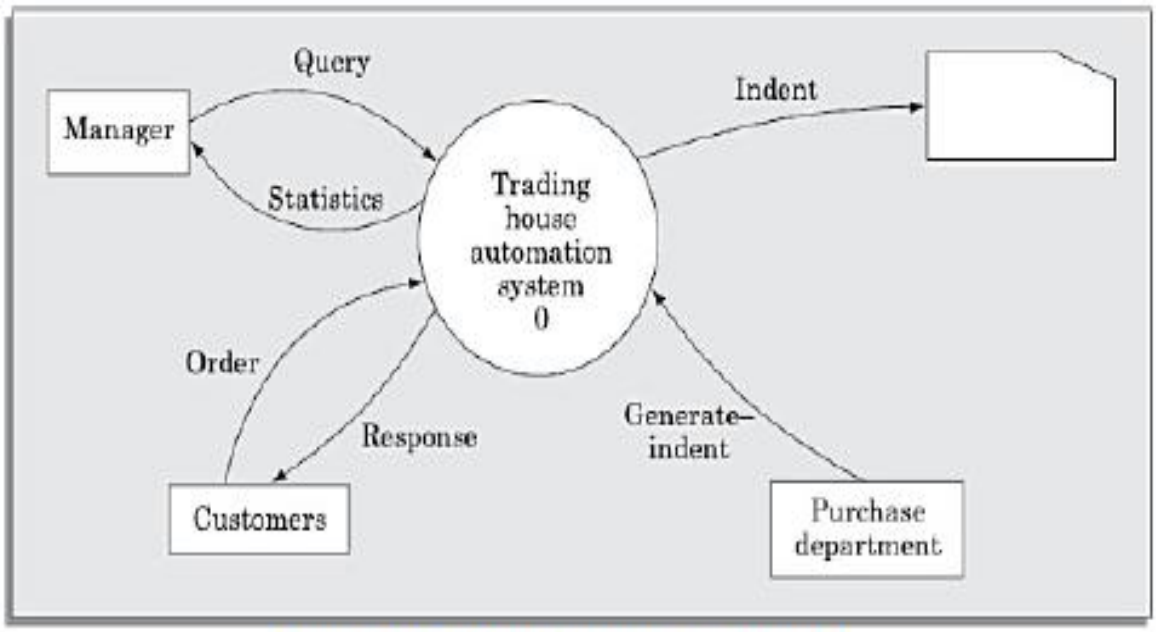


Figure 6.13: Context diagram for Example 6.4.

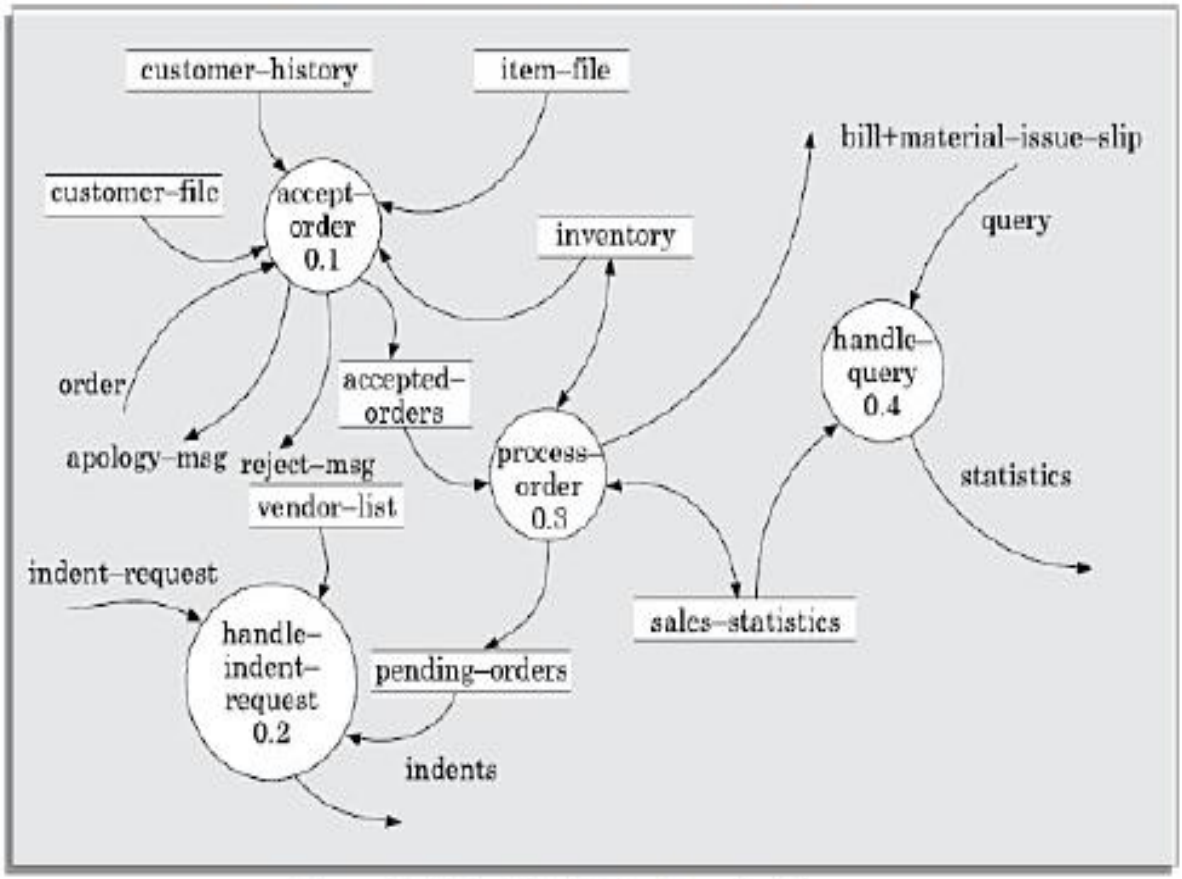


Figure 6.14: Level 1 DFD for Example 6.4

Data dictionary for the DFD model of TAS

response: [bill + material-issue-slip, reject-msg,apology-msg]
query: period /* query from manager regarding sales statistics*/
period: [date+date,month,year,day]
date: year + month + day year: integer
month: integer day: integer customer-id: integer
order: customer-id + {items + quantity}* + order#

a) Context Diagram

The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun).

The context diagram establishes the context in which the system operates; that is, who are the users, what data do they input to the system, and what data they received by the system.

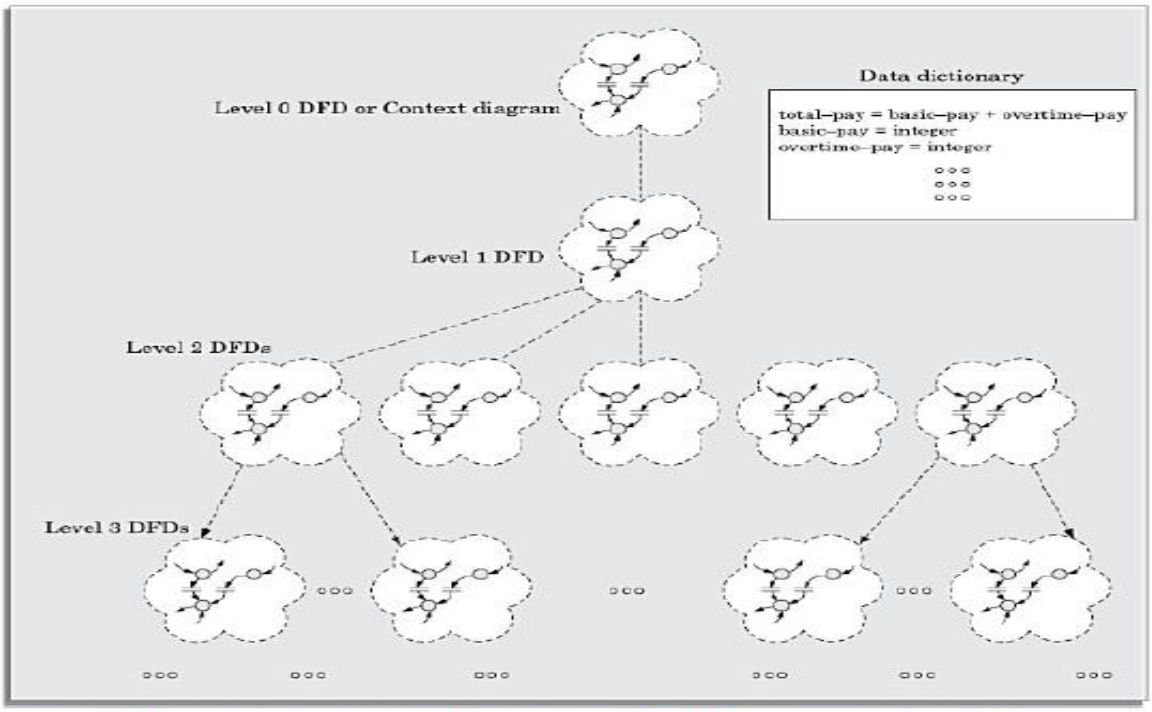


Figure 6.4: DFD model of a system consists of a hierarchy of DFDs and a single data dictionary.

