What is software engineering?

software engineering is: “A systematic collection of good program development practices and techniques”..An alternative definition of software engineering is: “An engineering approach to develop software”. Software engineering discusses systematic and cost-effective techniques for software development.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Evolution of an Engineering Discipline

Software engineering principles have evolved over the last sixty years with contributions from numerous researchers and software professionals. Over the years, it has emerged from a pure art to a craft, and finally to an engineering discipline. The early programmers used a exploratory programming style. This style of program development is now variously being referred to as 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 follow. Every programmer himself evolves his own software development techniques solely guided by his own intuition, experience.

The exploratory style usually yields poor quality and unmaintainable code and also makes program development very expensive as well as time-consuming.

In a build and fix style, a program is quickly developed without making any specification, plan, or

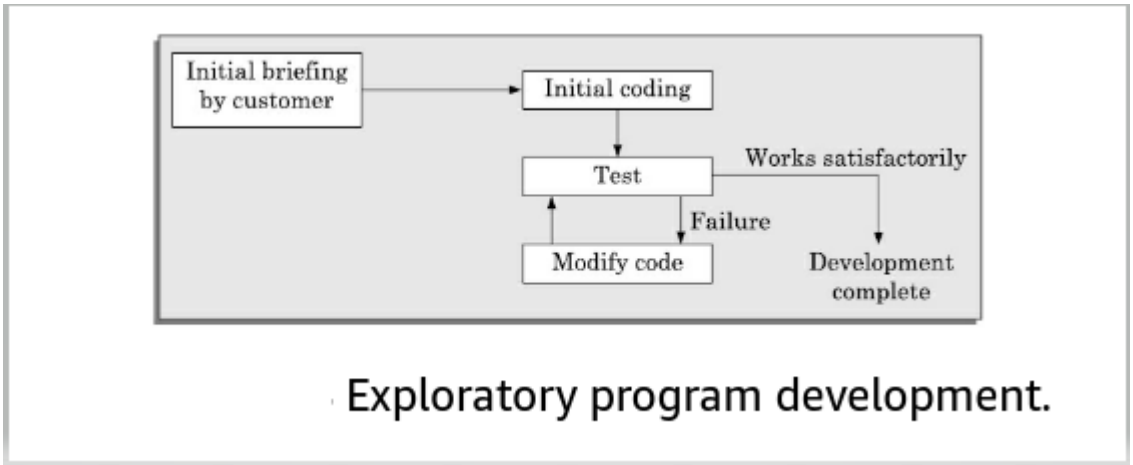
design. The different imperfections that are subsequently noticed are fixed.

In the early days of programming, there were good programmers and bad programmers. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers. Over the next several years, all good principles (or tricks) that were discovered by programmers along with research innovations have systematically been organised into a body of knowledge that forms the discipline of software engineering.

Software engineering principles are now being widely used in industry, and new principles are still continuing to emerge at a very rapid rate.

Exploratory style of software development

The exploratory program development style refers to an informal development style where the programmer makes use of his own intuition to develop a program rather than making use of the systematic body of knowledge categorized under the software engineering discipline. The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software. Though the exploratory style imposes no rules a typical development starts after an initial briefing from the customer. Based on this briefing, the developers start coding to develop a working program. The software is tested and the bugs found are fixed. This cycle of testing and bug fixing continues till the software works satisfactorily for the customer. A schematic of this work sequence in a build and fix style has been shown graphically in Figure . Observe that coding starts after an initial customer briefing



Exploratory program development.

about what is required. After the program development is complete, a test and fix cycle continues till the program becomes acceptable to the customer.

Early Computer Programming

Early commercial computers were very slow and too elementary as compared to today's standards.

Even simple processing tasks took considerable computation time on those computers. Those programs were usually written in assembly languages. Program lengths were typically limited to about a few hundreds of lines of monolithic assembly code. Every programmer developed his own individualistic style of writing programs according to his intuition .In simple words, programmers wrote programs without formulating any proper solution strategy, plan, or design. This style of programming as the build and fix (or the exploratory programming) style.

High-level Language Programming

Computers became faster with the introduction of the semiconductor technology in the early 1960s. Faster semiconductor transistors replaced the prevalent vacuum tube-based circuits in a computer. With the availability of more powerful computers, it became possible to solve larger and more complex problems. At this time, 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.

Control Flow-based Design

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs. To cope up with this problem, experienced programmers advised other programmers to pay particular attention to the design of a program's control flow structure. In order to help develop programs having good control flow structures, the flow charting technique was developed. The flow charting technique is being used to represent and design algorithms.A program's control flow structure indicates the sequence in which the program's instructions are executed.

Structured programming—a logical extension

A program is called structured when it uses only the sequence, selection, and iteration types of constructs and is modular. Structured programs avoid unstructured control flows by restricting the use of GO TO statements. Structured programming is facilitated, if the programming language being used supports single-entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, an important feature of structured programs is the design of good control structures.

Data Structure-oriented Design

Computers became even more powerful with the advent of integrated circuits . These are used to solve more complex problems. Software developers were expected to solve develop larger and more complicated software. This often required writing in excess of several tens of thousands of lines of source code. The control flow-based program development techniques could not be used satisfactorily any more to write those programs, and more effective program development techniques were needed. 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.

Data Flow-oriented Design

As computers became still faster and more powerful with the introduction of very large scale integrated (VLSI) Circuits and some new architectural concepts, more complex and sophisticated software were needed to solve further challenging problems. Thereforer more effective techniques for designing software and soon data flow-oriented techniques were proposed. 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).

For example, Figure shows the data-flow representation of an automated car assembly plant.

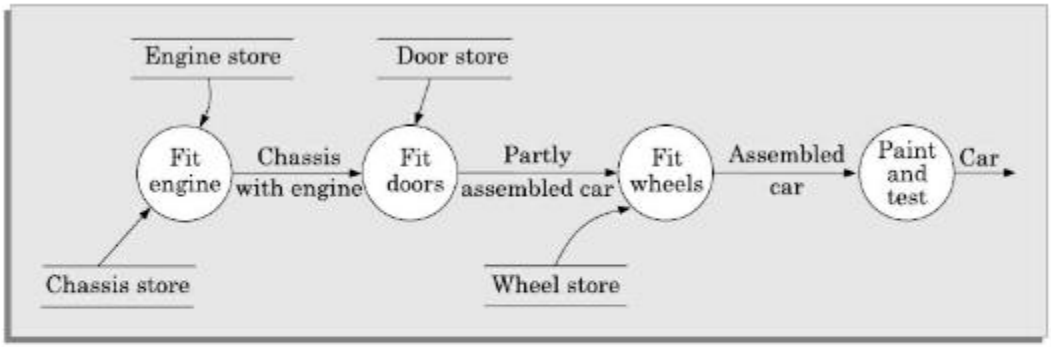


FIGURE Data flow model of a car assembly plant.

Object oriented design

Data flow-oriented techniques evolved into object-oriented design (OOD) techniques . Object-oriented design technique is an intuitively appealing approach,

where the natural objects (such as employees, pay-roll-register, etc.) 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. Object-oriented techniques have gained wide spread acceptance because of their simplicity, the scope for code and design reuse, promise of lower development time, lower development cost, more robust code, and easier maintenance.

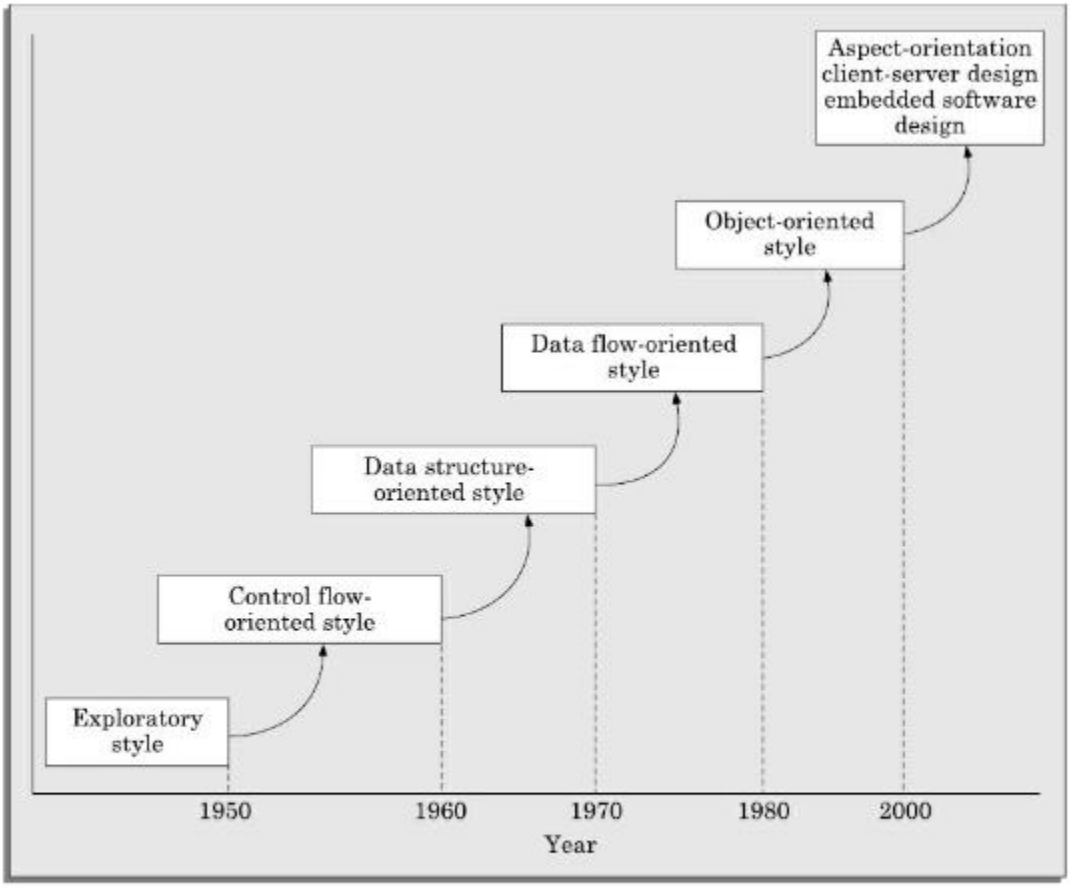


FIGURE Evolution of software design techniques.

Software Development Life Cycle (SDLC)

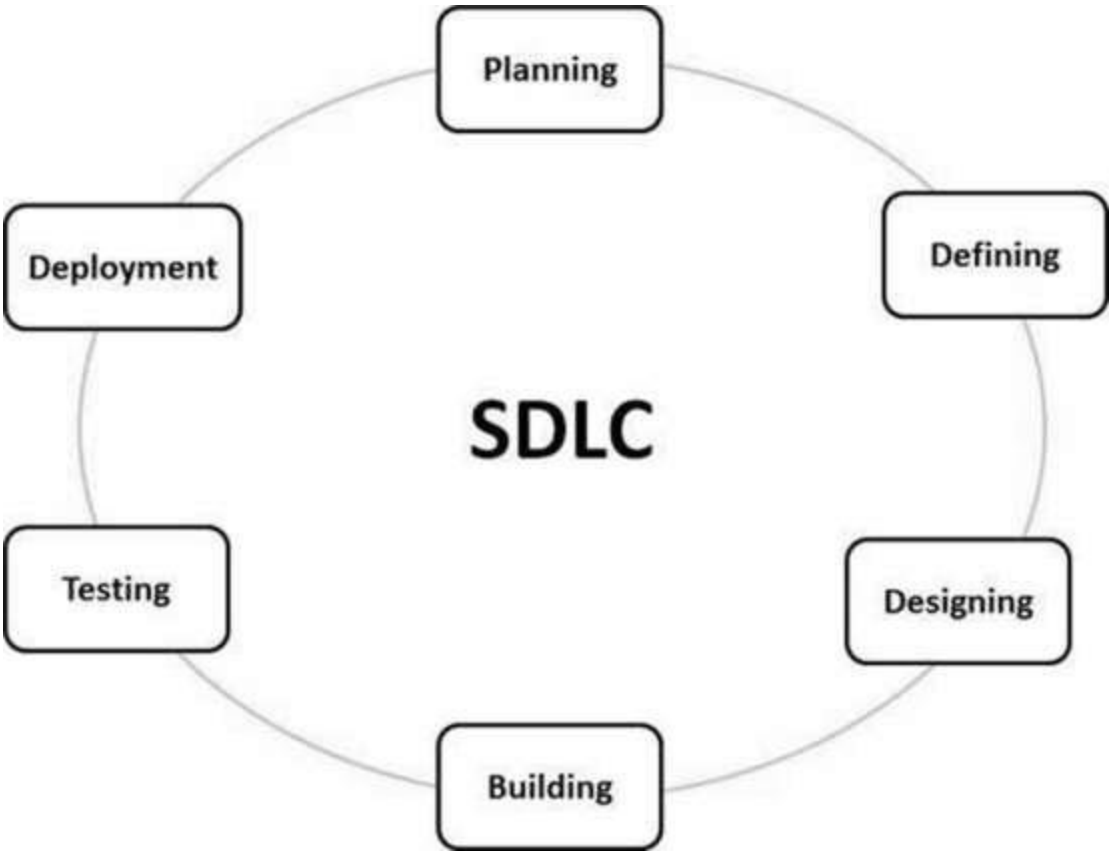
It is a process used by the software industry to design, develop and test high quality softwares. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

- SDLC is the acronym of Software Development Life Cycle.
- It is also called as Software Development Process.
- SDLC is a framework defining tasks performed at each step in the software development process.

What is SDLC?

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development Life Cycle consists of the following stages −

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java and PHP are used

for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

SDLC Models

There are various software development life cycle models defined and designed which are followed during the software development process. These models are also referred as Software Development Process Models". Each process model follows a Series of steps unique to its type to ensure success in the process of software development.

Following are the most important and popular SDLC models followed in the industry −

Classical Waterfall Model /Waterfall model

Classical waterfall model is the basic **software development life cycle** model. Classical waterfall model divides the life cycle into a set of phases. This model considers that one phase can be started after completion of the previous phase. That is the output of one phase will be the input to the next phase. Thus the development process can be considered as a sequential flow in the waterfall. Here the phases do not overlap with each other. The different sequential phases of the classical waterfall model are shown in the below figure:

The classical waterfall model divides the life cycle into six phases as shown in Figure . It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the model.

