MCA 403

Object Oriented Software Engineering

SATYAPRAKASH SWAIN

satyaimit@gmail.com

Module - 1

INTRODUCTION TO SOFTWARE ENGINEERING:

The term **software engineering** is composed of two words, software and engineering. **Software** is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

So, we can define **software engineering** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

IEEE defines software engineering as:

The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

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 absolutely necessary to achieve a good quality software cost effectively.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: ***abstraction*** and ***decomposition*****.** The principle of abstraction 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 is a 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 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.

NEED OF SOFTWARE ENGINEERING:

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software -** It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.
- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

CHARACTERESTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality

- Dependability
- Security
- Safety

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the ever- changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products

SOFTWARE CRISIS:

Software engineering appears to be among the few options available to tackle the present software crisis.

Let us explain the present software crisis in simple words, by considering 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 (Fig 1.6)

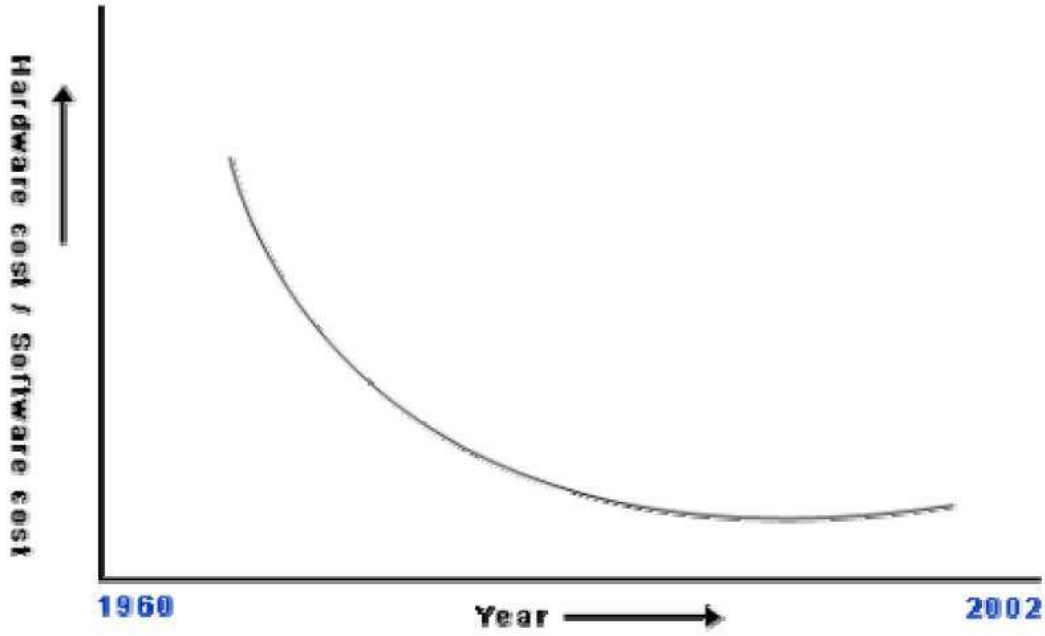


Fig. 1.6: Change in the relative cost of hardware and software over time

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 also presented a lot of other problems to the customers such as: 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.

Due to ineffective development of the product characterized by inefficient resource usage and time and cost over-runs. Other factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity Improvements.

S/W crisis from programmer point of view:-
i)Problem of compatibility.
ii)Problem of Portability.
iii)Proclaiming documentation.
iv)Problem of pirated s/w.
v)Problem in co-ordination of work of different people.
vi)Problem of proper maintenance.

S/W crisis from user point of view:-
i)s/w cost is very high.
ii)Price of h/w grows down.
iii)Lack of development specification.
iv)Problem of different s/w versions.
v)Problem of bugs or errors.

HANDLING COMPLEXITY THROUGH ABSTRACTION:

First thing is: what do you mean by **abstraction**? Abstraction is basically leaving out unnecessary parts and focusing on some parts that we require. Given a large problem if we look at the problem with entirety, then it appears very complex. We would like to focus only one aspect of the problem and ignore the rest. So, that is basically called as Obstruction. So, we focus our attention only one aspect of the problem and ignore other aspects that are irrelevant to the point that we are focusing and this is also called as Model Building.

Here we simplify a problem by omitting the unnecessary details and that is basically constructing a model. Every abstraction requires construction of a model and a model focuses on one aspect, and it ignores rest. For example: we have a very large and complex building to be developed and we just want to focus that how will it appear. Then we will construct a frontal model of a building, we ignore the rest what is its material it is build off, what is the thickness what is the floor plan, internal design, and so on we just focus on the frontal view of the building. So to summarize this abstraction, every abstraction require some model building and a model basically focuses on some aspect and ignores the rest.