1. **Construction of context diagram:** Examine the SRS document to determine:

• Different high-level functions that the system needs to perform.

• Data input to every high-level function.

• Data output from every high-level function.

• Interactions (data flow) among the identified high-level functions.

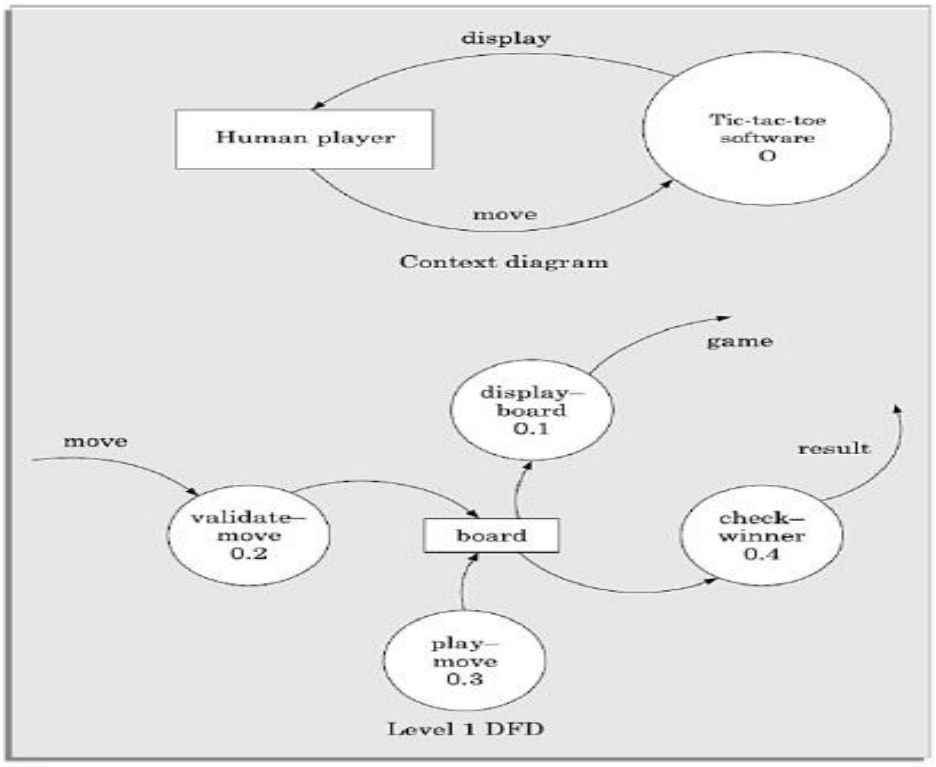


Figure 6.9: Context diagram and level 1 DFDs for Example 6.2.

Construction of level 1 diagram: Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

Construction of lower-level diagrams: Decompose each high-level function into its constituent subfunctions through the following set of activities:

- Identify the different subfunctions of the high-level function.
- Identify the data input to each of these subfunctions.
- Identify the data output from each of these subfunctions.
- Identify the interactions (data flow) among these subfunctions.

7. STRUCTURED DESIGN

- The aim of structured design is to transform the results of the structured analysis (that i s, the DFD model) into a structure chart.

- A structure chart represents the software architecture. The various modules making other modules), and the parameters that are passed among the different modules.
- The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects.

The basic building blocks using which structure charts are designed are as following:

Rectangular boxes: A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.

Module invocation arrows: An arrow connecting two modules implies that during program execution control is passed from one module to the other in the direction of the connecting arrow. However, just by looking at the structure chart, we cannot say whether a modules calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.

Data flow arrows: These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.

Library modules: A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called modules. Usually, when a module is invoked by many other modules, it is made into a library module.

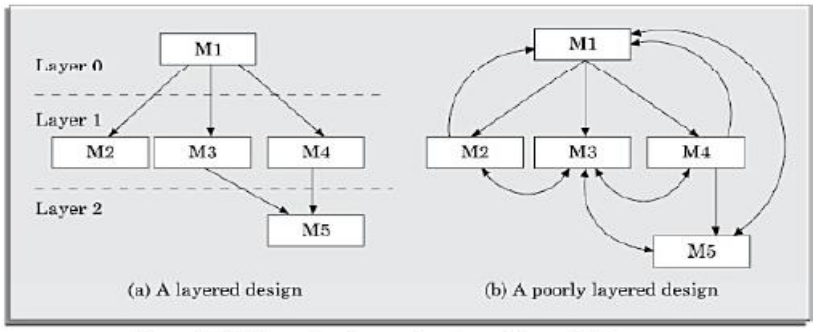


Figure 6.18: Examples of properly and poorly layered designs.

Selection: The diamond symbol represents the fact that one module of several modules connected with the diamond symbol i s invoked depending on the outcome of the condition attached with the diamond symbol.

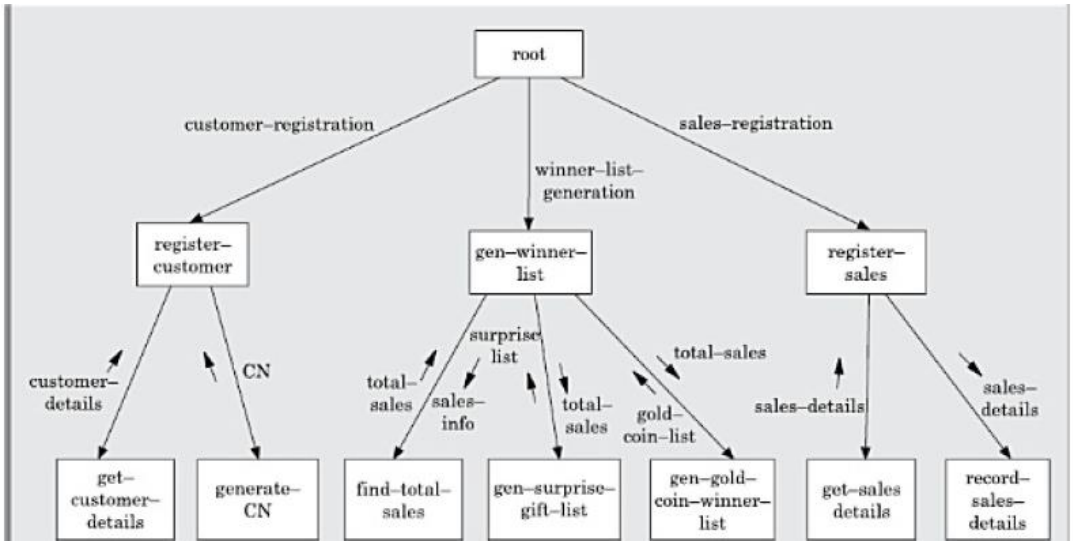
Repetition: A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.

Flow chart *versus* structure chart

Flow chart is a convenient technique to represent the flow of control in a program.

A structure chart differs from a flow chart in three principal ways: It is usually difficult to identify the different modules of a program from its flow chart representation. Data interchange among different modules is not represented in a flow chart. Sequential ordering of tasks that i s inherent to a flow chart is suppressed in a structure chart.

Structure chart of a super market.



8. DETAILED DESIGN

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart.
- These are usually described in the form of module specifications (MSPEC).
- MSPEC is usually written using structured English.
- The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out.
- To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

UNIT IV

User Interface Design

Characteristics of a good User Interface - Basic Concepts(508) – Types of User Interfaces – Fundamentals of Components based GUI Development: Window System. **Coding and Testing:-** Coding – Software Documentation – Testing: Basic Concepts and Terminologies – Testing Activities. – Unit Testing – Black-box Testing: Equivalence Class Partitioning – Boundary Value Analysis. – White-box Testing.

User Interface Design

The user interface part of a software product is responsible for all interactions with the end-user.

1. CHARACTERISTICS OF A GOOD USER INTERFACE

1) Speed of learning: A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memorize commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following **three** issues are crucial to enhance the speed of learning:

a) **U s e of metaphors and intuitive command names:** Speed of learning an interface is greatly facilitated if these are based on some day to-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called **metaphors**. If the user interface of a text editor uses concepts similar to the **tools** used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it.

b) **Consistency:** Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.

c) **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with. This can be achieved if the interfaces of different applications are developed using some standard user interface components.