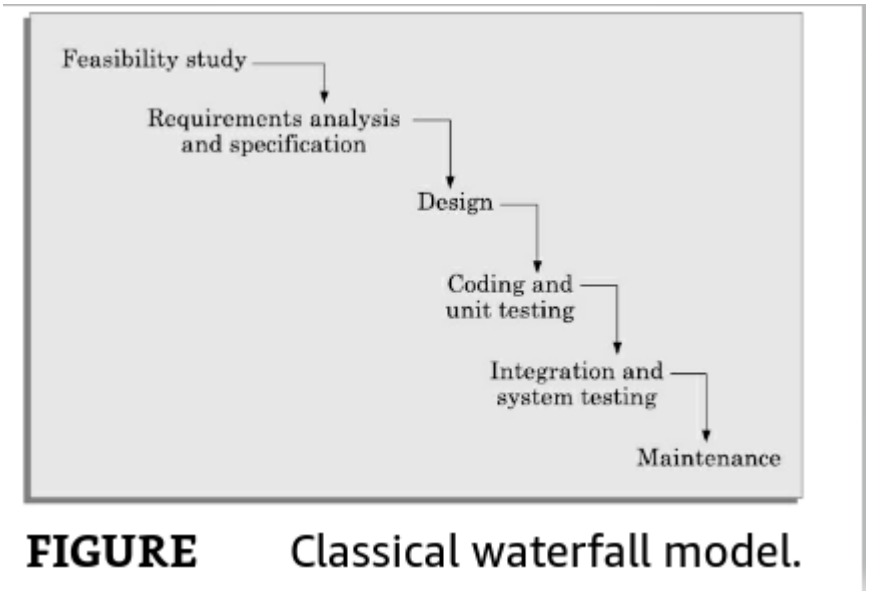


FIGURE Classical waterfall model.

Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in Figure. The different phases are—

1.feasibility study

2. requirements analysis and specification

3. design

4. coding and unit testing,

5.integration

6. system maintenance.

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. The 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.

The different identified solution schemes are analysed to evaluate their benefits and shortcomings. Such evaluation often requires making approximate estimates of the resources required, cost of development, and development time required. We can summarise the outcome of the feasibility study phase by noting that other than deciding whether to take up a project or not.

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

1. requirements gathering and analysis,
2. requirements specification.

1. 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 analysed. The goal of the requirements analysis activity is to clear the incompleteness and inconsistencies in these gathered requirements.

2. **Requirements specification**:

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

The SRS document normally serves as a contract between the development team and the customer. Any future dispute between the customer and the developers can be settled by examining the SRS document.

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. In technical terms, during the design phase the software architecture is derived from the SRS document.

Two different design approaches are popularly being used at present—the procedural and object-oriented design approaches.

Procedural design approach:

The traditional procedural design approach is in use in many software development projects at the present time. This traditional design technique is based on data flow modelling. During structured analysis, the functional requirements specified in the SRS document are decomposed into subfunctions and the data-flow among these subfunctions is analysed and represented diagrammatically in the form of DFDs(Data flow diagrams).

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.

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.

Integration and system testing

Integration of different modules are undertaken soon after they have been coded and unit tested. Integration of various modules is carried out incrementally over a number of steps. During each integration step, previously planned modules are added to the partially integrated system and the resultant system is tested. Finally, after all the modules have been successfully integrated and tested, the full working system is obtained and system testing is carried out on this.
System testing consists three different kinds of testing activities as described below :

1. **Alpha testing:** Alpha testing is the system testing performed by the development team.

2. **Beta testing:** Beta testing is the system testing performed by a friendly set of customers.
3. **Acceptance testing:** After the software has been delivered, the customer performed the acceptance testing to determine whether to accept the delivered software or to reject it.

Maintainence: Maintenance is the most important phase of a software life cycle. The effort spent on maintenance is the 60% of the total effort spent to develop a full software. There are basically three types of maintenance :

1. **Corrective Maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
2. **Perfective Maintenance:** This type of maintenance is carried out to enhance the functionalities of the system based on the customer's request.
3. **Adaptive Maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment such as work on a new computer platform or with a new operating system.

Advantages of Classical Waterfall Model

Classical waterfall model is an idealistic model for software development. It is very simple, so it can be considered as the basis for other software development life cycle models. Below are some of the major advantages of this SDLC model:

- This model is very simple and is easy to understand.
- Phases in this model are processed one at a time.
- Each stage in the model is clearly defined.
- This model has very clear and well undestood milestones.
- Process, actions and results are very well documented.
- This model works well for smaller projects and projects where requirements are well understood.

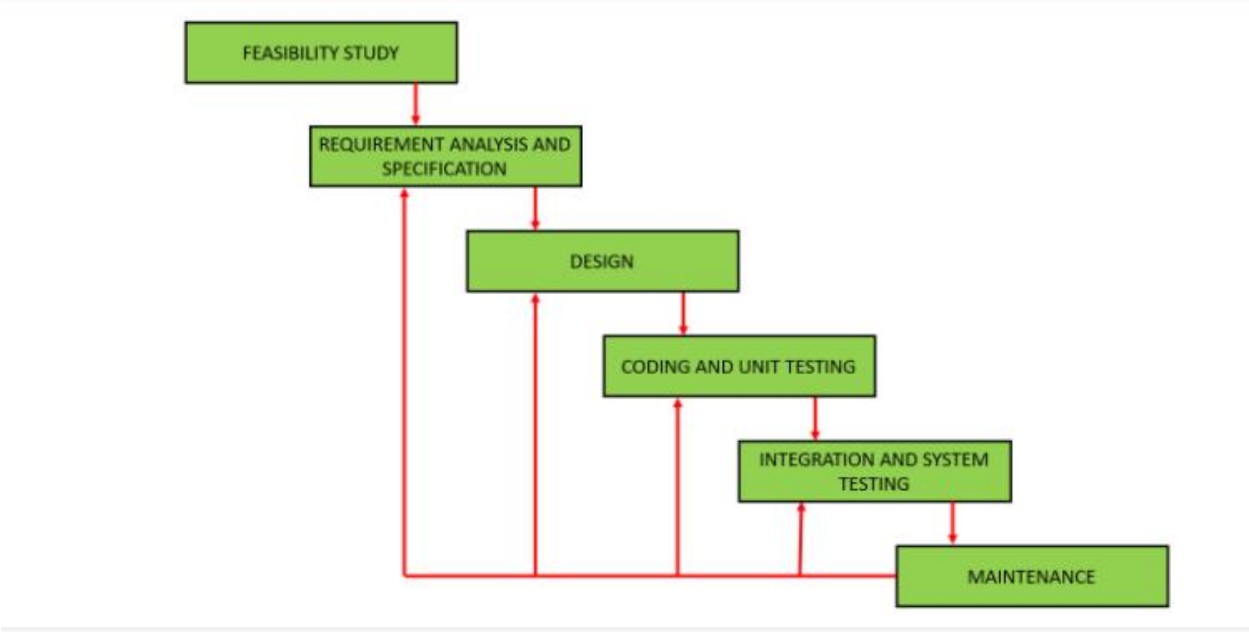
Drawbacks of Classical Waterfall Model

Classical waterfall model suffers from various shortcomings, basically we can't use it in real projects, but we use other software development lifecycle models which are based on the classical waterfall model. Below are some major drawbacks of this model:

- **No feedback path:** In classical waterfall model evolution of a software from one phase to another phase is like a waterfall. It assumes that no error is ever committed by developers during any phases. Therefore, it does not incorporate any mechanism for error correction.
- **Difficult to accommodate change requests:** This model assumes that all the customer requirements can be completely and correctly defined at the beginning of the project, but actually customers' requirements keep on changing with time. It is difficult to accommodate any change requests after the requirements specification phase is complete.
- **No overlapping of phases:** This model recommends that new phase can start only after the completion of the previous phase. But in real projects, this can't be maintained. To increase the efficiency and reduce the cost, phases may overlap.

Extension of Waterfall model or Iterative Waterfall Model

In a practical software development project, the classical waterfall model is hard to use. So, Iterative waterfall model can be thought of as incorporating the necessary changes to the classical waterfall model to make it usable in practical software development projects. It is almost same as the classical waterfall model except some changes are made to increase the efficiency of the software development.
The iterative waterfall model provides feedback paths from every phase to its preceding phases, which is the main difference from the classical waterfall model.
Feedback paths introduced by the iterative waterfall model are shown in the figure below.



When errors are detected at some later phase, these feedback paths allow correcting errors committed by programmers during some phase. The feedback paths allow the phase to be reworked in which errors are committed and these changes are reflected in the later phases. But, there is no feedback path to the stage – feasibility study, because once a project has been taken, does not give up the project easily.
It is good to detect errors in the same phase in which they are committed. It reduces the effort and time required to correct the errors.

Advantages of Iterative Waterfall Model

Feedback Path: In the classical waterfall model, there are no feedback paths, so there is no mechanism for error correction. But in iterative waterfall model feedback path from one phase to its preceding phase allows correcting the errors that are committed and these changes are reflected in the later phases.

Simple: Iterative waterfall model is very simple to understand and use. That's why it is one of the most widely used software development models.

Drawbacks of Iterative Waterfall Model

Difficult to incorporate change requests: The major drawback of the iterative waterfall model is that all the requirements must be clearly stated before starting of the development phase. Customer may change requirements after some time but the iterative waterfall model does not leave any scope to incorporate change requests that are made after development phase starts.

Incremental delivery not supported: In the iterative waterfall model, the full software is completely developed and tested before delivery to the customer. There is no scope for any intermediate delivery. So, customers have to wait long for getting the software.

Overlapping of phases not supported: Iterative waterfall model assumes that one phase can start after completion of the previous phase, But in real projects, phases may overlap to reduce the effort and time needed to complete the project.

Risk handling not supported: Projects may suffer from various types of risks. But, Iterative waterfall model has no mechanism for risk handling.

Limited customer interactions: Customer interaction occurs at the start of the project at the time of requirement gathering and at project completion at the time of software delivery. These fewer interactions with the customers may lead to many problems as the finally developed software may differ from the customers' actual requirements.

Agile Development Models

In earlier days Iterative Waterfall model was very popular to complete a project. But nowadays developers face various problems while using it to develop a software. The main difficulties included handling change requests from customers during project development and the high cost and time required to incorporate these changes. To overcome these drawbacks of Waterfall model, the Agile Software Development model was proposed.

The Agile model was primarily designed to help a project to adapt to change requests quickly. So, the main aim of the Agile model is to facilitate quick project completion. To accomplish this task agility is required. Agility is achieved by fitting the process to the project, removing activities that may not be essential for a specific project. Also, anything that is wastage of time and effort is avoided.

In the Agile model, the requirements are decomposed into many small parts that can be incrementally developed. The Agile model adopts Iterative development. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and that can be completed within a couple of weeks only. At a time one iteration is planned, developed and deployed to the customers. Long-term plans are not made.
Agile model is the combination of iterative and incremental process models. Steps involved in agile SDLC models are:

- Requirement gathering
- Requirement Analysis
- Design
- Coding
- Unit testing
- Acceptance testing

The time to complete an iteration is known as a Time Box. Time-box refers to the maximum amount of time needed to deliver an iteration to customers. The central principle of the Agile model is the delivery of an increment to the customer after each Time-box.

Advantages:

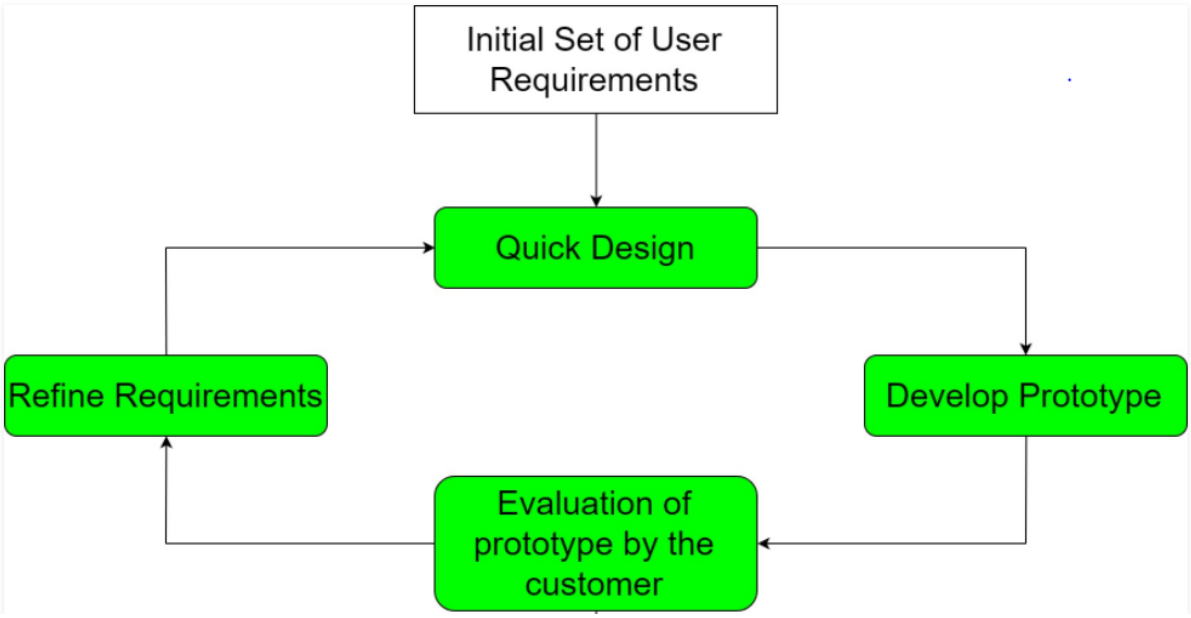
- It reduces total development time of the whole project.
- Customer representative get the idea of updated software products after each iretation. So, it is easy for him to change any requirement if needed.

Disadvantages:

- Due to lack of formal documents, it creates confusion and important decisions taken during different phases can be misinterpreted at any time by different team members.

Prototyping model

- Prototyping is defined as the process of developing a working model of a product or system that has to be engineered. It is used for obtaining customer feedback as described below:



The Prototyping Model is one of the most popularly used Software Development Life Cycle Models (SDLC models).This model is used when the customers do not know the exact project requirements beforehand. In this model, a prototype of the end product is first developed, tested and refined as per customer feedback repeatedly till a final acceptable prototype is achieved which forms the basis for developing the final product.

In this process model, the system is partially implemented before or during the analysis phase thereby giving the customers an opportunity to see the product early in the life cycle.

Once the customer figures out the problems, the prototype is further refined to eliminate them. The process continues until the user approves the prototype and finds the working model to be satisfactory.

Types of Prototyping Models

Four types of Prototyping models are:

1. Rapid Throwaway prototypes
2. Evolutionary prototype
3. Incremental prototype
4. Extreme prototype

Rapid Throwaway Prototype

Rapid throwaway is based on the preliminary requirement. It is quickly developed to show how the requirement will look visually.

