Publisher First Edition 2022 , NJSER , ISSN : 2582-5836

Dr. Manjeet Singh
Assistant Professor
Sainik Institute of Management and Technology,
Bathinda

NJESR

NATIONAL JOURNAL OF ENVIRONMENT AND SCIENTIFIC RESEARCH

Published by:

National Journal of Environment and Scientific Research

8-G-6 Jawahar Nagar Sriganganagar-335001Rajasthan

Mobile: 9780181083 8209583526 (Editor-in-Chief)

Email : editor@njesr.com

PREFACE

This book is an introduction to the art of software engineering. It is intended as a textbook for an undergraduate level course.

Software Engineering is about teams and it is about quality. The problems to solve are so complex or large, that a single developer cannot solve them anymore. Software engineering is also about communication on a team and with internal and external stakeholders. Teams do not consist only of developers, but also of quality assurance testers, systems architects, system/platform engineers, customers, project managers and other stakeholders.

Implementation is no longer just writing code, but it is also following guidelines, writing documentation and also writing unit tests. But unit tests alone are not enough. The different pieces have to fit together. I have tried to maintain it error free. Suggestions are welcome for further improvements.

I dedicate this book to my father Shri Angrej Singh and thank almighty God for empowering me to bring to a reality a dream of my father for his son to write a solo authored book for him students .I ‘m indebted to my partner Mrs. Sharandeep Kaur and all family members including my daughter to bring this book to existence. I also extend my thanks to the Chief Editor of NJESR,Dr. Brijesh Pathak for his support.

CONTENTS

Chapter-1
Basic Issues in Software Engineering

Scope and necessity of software engineering

Software engineering is an engineering approach for software development. We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively. These definitions can be elaborated with the help of a building construction analogy.

Suppose you have a friend who asked you to build a small wall as shown in fig.
1.1. You would be able to do that using your common sense. You will get buildingmaterials like bricks; cement etc. and you will then build the wall.

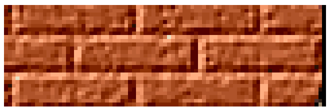


Fig.: A Small Wall

But what would happen if the same friend asked you to build a large multistoried building as shown in fig. ?

Fig. : A Multistoried Building

You don't have a very good idea about building such a huge complex. It would bevery difficult to extend your idea about a small wall construction into constructing a large building. Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the strength ofmaterials, testing, planning, architectural design, etc. Building a small wall and building a large building are entirely different ball games. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineeringprinciples.

Without using software engineering principles it would be difficult to develop largeprograms. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such largecommercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes as shown in fig. . For example, a program of size 1,000 lines of code has some complexity. But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn outto be 100 times more difficult unless software engineering principles are used. In such situations software engineering techniques come to rescue. Software engineering helps to reduce the programming complexity. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

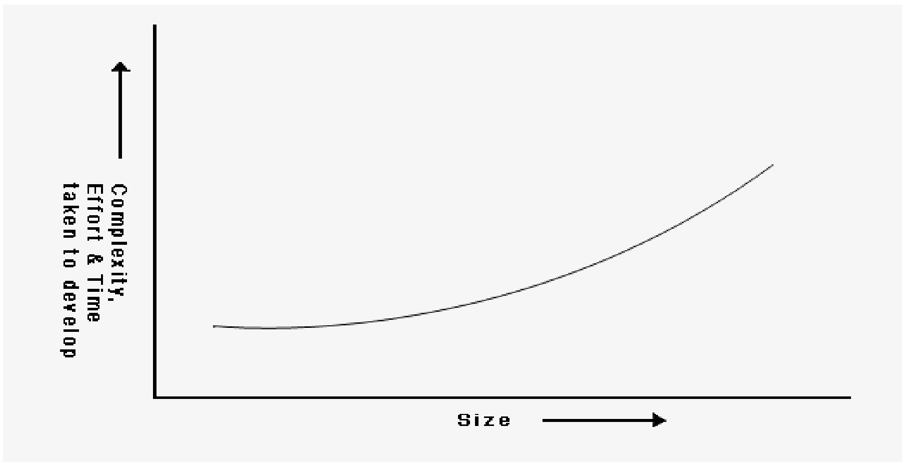


Fig.: Increase in development time and effort with problem size

The principle of abstraction (in fig.) implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction isa powerful way of reducing the complexity of the problem. The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem as shown in fig. should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

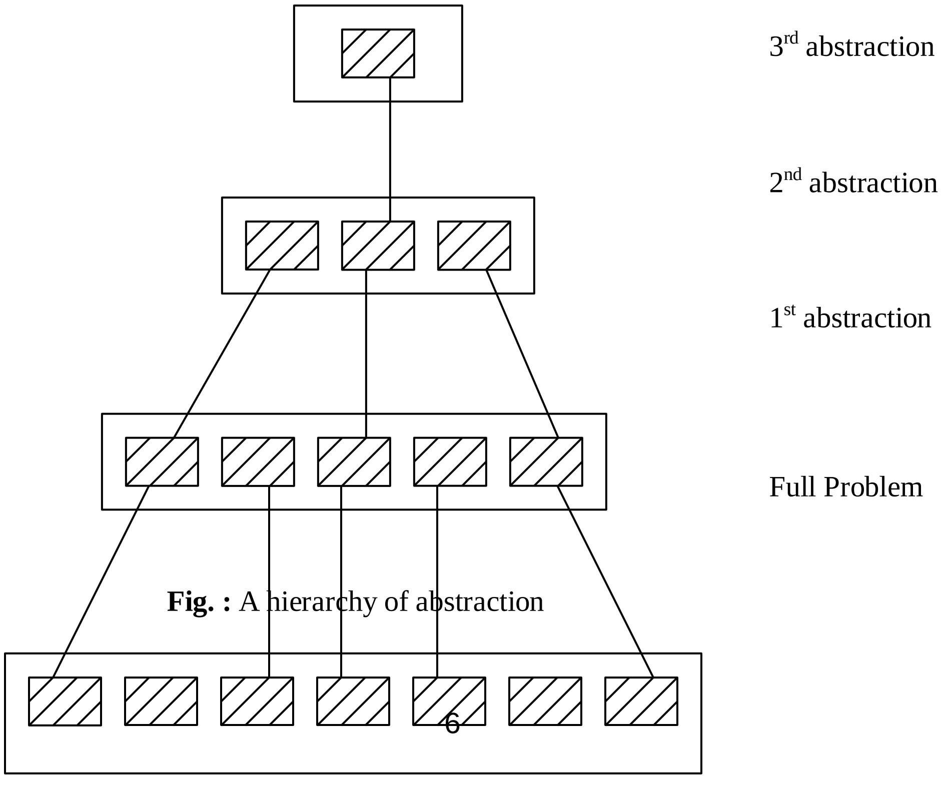


Fig. : A hierarchy of abstraction

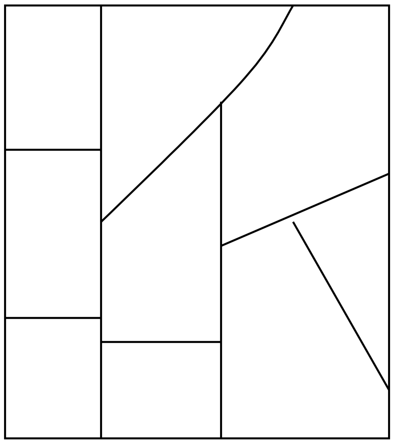


Fig. : Decomposition of a large problem into a set of smaller problems.

Causes of and solutions for software crisis.

Software engineering appears to be among the few options available to tackle the present software crisis. To explain the present software crisis in simple words, consider the following. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years (as shown in fig.)

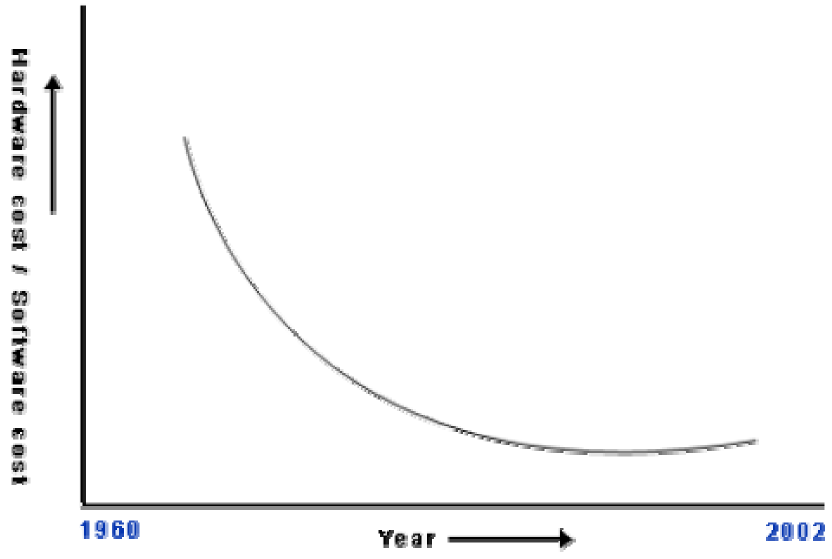


Fig. : Change in the relative cost of hardware and software over time

Organizations are spending larger and larger portions of their budget on software. Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources non- optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis. Remember that the cost we are talking of here is not on account of increased features, but due to ineffective development of the product characterized by inefficient resource usage, and time and cost over-runs.

There are many factors that have contributed to the making of the presentsoftware crisis. Factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity improvements.

It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself.

Program vs. software product

Programs are developed by individuals for their personal use. They are therefore,small in size and have limited functionality but software products are extremely large. In case of a program, the programmer himself is the sole user but on the other hand, in case of a software product, most users are not involved with the development. In case of a program, a single developer is involved but in case ofa software product, a large number of developers are involved. For a program, the user interface may not be very important, because the programmer is thesole user. On the other hand, for a software product, user interface must be carefully designed and implemented because developers of that product and users of that product are totally different. In case of a program, very little documentation is expected, but a software product must be well documented. A program can be developed according to the programmer’s individual style of development, but a software product must be developed using the accepted software engineering principles.

Important features of a structured program.

A structured program uses three types of program constructs i.e. selection,sequence and iteration. Structured programs avoid unstructured control flows by restricting the use of GOTO statements. A structured program consists of a well partitioned set of modules. Structured programming uses single entry, single-exit program constructs such as if-then-else, do-while, etc. Thus, the structured programming principle emphasizes designing neat control structures for programs.

Important advantages of structured programming.

Structured programs are easier to read and understand. Structured programs areeasier to maintain. They require less effort and time for development. They are amenable to easier debugging and usually fewer errors are made in the courseof writing such programs.

Evolution of software design techniques over the last 50years.

During the 1950s, most programs were being written in assembly language. These programs were limited to about a few hundreds of lines of assembly code,
i.e. were very small in size. Every programmer developed programs in his own individual style - based on his intuition. This type of programming was called Exploratory Programming.

The next significant development which occurred during early 1960s in thearea computer programming was the high-level language programming. Use of high-level language programming reduced development efforts and development time significantly. Languages like FORTRAN, ALGOL, and COBOL were introduced at that time.

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, but alsoto understand and maintain programs written by others. To cope with this problem, experienced programmers advised other programmers to pay particular attention to the design of the program’s control flow structure (in late 1960s). In the late 1960s, it was found that the "GOTO" statement was the main culprit which makes control structure of a program complicated and messy. At that time most of the programmers used assembly languages extensively. They considered use of "GOTO" statements in high-level languages were very natural because of their familiarity with JUMP statements which are very frequently used in assembly language programming. So they did not really accept that they can write programs without using GOTO statements, and considered the frequentuse of GOTO statements inevitable. At this time, **Dijkstra [1968]** published his (now famous) article “GOTO Statements Considered Harmful”. Expectedly, many programmers were enraged to read this article. They published several counter articles highlighting the advantages and inevitably of GOTO statements. But, soon it was conclusively proved that only three programming constructs – sequence, selection, and iteration – were sufficient to express any programming logic. This formed the basis of the structured programming methodology.

After structured programming, the next important development was data structure-oriented design. Programmers argued that for writing a good program, it is important to pay more attention to the design of data structure, of the program rather than to the design of its control structure. Data structure- oriented design techniques actually help to derive program structure from thedata structure of the program. Example of a very popular data structure-oriented design technique is Jackson's Structured Programming (JSP) methodology, developed by Michael Jackson in the1970s.

Next significant development in the late 1970s was the developmentof data flow-oriented design technique. Experienced programmers stated that to have a good program structure, one has to study how the data flows from input tothe output of the program. Every program reads data and then processes that data to produce some output. Once the data flow structure is identified, then fromthere one can derive the program structure.

Object-oriented design (1980s) is the latest and very widely used technique. It has an intuitively appealing design approach in which natural objects (such as employees, pay-roll register, etc.) occurring in a problem arefirst identified. Relationships among objects (such as composition, reference and inheritance) are determined. Each object essentially acts as a data hiding entity.

Exploratory style vs. modern style of software development.

An important difference is that the exploratory software development style is based on error correction while the software engineering principles are primarily based on error prevention. Inherent in the software engineering principles is the realization that it is much more cost-effective to prevent errors from occurringthan to correct them as and when they are detected. Even when errors occur, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible. In the exploratory style, errors are detected only during the final product testing. In contrast, the modern practiceof software development is to develop the software through several well-defined stages such as requirements specification, design, coding, testing, etc., andattempts are made to detect and fix as many errors as possible in the same phase in which they occur.

In the exploratory style, coding was considered synonymous with software development. For instance, exploratory programming style believed in developinga working system as quickly as possible and then successively modifying it until itperformed satisfactorily.

In the modern software development style, coding is regarded as only a small part of the overall software development activities. There are several development activities such as design and testing which typically require much more effort than coding.

A lot of attention is being paid to requirements specification. Significant effort is now being devoted to develop a clear specification of the problem before any development activity is started. Now there is a distinct design phase where standard design techniques are employed.

Periodic reviews are being carried out during all stages of the development process. The main objective of carrying out reviews is phase containment of errors, i.e. detect and correct errors as soon as possible. Defects are usually not detected as soon as they occur, rather they are noticed much later in the life cycle. Once a defect is detected, we have to go back to the phase where it was introduced and rework those phases - possibly change the design or change the code and so on.

Today, software testing has become very systematic and standard testing techniques are available. Testing activity has also become all encompassing in the sense that test cases are being developed right from the requirements specification stage. There is better visibility of design and code. By visibility we mean production of good quality, consistent and standard documents during every phase. In thepast, very little attention was paid to producing good quality and consistent documents. In the exploratory style, the design and test activities, even if carried out (in whatever way), were not documented satisfactorily. Today, consciously good quality documents are being developed during product development. This has made fault diagnosis and maintenance smoother.

Chapter-2
Basics of Software LifeCycle and Waterfall Model

Life cycle model

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out. For example, the design phase might consist of the structured analysis activity followed by the structured design activity.

The need for a software life cycle model

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. Fromthen on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models (such as classical waterfall model, iterative waterfall model, prototyping model, evolutionary model, spiral model etc.) it becomes difficult for software project managers to monitor the progress of the project.

Different software life cycle models

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonlyused life cycle models are as follows:

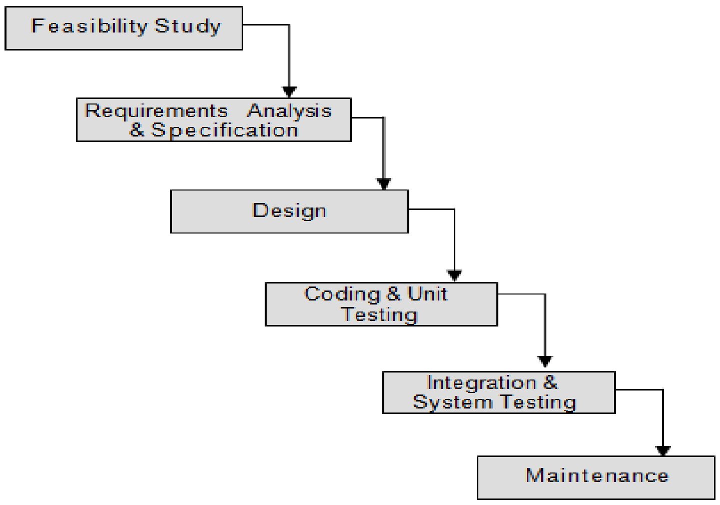
- Classical Waterfall Model
- Iterative Waterfall Model
- Prototyping Model
- Evolutionary Model
- Spiral Model

Different phases of the classical waterfall model

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious,it is not a practical model in the sense that it can not be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derivedfrom the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model.

Classical waterfall model divides the life cycle into the following phases asshown in fig.:

- Feasibility Study
- Requirements Analysis and Specification
- Design
- Coding and Unit Testing
- Integration and System Testing
- Maintenance


Feasibility Study
Requirements Analysis & Specification
Design
Coding & Unit Testing
Integration & System Testing
Maintenance

Activities in each phase of the life cycle

- **Activities undertaken during feasibility study: -**

- The main aim of feasibility study is to determine whether it would befinancially and technically feasible to develop the product.

○ At first project managers or team leaders try to have a roughunderstanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

○ After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what wouldbe the development time for each solution.

○ Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in thearea of development.

The following is an example of a feasibility study undertaken by an organization. It is intended to give you a feel of the activities and issues involved in the feasibility study phase of a typical software project.

Case Study

A mining company named Galaxy Mining Company Ltd. (GMC) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of mines at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident

fund (SPF) would be quickly distribute some compensation before the standard provident amount is paid. According to this scheme, each mine site would deduct SPF installments from each miner every month and deposit the same with the CSPFC (Central Special Provident Fund Commissioner). The CSPFC will maintain all details regarding the SPF installments collected from the miners. GMC employed a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. GMC realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. GMC indicated that the amount it can afford for thissoftware to be developed and installed is Rs. 1 million.
Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed the matter with the top managers of GMC to get an overview of the project. He also discussedthe issues involved with the several field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One was tohave a central database which could be accessed and updated via a satellite connection to various mine sites. The other approach was to have local databases at each mine site and to update the central database periodically through a dial-up connection. These periodicupdates could be done on a daily or hourly basis depending on the delay acceptable to GMC in invoking various functions of the software. The project manager found that the second approach was very affordable and more fault-tolerant as the local mine sites could still operate even when the communication link to the central database temporarily failed. The project manager quickly analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

Shortcomings of the classical waterfall model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of thelife cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

Phase-entry and phase-exit criteria of each phase

At the starting of the feasibility study, project managers or team leaders try to understand what is the actual problem by visiting the client side. At the end of that phase they pick the best solution and determine whether the solution is feasible financially and technically.
At the starting of requirements analysis and specification phase the required data is collected. After that requirement specification is carried out. Finally, SRS document is produced.
At the starting of design phase, context diagram and different levels of DFDs are produced according to the SRS document. At the end of this phase module structure (structure chart) is produced.
During the coding phase each module (independently compilation unit) of the design is coded. Then each module is tested independently as a stand-alone unit and debugged separately. After this each module is documented individually. The end product of the implementation phase is a set of

program modules that have been tested individually but not tested together.
After the implementation phase, different modules which have been tested individually are integrated in a planned manner. After all the modules have been successfully integrated and tested, system testing is carried out.
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. However, most products need maintenance due to the wear and tear caused by use.