2. **Speed of use:** Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is sometimes referred to as productivity support of the interface. It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal. This can be achieved through careful design of the interface. For example, an interface that requires users to type in lengthy commands or involves mouse movements to different areas of the screen that are wide apart for issuing commands can slow

down the operating speed of users. The most frequently used commands should have the smallest length or be available at the top of a menu to minimize the mouse movements necessary to issue commands.

3. Speed of recall: Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

4. Error prevention: A good user interface should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by average users while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users. Consistency of names, issue procedures, and behavior of similar commands and the simplicity of the command issue procedures minimize error possibilities. Also, the interface should prevent the user from entering wrong values.

5. Aesthetic and attractive: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

6. Consistency: The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.

7. Feedback: A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic. If required, the user should be periodically informed about the progress made in processing his command.

8. Support for multiple skill levels: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. When someone uses an application for the first time, his primary concern is speed of learning. Thus, the skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.

9. Error recovery (undo facility): While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors they commit while using software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

10. User guidance and on-line help: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

2. BASIC CONCEPTS

A) User Guidance and On-line Help

Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc.

a. On-line help system: Users expect the on-line help messages to be tailored to the context in which they invoke the "help system". Therefore, a good online help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. Also, the help messages should be tailored to the user's experience level. Further, a good on-line help system should take advantage of any graphics and animation characteristics of the screen and should not just be a copy of the user's manual.

b. Guidance messages: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc. A good guidance system should have different levels of sophistication for different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface. Also, users should have an option to turn off the detailed messages.

c. Error messages: Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc. Users do not like error messages that are either ambiguous or too general such as "invalid input or system error". Error messages should be polite. Error messages should not have associated noise which might embarrass the user. The message should suggest how a given error can be rectified. If appropriate, the user should be given the option of invoking the on-line help system to find out more about the error situation.

B) Mode-based versus Modeless Interface

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is, i.e., the mode at any instant is determined by the sequence of commands already issued by the user.

A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

C) Graphical User Interface (GUI) versus Text-based User Interface

In a GUI multiple windows with different information can simultaneously be displayed on the user screen. This is perhaps one of the biggest advantages of GUI over text- based interfaces since the user has the flexibility to simultaneously interact with several related items at any time and can have access to different system information displayed in different windows. Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation such as dragging an icon representing a file to a trash for deleting is intuitively very appealing and the user can instantly remember it.

A GUI usually supports command selection using an attractive and user-friendly menu selection system. In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy of command issue procedure. On the flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse.

On the other hand, a text-based user interface can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals. However, display terminals with graphics capability with bitmapped high-resolution displays and significant amount of local processing power have become affordable and over the years have replaced text-based terminals on all desktops. Therefore, the emphasis of this chapter is on GUI design rather than text-based user interface design.

3. TYPES OF USER INTERFACES

User interfaces can be classified into the following three categories:

- Command language-based interfaces
- Menu-based interfaces
- Direct manipulation interfaces

A) Command Language-based Interface

A command language-based interface, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type them appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. Thus, a command language-based interface can be made concise requiring minimal typing by the user. Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands.

Among the three categories of interfaces, the command language interface allows for most efficient command issue procedure requiring minimal typing. Further, a command language-based interface can be implemented even on cheap alphanumeric terminals. Also, a command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because compiler writing techniques are well developed. One can systematically develop a command language interface by using the standard compiler writing tools Lex and Yacc.

D**rawbacks**. Usually, command language-based interfaces are difficult to learn and require the User to memorize the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them. Further, in a command language-based interface, all interactions with the system is through a key-board and cannot take advantage of effective interaction devices such as a mouse. Obviously, for casual and inexperienced users, command language-based interfaces are not suitable.

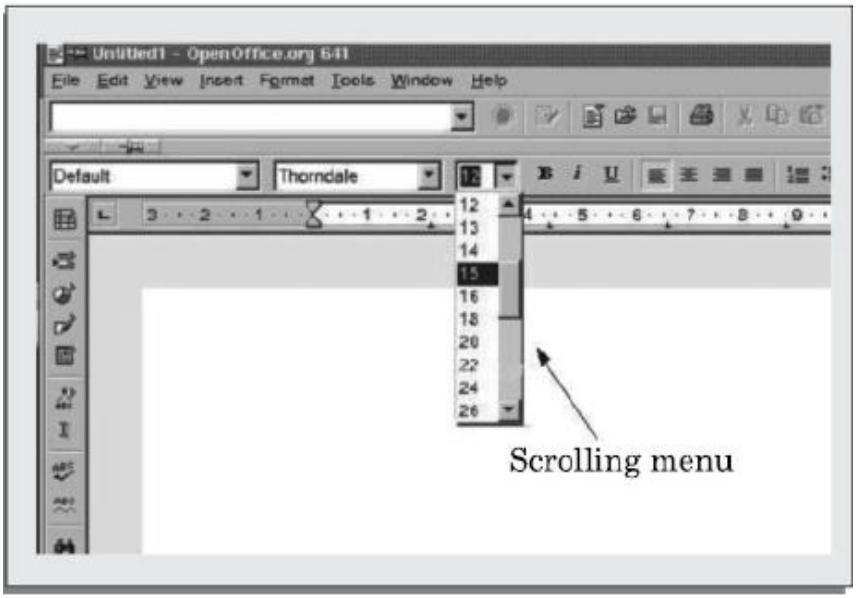
Issues in designing a command language-based interface

Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimize the total typing required. The designer should try to develop meaningful mnemonics and yet be concise to minimize the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.

B) Menu-based Interface

A menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Humans are much better in recognizing something than recollecting it. Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.

Also, if the number of choices is large, it is difficult to design a menu-based interface. A moderate-sized software might need hundreds or thousands of different menu choices. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms. Some of the techniques available to structure a large number of menu items:



Scrolling menu: Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs. This is important since the user cannot see all the commands at any one time.

A scrolling menu is frequently used is font size selection in a document processor (see Figure 9.1). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up or down to find the size he is looking for. However, if the commands do not have any definite ordering relation, then the user would have to in the worst case, scroll through all the commands to find the exact command he is looking for, making this organisation inefficient.

Walking menu: Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu. An example of a walking menu is shown in Figure 9.2. A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices since each adjacently displayed menu does take up screen space and the total screen area is after all limited.

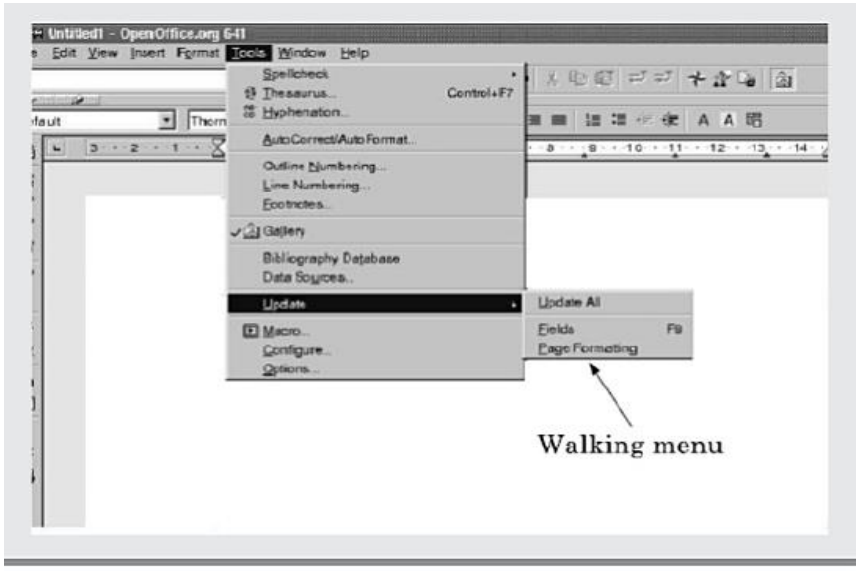


Figure 9.2: Example of walking menu.

Hierarchical menu: This type of menu is suitable for small screens with limited display area such as that in mobile phones. In a hierarchical menu, the menu items are organized in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus in this case, one can consider the menu and its various submenu to form a hierarchical tree-like structure.

Walking menu can be considered to be a form of hierarchical menu which is practicable when the tree is shallow. Hierarchical menu can be used to manage large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree. This probably is the main reason why this type of interface is very rarely used.

C) Direct Manipulation Interfaces

Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons or objects). For this reason, direct manipulation interfaces are sometimes called as iconic interfaces. In this type of interface, the user issues commands by performing actions on the visual representations of the objects, e.g., pull an icon representing a file into an icon

representing a trash box, for deleting the file. Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language independent. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files —which could be very easily done by issuing a command like delete *.*.

4. FUNDAMENTALS OF COMPONENT-BASED GUI DEVELOPMENT

Graphical user interfaces became popular in the 1980s. The main reason why there were very few GUI-based applications prior to the eighties is that graphics terminals were too expensive. For example, the price of a graphics terminal those days was much more than what a high-end personal computer costs these days. Also, the graphics terminals were of storage tube type and lacked raster capability.