In this method, a developed prototype will be discarded and will not be a part of the ultimately accepted prototype.

Evolutionary Prototyping

Here, the prototype developed is incrementally refined based on customer's feedback until it is finally accepted. It helps you to save time as well as effort.

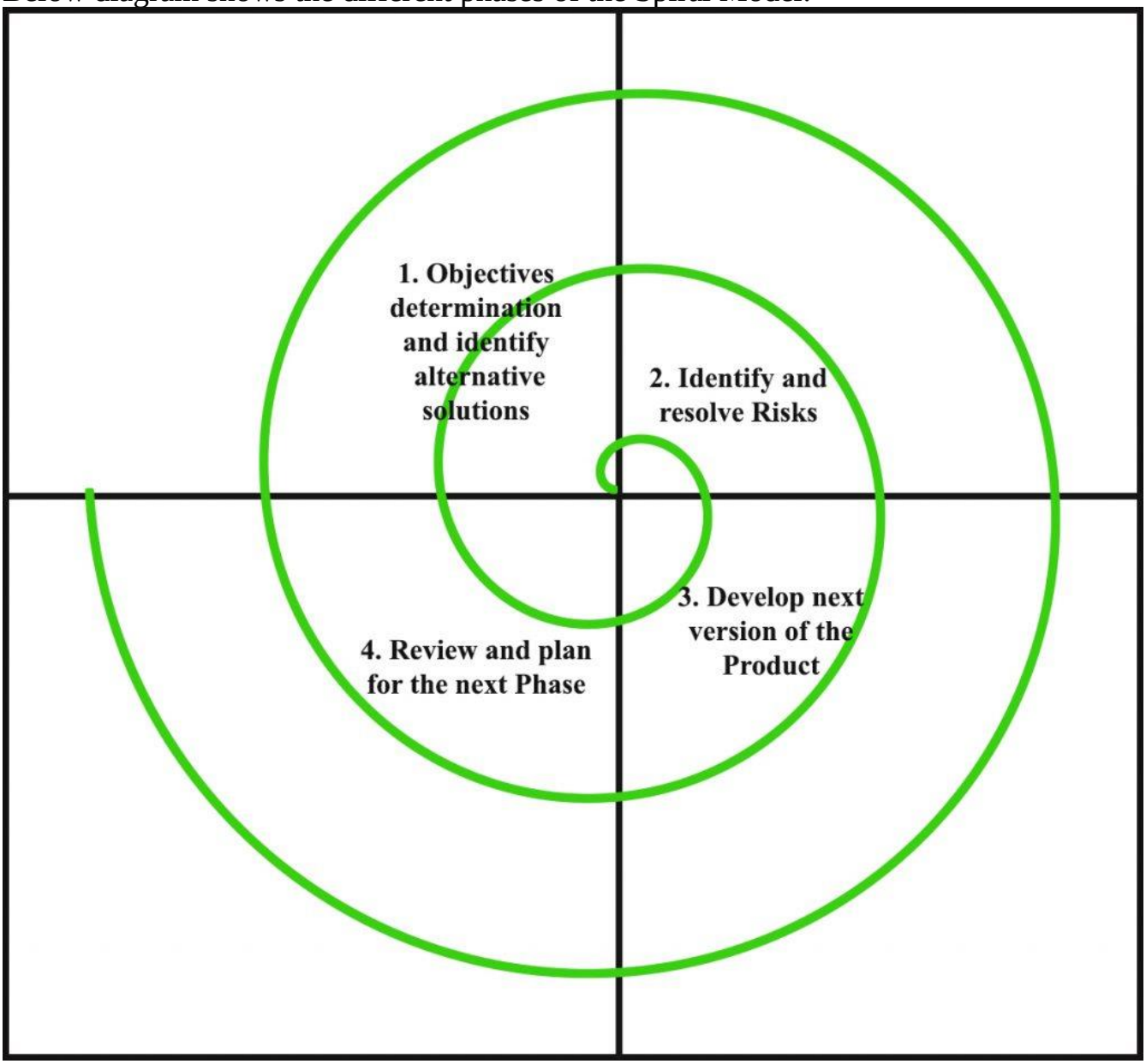
C) Incremental Prototyping – In this type of incremental Prototyping, the final expected product is broken into different small pieces of prototypes and being developed individually. In the end, when all individual pieces are properly developed, then the different prototypes are collectively merged into a single final product in their predefined order.

Extreme Prototyping – This method is mainly used for web development. It is consists of three sequential independent phases:

Spiral model

It is one of the most important Software Development Life Cycle models, which provides support for **Risk Handling**. In its diagrammatic representation, it looks like a spiral with many loops. The exact number of loops of the spiral is unknown and can vary from project to project. **Each loop of the spiral is called a Phase of the software development process.** The exact number of phases needed to develop the product can be varied by the project manager depending upon the project risks. As the project manager dynamically determines the number of phases, so the project manager has an important role to develop a product using spiral model.
The Radius of the spiral at any point represents the expenses(cost) of the project so far, and the angular dimension represents the progress made so far in the current phase.

Below diagram shows the different phases of the Spiral Model:



Each phase of Spiral Model is divided into four quadrants as shown in the above figure. The functions of these four quadrants are discussed below-

1. **Objectives determination and identify alternative solutions:**

   Requirements are gathered from the customers and the objectives are identified, elaborated and analyzed at the start of every phase. Then alternative solutions possible for the phase are proposed in this quadrant.

2. **Identify and resolve Risks:**

During the second quadrant all the possible solutions are evaluated to select the best possible solution. Then the risks associated with that solution is identified and the risks are resolved using the best possible strategy. At the end of this quadrant, Prototype is built for the best possible solution.

3. **Develop next version of the Product:**
   During the third quadrant, the identified features are developed and verified through testing. At the end of the third quadrant, the next version of the software is available.
4. **Review and plan for the next Phase:**
   In the fourth quadrant, the Customers evaluate the so far developed version of the software. In the end, planning for the next phase is started.

Risk Handling in Spiral Model

A risk is any adverse situation that might affect the successful completion of a software project. The most important feature of the spiral model is handling these unknown risks after the project has started. Such risk resolutions are easier done by developing a prototype. The spiral model supports coping up with risks by providing the scope to build a prototype at every phase of the software development.

Prototyping Model also support risk handling, but the risks must be identified completely before the start of the development work of the project. But in real life project risk may occur after the development work starts, in that case, we cannot use Prototyping Model. In each phase of the Spiral Model, the features of the product dated and analyzed and the risks at that point of time are identified and are resolved through prototyping. Thus, this model is much more flexible compared to other SDLC models.

Why Spiral Model is called Meta Model ?

The Spiral model is called as a Meta Model because it subsumes all the other SDLC models. For example, a single loop spiral actually represents the Iterative Waterfall Model. The spiral model incorporates the stepwise approach of the Classical Waterfall Model. The spiral model uses the approach of **Prototyping Model** by building a prototype at the start of each phase as a risk handling technique. Also, the spiral model can be considered as supporting the evolutionary model – the iterations along the spiral can be considered as evolutionary levels through which the complete system is built.

Advantages of Spiral Model:

Below are some of the advantages of the Spiral Model.

Risk Handling: The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling at every phase.

Good for large projects: It is recommended to use the Spiral Model in large and complex projects.

Flexibility in Requirements: Change requests in the Requirements at later phase can be incorporated accurately by using this model.

Customer Satisfaction: Customer can see the development of the product at the early phase of the software development and thus, they habituated with the system by using it before completion of the total product.

Disadvantages of Spiral Model:

Below are some of the main disadvantages of the spiral model.

Complex: The Spiral Model is much more complex than other SDLC models.

Expensive: Spiral Model is not suitable for small projects as it is expensive.

Too much dependable on Risk Analysis:

The successful completion of the project is very much dependent on Risk Analysis. Without very highly experienced expertise, it is going to be a failure to develop a project using this model.

Difficulty in time management:

As the number of phases is unknown at the start of the project, so time estimation is very difficult.

Comparison of different life cycle models

Classical Waterfall Model: The Classical Waterfall model can be considered as the basic model and all other life cycle models are based on this model. It is an ideal model. However, the Classical Waterfall model cannot be used in practical project development, since this model does not support any mechanism to correct the errors that are committed during any of the phases but detected at a later phase. This problem is overcome by the Iterative Waterfall model through the inclusion of feedback paths.

Iterative Waterfall Model: The Iterative Waterfall model is probably the most used software development model. This model is simple to use and understand. But this model is suitable only for well-understood problems and is not suitable for the development of very large projects and projects that suffer from a large number of risks.

Prototyping Model: The Prototyping model is suitable for projects, which either the customer requirements or the technical solutions are not well understood. This risks must be identified before the project starts. This model is especially popular for the development of the user interface part of the project.

Evolutionary Model: The Evolutionary model is suitable for large projects which can be decomposed into a set of modules for incremental development and delivery. This model is widely used in object-oriented development projects. This model is only used if incremental delivery of the system is acceptable to the customer.

Spiral Model: The Spiral model is considered as a meta-model as it includes all other life cycle models. Flexibility and risk handling are the main characteristics of this model. The spiral model is suitable for the development of technically challenging and large software that is prone to various risks that are difficult to anticipate at the start of the project. But this model is more complex than the other models.

Agile Model: The Agile model was designed to incorporate change requests quickly. In this model, requirements are decomposed into small parts that can be incrementally developed. But the main principle of the Agile model is to deliver an increment to the customer after each Time-box. The end date of an iteration is fixed, it can’t be extended. This agility is achieved by removing unnecessary activities that waste time and effort.

Module 2

Software project Management

Introduction

Effective project management is crucial to the success of any software development project. 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.

The job responsibilities of a project manager ranges from invisible activities like building up of team morale to highly visible customer presentations. Most managers take responsibilities for 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. These activities are certainly numerous and varied. The activities of project manager can be broadly classified into two major types—

1. project planning
2. project monitoring and control.

Three skills that are most critical to successful project management are the following:

- Knowledge of project management techniques.
- Decision taking capabilities.
- Previous experience in managing similar projects.

PROJECT PLANNING

Once a project has been found to be feasible, software project managers undertake project planning. Initial project planning is undertaken and completed before any development activity starts.

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays

can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. For this reason, project planning is undertaken by the project managers with utmost care and attention. For effective project planning, in addition to a thorough knowledge of the various estimation techniques, past experience is crucial. During project planning, the project manager performs the following activities. Note that the brief description of the activities.

Estimation:

The following project 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?

The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

Scheduling:

After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.

Staffing:

Staff organisation and staffing plans are made.

Risk management:

This includes risk identification, analysis, and planning.

Miscellaneous plans:

This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

Figure shows the order in which the planning activities are undertaken. Observe that size estimation is the first activity that a project manager undertakes during project planning. Size is the most fundamental parameter, based on which all other estimations and project plans are made. As can be seen from Figure , based on the

size estimation, the effort required to complete a project and the duration over which the development is to be carried out are estimated. Based on the effort estimation, the cost of the project is computed. The estimated cost forms the basis on which price negotiations with the customer is carried out. Other planning activities such as staffing, scheduling etc. are undertaken based on the effort and duration estimates made.

the staffing and scheduling issues.

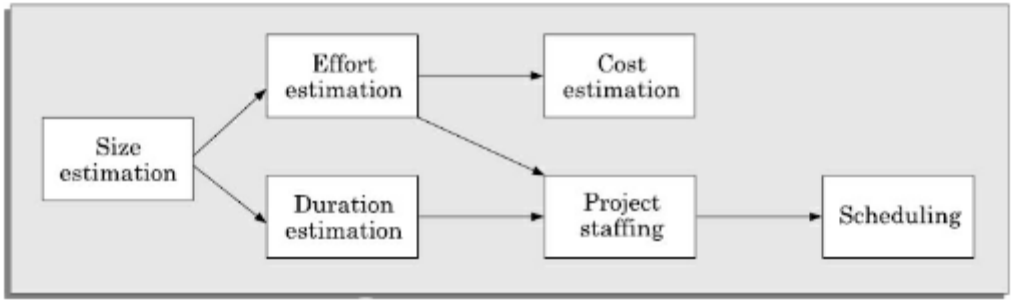


FIGURE Precedence ordering among planning activities.

Sliding Window Planning

It is usually very difficult to make accurate plans for large projects at project initiation. A part of the difficulty arises from the fact that large projects may take several years to complete. As a result, during the span of the project, the project parameters, scope of the project, project staff, etc., often change drastically resulting in the initial plans. In order to overcome this problem, sometimes project managers undertake project planning over several stages. That is, after the initial project plans have been made, these are revised at frequent intervals. 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. At the start of a project, the project manager has incomplete knowledge about the project. His information base gradually improves as the project progresses through different development phases. The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear. By taking these

developments into account, the project manager can plan the subsequent activities more accurately and with increasing levels of confidence.

The SPMP Document of Project Planning

Once project planning is complete, projectmanagers document their plans in a software project management plan (SPMP) document.

METRICS FOR PROJECT SIZE ESTIMATION

The size of a project is not the number of bytes that the source code occupies, neither is it the size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Software metrics can be classified into three categories −

- **Product metrics** − Describes the characteristics of the product such as size, complexity, design features, performance, and quality level.
- **Process metrics** − These characteristics can be used to improve the development and maintenance activities of the software.
- **Project metrics** − This metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity.

Currently, two metrics are popularly being used to measure size—lines of code (LOC) and function point (FP). Each of these metrics has its own advantages and disadvantages.

Lines of Code (LOC)

LOC is possibly the simplest among all metrics available to measure project size. Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored. Determining the LOC count at the end of a project is very simple. However, accurate estimation of LOC count at the beginning of a project is a very difficult task.

shortcomings of the LOC metric in the following subsections:

1. *LOC is a measure of coding activity alone.*

   A good problem size measure should consider the total effort needed to carry out various life cycle activities (i.e. specification, design, code, test, etc.) and not just the coding effort. The implicit assumption made by the LOC metric that the overall product development effort is solely determined from the coding effort alone is flawed.
2. *LOC count depends on the choice of specific instructions:*

   LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers. By coding style, we mean

the choice of code layout, the choice of the instructions in writing the program, and the specific algorithms used. Even for the same programing problem, different programmers might come up with programs having very different LOC counts. This situation does not improve, even if language tokens are counted instead of lines of code.

3. *Larger code size does not necessarily imply better quality of code or higher efficiency.*

   Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms. Calculating productivity as LOC generated per man-month may encourage programmers to write lots of poor-quality code rather than fewer lines of high-quality code achieve the same functionality.