Now, let us see some examples of how abstraction can help. Let us look at a hypothetical case that you have been asked to develop an overall understanding of some country. Maybe, you be later given the responsibility to rate the marketing department of some company for a country let us say, and you have been asked that please develop a overall understanding of a country. So, to do this what would you do? Would you go to that country, move around try to meet people, look at where are the mountains, where the rivers and so on? If you really did that then it will be extremely complex task, it will take you tens or hundreds of year still you will not be able to develop a proper understanding. You will not like to meet all the people examine every tree in the country, river, mountain and so on. But what you would really do is refer to various types of maps of the country. The maps are basically model or an abstraction of a country. You will study the political map, that focuses on the different provinces, capital, major cities, railway road connectivity and so on. You will look at the physical map: we will try to find out the vegetation, the elevation of the different places, rivers, sea shore and so on. So, an abstraction can help solve the complex problem very easily. And one thing to notice is that there are various types of abstraction for the same problem. But then, is it that for every problem we can be happy with just one abstraction. Can just one abstraction help us develop a full understanding of a problem? No. Several abstraction to the same problem can be created. These focus on some specific aspect and ignore the rest and the different types of models help to understand different aspects of the problem. For example, in the case of a building we would like a frontal model, we like to develop a prototype, we like to develop a floor plan, we like to develop a thermal model of the building and so on. So, for the same problem various models can be created each focusing on some aspect of the problem.

But, if the problem is extremely complex a single level of abstraction may itself be again complicated. We would like to create a hierarchy of obstruction, so if these are the problems we will create a first level abstraction, then we will create a second level abstraction, then an abstraction of those second level abstraction and so on. And if we look

at the root level that will be the simplest representation. And as we understand the simplest representation we might look at the next level of the models and so on.

So, for very complex problems we would create a hierarchy of obstructions. As we proceed with software design we will look at hierarchy of obstructions that are used, and in this hierarchical model a model at a any level is actually the details on which the next level of model is built. So, a model is an abstraction of a lower level model and we say that it is a implementation of the higher level model.

Let us look at abstraction of a complex problem; how a hierarchy can help, a hierarchical set up models can help. Let us look at a very complex problem that you have been asked to understand all the life forms that inhabit the earth. And remember there are billions or trillions of species.

So, in order to understand this life form if you go on examining various species nobody can complete the study in his lifetime. But then, what you do is you create an hierarchy of abstraction.

If you look at a biology book you will find such an abstraction, because they want to explain the reader the simplest way and they have constructed this abstraction that at the top level there are only three types of living organisms: the animals, plant, and the fungus. And then we have the mollusca, chordate, etcetera, etcetera. And there are further hierarchies until you reach the bottom most. And these are the species I was just saying that there will be trillions or species here.

So, for a complex problem the number of layers in the hierarchy can be large and each of these layers is basically a model.

Now, let us look at the other important principle which is **decomposition**. Decomposition as it means is to decompose a complex problem into small independent parts, because the problem in its entirety is extremely complex hard to understand it takes exponentially large time, but we would like to break it into small problems. And then we look at examine and understand the small problems and then once we put them together we have the understanding of the entire problem.

One of the very popular example of the decomposition principle is that if you try to break a bunch of sticks tied together it would be extremely complex, but then if you decompose it then you can break the sticks individually. But then, one what a question when using the decomposition principle is that a arbitrary decomposition of a problem may not help. We have to decompose such that each small problem can be solved separately. If we want to let us say draw an elephant, if you decompose in arbitrary ways it does not help. We thought that we will just draw each of them separately and then we will put them together, but each individual part drawing that may not be much easier.

So, we need to decompose properly; we will see how to properly decompose a problem and then we can solve it easily. And this is also a corner stone in all software engineering principles.

OVERVIEW OF SOFTWARE DEVELOPMENT ACTIVITIES:

Now, let us look at some major differences, in the way software is being written now one is that the software engineering principles. Themselves have evolved and the other is the software itself has evolved. The software writing problem is not the same as the problem that now we are solving.

The, problems have become much more complex, but at the same time these are only incrementally different from the problem that has been solved. Let us let us look at these issues now; the differences between exploratory style and the modern software development practices. Compared to 1950s or 60s where the exploratory style was used, now the life cycle models are used every program is developed according to an adopted life cycle model, where there are a set of stages and the development occurs through those stages. The stages typically the requirements analysis specification design, coding, testing, and so, on.

One major difference between the way program was written 1950s and 60s to now is that the emphasis has shifted from error correction to error prevention. The earlier technique the exploratory style was to first write the program complete somehow and then start correcting the errors, there will be hundreds of errors keep on eliminating one by one until all errors are eliminated. But, now the emphasis is error prevention as the program is written we track the errors and remove it from there itself. So, that during testing we have very less number of errors to correct, or in other words as soon as we have a mistake committed by the programmer in program development, we try to detect it as close to the time of commitment of the error as possible the main advantage of this is that it is very cost effective.

If, you correct the error after testing then you will have to first find out where exactly the error has occurred make changes to the code again test and so on. Here, before testing we just identify the place where the error is there we do not have to look at we just look through or review the code and so on, review the specification review the design identify the error corrected there itself and do not wait for error to be discussed detected and determined during the testing phase. The software is being developed through number of phases and in each phase, wherever a program commits a mistake. It is detected and corrected in that same phase as far as possible.

The exploratory style program development was basically coding, start writing the code as soon as the problem is given, but right now in the modern development practice coding is actually a small part of the development activities. The major activities are specification design then coding and then testing.

Of course, lot of attention now is given to requirement specification then the design has to be done that, standard design techniques that are available. And, at the end of every phase reviews are conducted the main idea behind review is to detect errors as soon as these are committed by the programmer, do not wait for the error to be detected during testing. Reviews help us to detect the errors much earlier, and this is an efficient way to detect errors and therefore, reduces cost of development and the time of development.