One of the first computers to support GUI-based applications was the Apple Macintosh computer. In fact, the popularity of the Apple Macintosh computer in the early eighties is directly attributable to its GUI. In those early days of GUI design, the user interface programmer typically started his interface development from the scratch. He would starting from simple pixel display routines, write programs to draw lines, circles, text, etc. He would then develop his own routines to display menu items, make menu choices, etc. The current user interface style has undergone a sea change compared to the early style.

The current style of user interface development is component-based. It recognizes that every user interface can easily be built from a handfull of predefined components such as menus, dialog boxes, forms, etc. Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the window system. The window system lets the application programmer create and manipulate windows without having to write the basic windowing functions.

1. Window System

Most modern graphical user interfaces are developed using some window system. A window system can generate displays through a set of windows. Since a window is the basic entity in such a graphical user interface, we need to first discuss what exactly a window is.

a. Window: A window is a rectangular area on the screen. A window can be considered to be a virtual screen, in the sense that it provides an interface to the user for carrying out independent

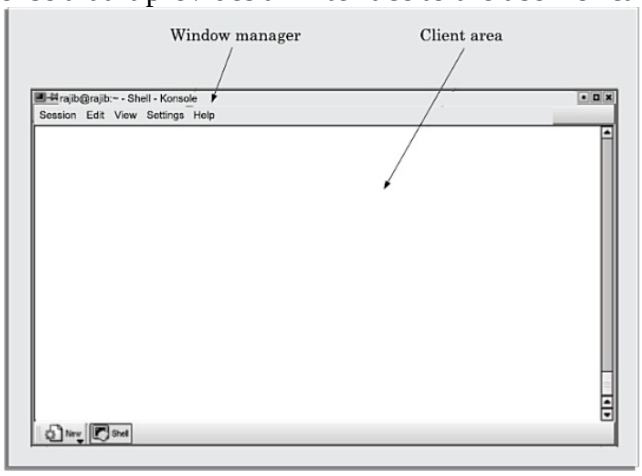


Figure: Window with client and user areas marked.

activities, e.g., one window can be used for editing a program and another for drawing pictures. A window can be divided into two parts—client part, and non-client part. The client area makes up the whole of the window, except for the borders and scroll bars. The client area is the area available to a client application for display. The non-client-part of the window determines the look and feel of the window. The look and feel defines a basic behaviour for all windows, such as creating, moving, resizing, iconifying of the windows. The window manager is responsible for managing and maintaining the non-client area of a window.

b. Window management system (WMS)

A graphical user interface typically consists of a large number of windows. Therefore, it is necessary to have some systematic way to manage these windows. Most graphical user interface development environments do this through a window management system (WMS). A window management system is primarily a resource manager. It keeps track of the screen area resource and allocates it to the different windows that seek to use the screen. From a broader perspective, a WMS can be considered as a user interface management system (UIMS) —which not only does resource management, but also provides the basic behaviour to the windows and provides several utility routines to the application programmer for user interface development.

A WMS simplifies the task of a GUI designer to a great extent by providing the basic behaviour to the various windows such as move, resize, iconify, etc. as soon as they are created and by providing the basic routines to manipulate the windows from the application program such as creating, destroying, changing different attributes of the windows, and drawing text, lines, etc.

A WMS consists of two parts (see Figure 9.4):

- a window manager, and
- a window system.

c. Window manager and window system:

The window manager is built on the top of the window system in the sense that it makes use of various services provided by the window system. The window manager and not the window system determines how the windows look and behave. In fact, several kinds of window managers can be developed based on the same window system.

The window manager can be considered as a special kind of client that makes use of the services (function calls) supported by the window system. The application programmer can also directly invoke the services of the window system to develop the user interface. The relationship between the window manager, window system, and the application program is shown in Figure 9.4.

This figure shows that the end-user can either interact with the application itself or with the window manager (resize, move, etc.) and both the application and the window manger invoke services of the window manager. Window manager is the component of WMS with which the end user interacts to do various window-related operations such as window repositioning, window resizing, iconification, etc.

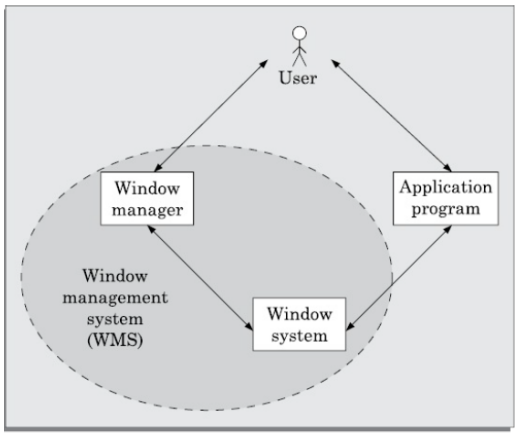


Figure 9.4: Window management system.

A widget is the short form of a window object. The data of an window object are the geometric attributes (such as size, location etc.) and other attributes such as its background and foreground colour, etc. The operations that are defined on these data include, resize, move, draw, etc.

Widgets are the standard user interface components. A user interface is usually made up by integrating several widgets. A few important types of widgets normally provided with a user interface development system are described.

d. Component-based development

A development style based on widgets is called component-based (or widget-based) GUI development style. There are several important advantages of using a widget-based design style. One of the most important reasons to use widgets as building blocks is because they help users learn an interface fast. In this style of development, the user interfaces for different applications are built from the same basic components. Therefore, the user can extend his knowledge of the behavior of the standard components from one application to the other.

e. Visual programming

Visual programming is the drag and drop style of program development. In this style of user interface development, a number of visual objects (icons) representing the GUI components are provided by the programming environment. The application programmer can easily develop the user interface by dragging the required component types (e.g., menu, forms, etc.) from the displayed icons and placing them wherever required.

Visual programming can be considered as program development through manipulation of several visual objects. Reuse of program components in the form of visual objects is an important aspect of this style of programming. Though popular for user interface development, this style of programming can be used for other applications such as Computer-Aided Design application (e.g., factory design), simulation, etc. User interface development using a visual programming language greatly reduces the effort required to develop the interface.

Examples of popular visual programming languages are Visual Basic, Visual C++, etc. Visual C++ provides tools for building programs with windowbased user interfaces for Microsoft Windows environments. In visual C++ you usually design menu bars, icons, and dialog boxes, etc. before adding them to your program. These objects are called as resources.

You can design shape, location, type, and size of the dialog boxes before writing any C++ code for the application.

f. Types of Widgets

Different interface programming packages support different widget sets.

a. Label widget: This is probably one of the simplest widgets. A label widget does nothing except to display a label, i.e., it does not have any other interaction capabilities and is not sensitive to mouse clicks. A label widget is often used as a part of other widgets.

b. Container widget: These widgets do not stand by themselves, but exist merely to contain other widgets. Other widgets are created as children of the container widget. When the container widget is moved or resized, its children widget also get moved or resized. A container widget has no callback routines associated with it.

c. Pop-up menu: These are transient and task specific. A pop-up menu
appears upon pressing the mouse button, irrespective of the mouse position.

d. Pull-down menu : These are more permanent and general. You have to move the cursor to a specific location and pull down this type of menu.

e. Dialog boxes: We often need to select multiple elements from a selection list. A dialog box remains visible until explicitly dismissed by the user. A dialog box can include areas for entering text as well as values. If an apply command is supported in a dialog box, the newly entered values can be tried without dismissing the box. Though most dialog boxes ask you to enter some information, there are some dialog boxes which are merely informative, alerting you to a problem with your system or an error you have made. Generally, these boxes ask you to read the information presented and then click OK to dismiss the box.

f. Push button: A push button contains key words or pictures that describe the action that is triggered when you activate the button. Usually, the action related to a push button occurs immediately when you click a push button unless it contains an ellipsis (. . .). A push button with an ellipsis generally indicates that another dialog box will appear.

g. Radio buttons: A set of radio buttons are used when only one option has to be selected out of many options. A radio button is a hollow circle followed by text describing the option it stands for. When a radio button is selected, it appears filled and the previously selected radio button from the group is unselected. Only one radio button from a group can be selected at any time. This operation is similar to that of the band selection buttons that were available in old radios.

h. Combo boxes: A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from. Normally a combo box is used to display either one-of-many choices when space is limited, the number of choices is large, or when the menu items are computed at run-time.

5. CODING

- Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.
- After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken.
- The input to the coding phase is the design document produced at the end of the design phase.
- Recollect that the design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design.