Prototype

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer’s needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and itsprototyping model.
Another reason for developing a prototype is that it is impossible to get theperfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

Examples for prototype model

A prototype of the actual product is preferred in situations such as:

- user requirements are not complete
- technical issues are not clear

Let’s see an example for each of the above category.
Example 1: User requirements are not complete

In any application software like billing in a retail shop, accounting in a firm, etc the users of the software are not clear about the different functionalities required. Once they are provided with the prototype implementation, they can try to use it and find out the missing functionalities.

Example 2: Technical issues are not clear

Suppose a project involves writing a compiler and the development team has never written a compiler.

In such a case, the team can consider a simple language, try to build a compiler in order to check the issues that arise in the process and resolve them. After successfully building a small compiler (prototype), they would extend it to one that supports a complete language.

Spiral model

The Spiral model of software development is shown in fig. . The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study. The next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. . The following activities are carried out during each phase of a spiral model.

- **First quadrant (Objective Setting)**
  - During the first quadrant, it is needed to identify the objectives ofthe phase.
  - Examine the risks associated with these objectives.
- **Second Quadrant (Risk Assessment and Reduction)**
  - A detailed analysis is carried out for each identified project risk.
  - Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may bedeveloped.

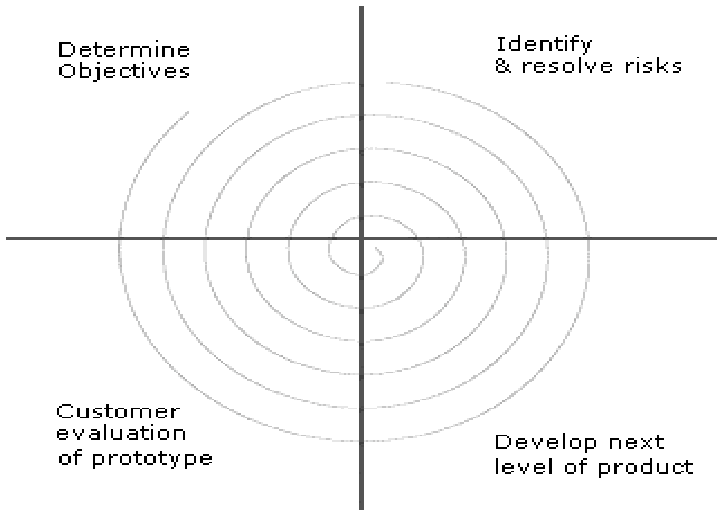


Fig. : Spiral Model

- **Third Quadrant (Development and Validation)**
  - Develop and validate the next level of the product after resolving the identified risks.
- **Fourth Quadrant (Review and Planning)**
  - Review the results achieved so far with the customer and plan the next iteration around the spiral.
  - Progressively more complete version of the software gets built with each iteration around the spiral.

Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

Comparison of different life-cycle models

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model can not be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However, this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery.This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other lifecycle models. Risk handling is inherently built into this model. The spiral model issuitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development teamis usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint,incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

Structured Analysis

Structured analysis is used to carry out the top-down decomposition of a set of high-level functions depicted in the problem description and to represent them graphically. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system performs is analyzed and hierarchically decomposed into more detailed functions. Structured analysis technique is based on the following essential underlying principles:

- Top-down decomposition approach.
- Divide and conquer principle. Each function is decomposed independently.
- Graphical representation of the analysis results using Data Flow Diagrams(DFDs).

Data Flow Diagram (DFD)

The DFD (also known as a bubble chart) 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 these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system. A DFD model uses a very limited number of primitive symbols [as shown in to represent the functions performed by a system and the data flow among these functions.

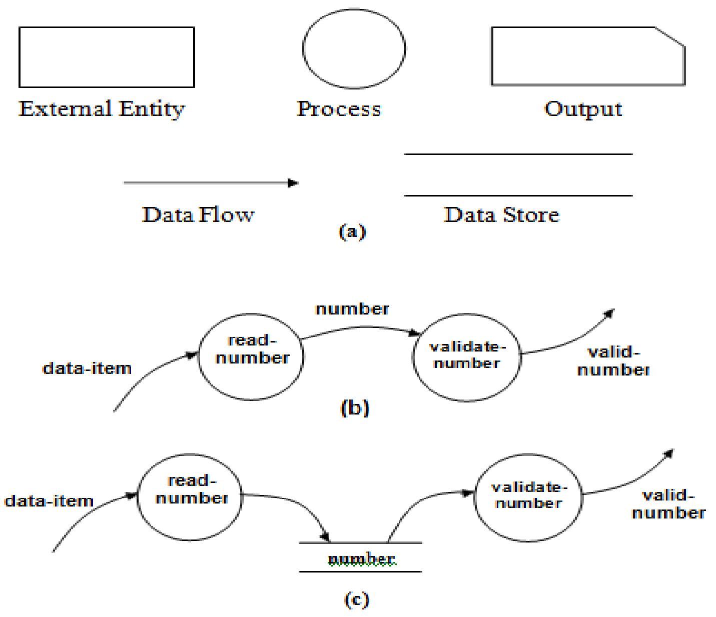


Fig. (a) Symbols used for designing DFDs (b) **, (c)** Synchronous and asynchronous data flow

Here, two examples of data flow that describe input and validation of data are considered. In Fig. 5.1(b), the two processes are directly connected by a data flow. This means that the 'validate-number' process can start only after the 'read-number' process had supplied data to it. However in Fig 5.1(c), the two processes are connected through a data store. Hence, the operations of the two bubbles are independent. The first one is termed 'synchronous' and the second one 'asynchronous'.

Context diagram

The context diagram is the most abstract data flow representation of a system. It represents the entire system as a single bubble. This bubble is labeled according to the main function of the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are also represented. 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. The name 'context diagram' is well justified because it represents the context in which the system is to exist, i.e. the external entities who would interact with the system and the specific data items they would be supplying the system and the data items they would be receiving from the system. The context diagram is alsocalled as the level 0 DFD.

To develop the context diagram of the system, it is required to analyze 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 the system. Here, the term "users of the system" also includes the external systems which supply data to or receive data from the system.

The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is in contrast with the bubbles in all other levels which are annotated with verbs. This is expected since thepurpose of the context diagram is to capture the context of the system rather than its functionality.

Example#1: 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 then determine the root mean square (rms) of the three input numbers and display it. In this example, the context diagram is simple to draw. The system acceptsthree integers from the user and returns the result to him.

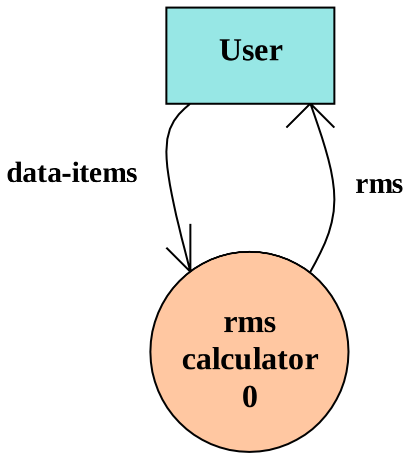


Fig. : Context Diagram

DFD model of a system

A DFD model of a system graphically depicts the transformation of the data input to the system to the final result through a hierarchy of levels. A DFD starts with the most abstract definition of the system (lowest level) and at each higher level DFD, more details are successively introduced. To develop a higher-level DFD model, processes are decomposed into their sub-processes and the data flow among these sub-processes is identified.

To develop the data flow model of a system, first the most abstractrepresentation of the problem is to be worked out. The most abstract representation of the problem is also called the context diagram. After, developing the context diagram, the higher-level DFDs have to be developed.

Context Diagram:-

This has been described earlier.

Level 1 DFD:-

To develop the level 1 DFD, examine the high-level functional requirements. If there are between 3 to 7 high-level functional requirements, then these can be directly represented as bubbles in the level 1 DFD. We can then examine the input data to these functions and the data output by these functions and represent them appropriately in the diagram.

If a system has more than 7 high-level functional requirements, then some of the related requirements have to be combined and represented in the form of a bubble in the level 1 DFD. Such a bubble can be split in the lower DFD levels. Ifa system has less than three high-level functional requirements, then some of them need to be split into their sub-functions so that we have roughly about 5 to7 bubbles on the diagram.

Decomposition:-

Each bubble in the DFD represents a function performed by the system. The bubbles are

decomposed into sub-functions at the successive levels of the DFD. Decomposition of a bubble is also known as factoring or exploding a bubble. Each bubble at any level of DFD is usually decomposed to anything between 3 to
7 bubbles. Too few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes redundant. Also, too many bubbles, i.e. more than 7 bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm.

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 by 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, etc. When a bubble numbered x is decomposed, its children bubble are numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Example:-

A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. Atthe end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every yearafter the prize winners' lists are generated. The context diagram for this problem is shown in fig.

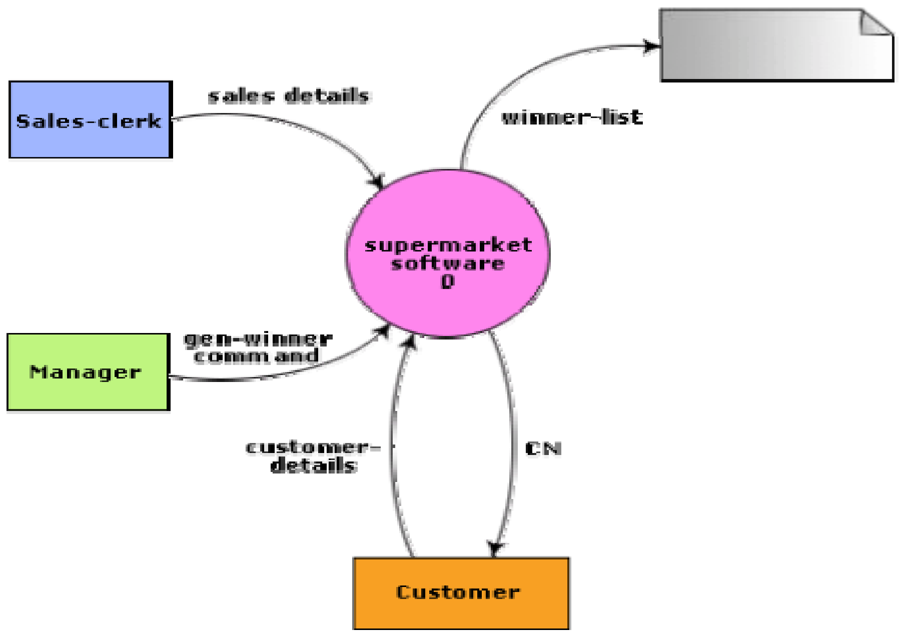


Fig. : Context diagram for supermarket problem

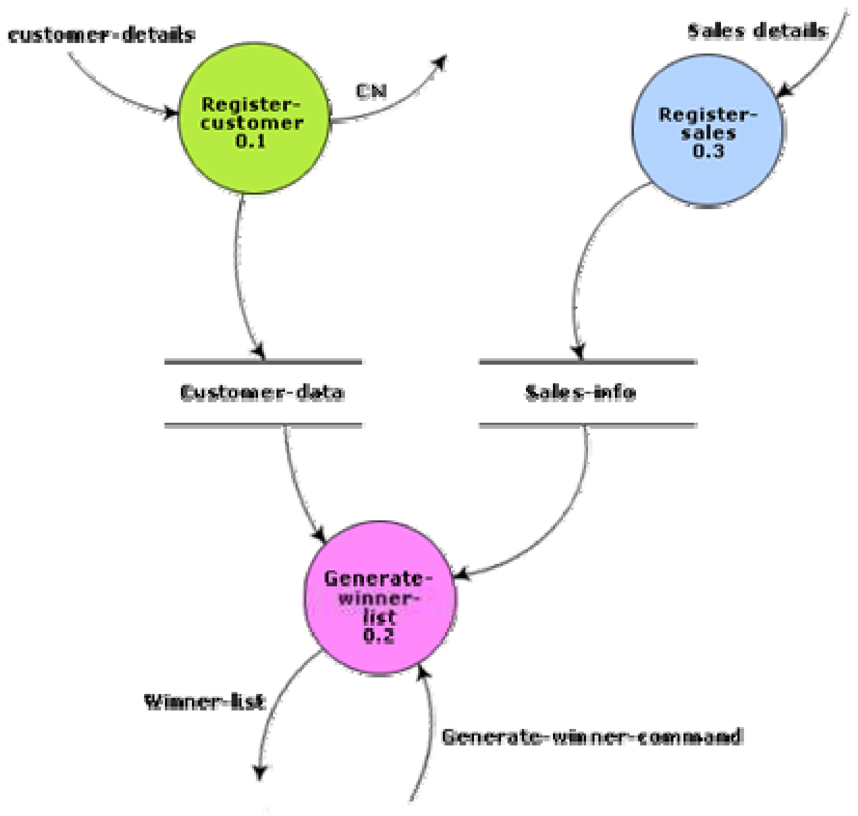


Fig.: Level 1 diagram for supermarket problem

Structured Design

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart.

- Transform analysis
- Transaction analysis

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. These are discussed in the subsequent sub-sections.

Structure Chart

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks which are used to design structure charts are thefollowing:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module toanother module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from

one module to another module in the direction of the arrow.

- **Library modules:** Represented by a rectangle with double edges.
- **Selection:** Represented by a diamond symbol.
- **Repetition:** Represented by a loop around the control flow arrow.

Structure Chart vs. Flow Chart

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

- It is usually difficult to identify the different modules of the softwarefrom its flow chart representation.
- Data interchange among different modules is not represented in a flowchart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in astructure chart.

Transform Analysis

Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

- Input
- Logical processing
- Output

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called the central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the central transform, and the afferent and efferent branches. These are drawn below a root module, which would invoke these modules.

Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central transform as a single box.

In the third step of transform analysis, the structure chart is refined by adding sub-functions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying customer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Example: Structure chart for the RMS software

For this example, the context diagram was drawn earlier.

To draw the level 1 DFD , from a cursory analysis of the problem description, we can see that there are four basic functions that the system needsto perform – accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result.

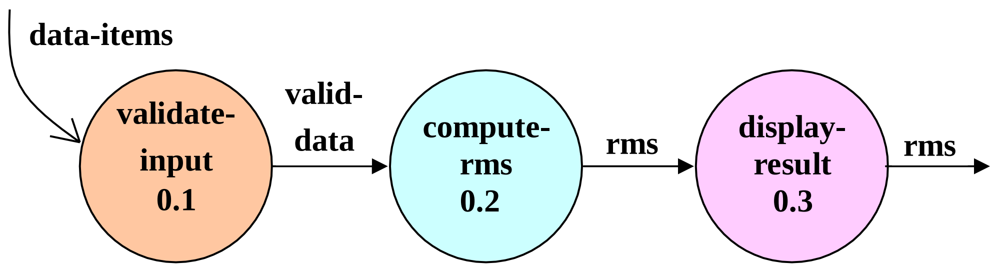


Fig. : Level 1 DFD

By observing the level 1 DFD, we identify the validate-input as the afferent branch and write-output as the efferent branch. The remaining portion (i.e. compute-rms) forms the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in fig.

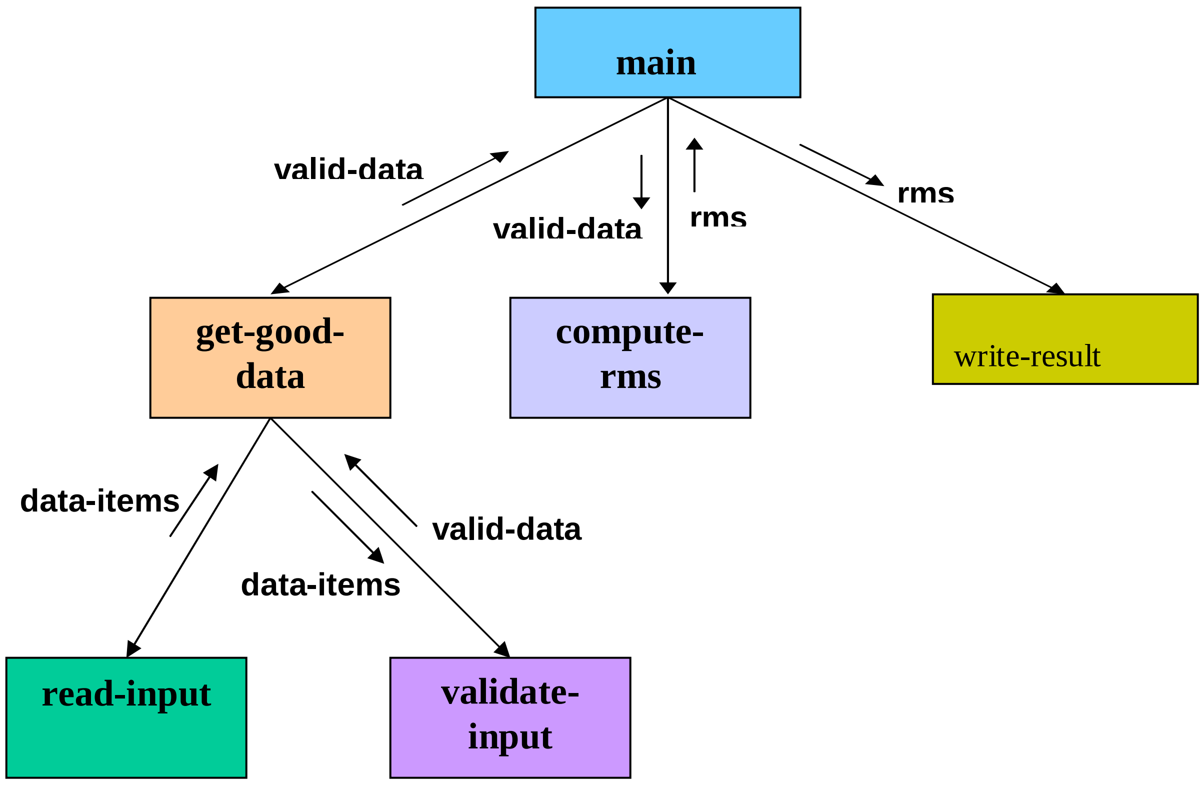


***Fig. :** Structure chart*

Transaction Analysis

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centered system which is characterized by similar processing steps for each dataitem. Each different way in which input data is handled is a transaction. A simple way to

identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transaction may not require any input data. These transactions can be identified from the experience of solving a large number of examples.
For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module. Every transaction carries a tag, which identifies its type. Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module. The structure chart for the supermarket prize scheme software is shown in fig.

Model

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

Need for a model

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following**:**

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.

Unified Modeling Language (UML)

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

Origin of UML

In the late 1980s and early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. These diverse notations used to give rise to a lot of confusion.

UML was developed to standardize the large number of object-oriented modeling notations that existed and were used extensively in the early 1990s. The principles ones in use were:

- Object Management Technology [Rumbaugh 1991]
- Booch's methodology [Booch 1991]
- Object-Oriented Software Engineering [Jacobson 1992]
- Odell's methodology [Odell 1992]
- Shaler and Mellor methodology [Shaler 1992]

It is needless to say that UML has borrowed many concepts from these modeling techniques. Especially, concepts from the first three methodologies have been heavily drawn upon. UML was adopted by Object ManagementGroup (OMG) as a *de facto* standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards. We shall see that UML contains an extensive set of notations and suggests construction of many types of diagrams. It has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and a strong industry backing have helped UML find widespread acceptance. UML is now being used in a large number of software development projects worldwide.

UML diagrams

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

Fig. shows the UML diagrams responsible for providing the different views.