The testing techniques have become really sophisticated we will look at them later in this course, and there are many standard testing techniques and tools are available. Earlier the programmer just wrote the code and then never know when it will get completed you will just have to ask the programmer, how far he has done in the coding?

But, even if he say that I am nearly complete completing the code, but then he might have too many bugs and he may take years to correct them. But, right now there is a visibility of the development somebody can see that the design is done, code is done or let us say the requirements are done. So, these documents improve the visibility, earlier there are no visibility you just have to ask the programmer that how far he has done? Right now you can see the documents and determine with what stages of development it is.

The increased visibility makes software project management much easier, earlier the project manager has to just ask the programmer how he is doing, how long he is? And, naturally the projects were going hey we are one year project taking 5 years was not uncommon, but now due to the increased visibility the project manager can very accurately determine when the project at what stage it is and how long it will take to complete the work?. Because, good documents now are produced later maintenance becomes easier several metrics are being used, which are used to determine the quality of the software, software project management and so on. There are many project management techniques that are being used like estimation scheduling monitoring mechanisms and also case tools are being used extensively.

SOFTWARE DEVELPOMENT PROCESS MODELS:

- n software development life cycle, various models are designed and defined. These models are called as Software Development Process Models.

- On the basis of project motive, the software development process model is selected for development.

Following are the different software development process models:

1) Big-Bang model
2) Code-and-fix model
3) Waterfall model
4) V model
5) Incremental model
6) RAD model
7) Agile model
8) Iterative model
9) Spiral model
10) Prototype model

1) Big-Bang Model

- Big-Bang is the SDLC(Software Development Life cycle) model in which no particular process is followed.

- Generally this model is used for small projects in which the development teams are small. It is specially useful in academic projects.
- This model is needs a little planning and does not follow formal development.
- The development of this model begins with the required money and efforts as an input.
- The output of this model is developed software, that may or may not be according to the requirements of the customer.

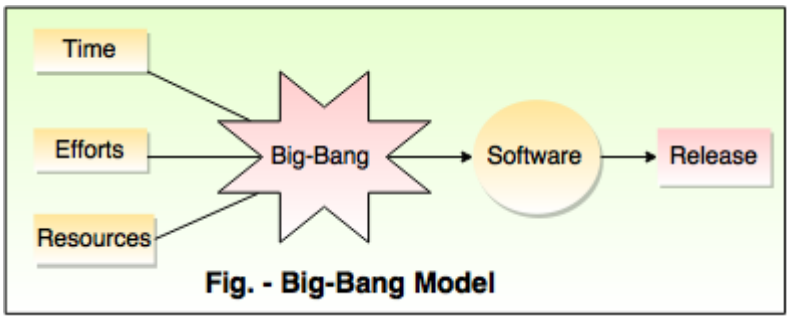


Fig. - Big-Bang Model

Advantages of Big-Bang model

- Big-Bang model is a simple model.
- It needs little planning.
- It is simple to manage. It needs just a few resources to be developed.
- It is useful for students and new comers.

Disadvantages of Big-Bang model

- It is a very high risk model.
- This model is not suitable for object oriented and complex projects.
- Big-Bang is poor model for lengthy and in-progress projects.

2) Code-and-fix Model

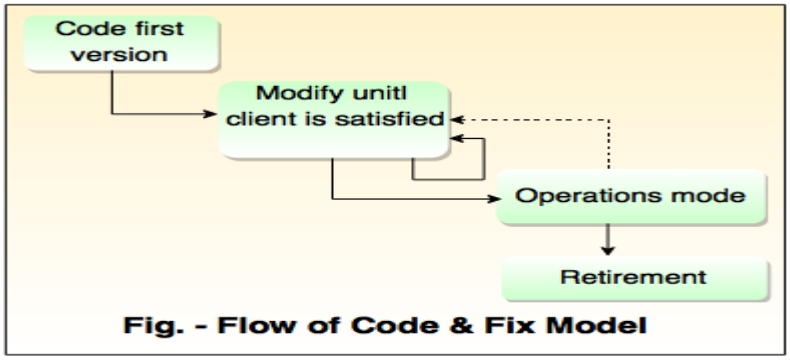


Fig. - Flow of Code & Fix Model

- Code and fix model is one step ahead from the Big-Bang model. It identifies the product that must be tested before release.
- The testing team find the bugs then sends the software back for fixing. To deliver the fixes developers complete some coding and send the software again for testing. This process is repeated till the bugs are found in it, at an acceptable level.

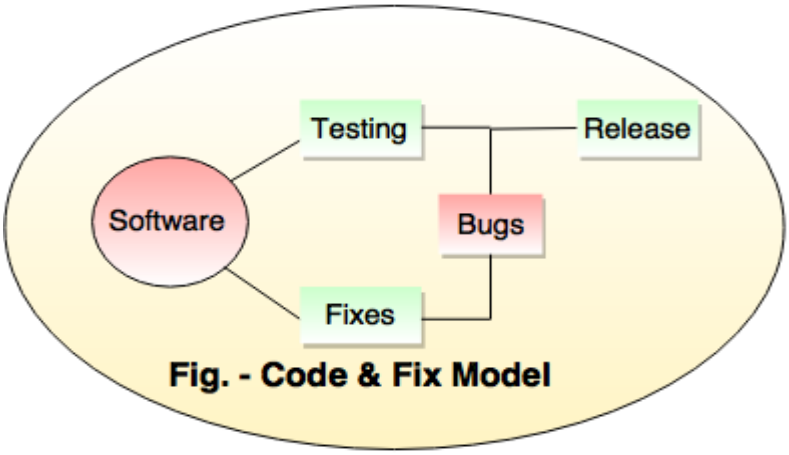


Fig. - Code & Fix Model

Advantages of Code-and-fix model

- This model is suitable for small projects.
- It needs less project planning.