- The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified.
- During the coding phase, different modules identified in the design document are coded according to their respective module specifications.
- The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code..
- Good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standard.
- The main advantages of adhering to a standard style of coding are the following:
  - A coding standard gives a uniform appearance to the codes written by different engineers.
  - It facilitates code understanding and code reuse.
  - It promotes good programming practices.

1. Coding Standards and Guidelines

- Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop.
- To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines that are commonly adopted by many software development organisations, rather than trying to provide an exhaustive list.

Representative coding standards:

a. Rules for limiting the use of globals: These rules list what types of data can be declared global and what cannot, with a view to limit the data that needs to be defined with global scope.

b. Standard headers for different modules: The header of different modules should have standard format and information for ease of understanding and maintenance. The following is an example of header format that is being used in some companies:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module. This is a small writeup about what the module does.
- Different functions supported in the module, along with their
- input/output parameters.
- Global variables accessed/modified by the module.

c. Naming conventions for global variables, local variables, and constant identifiers: A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g., GlobalData) and local variable names start with small letters (e.g., localData). Constant names should be formed using capital letters only (e.g., CONSTDATA).

d. Conventions regarding error return values and exception handling mechanisms: The way error conditions are reported by different functions in a program should be standard within an organisation. For example, all functions while encountering an error condition should either

return a 0 or 1 consistently, independent of which programmer has written the code. This facilitates reuse and debugging.

e. Representative coding guidelines: The following are some representative coding guidelines that are recommended by many software development organisations. Wherever necessary, the rationale behind these guidelines is also mentioned.

f. Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. C l e v e r coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.

g. **Avoid obscure side effects:** The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, suppose the value of a global variable is changed or some file I/O is performed obscurely in a called module. That is, this is difficult to infer from the function's name and header information. Then, it would be really hard to understand the code.

h. Do not use an identifier for multiple purposes: The rationale that they give for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas when the same variable is used for three different purposes, only one memory location is used. Use of variables for multiple purposes usually makes future enhancements more difficult. For example, while changing the final computed result from integer to float type, the programmer might subsequently notice that it has also been used as a temporary loop variable that cannot be a float type.

i. Code should be well-documented: As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.

j. Length of any function should not exceed 10 source lines: A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.

k. Do not use GO TO statements: Use of GO TO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

6. SOFTWARE DOCUMENTATION

- When a software is developed, in addition to the executable files and the source code, several **kinds of documents such as users' manual, software requirements specification (SRS) document,** design document, test document, installation manual, etc., are developed as part of the software engineering process.
- All these documents are considered a vital part of any good software development practice.
- Good documents are helpful in the following ways: Good documents help enhance understandability of code.
- As a result, the availability of good documents help to reduce the effort and time required for maintenance.
- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover problem.

- Even when an engineer leaves the organisation, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
- Production of good documents helps the manager to effectively track the progress of the project.
- The project manager would know that some measurable progress has been achieved, if the results of some pieces of work has been documented and the same has been reviewed.
- Different types of software documents can broadly be classified into the following:

Internal documentation: These are provided in the source code itself.

External documentation: These are the supporting documents such as SRS document, installation document, user manual, design document, and test document.

1. Internal Documentation

- Internal documentation is the code comprehension features provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following:
  - Comments embedded in the source code.
  - Use of meaningful variable names.
  - Module and function headers.
  - Code indentation.
  - Code structuring (i.e., code decomposed into modules and functions).
  - Use of enumerated types.
  - Use of constant identifiers.
  - Use of user-defined data types.
- Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand a piece of code.
- The above assertion, of course, is in contrast to the common expectation that code commenting would be the most useful.
- The research finding is obviously true when comments are written without much thought.
- For example, the following style of code commenting is not much of a help in understanding the code.

```
a=10; /* a made 10 */
```

- A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments.
- Good software development organisations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.
- Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding the code.

2. External Documentation

- External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc.
- A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.
- An important feature that is required of any good external documentation is consistency with the code.

- If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the software.
- In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents.
- Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion.
- Another important feature required for external documents is proper understandability by the category of users for whom the document is designed.
- For achieving this, Gunning's fog index is very useful.

Gunning's fog index

- Gunning's fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a document.
- The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document.
- That is, if a certain document has a fog index of 12, any one who has completed his 12^{th} class would not have much difficulty in understanding that document.
- The Gunning's fog index of a document D can be computed as follows:

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}} \right) + \text{per cent of words having 3 or more syllables}$$

- Observe that the fog index is computed as the sum of two different factors.
- The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences).
- This factor therefore accounts for the common observation that long sentences are difficult to understand.
- The second factor measures the percentage of complex words in the document.
- Note that a syllable is a group of words that can be independently pronounced.
- For example, the word "sentence" has three syllables ("sen", "ten", and "ce"). Words having more than three syllables are complex words and presence of many such words hamper readability of a document.

Example. Consider the following sentence: "The Gunning's fog index is based on the premise that use of short sentences and simple words makes a document easy to understand." Calculate its Fog index.

The fog index of the above example sentence is

$$0.4 \text{ x}(23/1) + (4/23) \text{ X } 100 = 26$$

If a users' manual is to be designed for use by factory workers whose educational qualification is class 8, then the document should be written such that the Gunning's fog index of the document does not exceed 8.

7. TESTING

Basic Concepts and Terminologies

How to test a program?

- Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected.
- If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction.
- A highly simplified view of program testing is schematically shown in Figure 10.1.
- The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs.

Unless the conditions under which a software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers.

For examples, a software might fail for a test case only when a network connection is enabled.

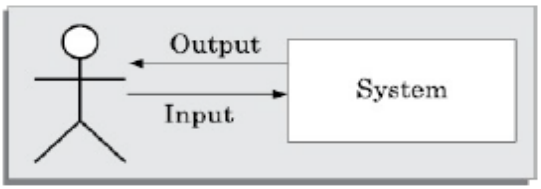


Figure 10.1: A simplified view of program testing.

Terminologies

- As is true for any specialised domain, the area of software testing has come to be associated with its own set of terminologies.

A **mistake** is essentially any programmer action that later shows up as an incorrect result during program execution.

- A programmer may commit a mistake in almost any development activity.
- For example, during coding a programmer might commit the mistake of not initializing a certain variable, or might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation.
- Both these mistakes can lead to an incorrect result.

An **error** is the result of a mistake committed by a developer in any of the development activities.

- Among the extremely large variety of errors that can exist in a program.
- One example of an error is a call made to a wrong function.
- The terms error, fault, bug, and defect are considered to be synonyms in the area of program testing.
- Though the terms error, fault, bug, and defect are all used interchangeably by the program testing community.

A **failure** of a program essentially denotes an incorrect behavior exhibited by the program during its execution.

- An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program.
- Every failure is caused by some bugs present in the program.

- In other words, we can say that every software failure can be traced to some bug or other present in the code.
- The number of possible ways in which a program can fail is extremely large.
- Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples: – The result computed by a program is 0, when the correct result is 10.
  - A program crashes on an input.
  - A robot fails to avoid an obstacle and collides with it.
- It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.

A **test case** is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program.

- The state of a program is also called its execution mode.
- As an example, consider the different execution modes of a certain text editor software.
- The text editor can at any time during its execution assume any of the following execution modes—edit, view, create, and display.
- In simple words, we can say that a test case is a set of test inputs, the mode in which the input is to be applied, and the results that are expected during and after the execution of the test case.

A **test scenario** is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output.

- A test case can be said to be an implementation of a test scenario.
- In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed.
- An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implement the test scenarios as test cases.

A **test script** is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.

- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality.
- A test case is said to be **negative test case**, if it is designed to test whether the software carries out something, that is not required of the system.
- As one example each of a positive test case and a negative test case, consider a program to manage user login.
- A positive test case can be designed to check if a login system validates a user with the correct user name and password.
- A negative test case in this case can be a test case that checks whether the login functionality validates and admits a user with wrong or bogus login user name or password.

A **test suite** is the set of all test that have been designed by a tester to test a given program.

Testability of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance.

- In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

A **failure mode** of a software denotes an observable way in which it can fail.

- In other words, all failures that have similar observable symptoms, constitute a failure mode.
- As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.

Equivalent faults denote two or more bugs that result in the system failing in the same failure mode.

- As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors.
- These two are equivalent faults, since each of these leads to a program crash.

Verification versus validation

- The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software.
- In spite of the apparent similarity between their objectives, the underlying principles of these two bug detection techniques and their applicability are very different.
- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase; whereas validation is the process of determining whether a fully developed software conforms to its requirements specification. Thus, the objective of verification is to check if the work products produced after a phase conform to that which was input to the phase.
- For example, a verification step can be to check if the design documents produced after the design step conform to the requirements specification.
- On the other hand, validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements.
- The primary techniques used for verification include review, simulation, formal verification, and testing.
- Review, simulation, and testing are usually considered as informal verification techniques. Formal verification usually involves use of theorem proving techniques or use of automated tools such as a model checker.
- On the other hand, validation techniques are primarily based on product testing.
- Note that we have categorized testing both under program verification and validation.
- The reason being that unit and integration testing can be considered as verification steps where it is verified whether the code is a s per the module and module interface specifications.
- On the other hand, system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.
- Verification does not require execution of the software, whereas validation requires execution of the software.
- Verification is carried out during the development process to check if the development activities are proceeding alright, whereas validation is carried out to check if the right as required by the customer has been developed.