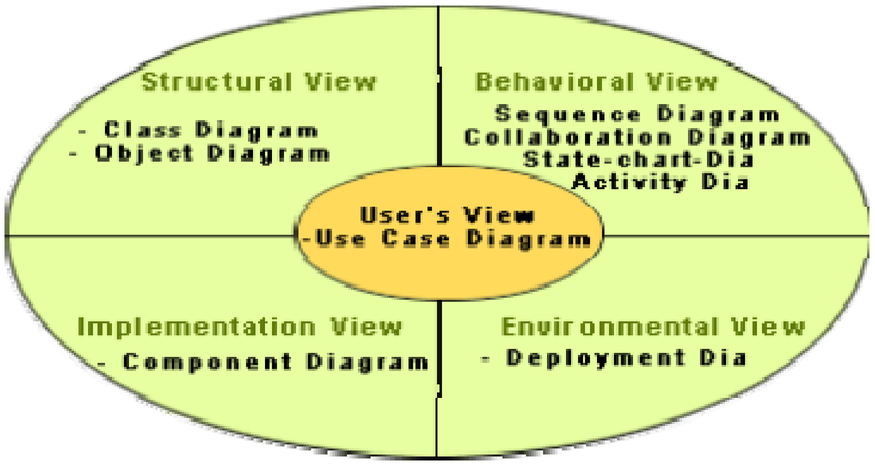


Fig. : Different types of diagrams and views supported in UML

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.
Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.
Implementation view: This view captures the important components of the system and their dependencies.
Environmental view: This view models how the different components are implemented on different pieces of hardware.

Chapter-3
Object Modeling using UML

Use Case Model

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?” Thus for the Library Information System (LIS), the use cases could be**:**

- issue-book
- query-book
- return-book
- create-member
- add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user’s point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

Purpose of use cases

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction

between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case canbe viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and systemactivity through the use case.

Representation of use cases

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written insidethe ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stickperson icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

Example 1:
The use case model for the Tic-tac-toe problem is shown in fig. 7.2. Thissoftware has only one use case "play move". Note that the use case "get-user- move" is not used here. The name "get-user-move" would be inappropriate because the use cases should be named from the users' perspective.

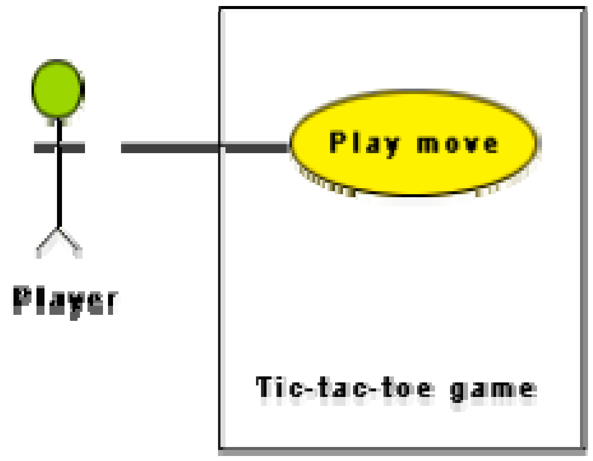


Fig. : Use case model for tic-tac-toe game

Text Description

Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use casein terms of the mainline sequence, different variations to the normalbehavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc. The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following aresome of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

Contact persons: This section lists the personnel of the client organization with whom the use case was discussed, date and time of themeeting, etc.

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Text description

U1: register-customer: Using this use case, the customer can registerhimself by providing the

necessary details.
Scenario 1: Mainline sequence
1. **Customer: select register customer option.**
2. **System: display prompt to enter name, address, and telephone number.**
3. **Customer: enter the necessary values.**
4. **System:** display the generated id and the message that the customer has been successfullyregistered.

Scenario 2: at step 4 of mainline sequence
1.System: displays the message that the customerhas already registered.
Scenario 2: at step 4 of mainline sequence

1. System: displays the message that some input information has not been entered. The system display a prompt to enter the missing value.
The description for other use cases is written in a similar fashion.

Utility of use case diagrams

From use case diagram, it is obvious that the utility of the use cases arerepresented by ellipses. They along with the accompanying text descriptionserve as a type of requirements specification of the system and form the core model to which all other models must conform. But, what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to. Another possible use is in preparing the documentation (e.g. users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

Factoring of use cases

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases are required under two situations. First, complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper. Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into manysmaller parts just for the shake of it.
UML offers three mechanisms for factoring of use cases as follows:

Generalization

Use case generalization can be used when one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the parent use case. The notation is the same too (as shown in fig.). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

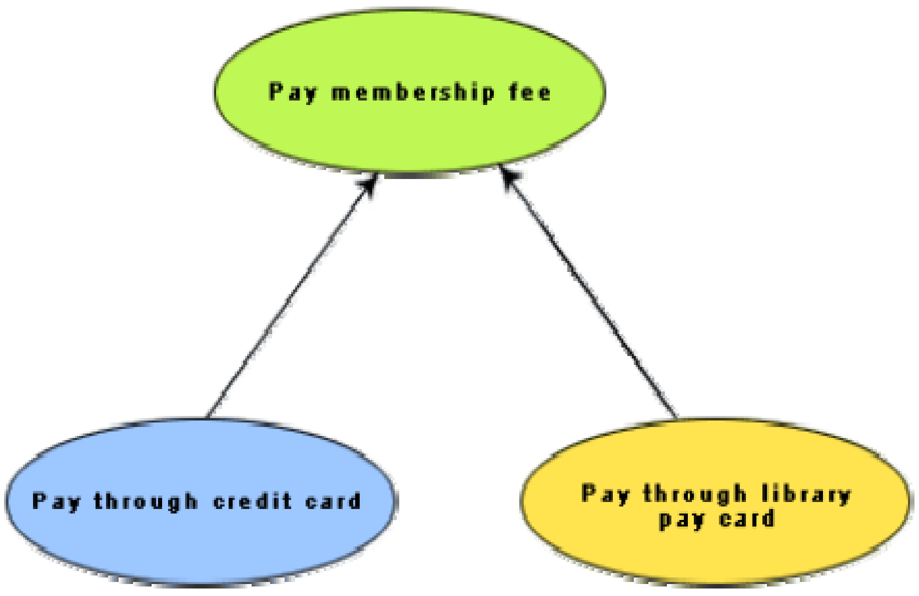


Fig. : Representation of use case generalization

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship involves one use case including the behavior of another use case in its sequenceof events and actions. The includes relationship occurs when a chunk of behavior that is similar across a number of use cases. The factoring of such behavior will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use cases into more manageable parts. As shown in fig., the includes relationship is represented using a predefined stereotype <<include>>. In the includes relationship, a base use case compulsorily and automatically includes the behavior of the common use cases. As shown in example fig. 7.6, issue-book and renew-book both include check-reservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case andthe independent text description should be provided for it.

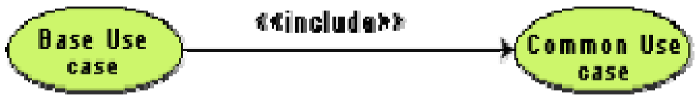


Fig.: Representation of use case inclusion

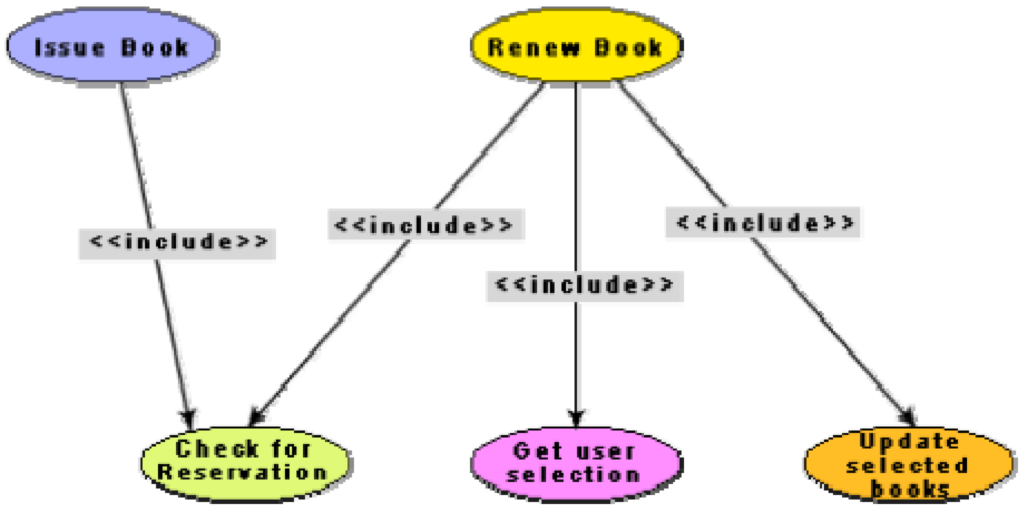


Fig. : Example use case inclusion

Extends

The main idea behind the extends relationship among the use cases is that it allows you to show optional system behavior. An optional system behavior is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in fig. . The extends relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.

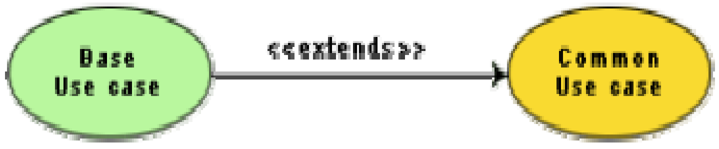


Fig. : Example use case extension

Organization of use cases

When the use cases are factored, they are organized hierarchically. The high- level use cases are refined into a set of smaller and more refined use cases as shown in fig. 7.8. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases. The functionality of the super-ordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases. In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. In the top- level diagram, only those use cases with which external users of the system. Thesubsystem-level use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

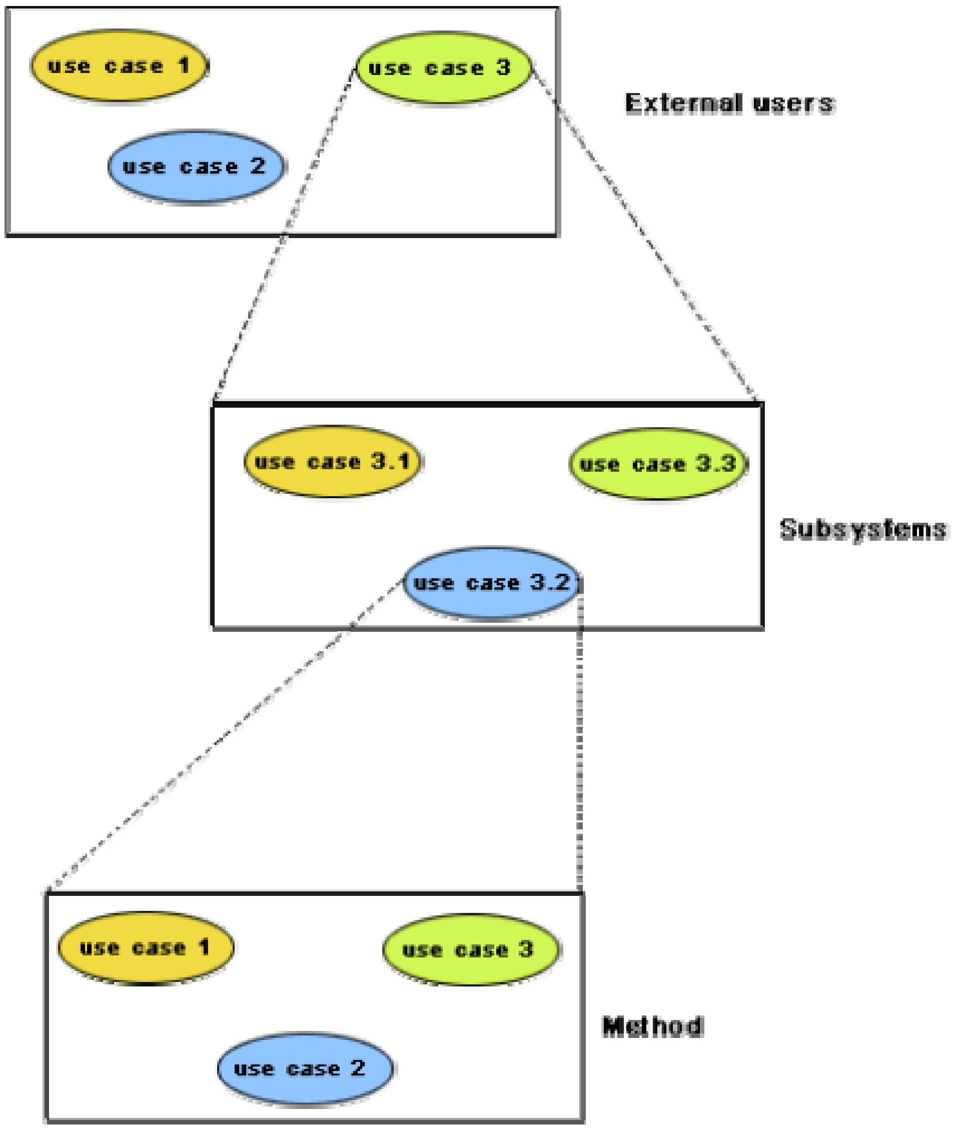


Fig. Hierarchical organization of use cases

Chapter-4
Class and Interaction Diagrams

Class diagrams

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

Classes

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns.

Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressedon all but one diagram.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name. Typically, the first letter of a class name is a small letter. An example for an attribute is given.

bookName : String

Operation

Operation is the implementation of a service that can be requested from any object of the class to affect behaviour. An object's data or state can be changed by invoking an operation of the object. A class may have any number ofoperations or no operation at all. Typically, the first letter of an operation name is a small letter. Abstract operations are written in italics. The parameters of an operation (if any), may have a kind specified, which may be 'in', 'out' or 'inout'. Anoperation may have a return type consisting of a single return type expression. An example for an operation is given.

issueBook(in bookName):Boolean

Association

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose Amit has borrowed the book Graph Theory. Here, borrowed is the connection between the objects Amit and Graph Theory book. Mathematically, a link can be considered to be a tuple, i.e. an ordered listof object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that Library Member borrows Books. Here, borrows is the association between the class LibraryMember and the class Book. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two differentobjects of the class are linked by the association relationship.

Association between two classes is represented by drawing a straight line between the concerned classes. Fig. illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1.5. An asterisk is a wild card and means many (zero or more). The association of fig. should be read as "Many books may be borrowed by a Library Member". Observe that associations (and links) appear as verbs in the problem statement.

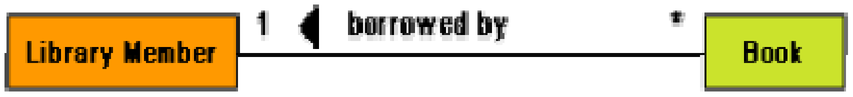


Fig. : Association between two classes

Associations are usually realized by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names ofthe association relation for the corresponding automatically generated attribute.

Aggregation

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibilityof delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as infig. .

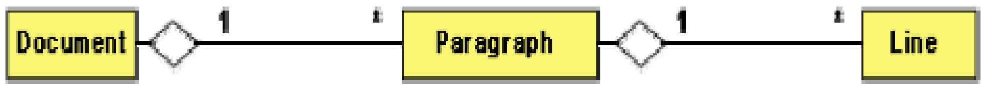


Fig. : Representation of aggregation

Aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts closely ties to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in fig..

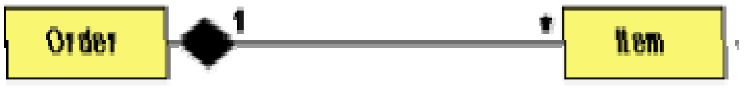


Fig. : Representation of composition

Association vs. Aggregation vs. Composition

- Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even stronger than aggregation, ties the lifecycle of the part and the whole together.
- Association relationship can be reflexive (objects can have relation to itself), but aggregation cannot be reflexive. Moreover, aggregation is anti-symmetric (If B is a part of A, A can not be a part of B).
- Composition has the property of exclusive aggregation i.e. an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window** whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.
- In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.

○ in general, the lifetime of parts and composite coincides
○ parts with non-fixed multiplicity may be created after compositeitself
○ parts might be explicitly removed before the death of thecomposite

For example, when a **Frame** is created, it has to be attached to anenclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

Inheritance vs. Aggregation/Composition

- Inheritance describes *'is a' / 'is a kind of'* relationship between classes (base class - derived class) whereas aggregation describes *'has a'* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation "cash payment *is a kind of* payment" is modeled using inheritance; "purchase order has a few items" is modeled using aggregation.

Inheritance is used to model a "generic-specific" relationship between classes whereas aggregation/composition is used to model a "whole-part" relationship between classes.

- Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of 'payment' in the system, we could substitute it withinstances of 'cash payment', but the reverse can not be done.
- Inheritance is defined statically. It can not be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig (a). But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we preferaggregation instead (see Fig (b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "C**ustomer_Supplier**" if it has both. Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations there of? The inheritance tree would be absolutely incomprehensible.

Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.

- The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The significant disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability. But the significant disadvantage is that it breaks encapsulation, which implies implementation dependence.

Interaction Diagrams

Interaction diagrams are models that describe how group of objects collaborateto realize some behavior. Typically, each interaction diagram realizes the behavior of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other. However, they are bothuseful. These two actually portray different perspectives of behavior of the system and different types of inferences can be drawn from them. The interactiondiagrams can be considered as a major tool in the design methodology.

Sequence Diagram

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participatingin the interaction are shown at the top of the chart as boxes attached to a verticaldashed line. Inside the box the name of the object is written with a colonseparating it from the name of the class and both the name of the object and the class are underlined. The objects appearing at the top signify that the object already existed when the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g. a method call), then the object should be shown at the appropriate place on the diagram where it is created. The vertical dashed line is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifetime is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifeline of twoobjects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is labeled with the message name. Some control information can also be included. Two types of control information are particularly valuable.

- A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements ofan array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig.. The development of the sequence diagram in the development methodology would help us in determining theresponsibilities of the different classes; i.e. what methods should be supportedby each class.