4. *LOC metric penalises use of higher-level programming languages and code reuse:*

   A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size, and in turn, would indicate lower effort! Thus, if managers use the LOC count to measure the effort put in by different developers (that is, their productivity), they would be discouraging code reuse by developers.

5. *LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities:*

   Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic.

6. *It is very difficult to accurately estimate LOC of the final program from problem specification:*

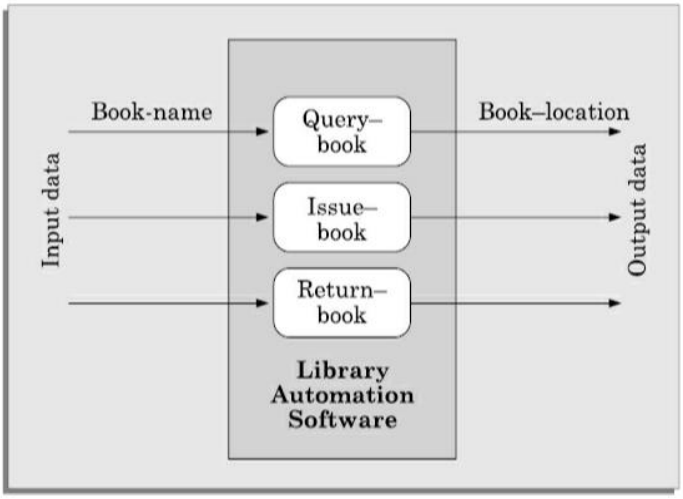
   At the project initiation time, it is a very difficult task to accurately estimate the number of lines of code (LOC) that the program would have after development. The LOC count can accurately be computed only after the code has fully been developed. Since project planning is carried out even before any development activity starts, the LOC metric is of little use to the project managers during project planning.

Function Point (FP) Metric

Function Point (FP) Metric overcomes many of the shortcomings of the LOC metric. Function point metric has several advantages over LOC metric. One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself. Using the LOC metric, on the other hand, the size can accurately be determined only after

the code has been fully written. Conceptually, the function point metric is based on the idea that a software product supporting many features would certainly be of larger size than a product with less number of features.

The conceptual idea behind the function point metric is the following. The size of a software product is directly dependent on the number of different high-level functions or features it supports. This assumption is reasonable, since each feature would take additional effort to implement.



System function as a mapping of input data to output data.

For example, the query book feature (see Figure 3.2) of a Library Automation Software takes the name of the book as input and displays its location in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data.

In addition to the number of basic functions that a software performs, its size also depends on the number of files and the number of interfaces associated with it. Here, interfaces refer to the different mechanisms for data transfer with external systems including the interfaces with the user, interfaces with external computers, etc.

Function point (FP) metric computation

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

Step 1: Compute the unadjusted function point (UFP) using a heuristic expression.

Step 2: Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.

Step 3: Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

We discuss these three steps in more detail in the following.

Step 1: UFP computation

The **unadjusted function points (UFP)** is computed as the weighted sum of five characteristics of a product as shown in the following expression. The weights associated with the five characteristics were determined empirically through data gathered from many projects.

UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

Number of inputs: Each data item input by the user is counted. However, it should be noted that data inputs are considered different from user inquiries. It needs to be further noted that individual data items input by the user are not simply added up to compute the number of inputs, but related inputs are grouped and considered as a single input.

Number of outputs: The outputs considered include reports printed, screen outputs, error messages produced, etc.

Number of inquiries: An inquiry is a user command (without any data input) and only requires some actions to be performed by the system.

Number of files: The files referred to here are logical files. A logical file represents a group of logically related data. Logical files include data structures as well as physical files.

Number of interfaces: Here the interfaces denote the different mechanisms that are used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems, etc.

Step 2: Refine parameters

UFP computed at the end of step 1 is a gross indicator of the problem size. This UFP needs to be refined by taking into account various peculiarities of the project. This is possible, since each parameter (input, output, etc.) has been implicitly assumed to be of average complexity. The weights for the different parameters are determined based on the numerical values shown in Table

TABLE 3.1 Refinement of Function Point Entities

Type	*Simple*	*Average*	*Complex*
Input (I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

Step 3: Refine UFP based on complexity of the overall project

Technical complexity factor (TCF) for the project is computed as

$$\mathbf{TCF = (0.65 + 0.01 \times DI).}$$

Where **DI is degree of influence**. There are 14 parameters that can influence the development effort. Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence).

As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.49.

Finally, FP is given as the product of UFP and TCF.

$$\mathbf{FP = UFP \times TCF.}$$

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, since these not only help in quoting an appropriate project cost to the customer, but also form the basis for resource planning and scheduling.

These can 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. We shall discuss two such formalisations of the basic empirical estimation techniques known as **expert judgement and the Delphi techniques**

i) Expert Judgement

Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analysing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. The outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant experience and knowledge of all aspects of a project.

A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimized when the estimation is done by a group of experts.

ii) Delphi Cost Estimation

Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a co-ordinator. In this approach, the co-ordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the co-ordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The co-ordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds.

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.

$$\textbf{Estimated Parameter} = c1 \times ed1$$

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 is an example of 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 × pd1 + c2 × pd2 + … 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 intermediate COCOMO is an example of multivariable cost estimation model.

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.

COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

COnstructive COst estimation MOdel (COCOMO) was proposed by Boehm. COCOMO prescribes a three stage process for project estimation. In the first stage, an initial estimate is arrived at. Over the next two stages, the initial estimate is refined to arrive at a more accurate estimate. COCOMO uses both single and multivariable estimation models at different stages of estimation. The three stages of COCOMO estimation technique are—

basic COCOMO,

intermediate COCOMO, and

complete COCOMO.

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity—

organic,

semidetached,

embedded.

Based on the category of a software development project, he gave different sets of formulas to estimate the effort and duration from the size estimate.

Organic:

We can classify a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

Semidetached:

A development project can be classify to be of semidetached type, if the development team consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

Embedded:

A development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent

regulations on the operational procedures exist. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

The basic COCOMO model

The basic COCOMO model is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms:

$$\text{Effort} = a1 \times (\text{KLOC})a2 \text{ PM}$$

$$\text{Tdev} = b1 \times (\text{Effort})b2 \text{ months}$$

- where, KLOC is the estimated size of the software
- product expressed in Kilo Lines Of Code.
- a1, a2, b1, b2 are constants for each category of software product.
- Tdev is the estimated time to develop the software, expressed in months.
- Effort is the total effort required to develop the software product, expressed in person-months (PMs).

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a1, a2, b1, b2 for different categories of products as given by Boehm. He derived these values by examining historical data collected from a large number of actual projects.

Estimation of development effort: For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

$$\begin{aligned}
&\text{Organic} && : \text{Effort} = 2.4(\text{KLOC})^{1.05} \text{ PM} \\
&\text{Semi-detached} && : \text{Effort} = 3.0(\text{KLOC})^{1.12} \text{ PM} \\
&\text{Embedded} && : \text{Effort} = 3.6(\text{KLOC})^{1.20} \text{ PM}
\end{aligned}$$

Estimation of development time: For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

$$\begin{aligned}
&\text{Organic} && : \text{Tdev} = 2.5(\text{Effort})^{0.38} \text{ Months} \\
&\text{Semi-detached} && : \text{Tdev} = 2.5(\text{Effort})^{0.35} \text{ Months} \\
&\text{Embedded} && : \text{Tdev} = 2.5(\text{Effort})^{0.32} \text{ Months}
\end{aligned}$$

Problem

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of a software developer is `15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.

Solution:

From the basic COCOMO estimation formula for organic software:

Effort = $2.4 \times (32)^{1.05}$

= 91 PM

Nominal development time = $2.5 \times (91)^{0.38}$

= 14 months

Staff cost required to develop the product = $91 \times$ `15,000

= `1,465,000

RISK MANAGEMENT

A risk is any anticipated unfavourable event or circumstance that can occur while a project is underway.

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

Risk Containment

1. **Risk Identification**

Risk Identification The project manager needs to anticipate the risks in a project as early as possible. As soon as a risk is identified, effective risk management plans are made, so that the possible impact of the risks is minimised. So, early risk identification is important.

A project can be subjected to a large variety of risks. In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorise risks into different classes. The project manager can then examine which risks from each class are relevant to the project. There are three main categories of risks which can affect a software project:

Project risks,

Technical risks,

Business risks.

i)Project risks:

Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very 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.

ii)Technical risks:

Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Most technical risks occur due the development team's insufficient knowledge about the product.

iii)Business risks:

This type of risks includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

2.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 (r). The consequence of the problems associated with that risk (s). Based on these two factors, the priority of each risk can be computed as follows:

$p = r \times s$ where, p is the priority with which the risk

3.Risk Containment:

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:

Avoid the risk:

Risks can be avoided in several ways such as Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.

Transfer the risk:

This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.

Risk reduction:

This involves planning ways to contain the damage due to a risk. The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.

Requirement Analysis and Specification

The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.

Requirements analysis and specification activity is usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather and analyse customer requirements and then write the requirements specification document are known as system analysts in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyse the collected data to conceptualise what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness. They then proceed to write the software requirements specification (SRS) document.

The SRS document is the final outcome of the requirements analysis and specification phase.

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

- Requirements gathering and analysis
- Requirements specification

Requirements gathering and analysis

The requirements have to be systematically gathered by the analyst from several sources in bits and pieces. These gathered requirements need to be analysed to remove several types of problems that frequently occur in the requirements that have been gathered piecemeal from different sources. We can conceptually divide the requirements gathering and analysis activity into two separate tasks: Requirements gathering

- Requirements analysis
- Requirements Gathering

Requirements gathering

Requirements gathering activity is also popularly 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 is concerned with the software.

Important ways in which an experienced analyst gathers requirements:

1. *Studying existing documentation:*
   The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the analyst.
2. *Interview:*
   Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each.
3. *Task analysis:*
   A service supported by a software is also called a task. Task analysis helps the analyst to understand the nitty-gritty of various user tasks and to represent each task as a hierarchy of subtasks.
4. *Scenario analysis:*
   A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different scenarios of a task, the behavior of the software can be different.
5. *Form analysis:*
   Form analysis is an important and effective requirements gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system.

Requirements Analysis

The main purpose of the requirements analysis activity is to analyse 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.

During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Anomaly:

It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible.

Inconsistency:

Two requirements are said to be inconsistent, if one of the requirements contradicts the other.

Incompleteness:

An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much later, possibly while using the software.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

The SRS document usually contains all the user requirements in a structured though an informal form. All the documents produced during a software development life cycle, SRS document is probably the most important document and is the toughest to write.

Users of SRS Document

Usually, a large number of different people need the SRS document for very different purposes. Some of the important categories of users of the SRS document and their needs for use are as follows:

- Users, customers, and marketing personnel:
- Software developers:
- Test engineers:
- User documentation writers:
- Project managers:
- Maintenance engineers:

The important uses of a well-formulated SRS document:

Forms an agreement between the customers and the developers:

A good SRS document sets the stage for the customers to form their expectation about the software and the developers about what is expected from the software.

Reduces future reworks:

Careful review of the SRS document can reveal omissions, misunderstandings, and inconsistencies early in the development cycle.

Provides a basis for estimating costs and schedules:

Project managers usually estimate the size of the software from an analysis of the SRS document.

Provides a baseline for validation and verification:

The SRS document provides a baseline against which compliance of the developed software can be checked. It is also used by the test engineers to create the test plan.

Facilitates future extensions:

The SRS document usually serves as a basis for planning future enhancements.

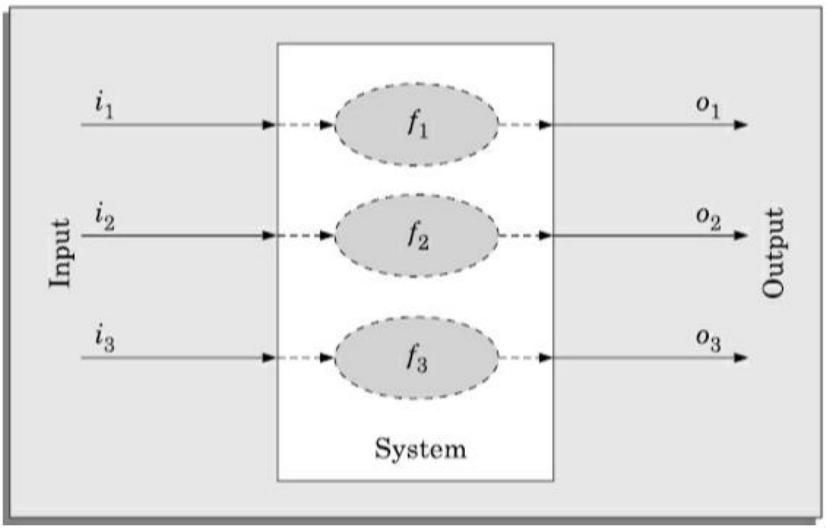
Characteristics of a Good SRS Document

Some of the identified desirable qualities of an SRS document are the following:
Concise:

The 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.

Implementation-independent:

The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements. It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the externally visible behavior of the system and not discuss the implementation issues. This view with which a requirement specification is written, has been shown in Figure



Traceable:

It should be possible to trace a specific requirement to the design elements that implement it and vice versa.

Modifiable:

Customers frequently change the requirements during the software development due to a variety of reasons. Therefore, in practice the SRS document undergoes several revisions during software development.

Identification of response to undesired events:

The SRS document should discuss the system responses to various undesired events and exceptional conditions that may arise.

Verifiable:

All requirements of the system as documented in the SRS document should be verifiable.

Attributes of Bad SRS Documents

Over-specification:

It occurs when the analyst tries to address the "how to" aspects in the SRS document.

Forward references:

One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.

Wishful thinking:

This type of problems concern description of aspects which would be difficult to implement.

Noise:

The term noise refers to presence of material not directly relevant to the software development process.

An SRS document should clearly document the following aspects of a software:

- Functional requirements
- Non-functional requirements
  - Design and implementation constraints
  - External interfaces required
  - Other non-functional requirements
- Goals of implementation.

Software Design

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

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 identified. Each module is a collection of functions and the data shared by these functions. 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.

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 identifies 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.

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.

Characteristics of good software design

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. This is an important requirement, since change requests usually keep coming from the customer even after product release.