Disadvantages of Code-and-fix model

- It is difficult to accommodate changes.
- It is not clear what will be delivered and when.
- It is difficult to assess quality.

3) Waterfall Model

- The waterfall model is the classic model or oldest model and is known as mother of all the model. It is widely used in government projects and many vital projects in company.
- The waterfall model is also called as **'Linear sequential model'** or **'Classic life cycle model'.**
- In this model, each phase is executed completely before the beginning of the next phase. Hence the phases do not overlap in waterfall model.
- This model is used for small projects.
- In this model, feedback is taken after each phase to ensure that the project is on the right path.
- Testing part starts only after the development is completed.

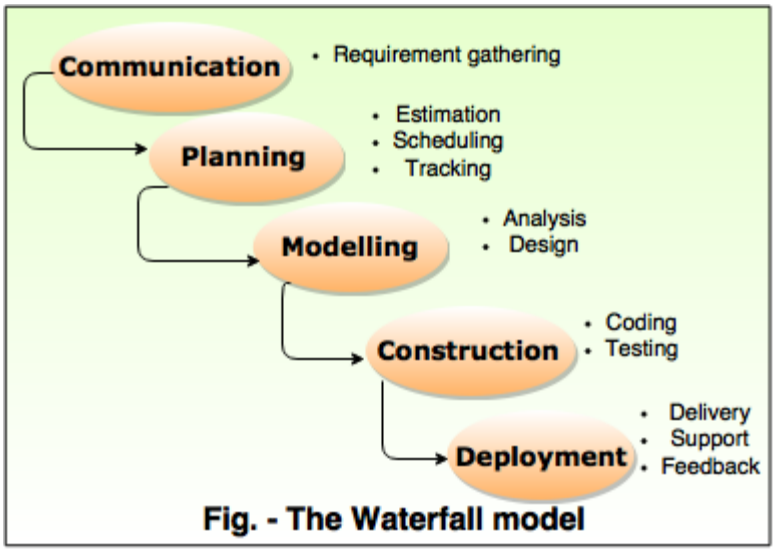


Fig. - The Waterfall model

Following are the phases in waterfall model:

i) Communication

The software development starts with the communication between customer and developer.

ii) Planning

It consists of complete estimation, scheduling for project development.

iii) Modelling

- Modelling consists of complete requirement analysis and the design of the project i.e algorithm, flowchart etc.
- The algorithm is the step-by-step solution of the problem and the flow chart shows a complete flow diagram of a program.

iv) Construction

- Construction consists of code generation and the testing part.
- Coding part implements the design details using an appropriate programming language.
- Testing is to check whether the flow of coding is correct or not.
- Testing also checks that the program provides desired output.

v) Deployment

- Deployment step consists of delivering the product to the customer and taking feedback from them.
- If the customer wants some corrections or demands for the additional capabilities, then the change is required for improvement in the quality of the software.

Advantages of Waterfall model

- The waterfall model is simple and easy to understand, to implement, and use.
- All the requirements are known at the beginning of the project, hence it is easy to manage.
- It avoids overlapping of phases because each phase is completed at once.
- This model works for small projects where the requirements are easily understood.
- This model is preferred for those projects where the quality is more important as compared to the cost of the project.

Disadvantages of the Waterfall model

- This model is not good for complex and object oriented projects.
- In this model, the changes are not permitted so it is not fit for moderate to high risk changes in project.
- It is a poor model for long duration projects.
- The problems with this model are uncovered, until the software testing.
- The amount of risk is high.

4) V Model

- V model is known as Verification and Validation model.
- This model is an extension of the waterfall model.
- In the life cycle of V-shaped model, processes are executed sequentially.
- Every phase completes its execution before the execution of next phase begins.

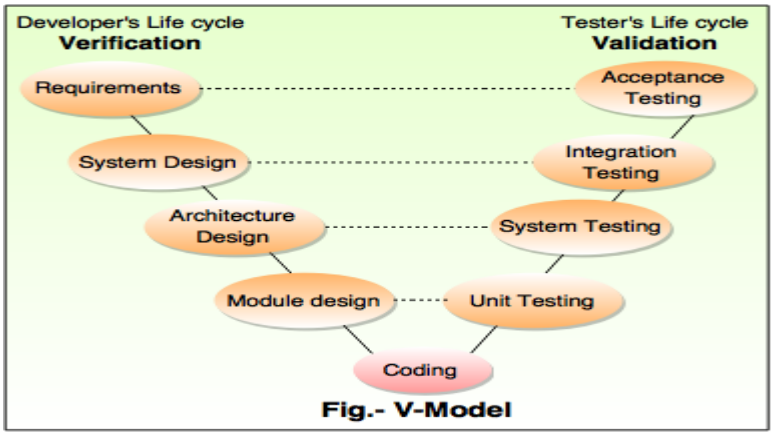


Fig.- V-Model

Following are the phases of V-model:

i) Requirements

- The requirements of product are understood from the customers point of view to know their exact requirement and expectation.
- The acceptance test design planning is completed at requirement stage because, business requirements are used as an input for acceptance testing.