- The primary objective of the verification steps are to determine whether the steps in product development are being carried out alright, whereas validation is carried out towards the end of the development process to determine whether the right product has been developed.
- Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and is an important software engineering principle.
- The principle of detecting errors as close to their points of commitment as possible is known as phase containment of errors.
- Phase containment of errors can reduce the effort required for correcting bugs.
- While verification is concerned with phase containment of errors, the aim of validation is to check whether the deliverable software is error free.
- The activities involved in these two types of bug detection techniques together are called the "V and V" activities. Based on the above discussions, we can conclude that:
- Error detection techniques = Verification techniques + Validation techniques

8. Testing Activities

- Testing involves performing the following main activities:

Test suite design: The set of test cases using which a program is to be tested is designed possibly using several test case design techniques.

Running test cases and checking the results to detect failures: Each test case is run and the results are compared with the expected results.

- A mismatch between the actual result and expected results indicates a failure.
- The test cases for which the system fails are noted down for later debugging.

Locate error: In this activity, the failure symptoms are analysed to locate the errors.

- For each failure observed during the previous activity, the statements that are in error are identified.

Error correction: After the error is located during debugging, the code is appropriately changed to correct the error.

- The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected. Of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

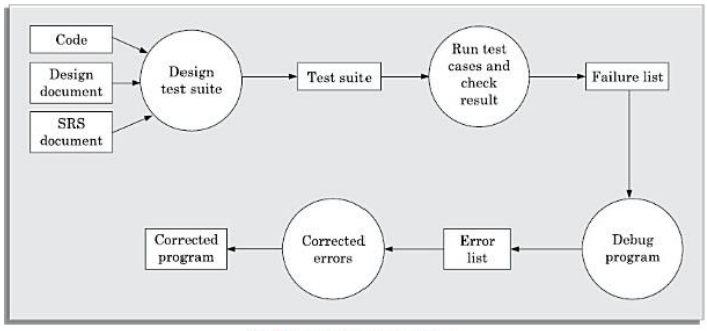


Figure 10.2: Testing process.

Why Design Test Cases?

- When test cases are designed based on random input data, many of the test cases donot contribute to the significance of the test suite,
- That is, they do not help detect any additional defects not already being detected by other test cases in the suite.
- Testing a software using a large collection of randomly selected test cases does not guarantee that all (or even most) of the errors in the system will be uncovered.
- Consider the following example code segment which determines the greater of two integer values x and y.
- This code segment has a simple programming error:

```
if (x>y) max = x;
else max = x;
```

- For the given code segment, the test suite {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.
- All the test cases in the larger test suite help detect the same error, while the other error in the code remains undetected.
- So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed.
- A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors.
- This is in contrast to testing using some random input values.

There are essentially two main approaches to systematically design test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software.

- That is, test cases are designed solely based on an analysis of the input/out behaviour (that is, functional behaviour) and does not require any knowledge of the internal structure of a program. For this reason, black-box testing is also known as functional testing.
- On the other hand, designing white-box test cases requires a thorough knowledge of the internal structure of a program, and therefore white-box testing is also called structural testing.
- Black- box test cases are designed solely based on the input-output behaviour of a program. In contrast, white-box test cases are based on an analysis of the code.
- These two approaches to test case design are complementary.
- That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

Testing in the Large versus Testing in the Small

- A software product is normally tested in three levels or stages:
  - Unit testing
  - Integration testing
  - System testing
- During unit testing, the individual functions (or units) of a program are tested.

- Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.
- After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing).
- Finally, the fully integrated system is tested (system testing). Integration and system testing are known as testing in the large.
- "Why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly?"
- There are two main reasons to it. First while testing a module, other modules with which this module needs to interface may not be ready.
- Moreover, it is always a good idea to first test the module in isolation before integration because it makes debugging easier.
- If a failure is detected when an integrated set of modules is being tested, it would be difficult to determine which module exactly has the error.

9. UNIT TESTING

- Unit testing is undertaken after a module has been coded and reviewed.
- This activity is typically undertaken by the coder of the module himself in the coding phase.
- Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed.

Driver and stub modules

- In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module. That is, besides the module under test, the following are needed to test the module:
- The procedures belonging to other modules that the module under test calls.
- Non-local data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.
- Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested.
- Stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

Stub: The role of stub and driver modules is pictorially shown in Figure 10.3.

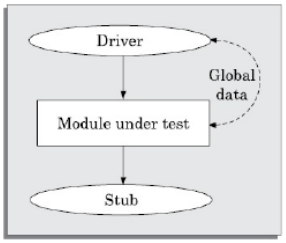


Figure 10.3: Unit testing with the help of driver and stub modules.

- A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behaviour. For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism.

Driver: A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

10. BLACK-BOX TESTING

- In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches available to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

1. Equivalence Class Partitioning

- In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes.
- The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.
- The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.
- Equivalence classes for a unit under test can be designed by examining the input data and output data.
- The following are two general guidelines for designing the equivalence classes:
- If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined. For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.

Example 10.6 For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and the black box test suite.
Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: {–5,500,6000}.

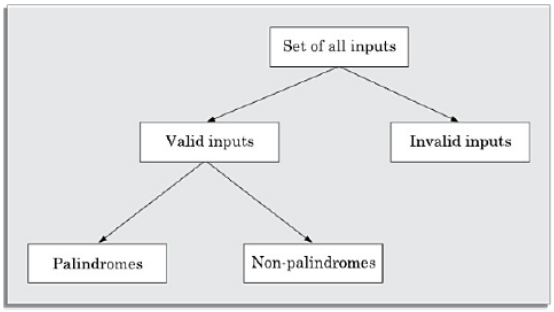


Figure 10.4: Equivalence classes for Example 10.6.

2. Boundary Value Analysis

- A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs.
- The reason behind programmers committing such errors might purely be due to psychological factors.
- Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes.
- For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.
- Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.
- To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values.
- For those equivalence classes that are not a range of values (i.e., consist of a discrete collection of values) no boundary value test cases can be defined.
- For an equivalence class that is a range of values, the boundary values need to be included in the test suite.
- For example, if an equivalence class contains the integers in the range 1 to 10, then the
- boundary value test suite is {0,1,10,11}.

Example 10.9 For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.

Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1,5000,5001}.

11. WHITE-BOX TESTING

- White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic.

Basic Concepts

- A white-box testing strategy can either be coverage-based or fault based.

Fault-based testing

- A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the **fault model** of the strategy. An example of a fault-based strategy is mutation testing.

Coverage-based testing

- A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.

Testing criterion for coverage-based testing

- A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures.

- The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.
- For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is statement coverage. We say that a test suite is adequate with respect to a criterion, if it covers all elements of the domain defined by that criterion.

Stronger versus weaker testing

- We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults.
- We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.
- A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.
- When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies.
- The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6.
- Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.
- If a stronger testing has been performed, then a weaker testing need not be carried out.

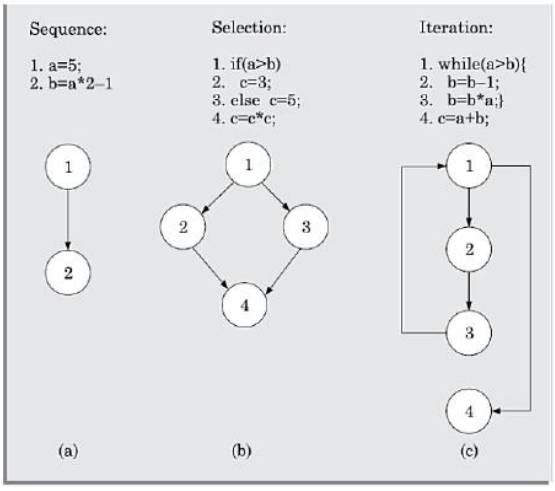


Figure 10.6: Illustration of stronger, weaker, and complementary testing strategies.

- A test suite should, however, be enriched by using various complementary testing strategies.