COHESION AND COUPLING

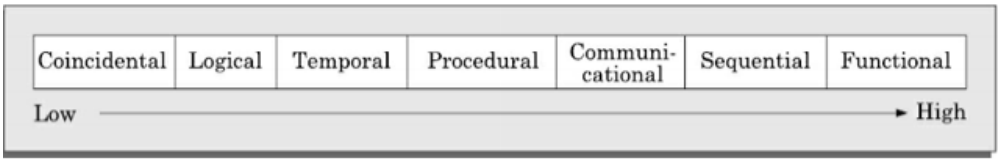
Cohesion is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

Cohesion

Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective. The different modules of a design can possess different degrees of freedom.

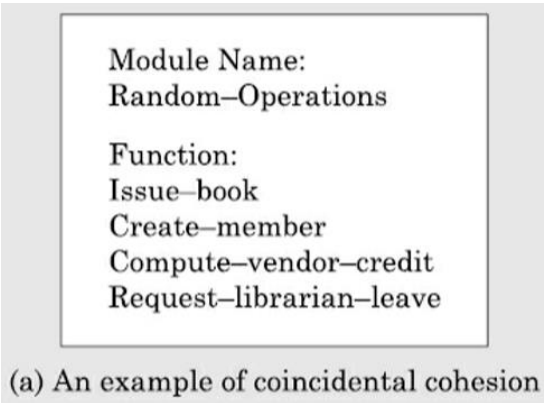
Different classes of cohesion are

1. Coincidental cohesion
2. Logical cohesion
3. Temporal cohesion
4. Procedural cohesion
5. Communicational cohesion
6. Sequential cohesion
7. Functional cohesion



Coincidental cohesion:

A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions. design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily.



(a) An example of coincidental cohesion

An example of a module with coincidental cohesion has been shown in Figure. Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

Logical cohesion:

A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc. As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.

Temporal cohesion:

When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion. As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.

Procedural cohesion:

A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data. Consider the activities associated with order processing in a trading house. The functions login(), place-order(), check-order(), print-bill(), place-order-on-vendor(), update-inventory(), and logout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.

Communicational cohesion:

A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admitStudent, enterMarks, printGradeSheet, etc. access and manipulate data stored in an array named studentRecords defined within the module.

Sequential cohesion:

A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.

Functional cohesion:

A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., computeOvertime(), computeWorkHours(), computeDeductions(), etc.) work together to generate the payslips of the employees.

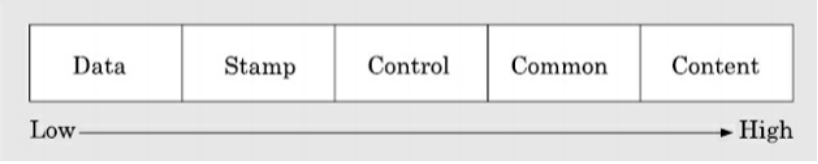
Coupling

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled.

The degree of coupling between two modules depends on their interface complexity. The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.

Classification of coupling.

1. Data coupling
2. Stamp coupling:
3. Control coupling:
4. Common coupling
5. Content coupling:



Data coupling:

Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.

Stamp coupling:

Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.

Control coupling:

Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling:

Two modules are common coupled, if they share some global data items.

Content coupling:

Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.

The degree of coupling increases from data coupling to content coupling. High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

Function Oriented Design

Function-oriented design view a system as a black-box that provides a set of services to the users of the software. These services provided by a software (e.g., issue book, serach book, etc.,)for a Library Automation Software to its users are also known as the high-level functions supported by the software.

During the design process, these high-level functions are successively decomposed into more detailed functions. After top-down decomposition has been carried out, the different identified functions are mapped to modules and a module structure is created.

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

SA/SD METHODOLOGY

The SA/SD technique can be used to perform the high-level design of a software.

SD METHODOLOGY

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

Structured analysis (SA)

Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have shown in Figure .

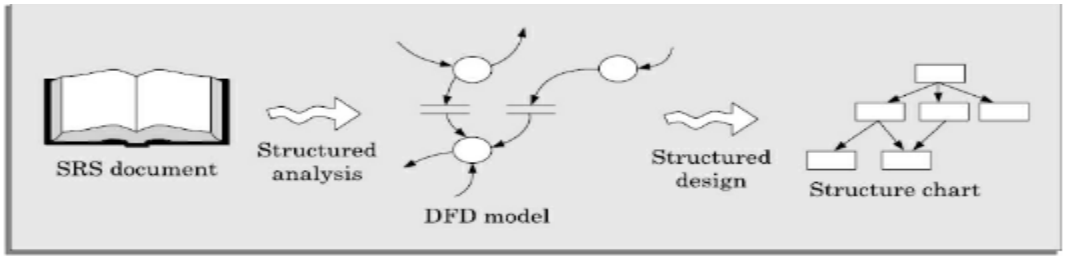


FIGURE Structured analysis and structured design methodology.

Observe the following from the figure:

- 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.

As shown in Figure , 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. That is, each function that the system needs to perform is analysed and hierarchically decomposed into more detailed functions. On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the high-level design or the software architecture for the given problem. This is represented using a structure chart.

STRUCTURED ANALYSIS

During structured analysis, the major processing tasks (high-level functions) of the system are analysed, and the data flow among these processing tasks are represented graphically.

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).

DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. It completely ignores aspects such as control flow, the specific algorithms used by the functions, etc. In the DFD terminology, each function is called a process or a bubble. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.

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.

Data Flow Diagrams (DFDs)

- The DFD (also known as the bubble chart) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.
- It is simple to understand and use.
- A DFD model uses a very limited number of primitive symbols (shown in Figure) to represent the functions performed by a system and the data flow among these functions.
- Abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy.

Different concepts associated with building a DFD model of a system.

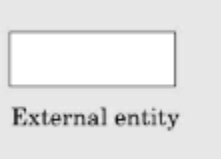
Primitive symbols used for constructing DFDs

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

Function symbol:

A function is represented using a circle. This symbol is called a process or a bubble. Bubbles are anno**tated with the names of the corresponding functions.**

External entity symbol:

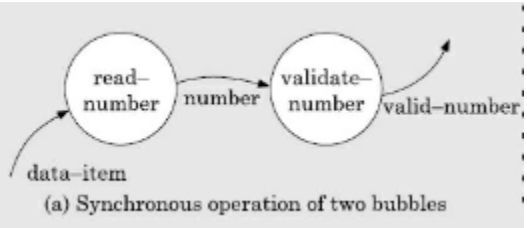


An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.

Data flow symbol:

A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example the DFD in

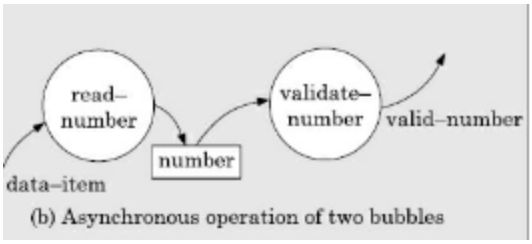
Figure shows three data flows—the data item number flowing from the process read-number to validate-number, data-item flowing into read-number, and valid-number flowing out of validate-number.



(a) Synchronous operation of two bubbles

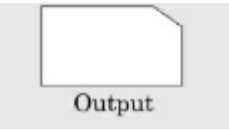
Data store symbol:

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. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. As an example of a data store, number is a data store in Figure .



(b) Asynchronous operation of two bubbles

Output symbol:



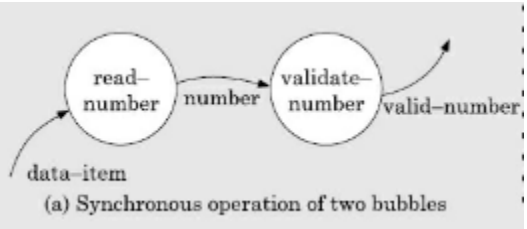
The output symbol is as shown in Figure . The output symbol is used when a hard copy is produced. The data store may look like a box with one end open.

Important concepts associated with constructing DFD models

Synchronous and asynchronous operations

Synchronous Operation

If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figure .

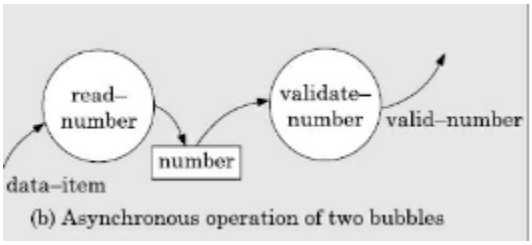


(a) Synchronous operation of two bubbles

Here, the validate-number bubble can start processing only after the read-number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.

Asynchronous operations

If two bubbles are connected through a data store, as in Figure then the speed of operation of the bubbles are independent.



(b) Asynchronous operation of two bubbles

The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

Data dictionary

Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1 DFD, level 2 DFDs.A single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.

For example, a data dictionary entry may represent that the data grossPay consists of the components regularPay and overtimePay.

grossPay = regularPay + overtimePay.

The dictionary plays a very important role in any software development process, especially for the following reasons:

- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project.
- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and vice versa.

Data definition

Composite data items can be defined in terms of primitive data items using the following data definition operators.

+ : denotes composition of two data items, e.g. a+b represents data a and b.

[,,] : represents selection, i.e. any one of the data items listed inside the square bracket can occur. For example, [a,b] represents either a occurs or b occurs.

() : the contents inside the bracket represent optional data which may or may not appear. a+(b) represents either a or a+ b occurs.

{} : represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.

= : represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.

/**/ : Anything appearing within /* and */ is considered as comment.

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 system is constructed by using a hierarchy of DFDs . **The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level).** It is the easiest to draw and understand. At each successive lower level DFDs, more and more details are gradually introduced.

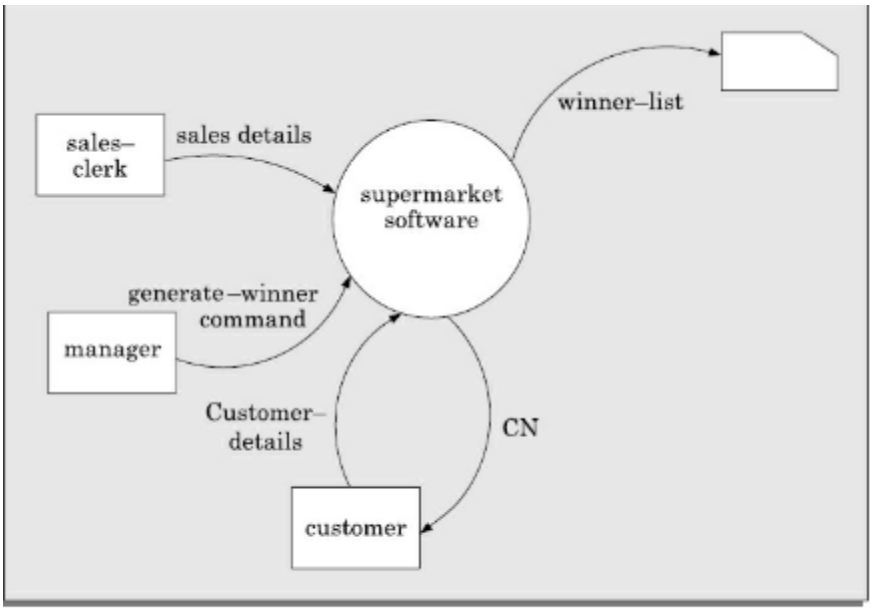
To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified. The DFD model of a problem consists of many DFDs and a single data dictionary.

To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model. All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

Context Diagram or Level 0 DFD

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). This is the only bubble in a DFD model, where a noun is used for naming the bubble. The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble.

. As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure).



Context diagram

The context diagram has been labelled as ‘Supermarket software’. 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 receive from the system. The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names. To develop the context diagram of the system, we have to analyse the SRS document to identify the different types of users who would be using the system and

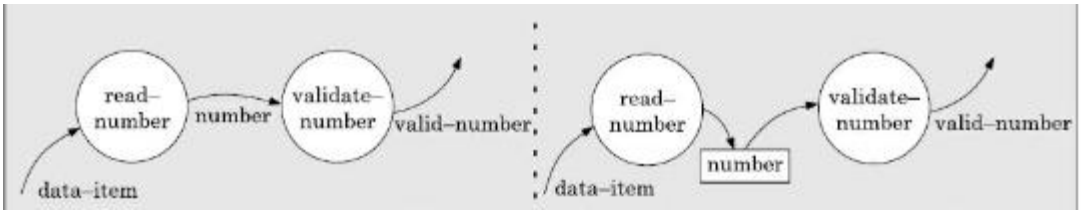
the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also includes any external systems

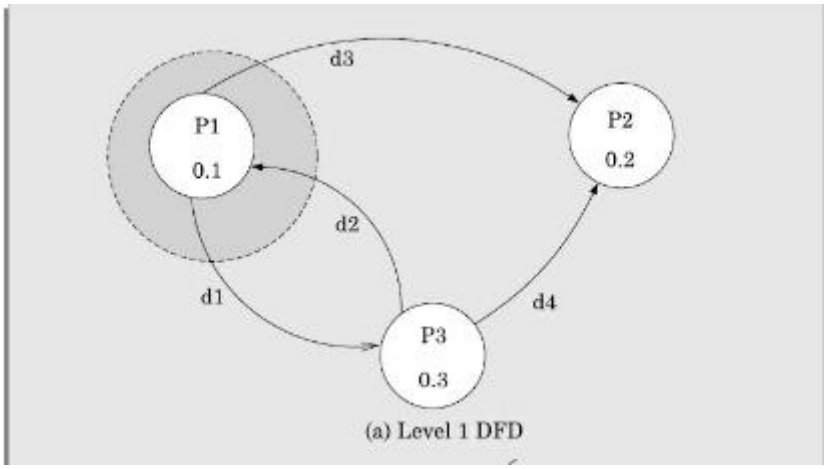
Level 1 DFD

The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high-level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

If a system has more than seven high-level requirements identified in the SRS document, some of the related requirements have to be combined and represented as a single bubble in the level 1 DFD. These can be split appropriately in the lower DFD levels.