ii) System Design

- In system design, high level design of the software is constructed.
- In this phase, we study how the requirements are implemented their technical use.

iii) Architecture design

- In architecture design, software architecture is created on the basis of high level design.
- The module relationship and dependencies of module, architectural diagrams, database tables, technology details are completed in this phase.

iv) Module design

- In module phase, we separately design every module or the software components.
- Finalize all the methods, classes, interfaces, data types etc.
- Unit tests are designed in module design phase based on the internal module designs.
- Unit tests are the vital part of any development process. They help to remove the maximum faults and errors at an early stage.

v) Coding Phase

- The actual code design of module designed in the design phase is grabbed in the coding phase.
- On the basis of system and architecture requirements, we decide the best suitable programming language.
- The coding is executed on the basis of coding guidelines and standards.

Advantages of V-model

- V-model is easy and simple to use.
- Many testing activities i.e planning, test design are executed in the starting, it saves more time.
- Calculation of errors is done at the starting of the project hence, less chances of error occurred at final phase of testing.
- This model is suitable for small projects where the requirements are easily understood.

Disadvantages of V-model

- V-model is not suitable for large and composite projects.
- If the requirements are not constant then this model is not acceptable.

5) Incremental Model

- The incremental model combines the elements of waterfall model and they are applied in an iterative fashion.
- The first increment in this model is generally a core product.
- Each increment builds the product and submits it to the customer for suggesting any modifications.
- The next increment implements the customer's suggestions and add additional requirements in the previous increment.
- This process is repeated until the product is completed. For example, the word-processing software is developed using the incremental model.

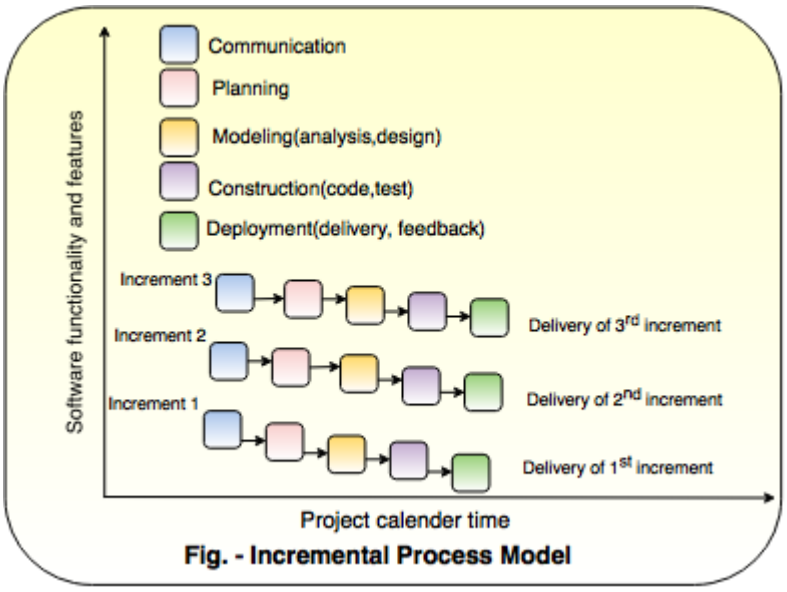


Fig. - Incremental Process Model

Following are the phases of Incremental model:

i)Communication

The software development starts with the communication between customer and developer.

ii)Planning

It consists of complete estimation, scheduling for project development.

iii) Modeling

- Modeling consists of complete requirement analysis and the design of the project like algorithm, flowchart etc.
- The algorithm is a step-by-step solution of the problem and the flow chart shows a complete flow diagram of a program.

iv) Construction

- Construction consists of code generation and the testing part.
- Coding part implements the design details using an appropriate programming language.
- Testing is to check whether the flow of coding is correct or not.
- Testing also checks that the program provides desired output.

v) Deployment

- Deployment step consists of delivering the product to the customer and taking feedback from them.
- If the customer wants some corrections or demands for the additional capabilities, then the change is required for improvement in the quality of the software.

Advantages of Incremental model

- This model is flexible because the cost of development is low and initial product delivery is faster.
- It is easier to test and debug in the smaller iteration.
- The working software is generated quickly in the software life cycle.
- The customers can respond to its functionalities after every increment.

Disadvantages of the incremental model

- The cost of the final product may cross the cost initially estimated.
- This model requires a very clear and complete planning.
- The planning of design is required before the whole system is broken into smaller increments.
- The demands of customer for the additional functionalities after every increment causes problem in the system architecture.

6) RAD Model

- RAD is a Rapid Application Development model.
- Using the RAD model, software product is developed in a short period of time.
- The initial activity starts with the communication between customer and developer.
- Planning depends upon the initial requirements and then the requirements are divided into groups.
- Planning is more important to work together on different modules.