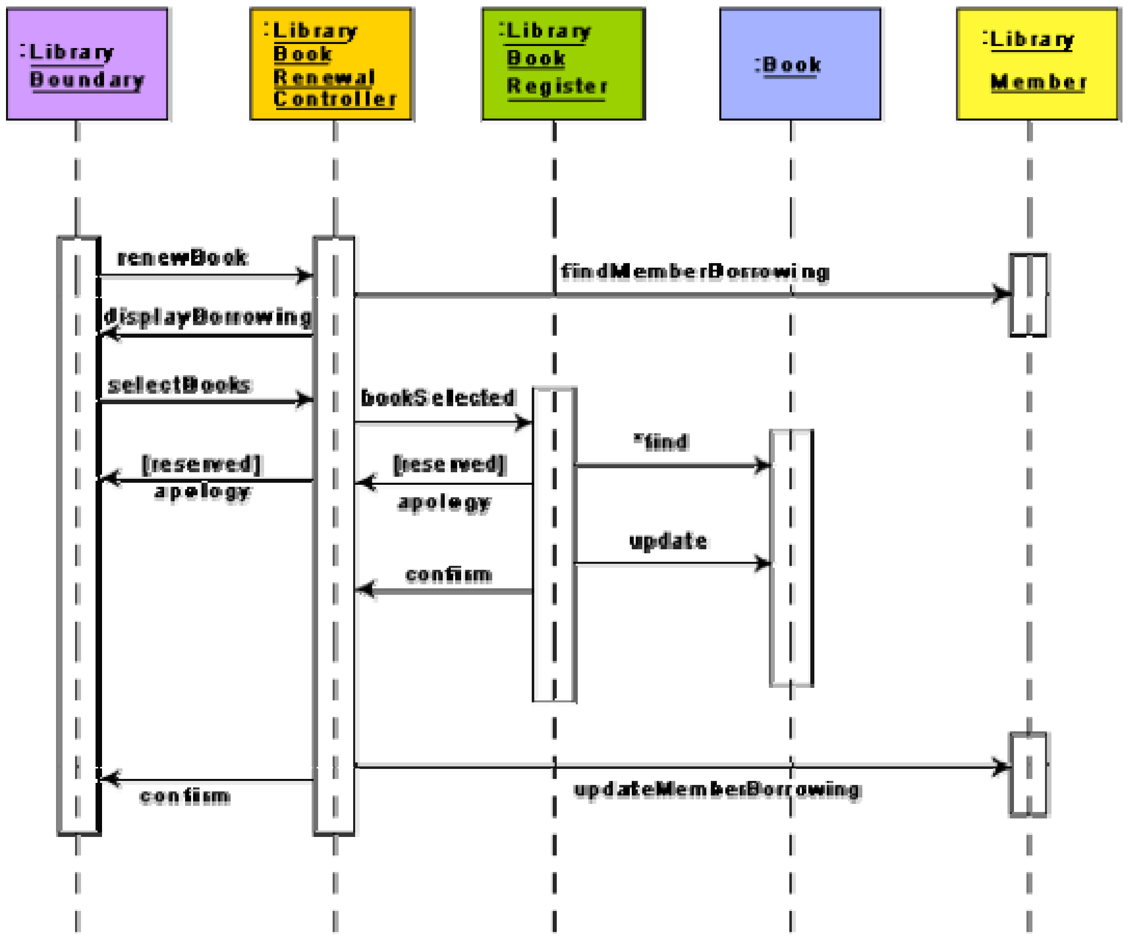


Fig.: Sequence diagram for the renew book use case

Collaboration Diagram

A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called acollaborator. The behavioral aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow placed near the link. Messages areprefixed with sequence numbers because they are only way to describe the relative sequencing of the messages in this diagram. The collaboration diagram for the example of fig. is shown in fig. . The use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

Chapter-5
Activity and StateChart Diagram

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Draw activity diagrams for any given problem.
- Differentiate between the activity diagrams and procedural flow charts.
- Develop the state chart diagram for any given class.
- Compare activity diagrams with state chart diagrams.

Activity diagrams

The activity diagram is possibly one modeling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh. It is possibly based on the event diagram of Odell [1992] through the notation is very different from that used by Odell. The activity diagram focuses on representing activities or chunks of processing which may or may not correspond to the methods of classes. An activity is a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity hasmore than one outgoing transitions, then these must be identified through conditions. An interesting feature of the activity diagrams is the swim lanes.Swim lanes enable you to group activities based on who is performing them, e.g. academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in a swim lane can be assigned to some model elements, e.g. classes or some component, etc.
Activity diagrams are normally employed in business process modeling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.
The student admission process in IIT is shown as an activity diagram in fig. .This shows the part played by different components of the Institute in the admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this synchronization is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

Activity diagrams vs. procedural flow charts

Activity diagrams are similar to the procedural flow charts. The difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities.

State chart diagram

A state chart diagram is normally used to model how the state of an object changes in its lifetime. State chart diagrams are good at describing how the behavior of an object changes across several use case executions. However, if we are interested in modeling some behavior that involves several objects collaborating with each other, state chart diagram is not appropriate. State chart diagrams are based on the finite state machine (FSM) formalism. An FSM consists of a finite number of states corresponding to those of the objectbeing modeled. The object undergoes state changes when specific events occur.The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications. Apart from modeling, it has even been used in theoretical computer science as a generator for regular languages.
A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. This

problem is overcome in UML by using state charts.The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called nested state).

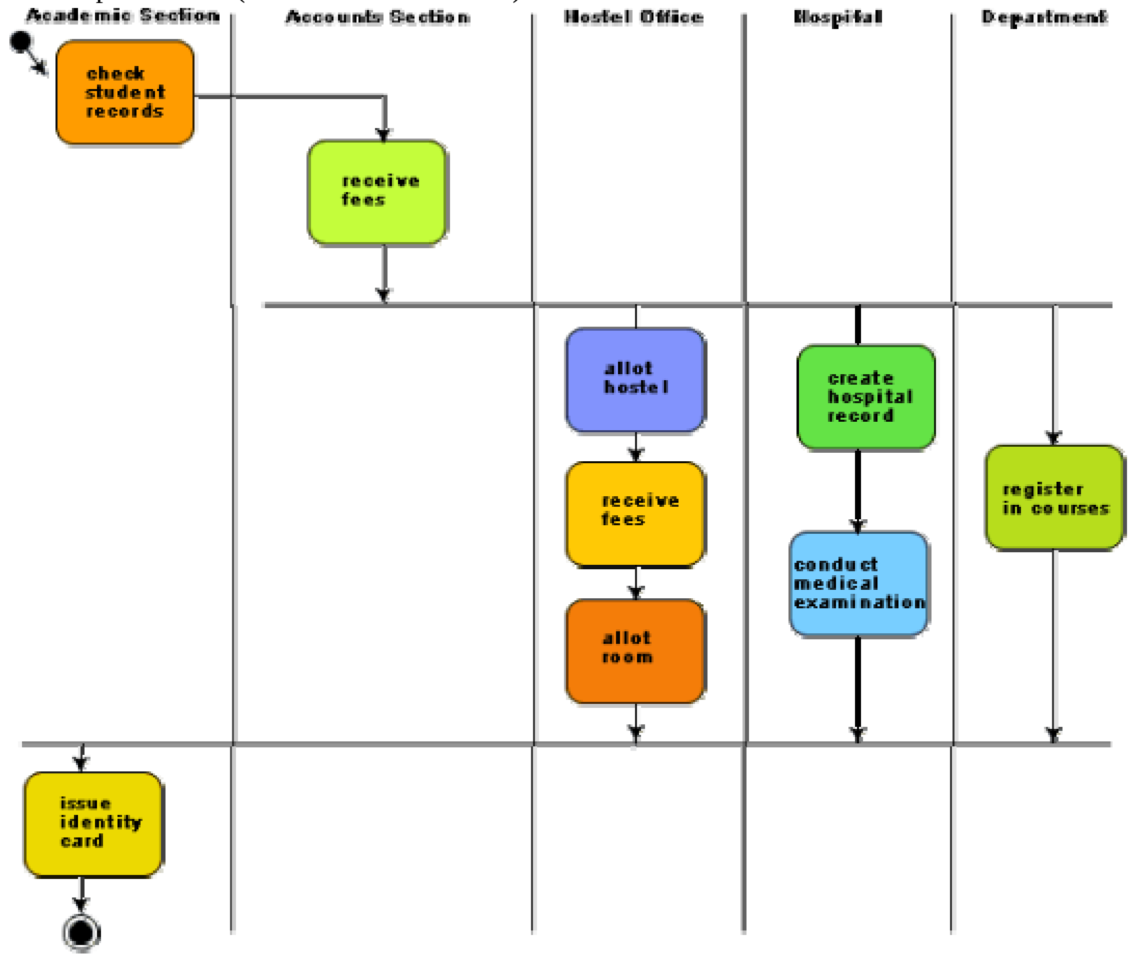


Fig. : Activity diagram for student admission procedure at IIT

Actions are associated with transitions and are considered to be processes that occur quickly and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.
The basic elements of the state chart diagram are as follows:

- **Initial state.** This is represented as a filled circle.
- **Final state.** This is represented by a filled circle inside a larger circle.
- **State.** These are represented by rectangles with rounded corners.
- **Transition.** A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is places along side the arrow. A guard to the transition can also be assigned. A guard is a Boolean logic condition. The transition can take place only if thegrade evaluates to true. The syntax for the label of the transition is shown in 3 parts: event[guard]/action.

An example state chart for the order object of the Trade House Automation software is shown in fig. .

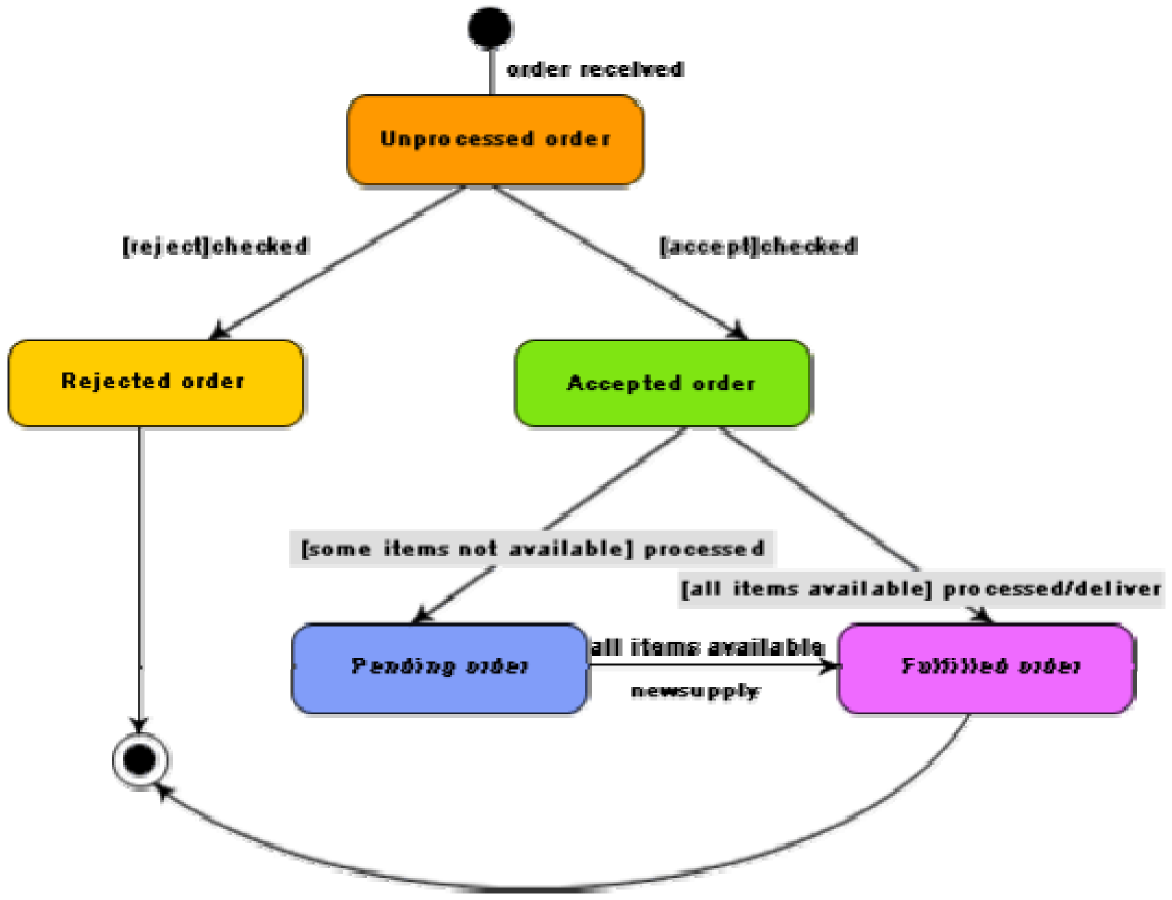


Fig. : State chart diagram for an order object

Activity diagram vs. State chart diagram

- Both activity and state chart diagrams model the dynamic behavior of the system. Activity diagram is essentially a flowchart showing flow of control from activity to activity. A state chart diagram shows a state machine emphasizing the flow of control from state to state.
- An activity diagram is a special case of a state chart diagram in which all or most of the states are activity states and all or most of the transitions are triggered by completion of activities in the source state (An activity is an ongoing non-atomic execution within a state machine).
- Activity diagrams may stand alone to visualize, specify, and document the dynamics of a society of objects or they may be used to model the flow of control of an operation. State chart diagrams may be attached to classes, use cases, or entire systems in order to visualize, specify, and document the dynamics of an individual object.

The following questions have been designed to test the objectives identified for this module:

1. Explain why is it necessary to create a model in the context of goodsoftware development.

Ans.: - An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following**:**

- Analysis
- Specification
- Code generation

- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specificationshould be very different from the one used for design. A model that is being usedfor analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a modelused for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has beendeveloped, along with the model.

2. Identify different types of views of a system captured by UML diagrams.

Ans.: - UML can be used to construct nine different types of diagrams to capture five different views of a system. Different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform tothis view. This thinking is in fact the crux of any user centric development style.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

Behavioral view: The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time- dependent (dynamic) behavior of the system.

Implementation view: This view captures the important components of the system and their dependencies.

Environmental view: This view models how the different components are implemented on different pieces of hardware.

Chapter-6
Design Patterns

Identify the basic difference between object-orientedanalysis (OOA) and object-oriented design (OOD).

The term object-oriented analysis (OOA) refers to a method of developing an initial model of the software from the requirements specification. The analysis model is refined into a design model. The design model can be implemented using a programming language. The term object-oriented programming refers to the implementation of programs using object-oriented concepts.

In contrast, object-oriented design (OOD) paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented. Object-oriented design (OOD) techniques not only identify objects but also identify the internal details of these identified objects. Also, the relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.

Explain why design patterns are important in creating goodsoftware design.

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern etc.

Explain what design patterns are.

Design patterns very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

Identify pattern solution for a particular problem in terms of class and interaction diagrams.

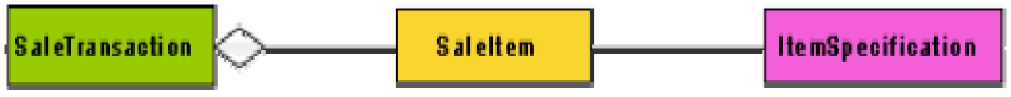
The design pattern solutions are typically described in terms of class and interaction diagrams.

Example:

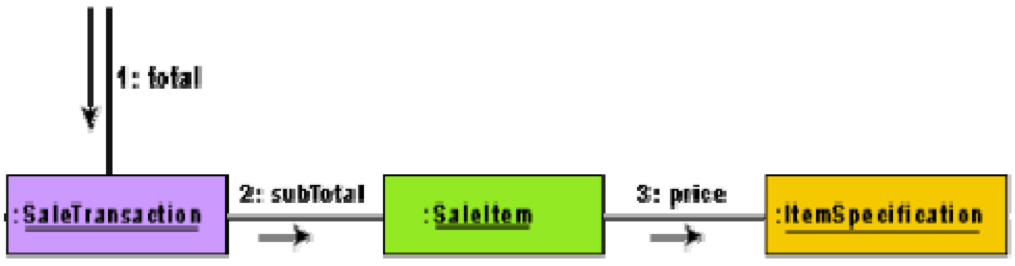
Expert Pattern

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig. .



(a)



(a)

Fig. : Expert pattern: (a) Class diagram (b) Collaboration diagram

Explain expert pattern and circumstances when it can beused.
Expert pattern was defined earlier.

Explain creator pattern and circumstances when it can beused.
Creator Pattern
Problem: Which class should be responsible for creating a new instance ofsome class?
Solution: Assign a class C1 the responsibility to create an instance of class C2,if one or more of the following are true**:**

- C1 is an aggregation of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialize the objects of type C2,when they are created.

Explain controller pattern and circumstances when it can beused.
Controller Pattern:

Problem: Who should be responsible for handling the actor requests?
Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request isreceived before arrange payment request.

Explain facade pattern and circumstances when it can beused.
Façade Pattern:
Problem: How should the services be requested from a service package?
Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.
Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.
Explain model view separation pattern and circumstanceswhen it can be used.
Model View Separation Pattern:

Problem: How should the non-GUI classes communicate with the GUI classes?

Context in which the problem occurs: This is a very commonly occurring pattern which occurs in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.

Solution: The model view separation pattern states that model objects shouldnot have direct knowledge (or be directly coupled) to the view objects. This

means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.

There are actually two solutions to this problem which work in different circumstances as follows:

Solution 1: Polling or Pull from above

It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required.

This model is frequently used. However, it is inefficient for certain applications. For example, simulation applications which require visualization, the GUI objects would not know when the necessary information becomes available. Other examples are, monitoring applications such as network monitoring, stock market quotes, and so on. In these situations, a “push-from-below” model of display update is required. Since “push-from-below” is not an acceptable solution, an indirect mode of communication from the other objects to the GUI objects is required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object. The event manager notifies all registered subscribers usually via a parameterized message (called a callback). Some languagesspecifically support event manager classes. For example, Java provides the EventListener interface for such purposes.

Explain intermediary pattern (i.e. proxy pattern) and circumstances when it can be used.

Intermediary Pattern or Proxy

Problem: How should the client and server objects interact with each other?

Context in the problem occurs: The client and server terms as used here refer to software components existing across a network. The clients are consumers of services provided by the servers.

Solution: A proxy object at the client side can be defined which is a local sit-infor the remote server object. The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the client request to the server, obtaining the server responseand seamlessly passing that to the client. The proxy can also augment (or filter) information that is exchanged between the client and the server. The proxy couldhave the same interface as the remote server object so that the client feels as if itis interacting directly with the remote server object and the complexities ofnetwork transmissions are abstracted out.

Chapter-7
Code Review

Coding

Good software development organizations normally require their programmers toadhere to some well-defined and standard style of coding called coding standards. Most software development organizations formulate their own coding standards that suit them most, and require their engineers to follow these standards rigorously. The purpose of requiring all engineers of an organization toadhere to a standard style of coding is the following:

- A coding standard gives a uniform appearance to the codes written bydifferent engineers.
- It enhances code understanding.
- It encourages good programming practices.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

Coding standards and guidelines

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and thetype of products they develop.
The following are some representative coding standards.

Rules for limiting the use of global: These rules list what types of data can be declared global and what cannot.

Contents of the headers preceding codes for different modules: The information contained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized inthe header can also be specified. The following are some standard header data:

- Name of the module.
- Date on which the module was created.
- Author's name.
- Modification history.
- Synopsis of the module.
- Different functions supported, along with their input/output parameters.
- Global variables accessed/modified by the module.

Naming conventions for global variables, local variables, and constantidentifiers: A possible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.

Error return conventions and exception handling mechanisms: The way error conditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

The following are some representative coding guidelines recommended by many software development organizations.

Do not use a coding style that is too clever or too difficult to understand: Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.

Avoid obscure side effects: The side effects of a function call include modification of parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from acasual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult toinfer from the function’s name and header information, it becomes difficult for anybody trying to understand the code.

Do not use an identifier for multiple purposes: Programmers often use the same identifier to denote several temporary entities. For example, some

programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approachand hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

- Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficultfor somebody trying to read and understand the code.
- Use of variables for multiple purposes usually makes futureenhancements more difficult.

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

The length of any function should not exceed 10 source lines: A function that is very lengthy is usually very difficult to understand as it probably carries outmany different functions. For the same reason, lengthy functions are likely tohave disproportionately larger number of bugs.

Do not use goto statements: Use of goto statements makes a programunstructured and makes it very difficult to understand.

Code review

Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module. These two types code review techniques are code inspection and code walk through.