We The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as **factoring or exploding a bubble.**



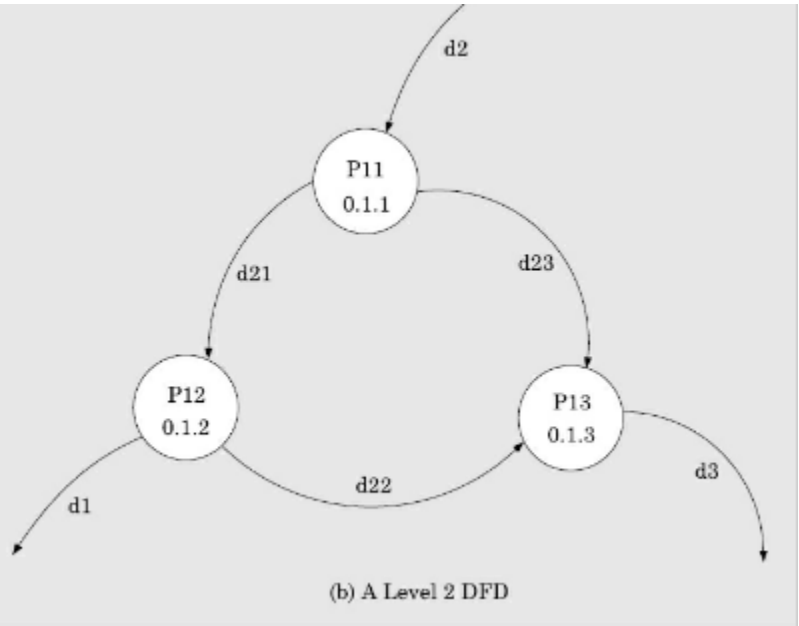


Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers help in uniquely identifying any bubble in the DFD from its bubble number. **The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubbles at level 1 are numbered, 0.1, 0.2, 0.3, etc.** In Figure above, the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1.

Level 2 DFD

2-level DFD goes one step deeper into parts of 1-level DFD. It can be used to plan or record the specific/necessary detail about the system's functioning.

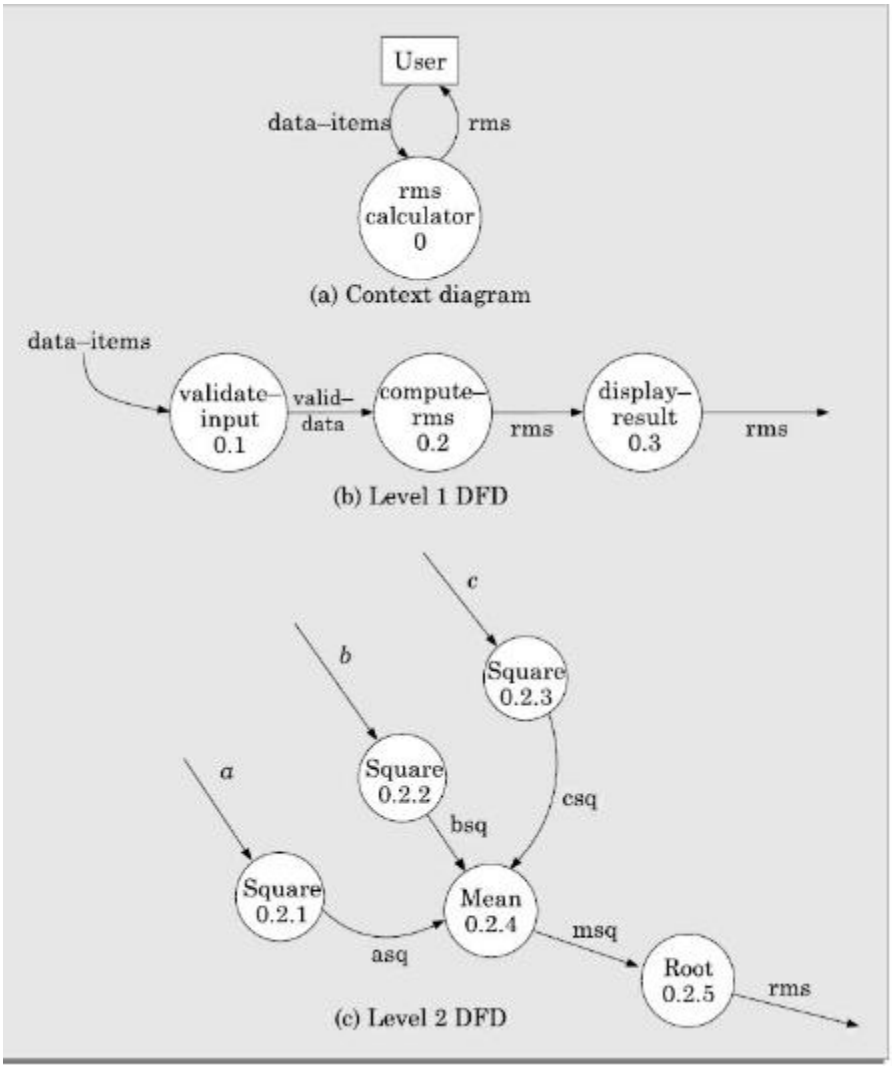


(b) A Level 2 DFD

In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1, 0.1.2, 0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in.

Example (RMS Calculating Software)

A software system called RMS calculating software would read three integral numbers from the user in the range of –1000 and +1000 and would determine the root mean square (RMS) of the three input numbers and display its LEVEL 0,LEVEL 1,LEVEL 2.



Data dictionary for the DFD model of Example

data-items: {integer}3

rms: float

valid-data:data-items

a: integer

b: integer

c: integer

asq: integer

bsq: integer

csq: integer

msq: integer

Shortcomings of the DFD model

DFD models suffer from several shortcomings. The important shortcomings of DFD models are the following:

- Imprecise DFDs.
- In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. e.g. what happens when some input information is missing or is incorrect.
- Not-well defined control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed.
- Further, many times it is not possible to say which DFD representation is superior or preferable to another one.
- Improper data flow diagram: The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart. A structure chart represents the software architecture. The various modules making up the system, the module dependency (i.e., which module calls which other modules), and the parameters that are passed among the different modules. The structure chart representation can

be easily implemented using some programming language. 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. 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 along side 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.

Selection:

The diamond symbol represents the fact that one module of several modules connected with the diamond symbol is 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.

In any structure chart, there should be one and only one module at the top, called the root. There should be at most one control relationship between any two modules

in the structure chart. This means that if module A invokes module B, module B cannot invoke module A.

The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules. However, it is possible for two higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design are shown in Figure.

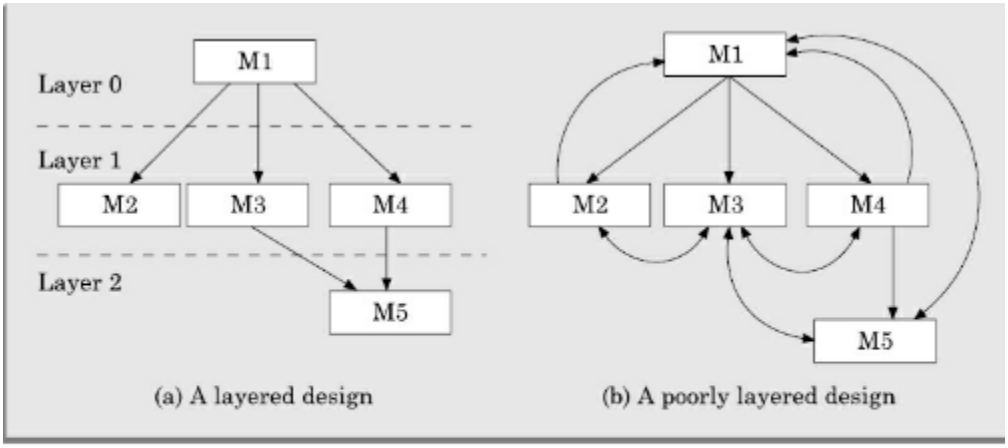
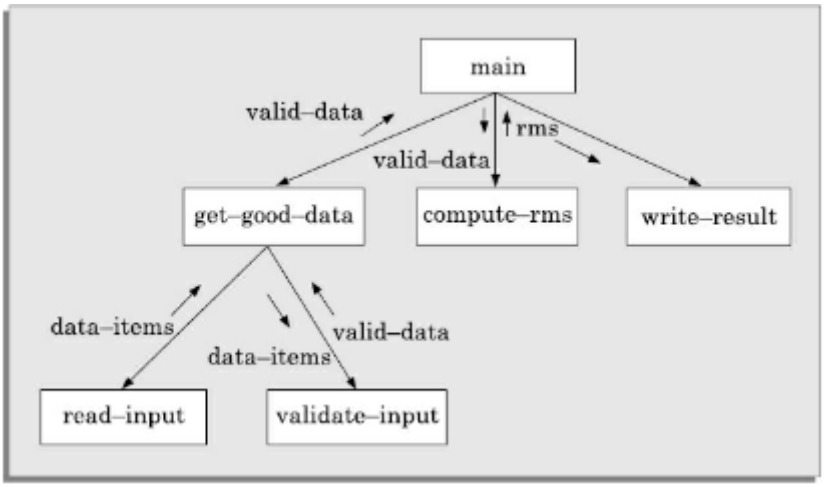


FIGURE Examples of properly and poorly layered designs.



USER INTERFACE DESIGN

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

CHARACTERISTICS OF A GOOD USER INTERFACE

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 memorise commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

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

Speed of recall:

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

Error prevention:

A good user interface should minimise 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 an average users while using the interface. Consistency of names, issue procedures, and behaviour of similar commands and the simplicity of the command issue procedures minimise error possibilities. Also, the interface should prevent the user from entering wrong values.

Aesthetic and attractive:

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

Consistency:

The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalise 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.

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.

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.

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 a software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

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.

TYPES OF USER INTERFACES

user interfaces can be classified into the following three categories:

Command language-based interfaces

Menu-based interfaces

Direct manipulation interfaces

Command Language-based Interface

A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands.

A simple command language-based interface might simply assign unique names to the different commands. A more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names one would have to remember. 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.

Advantages

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.

Disadvantages

Command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorise 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.

Menu-based Interface

An important advantage of a menu-based interface over a command language-based interface is that 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 recognising 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. However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible.

Some of the techniques available to structure a large number of menu items:

Scrolling menu: When a 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.

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.

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 organised 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 sub-menu to form a hierarchical tree-like structure.

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 recognised by the users very easily, and that icons are language-independent.

MODULE 4

CODING

The input to the coding phase is the design document produced at the end of the design phase. 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. Software development organisations usually formulate their own coding standards.

The main advantages of 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.

It is mandatory for the programmers to follow the coding standards.

Coding Standards and Guidelines

Coding standards and guidelines that are commonly adopted by many software development organisations,

coding standards

Rules for limiting the use of global:

Limit the data that needs to be defined with global scope.

Standard headers for different modules:

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 write-up about what the module does.
- Different functions supported in the module, along with their input/output
- parameters
- Global variables accessed/modified by the module

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).

Conventions regarding error return values and exception handling mechanisms:

For example, all functions while encountering an error condition should either return a 0 or 1.

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.

Do not use a coding style that is too clever or too difficult to understand:

Code should be easy to understand.

Do not use an identifier for multiple purposes:

Code should be well-documented:

Lengthy functions are likely to have disproportionately larger number of bugs.

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.

CODE REVIEW

Code review and testing are both effective defect removal mechanisms. That is, all the syntax errors have been eliminated from the module. Code review does not target to detect syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors. Code review has been recognised as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.

The two types of code review techniques. Normally, the following two types of reviews are carried out on the code of a module:

Code inspection.

Code walkthrough.

Code Walkthrough

Code walkthrough is an informal code analysis technique. In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand. That is, the reviewer mentally traces the execution through different statements and functions of the code. The main objective of code walkthrough is to discover the algorithmic and logical errors in the code. The members note down their findings of their code walkthrough and discuss those in a walkthrough meeting where the coder of the module is present. These guidelines are based on personal experience, common sense, and several other subjective factors.

Code Inspection

During code inspection, the code is examined to check for the presence of some common programming errors.

The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.

Such a list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.

The following is a checklist of some classical programming errors which can be used during code inspection:

Use of uninitialised variables

Jumps into loops

Non-terminating loops

Incompatible assignments

Array indices out of bounds

Improper storage allocation and deallocation

Mismatch between actual and formal parameter in procedure calls

Use of incorrect logical operators or incorrect precedence among operators

Cleanroom Technique

Cleanroom technique was pioneered at IBM. This technique relies heavily on walkthroughs, inspection, and formal verification for bug removal. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler.

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.

Good documents are helpful in several ways:

Good documents help to enhance understandability of a piece of code.

Code understanding is an important part of any maintenance activity.

Availability of good documents help to reduce the effort and time required for maintenance.

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.

good documents helps the manager to effectively track the progress of the project.

Different types of software documents can broadly be classified into the following

Internal documentation and External documentation.

Internal documentation:

These are provided in the source code itself.

Comments embedded in the source code

Use of meaningful variable names

Module and function headers

Use of enumerated types

Use of constant identifiers

Use of user-defined data types

External documentation:

These are the supporting documents such as SRS document, installation document, users' manual, design document, and test document.

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.

A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.

TESTING

The aim of program testing is to help in identifying all defects in a program. The careful testing can expose a large percentage of the defects existing in a program, and therefore, testing provides a practical way of reducing defects in a system.

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.

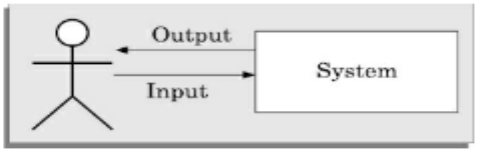


FIGURE 10.1 A simplified view of program testing.

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. When the system fails, it is necessary to note down the specific input values for which the failure occurs.

Terminologies

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

An error is the result of a mistake committed by a developer in any of the development activities. Mistakes can give rise to an extremely large variety of errors. One example error is a call made to a wrong function. The terms error, fault, bug, and defect are used interchangeably by the program testing community.

Testing Activities

Testing involves performing the following major activities:

Test suite design:

The test suite 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. A typical testing process in terms of the activities that are carried out has been shown schematically in Figure

As can be seen, the test cases are first designed. Subsequently, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected.

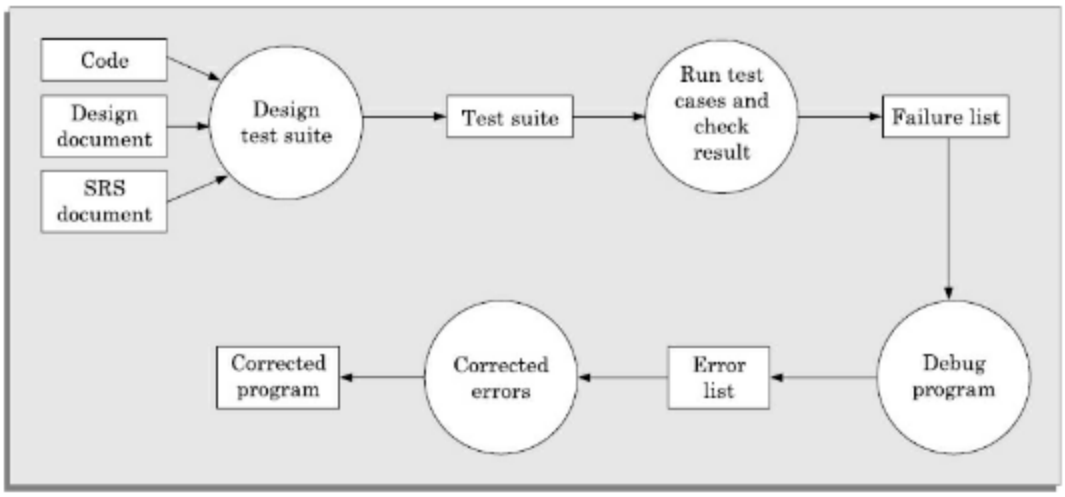


FIGURE 10.2 Testing process.