The RAD model consist of following phases:

1) Business Modelling
2) Data modelling
3) Process modelling
4) Application generation
5) Testing and turnover

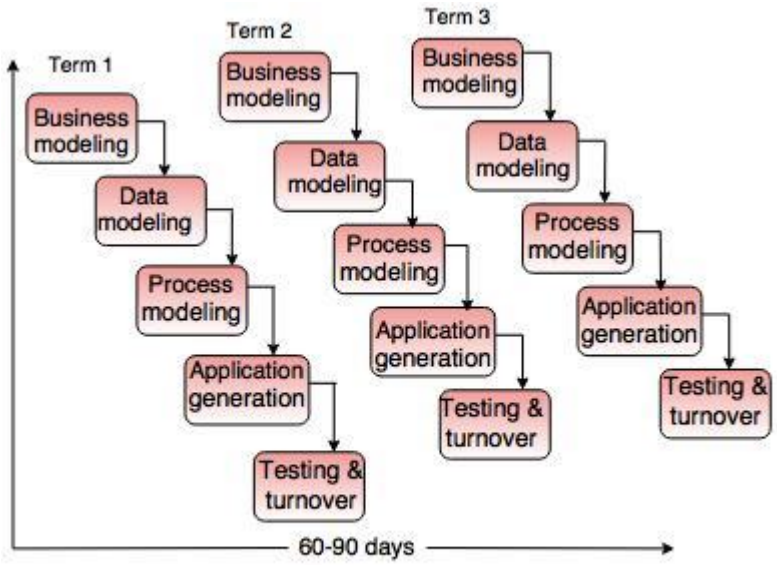


Fig. - RAD Model

1) Business Modelling

- Business modelling consists of the flow of information between various functions in the project.
  For example, what type of information is produced by every function and which are the functions to handle that information.
- It is necessary to perform complete business analysis to get the essential business information.

2) Data modelling

- The information in the business modelling phase is refined into the set of objects and it is essential for the business.
- The attributes of each object are identified and defined the relationship between objects.

3) Process modelling

- The data objects defined in the data modelling phase are changed to fulfil the information flow to implement the business model.
- The process description is created for adding, modifying, deleting or retrieving a data object.

4) Application generation

- In the application generation phase, the actual system is built.
- To construct the software the automated tools are used.

5) Testing and turnover

- The prototypes are independently tested after each iteration so that the overall testing time is reduced.
- The data flow and the interfaces between all the components are fully tested. Hence, most of the programming components are already tested.

Advantages of RAD Model

- The process of application development and delivery are fast.
- This model is flexible, if any changes are required.
- Reviews are taken from the clients at the staring of the development hence there are lesser chances to miss the requirements.

Disadvantages of RAD Model

- The feedback from the user is required at every development phase.
- This model is not a good choice for long term and large projects.

7) Agile Model

- Agile model is a combination of incremental and iterative process models.
- This model focuses on the users satisfaction which can be achieved with quick delivery of the working software product.
- Agile model breaks the product into individual iterations.
- Every iteration includes cross functional teams working on different areas such as planning, requirements, analysis, design, coding, unit testing and acceptance testing.
- At the end of an iteration working product shows to the users.

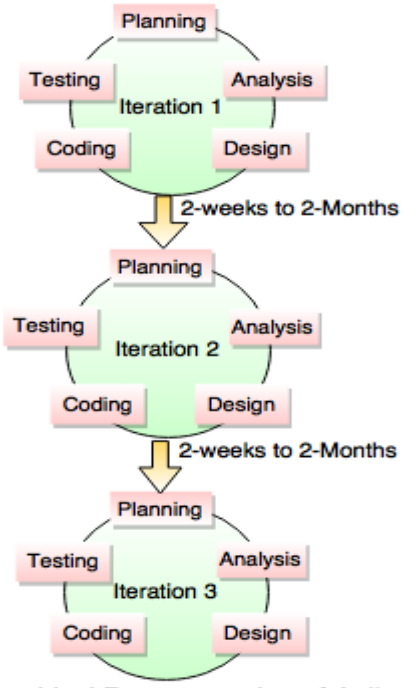


Fig. - Graphical Representation of Agile Model

- With every increment, features are incremented and the final increment hold all the features needed by the customers.
- The iterations in agile process are shorter in duration which can vary from 2 weeks to 2 months.

Advantages of Agile model

- Customers are satisfied because of quick and continuous delivery of useful software.
- Regular delivery of working software.
- Face to face interaction between the customers, developers and testers and it is best form of communication.

  Even the late changes in the requirement can be incorporated in the software.

Disadvantages of Agile model

- It is totally depends on customer interaction. If the customer is not clear with their requirements, the development team can go in the wrong direction.
- Documentation is less, so the transfer of technology to the new team members is challenging.

8) Iterative Model

- In Iterative model, the large application of software development is divided into smaller chunks and smaller parts of software which can be reviewed to recognize further requirements are implemented. This process is repeated to generate a new version of the software in each cycle of a model.
- With every iteration, development module goes through the phases i.e requirement, design, implementation and testing. These phases are repeated in iterative model in a sequence.

1) Requirement Phase

In this phase, the requirements for the software are assembled and analyzed. Generates a complete and final specification of requirements.

2) Design Phase

In this phase, a software solution meets the designed requirements which can be a new design or an extension of an earlier design.

3) Implementation and test phase

In this phase, coding for the software and test the code.

4) Evaluation

In this phase, software is evaluated, the current requirements are reviewed and the changes and additions in the requirements are suggested.

Advantages of an Iterative Model

- Produces working software rapidly and early in the software life cycle.
- This model is easy to test and debug in a smaller iteration.
- It is less costly to change scope and requirements.