Code Walk Throughs

Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code fewdays before the walk through meeting to read and understand code. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors inthe code. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present.

Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to beapplied dogmatically. Some of these guidelines are the following.

- The team performing code walk through should not be either too big or toosmall. Ideally, it should consist of between three to seven members.
- Discussion should focus on discovery of errors and not on how to fix the discovered errors.
- In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

Code Inspection

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an errorwill be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checkedduring code inspection:

- Use of uninitialized variables.
- Jumps into loops.
- Nonterminating loops.
- Incompatible assignments.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatches between actual and formal parameter in procedure calls.
- Use of incorrect logical operators or incorrect precedence amongoperators.
- Improper modification of loop variables.
- Comparison of equally of floating point variables, etc.

Clean room testing

Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, inspection, and formal verification. The programmers are notallowed to test any of their code by executing the code other than doing some syntax testing using a compiler. The software development philosophy is based on avoiding software defects by using a rigorous inspection process. The objective of this software is zero-defect software.

The name 'clean room' was derived from the analogy with semi-conductor fabrication units. In these units (clean rooms), defects are avoided bymanufacturing in ultra-clean atmosphere. In this kind of development, inspections to check the consistency of the components with their specifications has replacedunit-testing.

This technique reportedly produces documentation and code that is more reliableand maintainable than other development methods relying heavily on code execution-based testing.

The clean room approach to software development is based on five characteristics:

- **Formal specification:** The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
- **Incremental development:** The software is partitioned into increments which are developed and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.
- **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is process of stepwise refinement of the specification.
- **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
- **Statistical testing of the system:** The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on the operational profile which is developed in parallel with the system specification.

The main problem with this approach is that testing effort is increased as walk throughs, inspection, and verification are time- consuming.

Software documentation

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users' manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

- Good documents enhance understandability and maintainability of a software product. They reduce the effort and time required for maintenance.
- Use documents help the users in effectively using the system.
- Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he can build up the required knowledge easily.
- Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

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

- Internal documentation
- External documentation

Internal documentation is the code comprehension features provided as part of the source code itself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc. Careful experiments

suggest that out of all types of internal documentation meaningful variable namesis most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. Theresearch finding is obviously true when comments are written without thought. For example, the following style of code commenting does not in any way help in understanding the code.

a = 10;   /* a made 10 */

But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.

External documentation is provided through various types of supportingdocuments such as users' manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

Program Testing

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

- **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence of an error may not necessarily lead to a failure.

- **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the state of the system at which the data is input, and O isthe expected output of the system.
- **Test suite:** This is the set of all test cases with which a given software product is to be tested.

Aim of testing

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying outthe testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software productsis very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system.

Differentiate between verification and validation.

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

Design of test cases

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible. Actually, if test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected atrandom does not guarantee that all (or even most) of the errors in the system willbe uncovered. Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

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

For the above code segment, the test suite, **{(x=3,y=2);(x=2,y=3)}** can detect theerror, whereas a larger test suite **{(x=3,y=2);(x=4,y=3);(x=5,y=1)}** does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

Functional testing vs. Structural testing

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing.

On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing..

Chapter-8
Black-Box Testing

Testing in the large vs. testing in the small

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing inthe large.

Unit testing

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to testthe module:

- The procedures belonging to other modules that the module under testcalls.
- Nonlocal data structures that the module accesses.
- A procedure to call the functions of the module under test withappropriate parameters.

Modules required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in fig. . A stub procedure is a dummy procedure that has the same I/O parameters as thegiven procedure but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

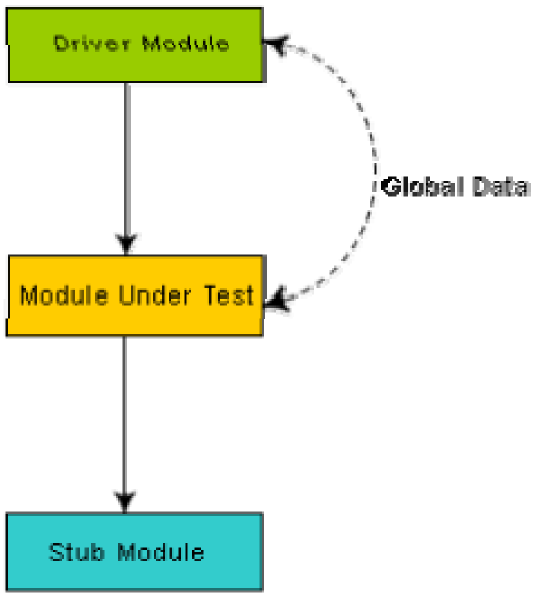


Fig.: Unit testing with the help of driver and stub modules

Black box testing

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design, or code is required. The following are the two main approaches to designing black box test cases.

- Equivalence class portioning
- Boundary value analysis

Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.
2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example#1: For a software that computes the square root of an input integer which can assume values in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

Example#2: Design the black-box test suite for the following program. Theprogram computes the intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c.

The equivalence classes are the following:

- Parallel lines (m1=m2, c1□c2)
- Intersecting lines (m1□m2)
- Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

Boundary Value Analysis

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < insteadof <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Example: For a function that computes the square root of integer values in the range of 0 and 5000, the test cases must include the following values: {0, - 1,5000,5001}.

Test cases for equivalence class testing and boundary valueanalysis for a problem

Let's consider a function that computes the square root of integer values in the range of 0 and 5000. For this particular problem, test cases corresponding to equivalence class testing and boundary value analysis have been found out earlier.

White box testing

One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The concepts of stronger and complementary testing are schematically illustrated in fig. .

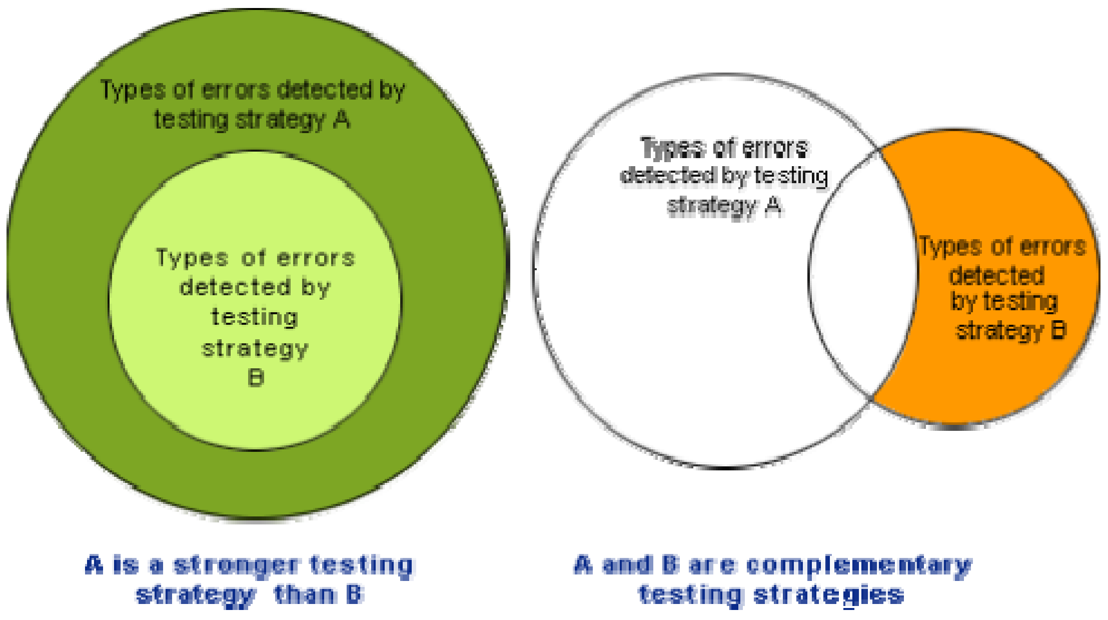

Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
Types of errors detected by testing strategy A
Types of errors detected by testing strategy B
A is a stronger testing strategy than B
A and B are complementary testing strategies

Fig. : Stronger and complementary testing strategies

Statement coverage

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

Example: Consider the Euclid's GCD computation algorithm: int

```
compute_gcd(x, y)
    int x, y;
 {
   1    while (x! = y){
  2         if (x>y) then
   3                 x= x – y;
  4         else  y= y – x;5
         }
  6   return x;
 }
```

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed at least once.

Branch coverage

In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program's control flow graph is traversed at least once.

It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid's GCD computation algorithm , the test cases for branch coverage canbe {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

Condition coverage

In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression ((c1.and.c2).or.c3), the components c1,c2 and c3 are each made to assume both true and false values. Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2^n test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

Path coverage

The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed 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. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig.). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig. 10.3 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown in fig. .

Path

A path through a program is a node and edge sequence from the starting node toa terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

Linearly independent path

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. Ifa path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath ofanother path is not considered to be a linearly independent path.

Control flow graph

In order to understand the path coverage-based testing strategy, it is very much necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

Linearly independent path

The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

Cyclomatic complexity

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity. Theanswers computed by the three methods are guaranteed to agree.

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$$V(G) = E – N + 2$$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in fig. , E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = Total number of bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe's cyclomatic complexity. But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

The number of bounded areas increases with the number ofdecision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For theCFG example shown in fig. , from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing with this method is also 2+1 = 3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1.

Data flow-based testing

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.

For a statement numbered S, let

DEF(S) = {X/statement S contains a definition of X}, andUSES(S) = {X/statement S contains a use of X}

For the statement **S:a=b+c;,** DEF(S) = {**a**}. USES(S) = {**b,c**}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.

The *definition-use chain* (or DU chain) of a variable X is of form [X, S, S1], where S and S1 are statement numbers, such that X Є DEF(S) and X Є USES(S1), and the definition of X in the statement S is live at statement S1. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

Mutation testing

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant. The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc. A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool whichwould run all the test cases automatically.

Chapter-9
COCOMO Model

Organic, Semidetached and Embedded software projects

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm's [1981] definition of organic, semidetached, and embeddedsystems are elaborated below.

Organic: A development project can be considered 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 considered of semidetached type, if the development 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 complex hardware, or if the stringent regulations on the operational procedures exist.

COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Where

$$\textbf{Effort} = a_1 \times (\textbf{KLOC})^{2} \textbf{ PM}$$
$$\textbf{Tdev} = b_1 \times (\textbf{Effort})^{2} \textbf{ Months}$$

- KLOC is the estimated size of the software product expressed in KiloLines of Code,
- a_1, a_2, b_1, b_2 are constants for each category of software products,
- Tdev is the estimated time to develop the software, expressed inmonths,
- Effort is the total effort required to develop the software product,expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in fig.). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve (as shown in fig.).

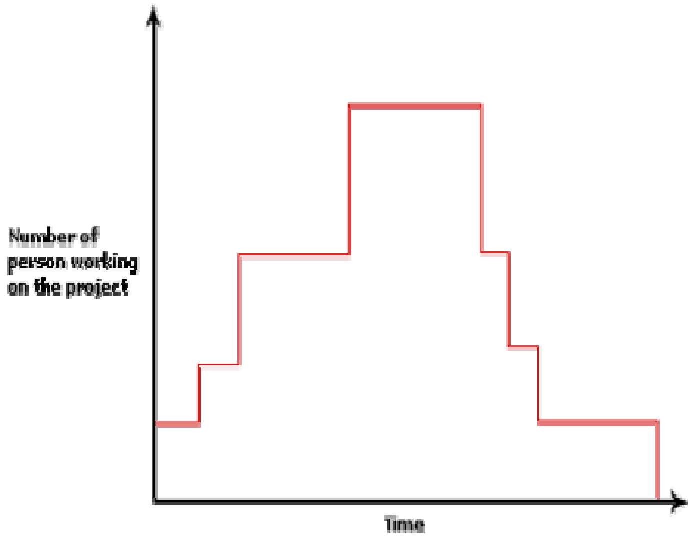


Fig. : Person-month curve

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 a_1, a_2, b_1, b_2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions 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:

Organic : **Effort = $2.4(KLOC)^{1.05}$ PM** Semi-detached : **Effort = $3.0(KLOC)^{1.12}$ PM** Embedded : **Effort = $3.6(KLOC)^{1.20}$ PM**

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:

Organic : **Tdev = $2.5(Effort)^{0.38}$ Months** Semi-detached : **Tdev = $2.5(Effort)^{0.35}$ Months** Embedded : **Tdev = $2.5(Effort)^{0.32}$ Months**

some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. shows a plot of estimated effort versus product size. From fig., we can observe that the effort is somewhat superlinear in the size of the software product. Thus, the effortrequired to develop a product increases very rapidly with project size.

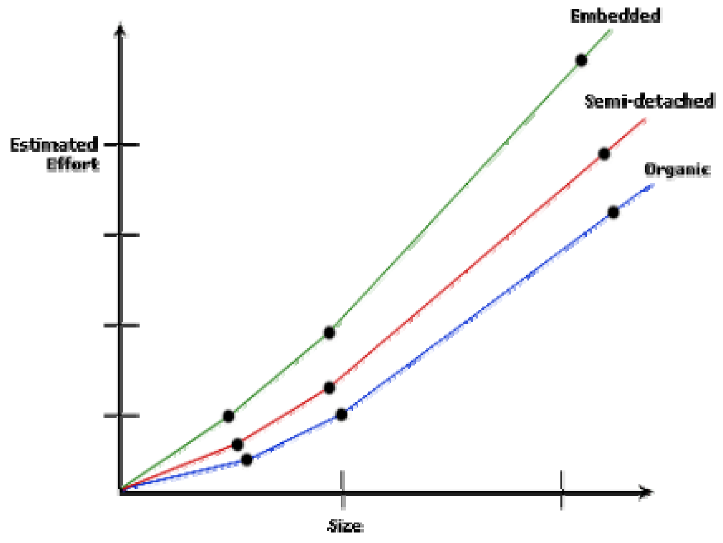


Fig. : Effort versus product size

The development time versus the product size in KLOC is plotted in fig. . From fig. , it can be observed that the development time is a sublinear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig. , it can be observed that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

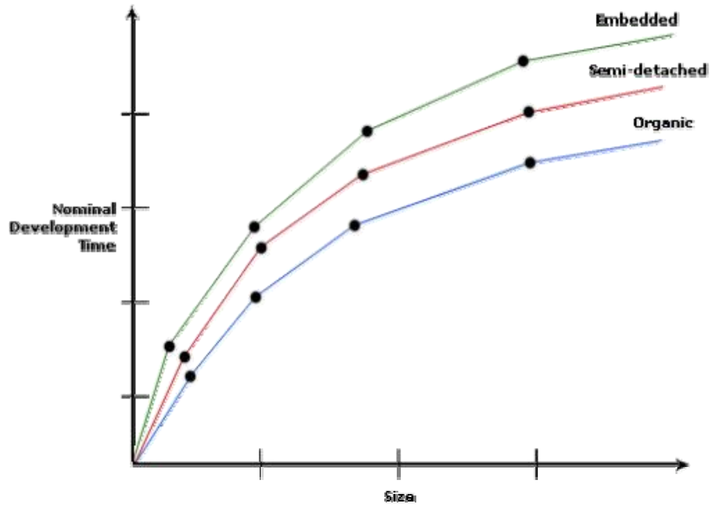


Fig. : Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the

COCOMO model are called as nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example:
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 software engineers be Rs. 15,000/- per month. Determine the effort required to developthe software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:Effort = **2.4 x** $\mathbf{(32)^{1.05}}$ **= 91 PM**

Nominal development time = **2.5 x** $\mathbf{(91)^{0.38}}$ = **14 months**

Cost required to develop the product = **14 x 15,000**
= **Rs. 210,000/-**

Intermediate COCOMO model

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using aset of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to ratethese 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of thefollowing items:
Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.
Computer: Characteristics of the computer that are considered include the execution speed required, storage space required etc.
Personnel: The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.
Development Environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for softwaredevelopment.

Complete COCOMO model

A major shortcoming of both the basic and intermediate COCOMO models is thatthey consider a software product as a single homogeneous entity. However,most large systems are made up several smaller sub-systems. These sub- systems may have widely different characteristics. For example, some sub- systems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystemsmay be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. Thecost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

- Database part
- Graphical User Interface (GUI) part
- Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately,and summed up to give the overall cost of the system.

Chapter-10
Risk Management andSoftware Configuration Management

Risk management

A software project can be affected by a large variety of risks. In order to be able to systematically identify the important risks which might affect a software project,it is necessary to categorize 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. Project risks concern varies 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. It is very difficult to control something which cannot be seen. For any manufacturing project, such as manufacturing of cars, the project manager can see the product taking shape. He can for instance, see that the engine is fitted, after that the doors are fitted, the car is getting painted, etc. Thus he can easily assess the progress of the work and control it. The invisibility of theproduct being developed is an important reason why many software projects suffer from the risk of schedule slippage.

Technical risks. Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur dueto the development team's insufficient knowledge about the project.

Business risks. This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments, etc.

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 coming true (denoted as r).
- The consequence of the problems associated with that risk (denotedas s).

Based on these two factors, the priority of each risk can be computed:

$$p = r * s$$

Where, p is the priority with which the risk must be handled, r is the probability of the risk becoming true, and s is the severity of damage caused due to the risk becoming true. If all identified risks are prioritized, then the most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for these risks.

Risk containment

After all the identified risks of a project are assessed, plans must be made to contain the most damaging and the most likely risks. Different risks requiredifferent containment procedures. In fact, most risks require ingenuity on the part of the project manager in tackling the risk.

There are three main strategies to plan for risk containment:

Avoid the risk: This may take several forms such as discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the engineers to avoid the risk of manpower turnover, etc.

Transfer the risk: This strategy involves getting the risky component developedby a third party, buying insurance cover, etc.

Risk reduction: This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

Risk leverage

To choose between the different strategies of handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this the risk leverage of the different risks can be computed.

Risk leverage is the difference in risk exposure divided by the cost of reducingthe risk. More formally, risk leverage = (risk exposure before reduction – risk exposure afterreduction) / (cost of reduction)

Risk related to schedule slippage

Even though there are three broad ways to handle any risk, but still risk handling requires a lot of ingenuity on the part of a project manager. As an example, it can be considered the options available to contain an important type of risk that occurs in many software projects – that of schedule slippage. Risks relating to schedule slippage arise primarily due to the intangible nature of software. Therefore, these can be dealt with by increasing the visibility of the software product. Visibility of a software product can be increased by producing relevant documents during the development process wherever meaningful and getting these documents reviewed by an appropriate team. Milestones should be placed at regular intervals through a software engineering process to provide a manager with regular indication of progress. Completion of a phase of the development process before followed need not be the only milestones. Every phase can be broken down to reasonable-sized tasks and milestones can be scheduled for these tasks too. A milestone is reached, once documentation produced as part ofa software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb isto set a milestone every 10 to 15 days.

Software configuration management

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g. source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software engineers through out the life cycle of the software. The state of all these objects at any point of time is called the configuration of the software product. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

Release vs. Version vs. Revision

A new version of a software is created when there is a significant change in functionality, technology, or the hardware it runs on, etc. On the other hand anew revision of a software refers to minor bug fix in that software. A new release is created if there is only a bug fix, minor enhancements to the functionality, usability, etc.