Unit Testing

Unit testing is undertaken after coding of a module is complete, all syntax errors have been removed.

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, provide the required global data, or are called by the module under test) .

stubs and drivers

They are designed to provide the complete environment for a module so that testing can be carried out.

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

Stub:

A stub module consists of several stub procedures that are called by the module under test. A stub procedure is a dummy procedure that takes the same parameters as the function called by the unit under test but has a highly simplified behavior. For example, a stub procedure may produce the expected behaviour using a simple table look up mechanism, rather than performing actual computations.

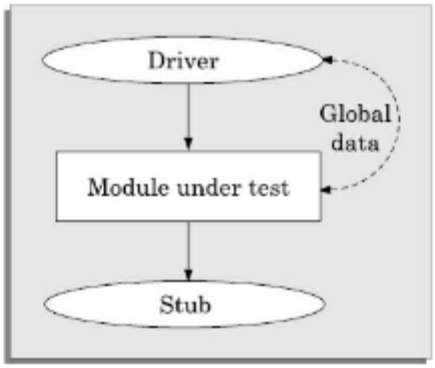


FIGURE 10.3 Unit testing with the help of driver and stub modules.

Driver:

A driver module contains the non-local data structures that are 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.

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.

B**lack Box Testing** is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.

How to do Black Box Testing

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether System processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the System is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

The following are the two main approaches available to design black box test cases:

Equivalence class partitioning

In the equivalence class partitioning approach, the domain of input values to the unit 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. Equivalence classes for a unit under test can be designed by examining the input data and output data.

Boundary value analysis

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}.

The important steps in the black-box test suite design approach:

Examine the input and output values of the program.

Identify the equivalence classes.

Design equivalence class test cases by picking one representative value from each equivalence class.

Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

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. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.

White Box Testing

is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security. In white box testing, code is visible to testers so it is also called Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output
- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

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. An example of a fault-based strategy is mutation testing, which is discussed later in this section.

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.

Statement Coverage

Statement coverage is a metric to measure the percentage of statements that are executed by a test suite in a program at least once.

Branch Coverage

Branch coverage is also called decision coverage (DC). It is also sometimes referred to as all edge coverage. A test suite achieves branch coverage, if it makes the decision expression in each branch in the program to assume both true and false values. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed. Branch testing is also known as all edge testing,

Condition Coverage

Condition coverage testing is also known as basic condition coverage (BCC) testing. A test suite is said to achieve basic condition coverage (BCC), if each basic condition in every conditional expression assumes both true and false values during testing. For example, for the following decision statement: if(A||B && C) …; the basic conditions A, B, and C assume both true and false values.

Condition and Decision Coverage

A test suite is said to achieve condition and decision coverage, if it achieves condition coverage as well as decision (that is, branch) coverage. Obviously, condition and decision coverage is stronger than both condition coverage and decision coverage.

Multiple Condition Coverage

Multiple condition coverage (MCC) is achieved, if the test cases make the component conditions of a composite conditional expression to assume all possible combinations of true and false values.

For example, consider the composite conditional expression [(c1 and c2) or c3]. A test suite would achieve MCC, if all the component conditions c1, c2, and c3 are each made to assume all combinations of true and false values.

Path Coverage

A test suite achieves path coverage if it exeutes each linearly independent paths (or basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control flow graph (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed.

Mutation Testing

mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program. In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed. After the initial testing is complete, mutation testing can be taken up. The idea behind mutation testing is to make a few arbitrary changes to a program at a time. These changes correspond to simple programming errors such as using an inappropriate operator in an arithmetic expression. Each time the program is changed, it is called a mutated program and the specific change effected is called a mutant. An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of several simple errors.

There are automated tools available to perform Code coverage analysis. Below are a few coverage analysis techniques a box tester can use:

Advantages of White Box Testing

- Code optimization by finding hidden errors.

- White box tests cases can be easily automated.
- Testing is more thorough as all code paths are usually covered.
- Testing can start early in SDLC even if GUI is not available.

Disadvantages of WhiteBox Testing

- White box testing can be quite complex and expensive.
- Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
- White box testing requires professional resources, with a detailed understanding of programming and implementation.
- White-box testing is time-consuming, bigger programming applications take the time to test fully.

Integration Testing

Integration testing is carried out after all (or at least some of) the modules have been unit tested. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters). For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module. Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module. The objective of integration testing is to check whether the different modules of a program interface with each other properly.

the following approaches can be used to develop the test plan:

Big-bang approach to integration testing

Top-down approach to integration testing

Bottom-up approach to integration testing

Mixed (also called sandwiched) approach to integration testing

Big-bang approach to integration testing

In this approach, all the modules making up a system are integrated in a single step. In simple words, all the unit tested modules of the system are simply linked together and tested.

Bottom-up approach to integration testing

A subsystem might consist of many modules which communicate among each other through well-defined interfaces. In bottom-up integration testing, first the modules for the each subsystem are integrated.

Top-down integration testing

It starts with the root module in the structure chart and one or two subordinate modules of the root module. After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested.

Mixed approach to integration testing

The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready.

SYSTEM TESTING

After all the units of a program have been integrated together and tested, system testing is taken up. System tests are designed to validate a fully developed system to assure that it meets its requirements.

There are three main kinds of system testing. These are essentially similar tests, but differ in who carries out the testing:

Alpha Testing:

Alpha testing refers to the system testing carried out by the test team within the developing organisation.

Beta Testing:

Beta testing is the system testing performed by a select group of friendly customers.

Acceptance Testing:

Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

Smoke Testing

For smoke testing, a few test cases are designed to check whether the basic functionalities are working. For example, for a library automation system, the smoke tests may check whether books can be created and deleted.

Performance Testing

Performance testing is carried out to check whether the system meets its non-functional requirements identified in the SRS document.

Stress testing

Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time. Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software.

Volume testing

Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.

Configuration testing

Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements.

Compatibility testing

This type of testing is required when the system interfaces with external systems (e.g., databases, servers, etc.). Compatibility aims to check whether the interfaces with the external systems are performing as required.

Regression testing

This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc.

Recovery testing

Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily.

Maintenance testing

This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system.

Documentation testing

It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent.

Usability testing

Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested.

Security testing

Security testing is essential for software that handle or process confidential data that is to be guarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers.

DEBUGGING

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed.

Debugging Approaches

The following are some of the approaches that are popularly adopted by the programmers for debugging:

Brute force method

This is the most common method of debugging but is the least efficient method. In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger),

Backtracking

This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.

Cause elimination method

In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is identified and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

Program slicing

This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements. The search space is reduced by defining slices. A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable .Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

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.

Intuitively, it is obvious that a software product having a large number of defects is unreliable. It is also very reasonable to assume that the reliability of a system improves, as the number of defects in it is reduced.

It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the instruction having the error is executed. If an error is removed from an instruction that is frequently executed (i.e., belonging to the core of the program), then this would show up as a large improvement to the reliability figure. On the other hand, removing errors from parts of the program that are rarely used, may not cause any appreciable change to the reliability of the product.

Reliability also depends upon how the product is used, or on its execution profile. If the users execute only those features of a program that are "correctly" implemented, none of the errors will be exposed and the perceived reliability of the product will be high.

For example, for a Library Automation Software the library members would use functionalities such as issue book, search book, etc., on the other hand the librarian would normally execute features such as create member, create book record, delete member record, etc. So defects which show up for the librarian, may not show up for the members.

Suppose the functions of a Library Automation Software which the library members use are error-free; and functions used by the Librarian have many bugs. Then, these two categories of users would have very different opinions about the reliability of the software. Therefore, the reliability figure of a software product is observer-dependent and it is very difficult to absolutely quantify the reliability of the product.

Software quality

the quality of a product is defined in terms of its fitness of purpose. That is, 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. That is, it correctly performs all the functions that have been specified in its SRS document. Even though it may be functionally correct, it may not be considered it to be a quality product, if it has an almost unusable user interface.

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.

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 analyse 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 summarising the effectiveness of the quality system in the organisation.

Software maintenance

Software maintenance denotes any changes made to a software product after it has been delivered to the customer. Maintenance is inevitable for almost any kind of product. software products need maintenance to correct errors, enhance features, port to new platforms, etc.

CHARACTERISTICS OF SOFTWARE MAINTENANCE

Software maintenance is becoming an important activity of a large number of organisations. 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 or the software needs to run over hand held devices. 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 to overcome the failures 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:

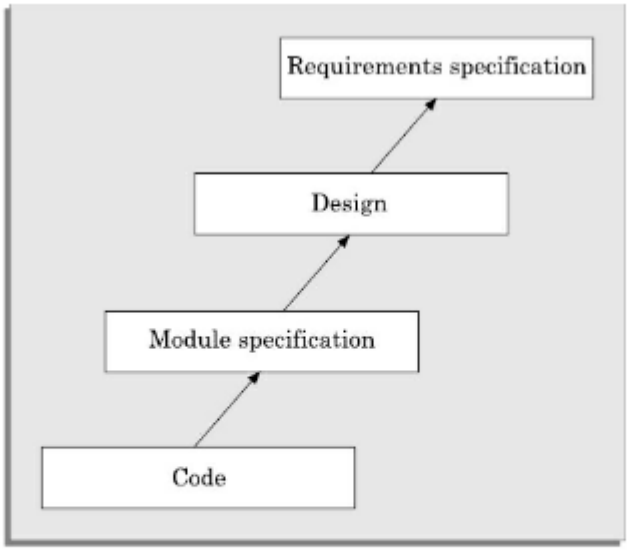
A software product needs maintenance to support any new features that the users may want it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

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.

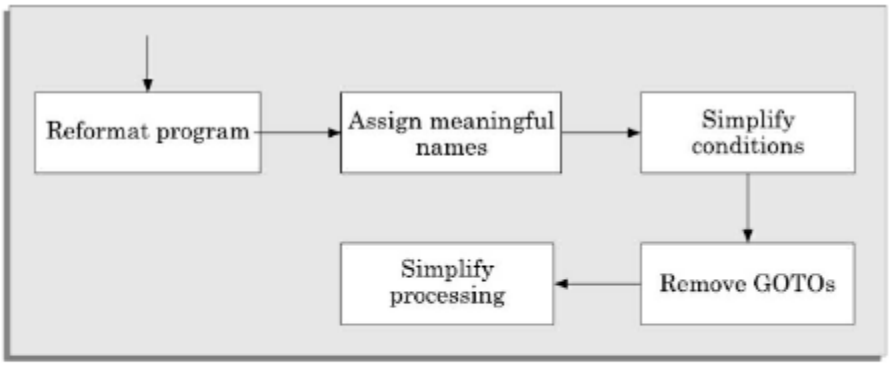
Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts. The first stage of reverse engineering usually focuses 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 .

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.



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.



Cosmetic changes carried out before reverse engineering.

The following software engineering trends are becoming noticeable:

- **Client-server software**
- **Service-oriented architecture (SOA)**
- **Software as a service (SaaS)**

CLIENT-SERVER SOFTWARE

In a client-server software, both clients and servers are essentially software components. A client is a consumer of services and a server is a provider of services.

Advantages of client-server software

There are many reasons for the popularity of client-server software. A few important reasons are as follows:

Concurrency:

A client-server software divides the computing work among many different client and server components that could be residing on different machines. Thus client-server solutions are inherently concurrent and as a result offer the advantage of faster processing.

Loose coupling:

Client and server components are inherently loosely-coupled, making these easy to understand and develop.

Flexibility.

A client-server software is flexible in the sense that clients and servers can be attached and removed as and when required. Also, clients can access the servers from anywhere.

Cost-effectiveness:

The client-server paradigm usually leads to cost-effective solutions. Clients usually run on cheap desktop computers, whereas severs may run on sophisticated and expensive computers. Even to use a sophisticated software, one needs to own only a cheap client machine to invoke the server.

Heterogeneous hardware:

In a client-server solution, it is easy to have specialised servers that can efficiently solve specific problems. It is possible to efficiently integrate heterogeneous computing platforms to support the requirements of different types of server software.

Fault-tolerance:

Client-server solutions are usually fault-tolerant. It is possible to have many servers providing the same service. If one server becomes unavailable, then client requests can be directed to any other working server.

Mobile computing:

Mobile computing implicitly requires uses of client-server technique. Cell phones are, of late, evolving as handheld computing and communicating devices and are being provided with small processing power, keyboard, small memory, and LCD display.

Application service provisioning:

There are many application software products that are extremely expensive to own. In this approach, an application service provider (ASP) would own it, and the users would pay the ASP based on the charges per unit time of usage.

Component-based development:

In the component paradigm, software development consists of integrating off-the-shelf software components and writing only the missing parts.

Disadvantages of client-server software

There are several disadvantages of client-server software development. The main disadvantages are:

Security:

For example, a client can connect to a server from anywhere. This makes it easy for hackers to break into the system. Therefore, ensuring security of a client-server system is a very challenging task.

Servers can be bottlenecks:

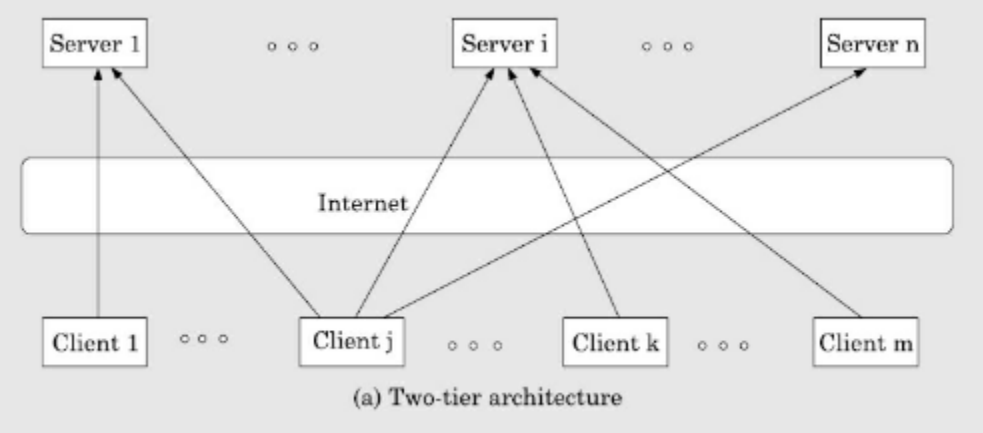
Servers can turn out to be bottlenecks because many clients might try to connect to a server at the same time.