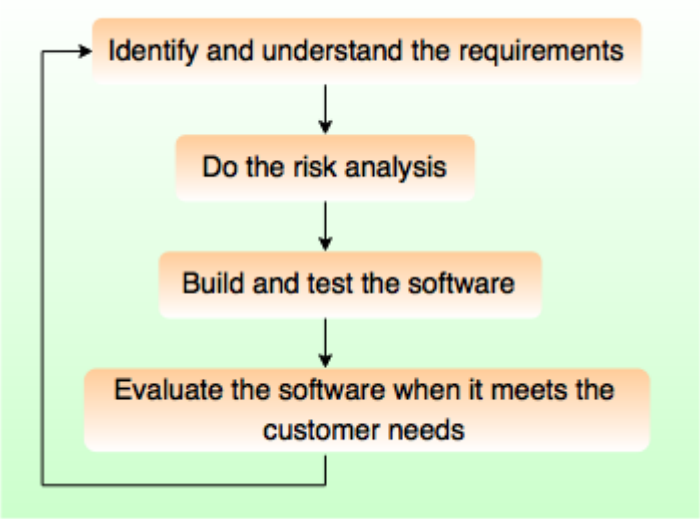
Disadvantages of an Iterative Model

- The system architecture is costly.
- This model is not suitable for smaller projects.

9) Spiral model

- It is a combination of prototype and sequential or waterfall model.
- This model was developed by Boehm.
- It is used for generating the software projects. This model is a risk driven process model.
- Every phase in the Spiral model is start with a design goal and ends with the client review.
- The development team in this model begins with a small set of requirements and for the set of requirements team goes through each development phase.
- The development team adds the functionality in every spiral till the application is ready.

Following are the steps involved in spiral model:



Phases of Spiral model are:

1) Planning
2) Risk Analysis
3) Engineering
4) Evaluation

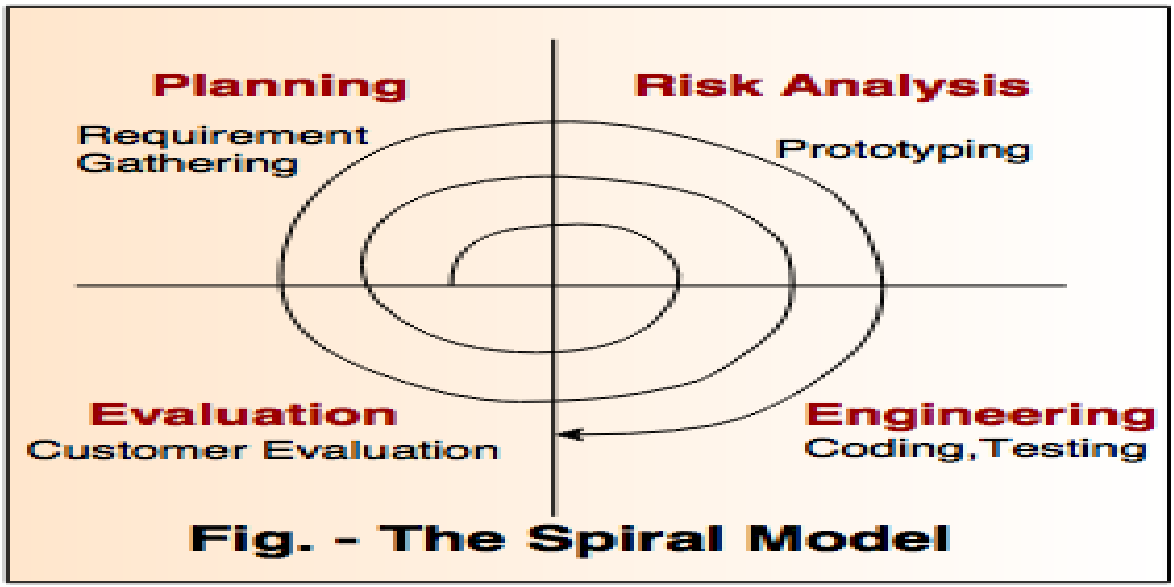


Fig. – The Spiral Model

1) Planning

- This phase, studies and collects the requirements for continuous communication between the customer and system analyst.
- It involves estimating the cost and resources for the iteration.

2) Risk Analysis

This phase, identifies the risk and provides the alternate solutions if the risk is found.

3) Engineering

In this phase, actual development i.e coding of the software is completed. Testing is completed at the end of the phase.

4) Evaluation

Get the software evaluated by the customers. They provide the feedback before the project continues to the next spiral.

Advantages of Spiral Model

- It reduces high amount of risk.
- It is good for large and critical projects.
- It gives strong approval and documentation control.
- In spiral model, the software is produced early in the life cycle process.

Disadvantages of Spiral Model

- It can be costly to develop a software model.
- It is not used for small projects.

10) Prototype Model:

- Prototype is defined as first or preliminary form using which other forms are copied or derived.
- Prototype model is a set of general objectives for software.
- It does not identify the requirements like detailed input, output.
- It is software working model of limited functionality.
- In this model, working programs are quickly produced.