For example, one version of a mathematical computation package might run on Unix-based machines, another on Microsoft Windows and so on. As a software is released and used by the customer, errors are discovered that need correction. Enhancements to the functionalities of the software may also be needed. A new release of software is an improved system intended to replace anold one. Often systems are described as version m, release n; or simple m.n. Formally, a history relation is version of can be defined between objects. This relation can be split into two sub relations *is revision of* and *is variant of.*

Necessity of software configuration management

There are several reasons for putting an object under configuration management.But, possibly the most important reason for configuration management is to control the access to the different deliverable objects. Unless strict discipline is enforced regarding updation and storage of different objects, several problems appear. The following are some of the important problems that appear if configuration management is not used.

Inconsistency problem when the objects are replicated. A scenario can be considered where every software engineer has a personal copy of an object (e.g. source code). As each engineer makes changes to his local copy, he is expected to intimate them to other engineers, so that the changes in interfaces are uniformly changed across all modules. However, many times an engineer makes changes to the interfaces in his local copies and forgets to intimate other teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the product is integrated, it does not work. Therefore, when several team members work on developing an object, it is necessary for them to work on a single copy of the object, otherwise inconsistency may arise.

Problems associated with concurrent access. Suppose there is a single copy of a problem module, and several engineers are working on it. Two engineers may simultaneously carry out changes to different portions of the same module, and while saving overwrite each other. Though the problem associated with concurrent access to program code has been explained, similar problems occur for any other deliverable object.

Providing a stable development environment. When a project is underway, the team members need a stable environment to make progress. Suppose somebody is trying to integrate module A, with the modules B and C, he cannot make progress if developer of module C keeps changing C; this can be especially frustrating if a change to module C forces him to recompile A. Whenan effective configuration management is in place, the manager freezes the objects to form a base line. When anyone needs any of the objects under configuration control, he is provided with a copy of the base line item. The requester makes changes to his private copy. Only after the requester is through with all modifications to his private copy, the configuration is updated and a new

base line gets formed instantly. This establishes a baseline for others to use and depend on. Also, configuration may be frozen periodically. Freezing a configuration may involve archiving everything needed to rebuild it. (Archiving means copying to a safe place such as a magnetic tape).

System accounting and maintaining status information. System accounting keeps track of who made a particular change and when the change was made.

Handling variants. Existence of variants of a software product causes some peculiar problems. Suppose somebody has several variants of the same module,and finds a bug in one of them. Then, it has to be fixed in all versions and revisions. To do it efficiently, he should not have to fix it in each and everyversion and revision of the software separately.

Software configuration management activities

Normally, a project manager performs the configuration management activity by using an automated configuration management tool. A configurationmanagement tool provides automated support for overcoming all the problems mentioned above. In addition, a configuration management tool helps to keep track of various deliverable objects, so that the project manager can quickly and unambiguously determine the current state of the project. The configuration management tool enables the engineers to change the various components in a controlled manner.

Configuration management is carried out through two principal activities:

- Configuration identification involves deciding which parts of the systemshould be kept track of.
- Configuration control ensures that changes to a system happensmoothly.

Configuration identification

The project manager normally classifies the objects associated with a software development effort into three main categories: controlled, precontrolled, and uncontrolled. Controlled objects are those which are already put under configuration control. One must follow some formal procedures to change them. Precontrolled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subjected to configuration control. Controllable objects include both controlled and precontrolled objects. Typical controllable objects include:

- Requirements specification document
- Design documents
- Tools used to build the system, such as compilers, linkers, lexicalanalyzers, parsers, etc.
- Source code for each module
- Test cases
- Problem reports

The configuration management plan is written during the project planning phase and it lists all controlled objects. The managers who develop the plan must strike a balance between controlling too much, and controlling too little. If too much is controlled, overheads due to configuration management increase tounreasonably high levels. On the other hand, controlling too little might lead to confusion when something changes.

Configuration control

Configuration control is the process of managing changes to controlled objects. Configuration control is the part of a configuration management system that mostdirectly affects the day-to-day operations of developers. The configuration controlsystem prevents unauthorized changes to any controlled objects. In order to change a controlled object such as a module, a developer can get a private copy of the module by a reserve operation as shown in fig. . Configuration management tools allow only one person to reserve a module at a time. Once anobject is reserved, it does not allow any one else to reserve this module until the reserved module is restored as shown in fig. . Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

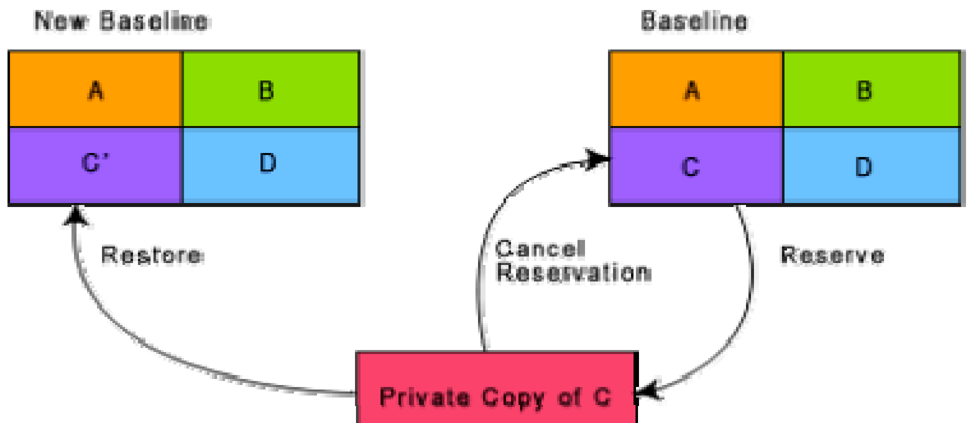


Fig. : Reserve and restore operation in configuration control

It can be shown how the changes to any object that is under configuration control can be achieved. The engineer needing to change a module first obtains a private copy of the module through a reserve operation. Then, he carries out all necessary changes on this private copy. However, restoring the changed moduleto the system configuration requires the permission of a change control board (CCB). The CCB is usually constituted from among the development team members. For every change that needs to be carried out, the CCB reviews the changes made to the controlled object and certifies several things about the change:

1. Change is well-motivated.
2. Developer has considered and documented the effects of the change.
3. Changes interact well with the changes made by other developers.
4. Appropriate people (CCB) have validated the change, e.g. someone has tested the changed code, and has verified that the change is consistent with the requirement.

The change control board (CCB) sounds like a group of people. However, except for very large projects, the functions of the change control board are normally discharged by the project manager himself or some senior member of the development team. Once the CCB reviews the changes to the module, the project manager updates the old base line through a restore operation (as shown in fig. 12.5). A configuration control tool does not allow a developer to replace an object he has reserved with his local copy unless he gets an authorization from the CCB. By constraining the developers' ability to replace reserved objects, a stable environment is achieved. Since a configuration management tool allows only one engineer to work on one module at any one time, problem of accidental overwriting is eliminated. Also, since only the manager can update the baseline after the CCB approval, unintentional changes are eliminated.

Configuration management tools

SCCS and RCS are two popular configuration management tools available on most UNIX systems. SCCS or RCS can be used for controlling and managing different versions of text files. SCCS and RCS do not handle binary files (i.e. executable files, documents, files containing diagrams, etc.) SCCS and RCS provide an efficient way of storing versions that minimizes the amount of occupied disk space. Suppose, a module MOD is present in three versions MOD1.1, MOD1.2, and MOD1.3. Then, SCCS and RCS stores the original module MOD1.1 together with changes needed to transform MOD1.1 into MOD1.2 and MOD1.2 to MOD1.3. The changes needed to transform each base lined file to the next version are stored and are called deltas. The main reason behind storing the deltas rather than storing the full version files is to save disk space.

The change control facilities provided by SCCS and RCS include the ability to incorporate restrictions on the set of individuals who can create new versions, and facilities for checking components in and out (i.e. reserve and restore operations). Individual developers check out components and modify them. After they have made all necessary changes to a module and after the changes have been reviewed, they check in the changed module into SCCS or RCS. Revisions are denoted by numbers in ascending order, e.g., 1.1, 1.2, 1.3 etc. It is also possible to create variants or revisions of a component by creatinga fork in the development history.

Chapter-11
Software Reliability Issues

Repeatable vs. non-repeatable software development organization

A repeatable software development organization is one in which the software development process is person-independent. In a non-repeatable software development organization, a software development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals. Thus, in a non-repeatable software development organization, the chances of successful completion of a software project is to a great extent depends on the team members.

Software reliability

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. 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 is the corresponding instruction executed.

Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the "correctly" implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

Reasons for software reliability being difficult to measure

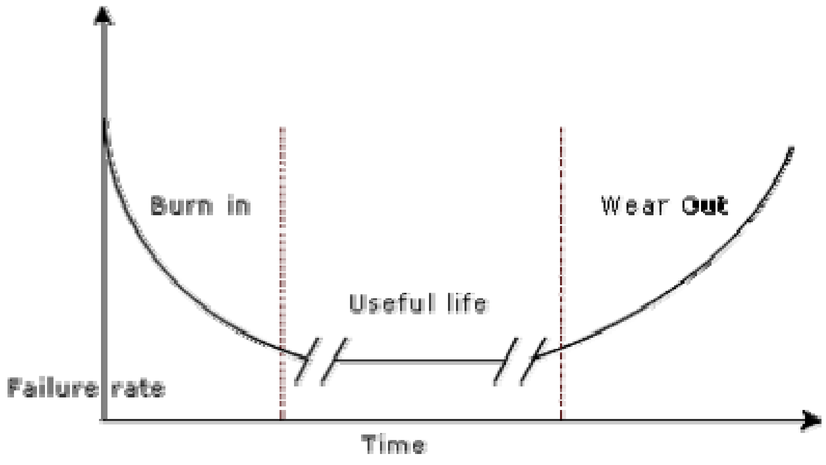
The reasons why software reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

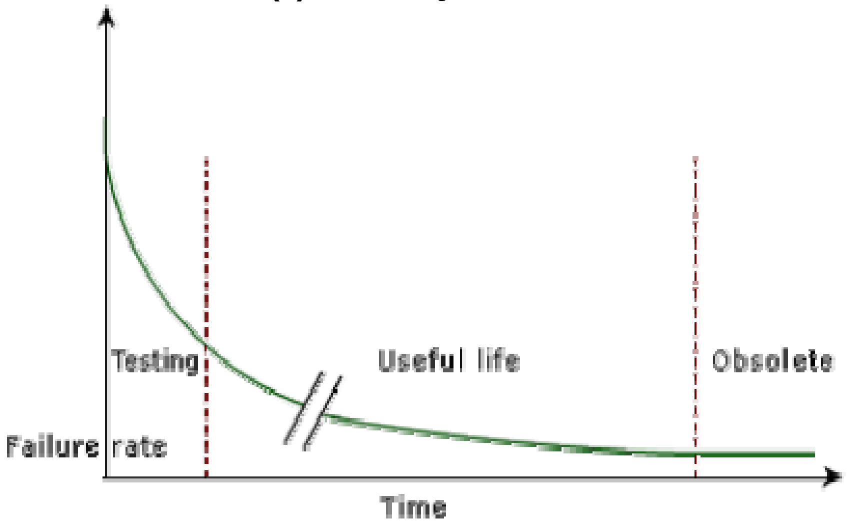
Hardware reliability vs. software reliability differ

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decreaseif a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase).

The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. . For hardware products, it canbe observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and thefailure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rateis at it’s highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. Thiserror removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.



(a) Hardware product



(b) Software product

Fig. : Change in failure rate of a product

Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performancemeasurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

- **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). ROCOF measure of a software product can be obtained by observing thebehavior of a software product in operation over a specified time interval and then recording the total number of failures occurring duringthe interval.
- **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t_1, t_2, …, t_n. Then, MTTF can be calculated as $\sum_{i=1}^{n} \frac{t_{i+1} - t_i}{(n-1)}$. It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements and the clock is stopped at these times.
- **Mean Time To Repair (MTTR).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
- **Mean Time Between Failure (MTBR).** MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failureis expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.
- **Probability of Failure on Demand (POFOD).** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in afailure.
- **Availability.** Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, which aresupposed to be never down and where repair and restart time are significant and loss of service during that time is important.

Classification of software failures

A possible classification of failures of software products into five different types isas follows:

- **Transient.** Transient failures occur only for certain input values while invoking a function of the system.
- **Permanent.** Permanent failures occur for all input values while invoking a function of the system.
- **Recoverable.** When recoverable failures occur, the system recovers with or without operator intervention.
- **Unrecoverable.** In unrecoverable failures, the system may need to berestarted.
- **Cosmetic.** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of onceto invoke a given function through the graphical user interface.

Reliability growth models

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

Jelinski and Moranda Model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in fig. However, thissimple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.

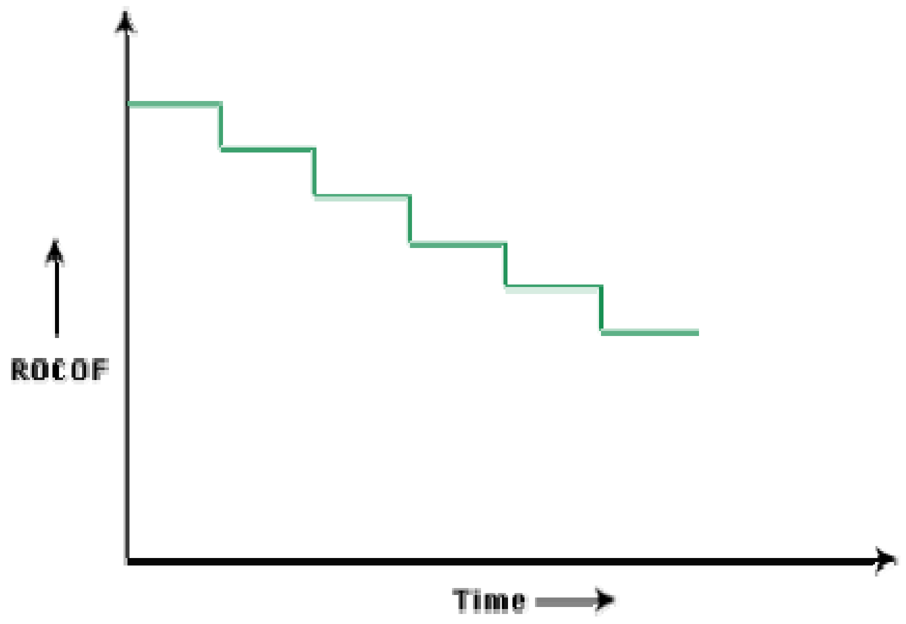


Fig. 13.2: Step function model of reliability growth

Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases (Fig.). It treat's an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

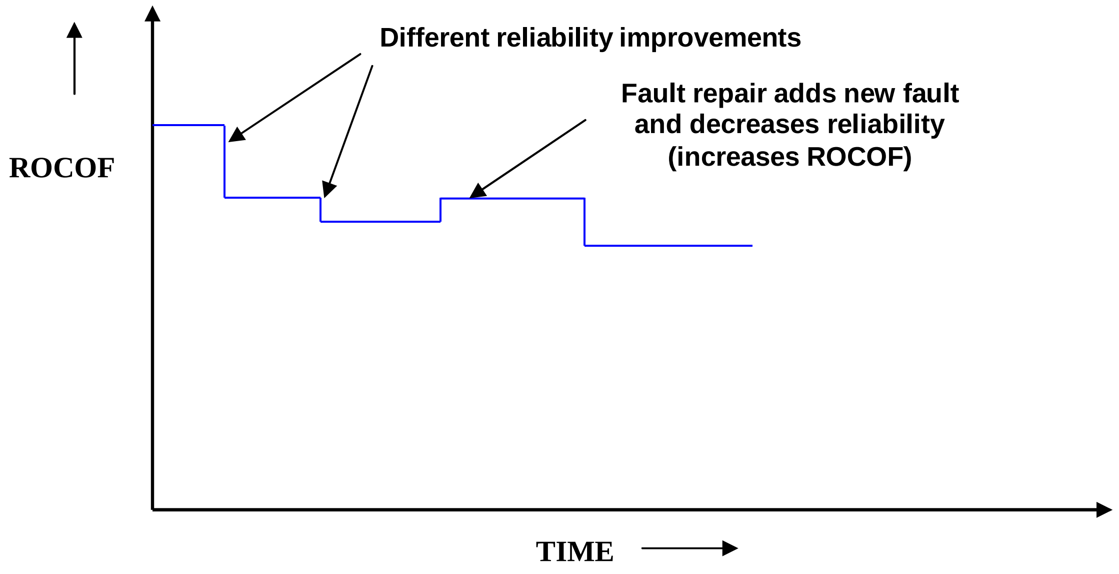

Different reliability improvements
Fault repair adds new fault and decreases reliability (increases ROCOF)
ROCOF
TIME

Fig. : Random-step function model of reliability growth

Chapter-12
Characteristics ofSoftware Maintenance

Necessity of software maintenance

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, 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 upwith the newer interface. For instance, a software product may need to bemaintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Thereforeit can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

Types of software maintenance

There are basically three types of software maintenance. These are:

- **Corrective:** Corrective maintenance of a software product is necessary torectify the bugs observed while the system is in use.
- **Adaptive:** A software product might need maintenance when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware orsoftware.
- **Perfective:** A software product needs maintenance to support the new features that users want it to support, to change different functionalities of

  the system according to customer demands, or to enhance the performance of the system.

Problems associated with software maintenance

Software maintenance work typically is much more expensive than what it shouldbe and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason beingthat software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenancework. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else’s work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

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 of its functionalities. A process model for reverse engineering has been shown in fig.1. A program can be reformatted using any of the several available prettyprinter programs which layout the program neatly. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex

nested conditionals in the program can be replaced by simpler conditionalstatements or whenever appropriate by case statements.

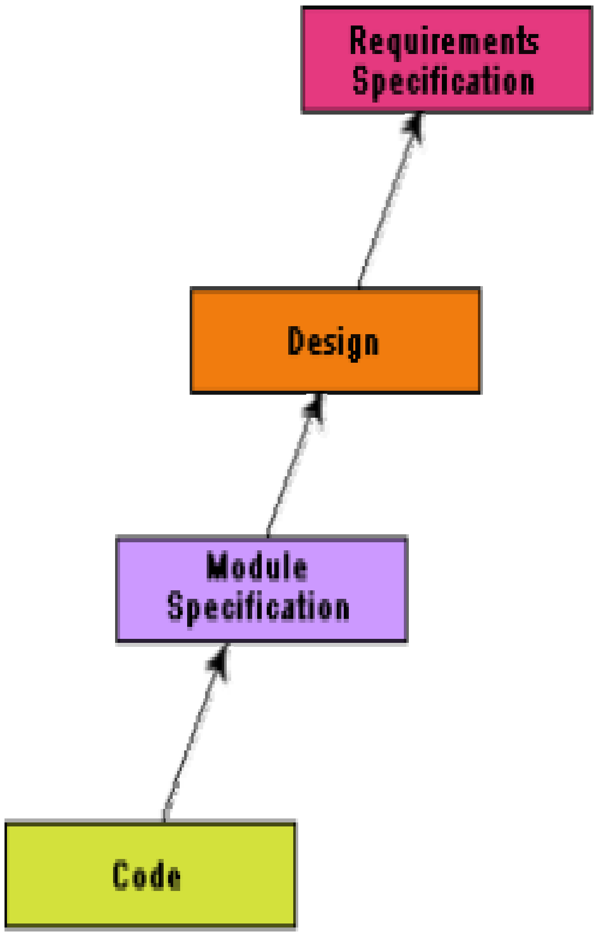


Fig. : A process model for reverse engineering

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 fig. . In order to extract the design, a full understanding of the code is needed. Some automatic tools canbe 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.