Unit V

Software Reliability and Quality Management:- Software Reliability: Hardware versus Software Reliability. – Software Quality – Software Quality Management System – ISO 9000: What is ISO 9000 Certification? – ISO 9000 for Software Industry – Shortcomings of ISO 9000 Certification. – SEI Capability Maturity Model: Level 1 to Level 5. **Software Maintenance:-** Characteristics of Software Maintenance: Characteristics of Software Evolution – Software Reverse Engineering. (12L)

1. SOFTWARE RELIABILITY

- The reliability of a software product essentially denotes its *trustworthiness or dependability*. Alternatively, the reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time. the main reasons that make software reliability more difficult to measure than hardware reliability:
- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

2. HARDWARE VERSUS SOFTWARE RELIABILITY

- An important characteristic feature that sets hardware and software reliability issues apart is the difference between their failure patterns.
- Hardware components fail due to very different reasons as compared to software components. Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.

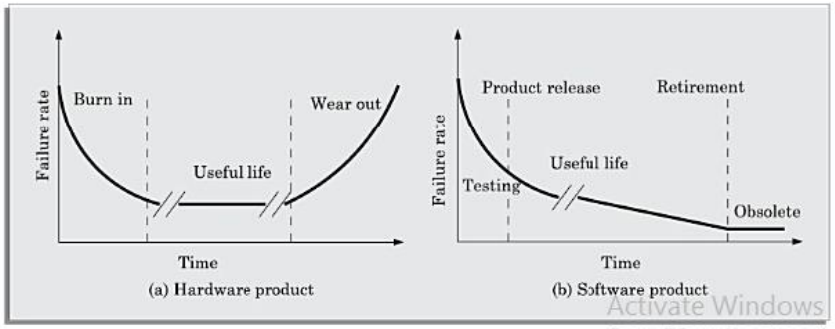


Figure 11.1: Change in failure rate of a product.

3. SOFTWARE QUALITY

A good quality product does exactly what the users want it to do, since for almost every product, fitness of purpose is interpreted in terms of satisfaction of the requirements laid down in the SRS document.

Although "fitness of purpose" is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc.—"fitness of purpose" is not a wholly satisfactory definition of quality for software products.

- To give an example of why this is so, consider a software product that is functionally correct. That is, it correctly performs all the functions that have been specified in its SRS document. Even though it may be functionally correct, we cannot consider it to be a quality product, if it has an almost unusable user interface.
- Another example is that of a product which does everything that the users wanted but has an almost incomprehensible and unmaintainable code.
- Therefore, the traditional concept of quality as "fitness of purpose" for software products is not wholly satisfactory.
- Unlike hardware products, software lasts a long time, in the sense that it keeps evolving to accommodate changed circumstances.
- The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:

Portability : A software product is said to be portable, if it can be easily made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.

Usability: A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.

Reusability: A software product has good reusability, if different modules of the product can easily be reused to develop new products.

Correctness: A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

McCall's quality factors

- McCall distinguishes two levels of quality attributes [McCall].
- The higher level attributes, known as quality factor s or external attributes can only be measured indirectly.
- The second-level quality attributes are called quality criteria.
- Quality criteria can be measured directly, either objectively or subjectively.
- By combining the ratings of several criteria, we can either obtain a rating for the quality factors, or the extent to which they are satisfied.
- For example, the reliability cannot be measured directly, but by measuring the number of defects encountered over a period of time.
- Thus, reliability is a higher-level quality factor and number of defects is a low-level quality factor.

ISO 9126

- ISO 9126 defines a set of hierarchical quality characteristics.
- Each sub characteristic in this is related to exactly one quality characteristic.
- This is in contrast to the McCall's quality attributes that are heavily interrelated. Another difference is that the ISO characteristic strictly refers to a software product, whereas McCall's attributes capture process quality issues as well.
- The users as well as the managers tend to be interested in the higher-level quality attributes (quality factors).

4. SOFTWARE QUALITY MANAGEMENT SYSTEM

A quality management system (often referred to as quality system) is the principal methodology used by organisations to ensure that the products they develop have the desired quality. Some of the important issues associated with a quality system:

Managerial structure and individual responsibilities

- A quality system is the responsibility of the organisation as a whole.
- Every organisation has a separate quality department to perform several quality system activities. The quality system of an organisation should have the full support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

Quality system activities

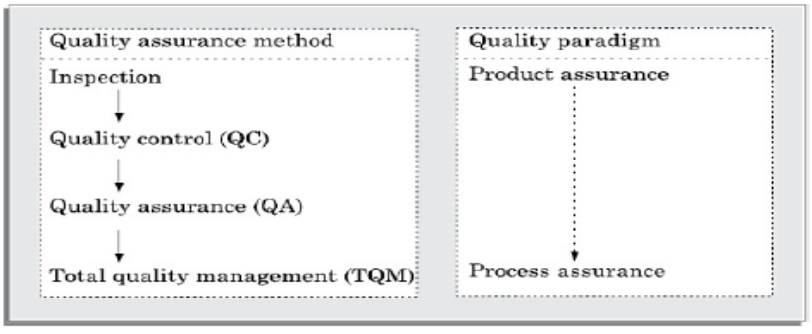
The quality system activities encompass the following: Auditing of projects to check if the processes are being followed.

- Collect process and product metrics and analyze them to check if quality goals are being met.
- Review of the quality system to make it more effective.
- Development of standards, procedures, and guidelines.
- Produce reports for the top management summarizing the effectiveness of the quality system in the organisation.
- A good quality system must be well documented.

Without a properly documented quality system, the application of quality controls and procedures become ad hoc, resulting in large variations in the quality of the products delivered. Also, an undocumented quality system sends clear messages to the staff about the attitude of the organisation towards quality assurance. International standards such as ISO 9000 provide guidance on how to organize a quality system.

Evolution of Quality Systems

Quality systems have rapidly evolved over the last six decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. For example, a company manufacturing nuts and bolts would inspect its finished goods and would reject those nuts and bolts that are outside certain specified tolerance range. Since that time, quality systems of organisations have undergone four stages of evolution as shown in Figure 11.3.



The initial product inspection method gave way to quality control (QC) principles. Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

Thus, quality control aims at correcting the causes of errors and not just rejecting the defective products.
The next breakthrough in quality systems, was the development of the quality assurance (QA) principles.
The basic premise of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality.
The modern quality assurance paradigm includes guidance for recognizing, defining, analyzing, and improving the production process.
Total quality management (TQM) advocates that the process followed by an organisation must continuously be improved through process measurements.
TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign.
A term related to TQM is business process re-engineering BPR), which is aims at re-engineering the way business is carried out in an organisation, whereas our focus in this text is re-engineering of the software development process.

5. ISO 9000

International standards organisation (ISO) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987.

6. WHAT IS ISO 9000 CERTIFICATION?

- ISO 9000 certification serves as a reference for contract between independent parties.
- In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether the vendor has obtained ISO 9000 certification or not
- In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system. The ISO standard addresses both operational aspects (that is, the process) and organizational aspects such as responsibilities, reporting, etc.
- In a nutshell, ISO 9000 specifies a set of recommendations for repeatable and high quality product development.
- It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product it self.
- ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO

The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.
The types of software companies to which the different ISO standards apply are as follows:
ISO 9001: This standard applies to the organisations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.
ISO 9002: This standard applies to those organisations which do not design products but are only involved in production. Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organisations.

ISO 9003: This standard applies to organisations involved only in installation and testing of products.

7. ISO 9000 FOR SOFTWARE INDUSTRY

- ISO 9000 is a generic standard that is applicable to a large gamut of industries, starting from a steel manufacturing industry to a service rendering company.
- Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very difficult to interpret them in the context of software development organisations.
- An important reason behind such a situation is the fact that software development is in many respects radically different from the development of other types of products.
- Two major differences between software development and development of other kinds of products are as follows:
- Software is intangible and therefore difficult to control. It means that software would not be visible to the user until the development is complete and the software is up and running. It is difficult to control and manage anything that you cannot see and feel.
- In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to accurately determine how much work has been completed and to estimate how much more time will it take.
- During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product.
- As an example, consider a steel making company. The company would consume large amounts of raw material such as iron-ore, coal, lime, manganese, etc. Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organisations.
- Due to such radical differences between software and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of software industry.
- Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for software industry.
- At present, official guidance is inadequate regarding the interpretation of various clauses of ISO 9000 standard in the context of software industry and one has to keep on cross referencing the ISO 9000-3 document.

8. SHORTCOMINGS OF ISO 9000 CERTIFICATION

Even though ISO 9000 is widely being used for setting up an effective quality system in an organisation, it suffers from several shortcomings.

Some of these shortcomings of the ISO 9000 certification process are the following:

- ISO 9000 requires a software production process to be adhered to, but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
- ISO 9000 certification process is not fool-proof and no international accredition agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accredition agencies and also among the registrars.