Compatibility:

Clients and servers may not be compatible to each other. Since the client and server components may be manufactured by different vendors, they may not be compatible with respect to data types, languages, number representation, etc.

CLIENT-SERVER ARCHITECTURES

The simplest way to connect clients and servers is by using a two-tier architecture shown in Figure . In a two-tier architecture, any client can get service from any server by sending a request over the network.



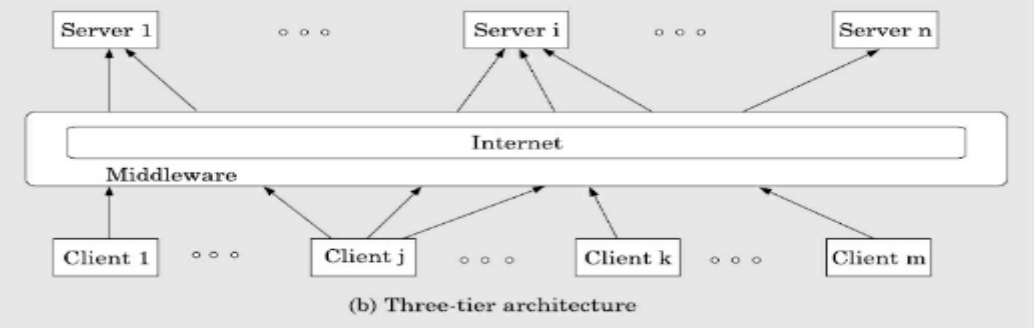
(a) Two-tier architecture

Limitations of two-tier client-server architecture

The main problem is that client and server components are usually manufactured by different vendors, who may adopt their own interfacing and implementation solutions. As a result, the different components may not interface with (talk to) each other easily.

Three-tier client-server architecture

The three-tier architecture overcomes the main limitations of the two-tier architecture. In the three-tier architecture, a middleware is added between client and the server components as shown in Figure . The middleware keeps track of all servers. It also translates client requests into server understandable form. For example, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer to the client.



(b) Three-tier architecture

Functions of middleware

The important activities of the middleware include the following:

The middleware keeps track of the addresses of servers. Based on a client request, it can therefore easily locate the required server.

It can translate between client and server formats of data and vice versa.

Two popular middleware standards are:

Common Object Request Broker Architecture (CORBA)

CORBA

Common object request broker architecture (CORBA) is a specification of a standard architecture for middleware. Using a CORBA implementation, a client can transparently invoke a service of a server object, which can be on the same machine or across a network. CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing;framing and error-handling,operation dispatching.

CORBA Reference Model

The CORBA reference model has been shown in Figure 15.2. In the following subsection, we briefly discuss the major components of the CORBA reference model.

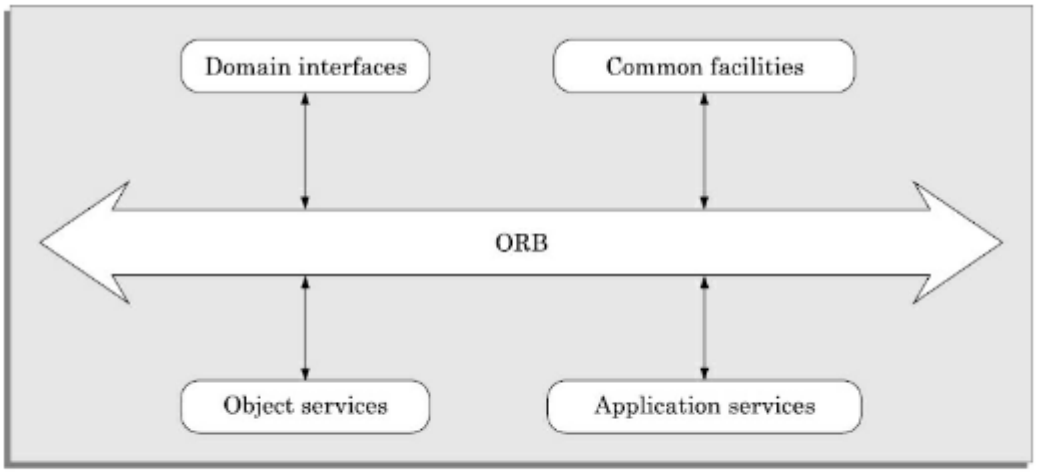


FIGURE 15.2 CORBA reference model.

ORB

ORB is also known as the object bus, since ORB supports communication among the different components attached to it. This is akin to a bus on a printed circuit board (PCB) on which

the different hardware components (ICS) communicate. The ORB handles client requests for any service, and is responsible for finding an object that can

implement the request, passing it the parameters, invoking its method, and returning the results of the invocation.

Domain interfaces

These interfaces provide services pertaining to specific application domains. Several domain services have been in use, including manufacturing, telecommunication, medical, and financial domains.

Object services

These are domain-independent interfaces that are used by many distributed object programs.

Two examples of object services that fulfill this role are the following:

Naming Service: This allows clients to find objects based on names. Naming service is also called white page service.

Trading Service: This allows clients to find objects based on their properties. Trading service is also called yellow page service. Using trading service a specific service can be searched. This is akin to searching a service such as automobile repair shop in a yellow page directory.

Common facilities

Like object service interfaces, these interfaces are also horizontally-oriented, but unlike object services they are oriented towards end-user applications.

An example of such a facility is the distributed document component facility (DDCF).

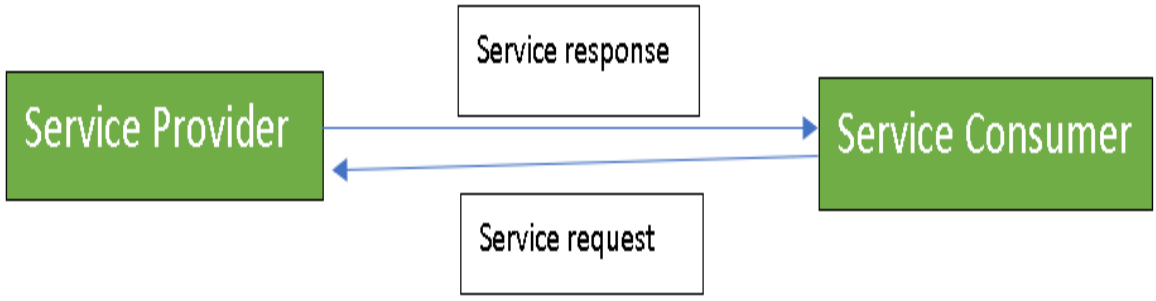
Service-Oriented Architecture

Service-Oriented Architecture (SOA) is an architectural approach in which applications make use of services available in the network.

In this architecture, services are provided to form applications, through a communication call over the internet.

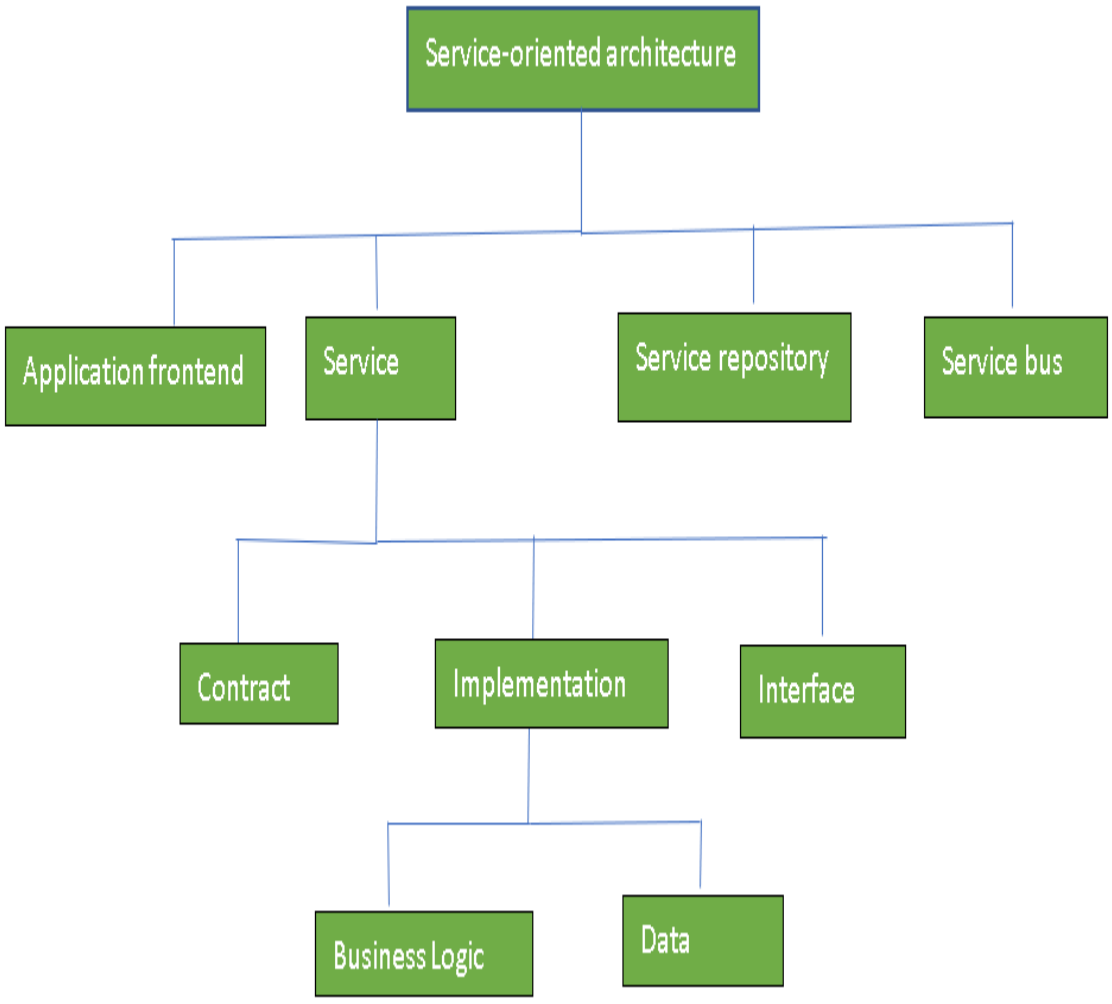
SOA allows users to combine a large number of facilities from existing services to form applications.
SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system.
SOA based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains.
There are two major roles within Service-oriented Architecture:

Service provider: The service provider is the maintainer of the service and the organization that makes available one or more services for others to use. To advertise services, the provider can publish them in a registry, together with a service contract that specifies the nature of the service, how to use it, the requirements for the service, and the fees charged.
Service consumer: The service consumer can locate the service metadata in the registry and develop the required client components to bind and use the service.



Services might aggregate information and data retrieved from other services or create workflows of services to satisfy the request of a given service consumer. This practice is known as **service orchestration** .Another important interaction pattern is service choreography, which is the coordinated interaction of services without a single point of control.

Components of SOA:



Principles of SOA:

Standardized service contract: Specified through one or more service description documents.
Loose coupling: Services are designed as self-contained components, maintain relationships that minimize dependencies on other services.
Abstraction: A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation.
Reusability: Designed as components, services can be reused more effectively, thus reducing development time and the associated costs.

Discoverability: Services are defined by description documents that constitute supplemental metadata through which they can be effectively discovered. Service discovery provides an effective means for utilizing third-party resources.
Composability: Using services as building blocks, sophisticated and complex operations can be implemented. Service orchestration and choreography provide solid support for composing services and achieving business goals.

Advantages of SOA:

Service reusability: In SOA, applications are made from existing services. Thus, services can be reused to make many applications.
Easy maintenance: As services are independent of each other they can be updated and modified easily without affecting other services.
Platform independent: SOA allows making a complex application by combining services picked from different sources, independent of the platform.
Availability: SOA facilities are easily available to anyone on request.
Reliability: SOA applications are more reliable because it is easy to debug small services rather than huge codes
Scalability: Services can run on different servers within an environment, this increases scalability

Disadvantages of SOA:

High investment: A huge initial investment is required for SOA.
Complex service management: When services interact they exchange messages to tasks. the number of messages may go in millions. It becomes a cumbersome task to handle a large number of messages.

Software as a Service | SaaS

SaaS is also known as "**On-Demand Software**". It is a software distribution model in which services are hosted by a cloud service provider. These services are available to end-users over the internet so, the end-users do not need to install any software on their devices to access these services.

There are the following services provided by SaaS providers -

Business Services - SaaS Provider provides various business services to start-up the business. The SaaS business services include **ERP** (Enterprise Resource Planning), **CRM** (Customer Relationship Management), **billing**, and **sales**.

Document **Management** - SaaS document management is a software application offered by a third party (SaaS providers) to create, manage, and track electronic documents.

Example: Slack, Samepage, Box, and Zoho Forms.

Social Networks - As we all know, social networking sites are used by the general public, so social networking service providers use SaaS for their convenience and handle the general public's information.

Mail Services - To handle the unpredictable number of users and load on e-mail services, many e-mail providers offering their services using SaaS.

Advantages of SaaS cloud computing layer

SaaS is easy to buy

SaaS pricing is based on a monthly fee or annual fee subscription, so it allows organizations to access business functionality at a low cost, which is less than licensed applications.

Less hardware required for SaaS

The software is hosted remotely, so organizations do not need to invest in additional hardware.

Low maintenance required for SaaS

Software as a service removes the need for installation, set-up, and daily maintenance for the organizations. The initial set-up cost for SaaS is typically less than the enterprise software.

No special software or hardware versions required

All users will have the same version of the software and typically access it through the web browser. SaaS reduces IT support costs by outsourcing hardware and software maintenance and support to the IaaS provider.

Disadvantages of SaaS cloud computing layer

Security

Actually, data is stored in the cloud, so security may be an issue for some users. However, cloud computing is not more secure than in-house deployment.

Latency issue

Since data and applications are stored in the cloud at a variable distance from the end-user, there is a possibility that there may be greater latency when interacting with the application compared to local deployment. **Total Dependency on Internet**

Without an internet connection, most SaaS applications are not usable.

Switching between SaaS vendors is difficult

Switching SaaS vendors involves the difficult and slow task of transferring the very large data files over the internet and then converting and importing them into another SaaS also.