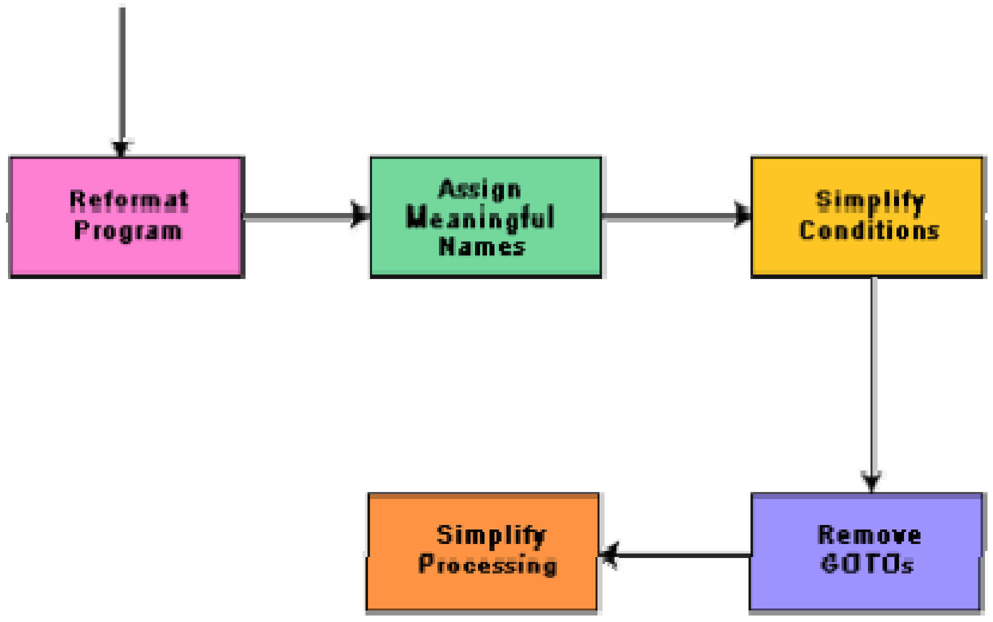


Fig. : Cosmetic changes carried out before reverse engineering

Legacy software products

It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), andlack of personnel knowledgeable in the product. Many of the legacy systemswere developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

Factors on which software maintenance activities depend

The activities involved in a software maintenance project are not unique and depend on several factors such as:

- the extent of modification to the product required
- the resources available to the maintenance team
- the conditions of the existing product (e.g., how structured it is, howwell documented it is, etc.)
- the expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all

the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineeringcycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

Software maintenance process models

Two broad categories of process models for software maintenance can be proposed. The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later. This maintenance process is graphically presented in fig.. In this approach, the project starts by gathering the requirements for changes. The requirements are next analyzed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle team, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localize the bugs.

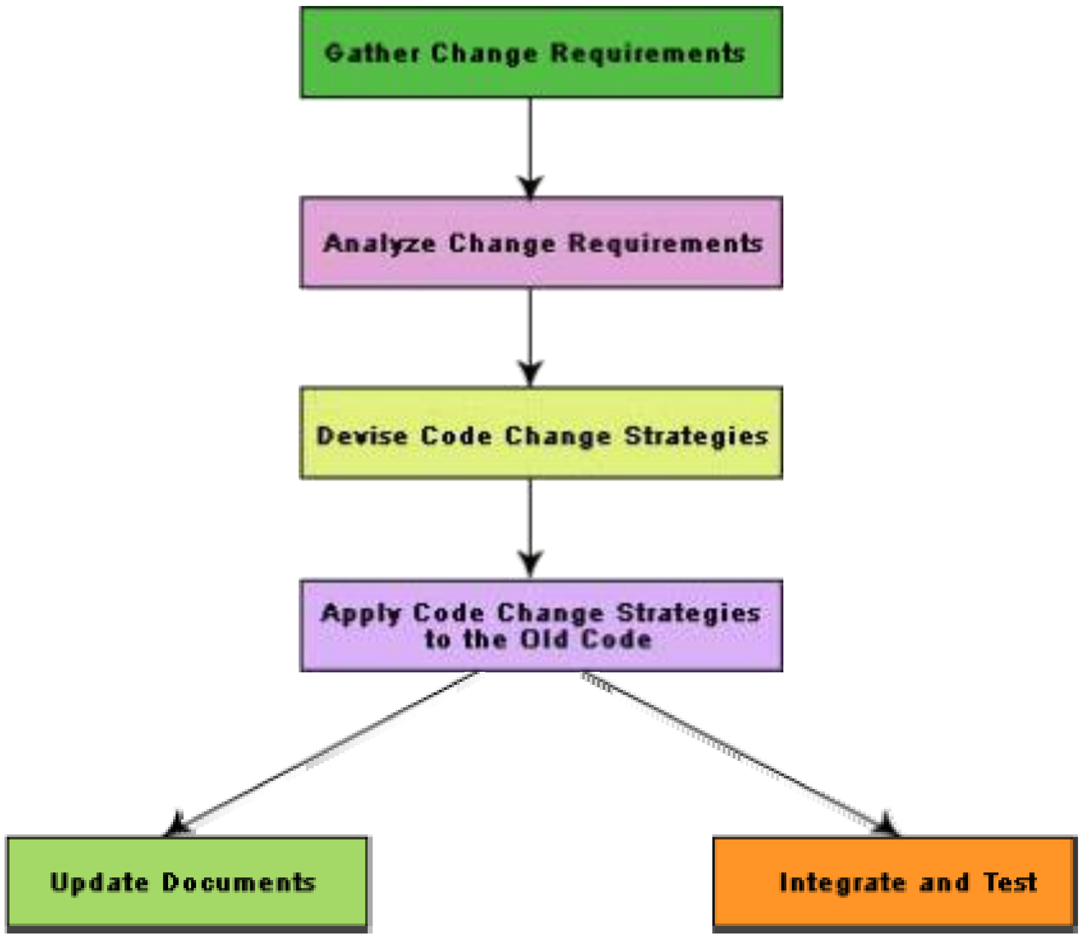

Gather Change Requirements
Analyze Change Requirements
Devise Code Change Strategies
Apply Code Change Strategies to the Old Code
Update Documents
Integrate and Test

Fig. : Maintenance process model 1

The second process model for software maintenance is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software reengineering. This process model is depicted in fig. . The reverse engineering cycle is required for legacy products. During the reverse engineering, the old code is analyzed (abstracted) to extract the module specifications. The module specifications are then analyzed to produce the design. The design is analyzed (abstracted) to produce the original requirements specification. The change requests are then applied to this requirements specification to arrive at the new requirements specification. At the design, module specification, and coding a substantial reuse is made from the reverse engineered products. An important advantage of this approach is that it produces a more structured design compared to what the original product had, produces good documentation, and very often results in increased efficiency. The efficiency improvements are brought about by a more efficient design. However, this approach is more costly than the first approach. An empirical study indicates that process 1 is preferable when the amount of rework is no more than 15% (as shown in fig. 14.5). Besides the amount of rework, several other factors might affect the decision regarding using process model 1 over process model 2:

- Reengineering might be preferable for products which exhibit a highfailure rate.
- Reengineering might also be preferable for legacy products havingpoor design and code structure.

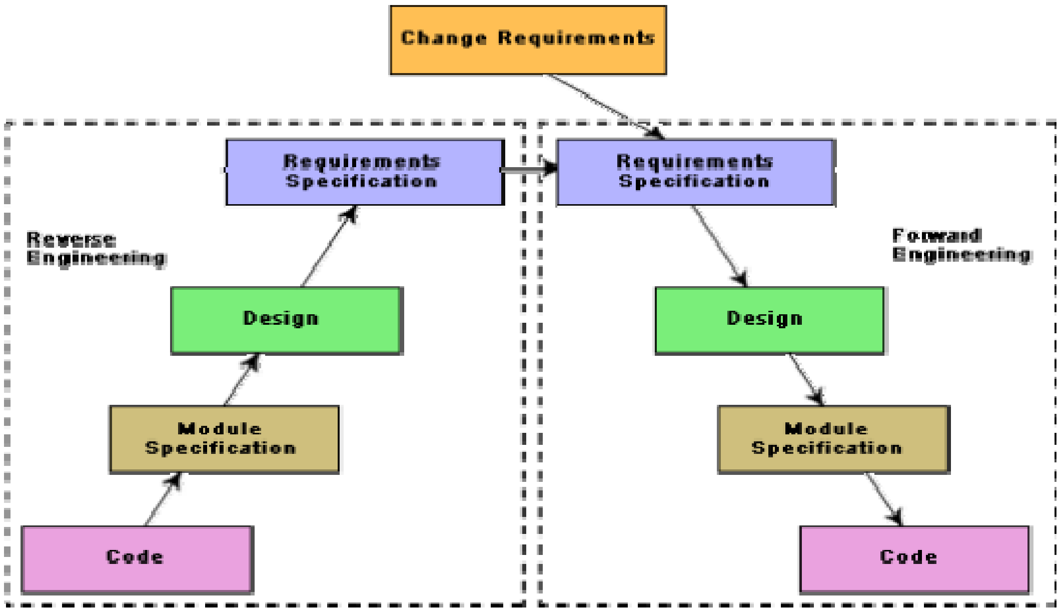


Fig. : Maintenance process model 2

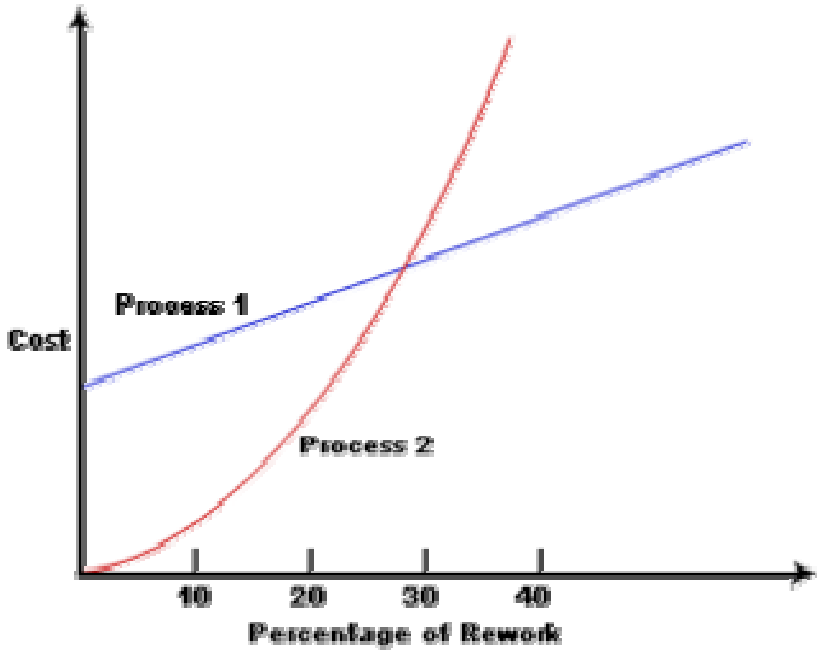


Fig.: Empirical estimation of maintenance cost versus percentage rework

Software reengineering

Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering as shown in the fig. .

Estimation of approximate maintenance cost

It is well known that maintenance efforts require about 60% of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the Annual Change Traffic (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

where, $KLOC_{added}$ is the total kilo lines of source code added during maintenance. $KLOC_{deleted}$ is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and the code deleted. The annual change traffic (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

$$\text{maintenance cost} = ACT \times \text{development cost.}$$

Most maintenance cost estimation models, however, yield only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardwarerequirements, software complexity, etc.

Different Characteristics of CASE Tools

Hardware and environmental requirements

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, it can be emphasized on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and this can be chosen for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients who run different modules access data dictionary through this server. The data dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common.

The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows to share information between users and projects. The data dictionary provides consistent view of all project entities, e.g. a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view ofthe data dictionary. This means that it should allow back up/restore, copy, cleaning part of the data dictionary, etc.

The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

Documentation support

The deliverable documents should be organized graphically and should be ableto incorporate text and diagrams from the central repository. This helps in producing up-to-date documentation. The CASE tool should integrate with one or more of the commercially available desktop publishing packages. It should be possible to export text, graphics, tables, data dictionary reports to the DTPpackage in standard forms such as PostScript.

Project management support

The CASE tool should support collecting, storing, and analyzing information on the software project's progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

External interface

The CASE tool should allow exchange of information for reusability of design. The information which is to be exported by the CASE tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information. It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

Reverse engineering

The CASE tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code. If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchicaland network database to relational database systems.

Data dictionary interface

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

Second-generation CASE tool

An important feature of the second-generation CASE tool is the direct support of any adapted methodology. This would necessitate the function of a CASE administrator organization who can tailor the CASE tool to a particular methodology. In addition, the second-generation CASE tools have following features:

- **Intelligent diagramming support.** The fact that diagramming techniques are useful for system analysis and design is wellestablished. The future CASE tools would provide help to aesthetically and automatically lay out the diagrams.
- **Integration with implementation environment.** The CASE tools should provide integration between design and implementation.
- **Data dictionary standards.** The user should be allowed to integrate many development tools into one environment. It is highly unlikely that any one vendor will be able to deliver a total solution. Moreover, a preferred tool would require tuning up for a particular system. Thus the user would act as a system integrator. This is possibly only if some standard on data dictionary emerges.
- **Customization support.** The user should be allowed to define new types of objects and connections. This facility may be used to build some special methodologies. Ideally it should be possible to specifythe rules of a methodology to a rule engine for carrying out the necessary consistency checks.

Architecture of a CASE environment

The architecture of a typical modern CASE environment is shown diagrammatically in fig. . The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository. Characteristics of a tool set have been discussed earlier.

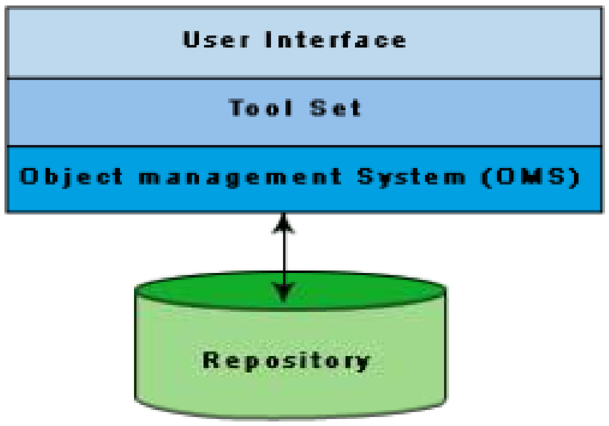


Fig.: Architecture of a Modern CASE Environment

User Interface

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with thedifferent tools and reducing the overhead of learning how the differenttools are used.

Object Management System (OMS) and Repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities such into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. Thereare a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with perhaps a few instances of each. Thus the object management system takes care of appropriately mapping into the underlying storage management system.

Basic Ideas on Client-Server Software Development andClient-Server Architecture

Client-server software

A client is basically a consumer of services and server is a provider of servicesas shown in fig. . A client requests some services from the server and the server provides the required services to the client. Client and server are usually software components running on independent machines. Even a single machine can sometimes acts as a client and at other times a server depending on the situations. Thus, client and server are mere roles.

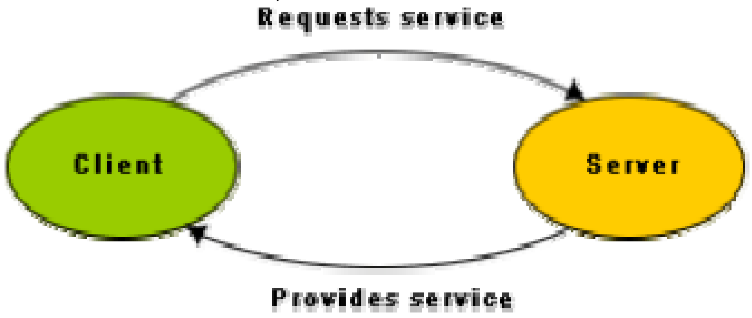


Fig. : Client-server model

Example:
A man was visiting his friend's town in his car. The man had a handheldcomputer (client). He knew his friend's name but he didn't know his friend's address. So he sent a wireless message (request) to the nearest "address server" by his handheld computer to enquire his friend's address. The message first came to the base station. The base station forwarded that message through landline to local area network where the server is located. After some processing,LAN sent back that friend's address (service) to the man.

Advantages of client-server software

The client-server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability as compared to centralized, mainframe, time sharing computing.

Factors for feasibility and popularity of client-server solutions

Client-server concept is not a new concept. It already existed in the society for long time. A doctor is a client of a barber, who in turn is a client of the lawyer and so forth. Something can be a server in some context and a client in some other context. So client and server are mere roles as shown in fig. .

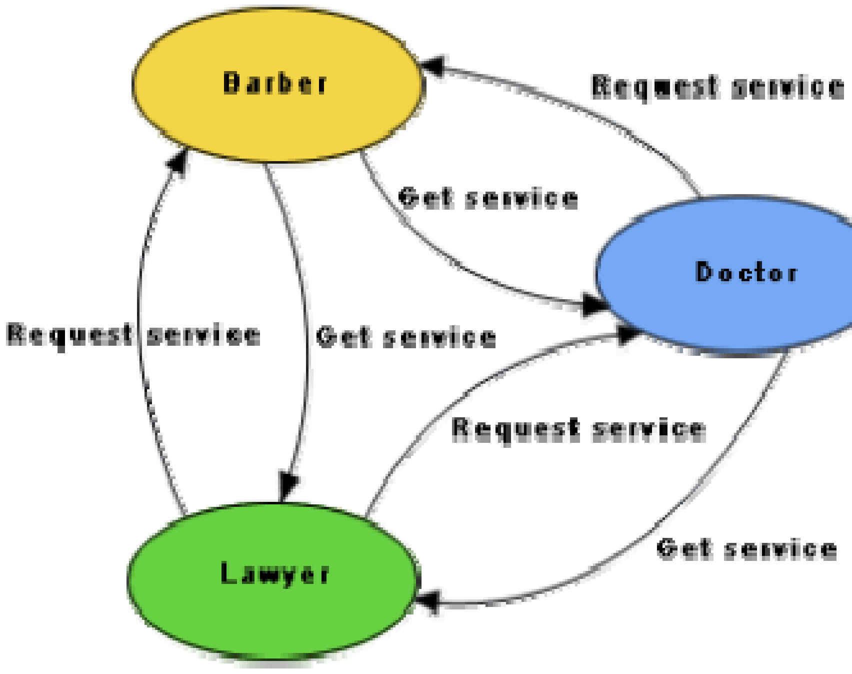


Fig.: Client and server as roles

There are many reasons for the popularity of client-server software development. Some reasons are:

- Computers have become small, decentralized and cheap
- Networking has become affordable, reliable, and efficient.
- Client-server systems divide up the work of computing among many separate machines. Thus client-server solutions are modular and loosely coupled. So they are easy to develop and maintain.

Advantages of client-server software development

There are many advantages of client-server software products as compared tomonolithic ones.