- Organisations getting ISO 9000 certification often tend to downplay domain expertise and the ingenuity of the developers. These organisations start to believe that since a good process is in place, the development results are truly person-independent.
- That is, any developer is as effective as any other developer in performing any particular software development activity. In manufacturing industry there is a clear link between process quality and product quality.
- Once a process is calibrated, it can be run again and again producing quality goods. Many areas of software development are so specialised that special expertise and experience in these areas (domain expertise) is required.
- Also, unlike in case of general product manufacturing, ingenuity and effectiveness of personal practices play an important art in determining the results produced by a developer.
- In other words, software development is a creative process and individual skills and experience are important.
- ISO 9000 does not automatically lead to continuous process improvement. In other words, it does not automatically lead to TQM.

9. SEI CAPABILITY MATURITY MODEL

- S E I capability maturity model (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. CMM is patterned after the pioneering work of Philip Crosby who published his maturity grid of five evolutionary stages in adopting quality practices in his book "Quality is Free" [Crosby79].
- The Unites States Department of Defence (US DoD) is the largest buyer of software product. It often faced difficulties in vendor performances, and had to many times live with low quality products, late delivery, and cost escalations. In this context, SEI CMM was originally developed to assist the U.S. Department of Defense (DoD) in software acquisition.
- Most of the major DoD contractors began CMM-based process improvement initiatives as they vied for DoD contracts. It was observed that the SEI CMM model helped organisations to improve the quality of the software they developed and therefore adoption of SEI CMM model had significant business benefits.
- In simple words, CMM is a reference model for apprising the software process maturity into different levels. This can be used to predict the most likely outcome to be expected from the next project that the organisation undertakes. It must be remembered that SEI CMM can be used in two ways— **capability evaluation and software process assessment**.
- **Capability evaluation** and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organisation. Capability evaluation is administered by the contract awarding authority, and therefore the results would indicate the likely contractor performance if the contractor is awarded a work.
- On the other hand, **software process assessment** is used by an organisation with the objective to improve its own process capability. Thus, the latter type of assessment is for purely internal use by a company.
- The different levels of SEI CMM have been designed so that it is easy for an organisation to slowly build its quality system starting from scratch. SEI CMM classifies software development industries into the following **five** maturity levels:

Level 1: Initial

A software development organisation at this level is characterized by adhoc activities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level.

The success of projects depend on individual efforts and heroics. When a developer leaves the organisation, the successor would have great difficulty in understanding the process that was followed and the work completed. Also, no formal project management practices are followed. As a result, time pressure builds up towards the end of the delivery time, as a result short-cuts are tried out leading to low quality products.

Level 2: Repeatable

- At this level, the basic project management practices such as tracking cost and schedule are established. Configuration management tools are used on items identified for configuration control. Size and cost estimation techniques such as function point analysis, COCOMO, etc.,are used.
- The necessary process discipline is in place to repeat earlier success on projects with similar applications. Though there is a rough understanding among the developers about the process being followed, the process is not documented.
- Configuration management practices are used for all project deliverables. Please remember that opportunity to repeat a process exists only when a company produces a family of products. Since the products are very similar, the success story on development of one product can repeated for another.
- In a non repeatable software development organisation, a software product development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals.
- On the other hand, in a non-repeatable software development organisation, the chances of successful completion of a software project is to a great extent depends on who the team members are. For this reason, the successful development of one product by such an organisation doesnot automatically imply that the next product development will be successful.

Level 3: Defined

At this level, the processes for both management and development activities are defined and documented. There is a common organisation-wide understanding of activities, roles, and responsibilities. The processes though defined, the process and product qualities are not measured. At this level, the organisation builds up the capabilities of its employees through periodic training programs. Also, review techniques are emphasized and documented to achieve phase containment of errors. ISO 9000 aims at achieving this level.

Level 4: Managed

At this level, the focus is on software metrics. Both process and product metrics are collected. Quantitative quality goals are set for the products and at the time of completion of development it was checked whether the quantitative quality goals for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

Level 5: Optimizing

At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement. For example, if from an analysis of the process measurement results, it is found that the code reviews are not very effective and a large

number of errors are detected only during the unit testing, then the process would be fine tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated into the process. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. At CMM level 5, an organisation would identify the best software engineering practices and innovations (which may be tools, methods, or processes) and would transfer these organisation wide. Level 5 organisations usually have a department whose sole responsibility is to assimilate latest tools and technologies and propagate them organisation-wide. The focus of each level and the corresponding key process areas are shown in the Table 11.1:

Table 11.1 Focus areas of CMM levels and Key Process Areas

CMM Level	*Focus*	*Key Process Areas (KPAs)*
Initial	Competent people	
Repeatable	Project management	Software project planning Software configuration management
Defined	Definition of processes	Process definition Training program Peer reviews
Managed	Product and process quality	Quantitative process metrics Software quality management
Optimising	Continuous process improvement	Defect prevention Process change management Technology change management

SEI CMM provides a list of key areas on which to focus to take an organisation from one level of maturity to the next. Thus, it provides a way for gradual quality improvement over several stages. Each stage has been carefully designed such that one stage enhances the capability already built up.

For example, trying to implement a defined process (level 3) before a repeatable process (level 2) would be counterproductive as it becomes difficult to follow the defined process due to schedule and budget pressures.

Substantial evidence has now been accumulated which indicate that adopting SEI CMM has several business benefits. However, the organisations trying out the CMM frequently face a problem that stems from the characteristic of the CMM itself.

10. CHARACTERISTICS OF SOFTWARE MAINTENANCE

First classify the different maintenance efforts into a few classes. Software maintenance is becoming an important activity of a large number of organisations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts.

Types of Software Maintenance

There are **three** types of software maintenance, which are described as follows:

Corrective: Corrective maintenance of a software product is necessary either to rectify the bugs observed while the system is in use.

Adaptive: A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.

Perfective: A software product needs maintenance to support the new features that users want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

11. CHARACTERISTICS OF SOFTWARE EVOLUTION

Lehman and Belady have studied the characteristics of evolution of s e v e r a l software products [1980]. They have expressed their observations in the form of laws. Their important laws are presented in the following subsection. But a word of caution here is that these are generalizations and may not be applicable to specific cases and also most of these observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

Lehman's first law: A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts. This law clearly shows that every product irrespective of how well designed must undergo maintenance. In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law: The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that when you add a function during maintenance, you build on top of an existing program, often in a way that the existing program was not intended to support. If you do not redesign the system, the additions will be more complex that they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law: Over a program's lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

12. SOFTWARE REVERSE ENGINEERING

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code.

The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1. A program can be reformatted using any of the several available pretty printer programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important because we had seen in that meaningful variable names is the most helpful code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

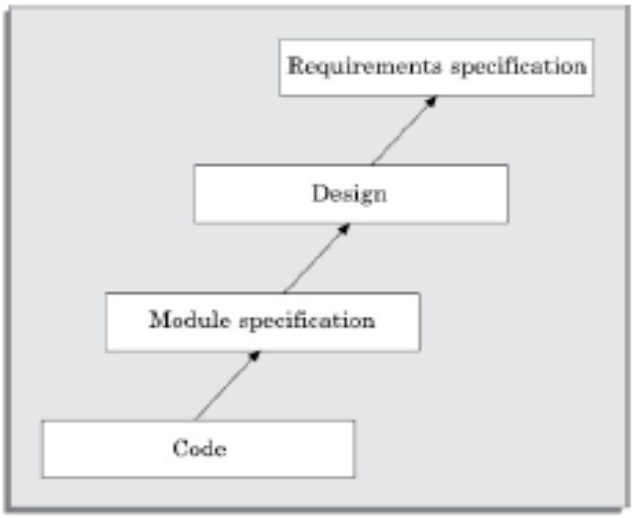


Figure 13.1: A process model for reverse engineering.

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

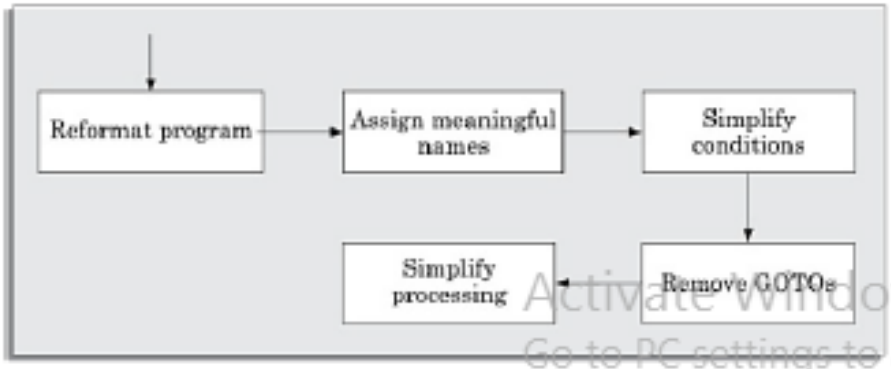


Figure 13.2: Cosmetic changes carried out before reverse engineering.