These advantages are:

- **Simplicity and modularity –** Client and server components are loosely coupled and therefore modular. These are easy to understand and develop.
- **Flexibility** – Both client and server software can be easily migrated acrossdifferent machines in case some machine becomes unavailable or crashes. The client can access the service anywhere. Also, clients and servers can be added incrementally.
- **Extensibility** – More servers and clients can be effortlessly added.
- **Concurrency** – The processing is naturally divided across several machines. Clients and servers reside in different machines which can operate in parallel and thus processing becomes faster.
- **Cost Effectiveness** – Clients can be cheap desktop computers whereas severs can be sophisticated and expensive computers. To use a sophisticated software, one needs to own only a cheap client and invoke the server.
- **Specialization** – One can have different types of computers to run different types of servers. Thus, servers can be specialized to solve some specific problems.
- **Current trend** – Mobile computing implicitly uses client-server technique. Cell phones (handheld computers) are being provided with small processing power, keyboard, small memory, and LCD display. Cell phones cannot really compute much as they have very limited processing power and storage capacity but they can act as clients. The handhold computers only support the interface to place requests on some remote servers.
- **Application Service Providers (ASPs) –** There are many application software products which are very expensive. Thus it makes prohibitively costly to own those applications. The cost of those applications often runs into millions of dollars. For example, a Chemical Simulation Software named "*Aspen*" is very expensive but very powerful. For small industries it would not be practical to own that software. Application Service Providers can own ASPEN and let the small industries use it as client and charge them based on usage time. A client and simply logs in and ASP charges according to the time that the software is used.
- **Component-based development** – It is the enabler of the client-server technology. Component-based development is radically different from traditional software development. In component-based development, adeveloper essentially integrates pre-built components purchased off-the- shelf. This is akin to the way hardware developers integrate ICs on a Printed Circuit Board (PCB). Components might reside on different computers which act as servers and clients.
- **Fault-tolerance** – Client-server based systems are usually fault-tolerant. There can be many servers. If one server crashes then client requests canbe switched to a redundant server.

There are many other advantages of client-server software. For example, we canlocate a server near to the client. There might be several servers and the client requests can be routed to the nearest server. This would reduce thecommunication overhead.

Disadvantages of client-server software

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

- **Security** – In a monolithic application, implementation of security is very easy. But in a client-server based development a lot of flexibility is provided and a client can connect from anywhere. This makes it easy for hackers to break into the system. Therefore, ensuring security in client-server system is very challenging.
- **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. This problem arises due to the flexibility given that any client can connect anytime required.
- **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, language, etc.
- **Inconsistency** – Replication of servers is a problem as it can make datainconsistent.

Host-slave computing vs. client-server computing

An example of a host-slave computing is a Railway-reservation system. The software is divided into two parts – one resides on the terminals of the booking clerks. The master at any time directs the slaves what to do. A slave can only make requests and master takes over and tells what to do.

On the other hand, in a client-server computing, different components are interfaced using an open protocol. In a master-slave they are proprietary. An example of a client-server system is a world wide web.

Two-tier client-server architecture

The simplest way to connect clients and servers is a two-tier architecture as shown in fig. . In a two-tier architecture, any client can get service from any server by initiating a request over the network. With two tier client-server architectures, the user interface is usually located in the user's desktop and the services are usually supported by a server that is a powerful machine that can service many clients. Processing is split between the user interface and the database management server. There are a number of software vendors who provide tools to simplify development of applications for the two-tier client-server architecture.

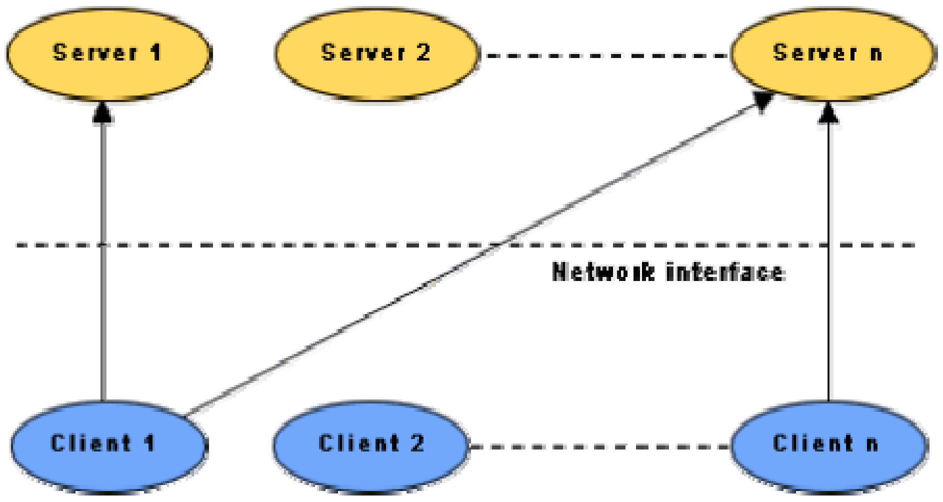


Fig. : Two-tier client-server architecture

Limitations of two-tier client-server architecture

A two tier architecture for client-server applications is an ideal solution but is not practical. The problem is that client and server components are manufactured by different vendors and the different vendors come up with different sets of interfaces and different implementation standards. That's the reason why clients and servers can often not talk to each other. A two tier architecture can work onlyin an open environment. In an open environment all components have standard interfaces. However, till date an open environment is still far from becoming practical.

Three-tier client-server architecture

The three-tier architecture overcomes the important limitations of the two-tier architecture. In the three-tier architecture, a middleware was added between the user system interface client environment and the server environment as shown in fig. . The middleware keeps track of all server locations. It also translates client's requests into server understandable form. For example, if the middleware provides queuing, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer tothe client.

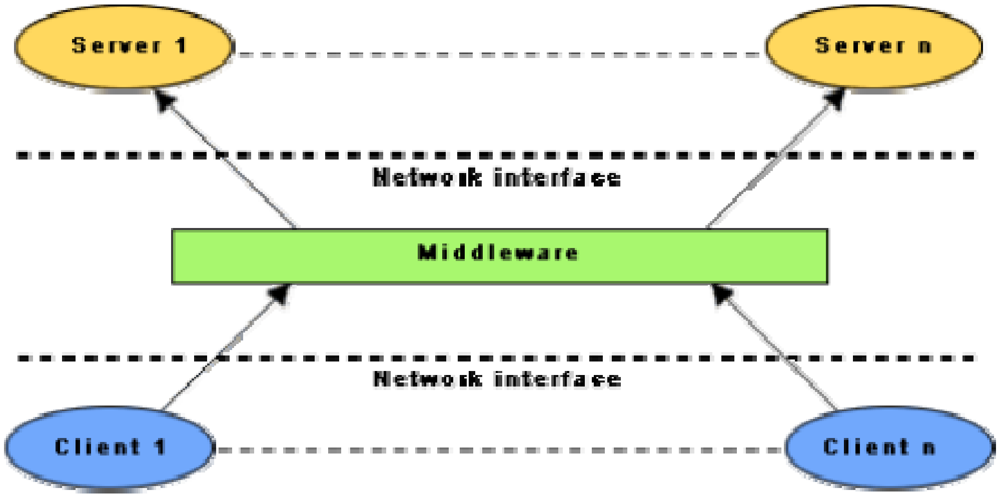


Fig. : Three-tier client-server architecture

Functions of middleware

The middleware performs many activities such as:

- It knows the addresses of servers. So, based on client requests, it canlocate the servers.
- It can translate between client and server formats of data and vice versa.

Popular middleware standards

Two popular middleware standards are:

- CORBA (Common Object Request Broker Architecture)
- COM/DCOM

CORBA is being promoted by Object Management Group (OMG), a consortium of a large number of computer industries such as IBM, HP, Digital etc. Actually OMG is not a standards body, they only try to promotede facto standards. They don't have any authority to make or enforce standards. They just try to popularize good solutions with the hope that if they become highly popular they would automatically become standard.

COM/DCOM is being promoted by Microsoft alone.

Client-server software

A client is basically a consumer of services and server is a provider of servicesas shown in fig. . A client requests some services from the server and the server provides the required services to the client. Client and server are usually software components running on independent machines. Even a single machine can sometimes acts as a client and at other times a server depending on the situations. Thus, client and server are mere roles.

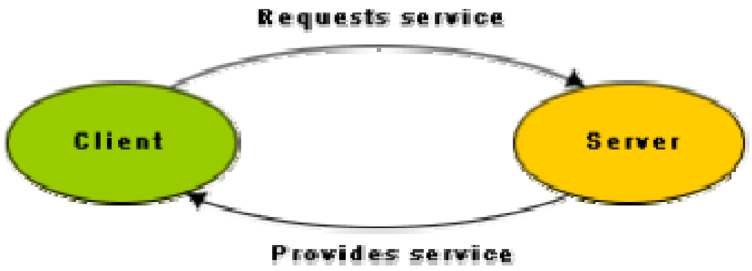


Fig. : Client-server model

Example:
A man was visiting his friend's town in his car. The man had a handheldcomputer (client). He knew his friend's name but he didn't know his friend's address. So he sent a wireless message (request) to the nearest "address server" by his handheld computer to enquire his friend's address. The message first came to the base station. The base station forwarded that message through landline to local area network where the server is located. After some processing,LAN sent back that friend's address (service) to the man.

Advantages of client-server software

The client-server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability as compared to centralized, mainframe, time sharing computing.

Factors for feasibility and popularity of client-server solutions

Client-server concept is not a new concept. It already existed in the society for long time. A doctor is a client of a barber, who in turn is a client of the lawyer and so forth. Something can be a server in some context and a client in some other context. So client and server are mere roles as shown in fig. .

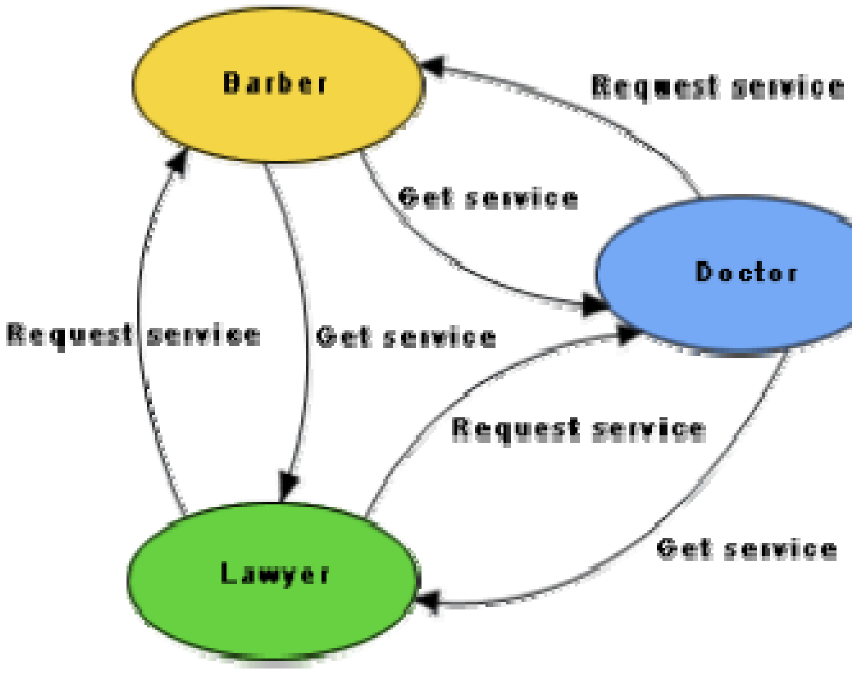


Fig. : Client and server as roles

There are many reasons for the popularity of client-server software development. Some reasons are:

- Computers have become small, decentralized and cheap
- Networking has become affordable, reliable, and efficient.
- Client-server systems divide up the work of computing among many separate machines. Thus client-server solutions are modular and loosely coupled. So they are easy to develop and maintain.

Advantages of client-server software development

There are many advantages of client-server software products as compared tomonolithic ones.

These advantages are:

- **Simplicity and modularity –** Client and server components are loosely coupled and therefore modular. These are easy to understand and develop.
- **Flexibility** – Both client and server software can be easily migrated across different machines in case some machine becomes unavailable or crashes. The client can access the service anywhere. Also, clients and servers can be added incrementally.
- **Extensibility** – More servers and clients can be effortlessly added.
- **Concurrency** – The processing is naturally divided across several machines. Clients and servers reside in different machines which can operate in parallel and thus processing becomes faster.
- **Cost Effectiveness** – Clients can be cheap desktop computers whereas severs can be sophisticated and expensive computers. To use a sophisticated software, one needs to own only a cheap client and invoke the server.
- **Specialization** – One can have different types of computers to run different types of servers. Thus, servers can be specialized to solve some specific problems.
- **Current trend** – Mobile computing implicitly uses client-server technique. Cell phones (handheld computers) are being provided with small processing power, keyboard, small memory, and LCD display. Cell phones cannot really compute much as they have very limited processing power and storage capacity but they can act as clients. The handhold computers only support the interface to place requests on some remote servers.
- **Application Service Providers (ASPs) –** There are many application software products which are very expensive. Thus it makes prohibitively costly to own those applications. The cost of those applications often runs into millions of dollars. For example, a Chemical Simulation Softwarenamed "*Aspen*" is very expensive but very powerful. For small industries it would not be practical to own that software. Application Service Providers can own ASPEN and let the small industries use it as client and charge them based on usage time. A client and simply logs in and ASP charges according to the time that the software is used.
- **Component-based development** – It is the enabler of the client-server technology. Component-based development is radically different from traditional software development. In component-based development, a developer essentially integrates pre-built components purchased off-the- shelf. This is akin to the way hardware developers integrate ICs on a Printed Circuit Board (PCB). Components might reside on different computers which act as servers and clients.
- **Fault-tolerance** – Client-server based systems are usually fault-tolerant. There can be many servers. If one server crashes then client requests canbe switched to a redundant server.

There are many other advantages of client-server software. For example, we canlocate a server near to the client. There might be several servers and the client requests can be routed to the nearest server. This would reduce thecommunication overhead.

Disadvantages of client-server software

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

- **Security** – In a monolithic application, implementation of security is very easy. But in a client-server based development a lot of flexibility is provided and a client can connect from anywhere. This makes it easy for hackers to break into the system. Therefore, ensuring security in client- server system is very challenging.
- **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. This problem arises due to the flexibility given that any client can connect anytime required.

- **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, language, etc.
- **Inconsistency** – Replication of servers is a problem as it can make datainconsistent.

Host-slave computing vs. client-server computing

An example of a host-slave computing is a Railway-reservation system. The software is divided into two parts – one resides on the terminals of the booking clerks. The master at any time directs the slaves what to do. A slave can only make requests and master takes over and tells what to do.

On the other hand, in a client-server computing, different components are interfaced using an open protocol. In a master-slave they are proprietary. An example of a client-server system is a world wide web.

Two-tier client-server architecture

The simplest way to connect clients and servers is a two-tier architecture as shown in fig. . In a two-tier architecture, any client can get service from any server by initiating a request over the network. With two tier client-server architectures, the user interface is usually located in the user's desktop and the services are usually supported by a server that is a powerful machine that can service many clients. Processing is split between the user interface and the database management server. There are a number of software vendors who provide tools to simplify development of applications for the two-tier client-server architecture.

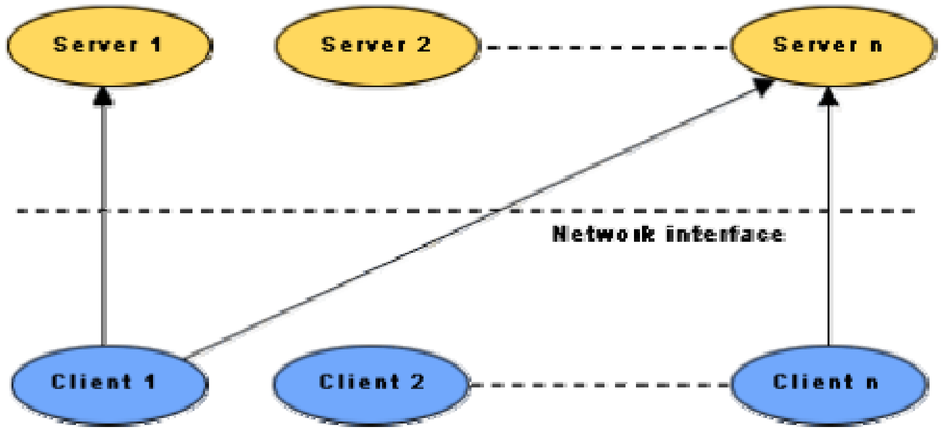


Fig.: Two-tier client-server architecture

Limitations of two-tier client-server architecture

A two tier architecture for client-server applications is an ideal solution but is not practical. The problem is that client and server components are manufactured by different vendors and the different vendors come up with different sets of interfaces and different implementation standards. That's the reason why clients and servers can often not talk to each other. A two tier architecture can work onlyin an open environment. In an open environment all components have standard interfaces. However, till date an open environment is still far from becoming practical.

Three-tier client-server architecture

The three-tier architecture overcomes the important limitations of the two-tier architecture. In the three-tier architecture, a middleware was added between the user system interface client environment and the server environment as shown infig. 17.4. The middleware keeps track of all server locations. It also translates client's requests into server understandable form. For example, if the middleware provides queuing, the client can deliver its request to the middleware and disengage because the middleware will access the data and return the answer tothe client.

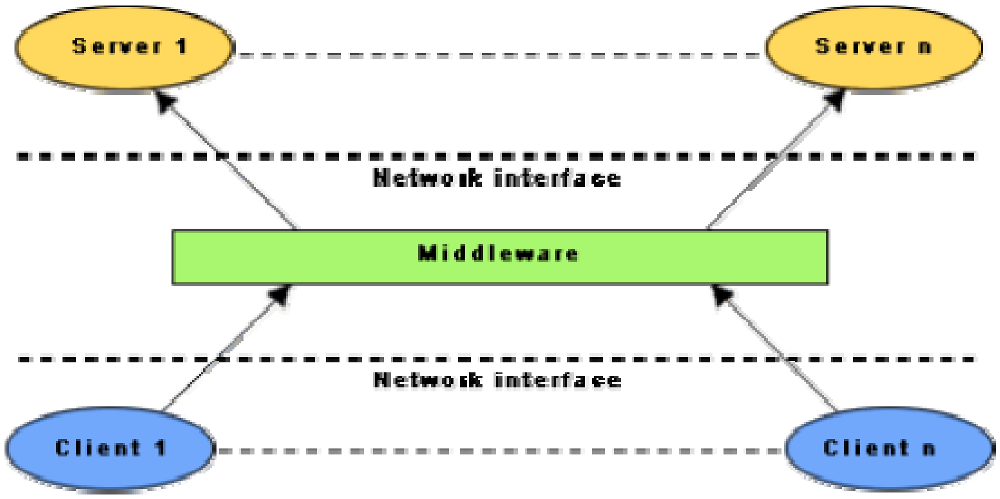


Fig. : Three-tier client-server architecture

Functions of middleware

The middleware performs many activities such as:

- It knows the addresses of servers. So, based on client requests, it canlocate the servers.
- It can translate between client and server formats of data and vice versa.

Popular middleware standards

Two popular middleware standards are:

- CORBA (Common Object Request Broker Architecture)
- COM/DCOM

CORBA is being promoted by Object Management Group (OMG), a consortium of a large number of computer industries such as IBM, HP, Digital etc. Actually OMG is not a standards body, they only try to promotede facto standards. They don’t have any authority to make or enforce standards. They just try to popularize good solutions with the hope that if they become highly popular they would automatically become standard. COM/DCOM is being promoted by Microsoft alone.