The different phases of Prototyping model are:

1) Communication
2) Quick design
3) Modelling and quick design
4) Construction of prototype
5) Deployment, delivery, feedback

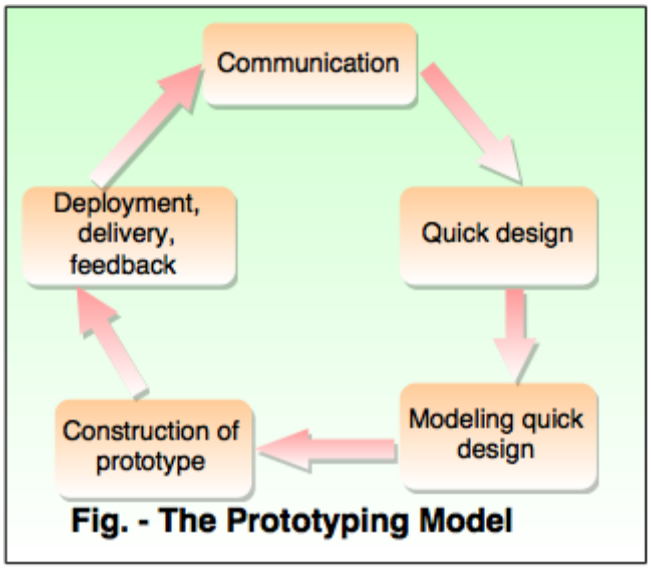


Fig. - The Prototyping Model

1. Communication
In this phase, developer and customer meet and discuss the overall objectives of the software.

2. Quick design

- Quick design is implemented when requirements are known.
- It includes only the important aspects i.e input and output format of the software.
- It focuses on those aspects which are visible to the user rather than the detailed plan.
- It helps to construct a prototype.

3. Modelling quick design

- This phase gives the clear idea about the development of software as the software is now constructed.
- It allows the developer to better understand the exact requirements.

4. Construction of prototype
The prototype is evaluated by the customer itself.

5. Deployment, delivery, feedback

- If the user is not satisfied with current prototype then it is refined according to the requirements of the user.
- The process of refining the prototype is repeated till all the requirements of users are met.
- When the users are satisfied with the developed prototype then the system is developed on the basis of final prototype.

Advantages of Prototyping Model

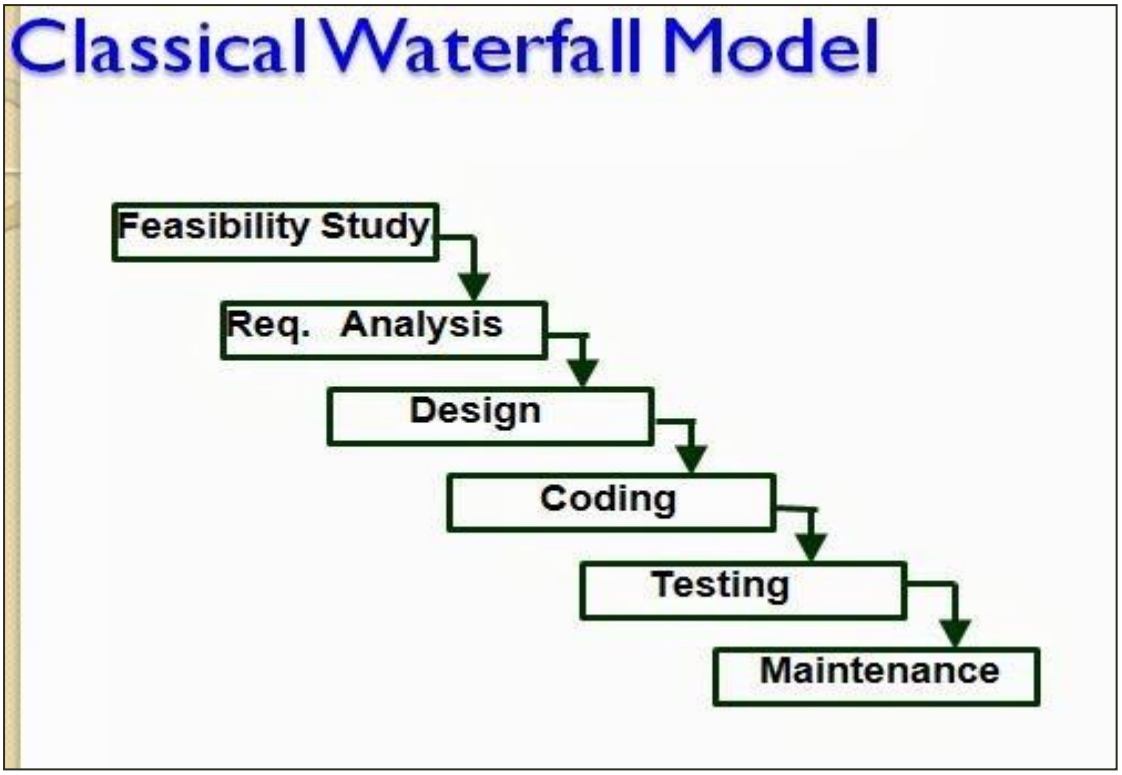
- In the development process of this model users are actively involved.
- The development process is the best platform to understand the system by the user.
- Earlier error detection takes place in this model.
- It gives quick user feedback for better solutions.
- It identifies the missing functionality easily. It also identifies the confusing or difficult functions.

Disadvantages of Prototyping Model

- The client involvement is more and it is not always considered by the developer.
- It is a slow process because it takes more time for development.
- Many changes can disturb the rhythm of the development team.
- It is a throw away prototype when the users are confused with it.

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 cannot 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 derived from 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 as shown in bellow figure.



Classical Waterfall Model

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

- At first project managers or team leaders try to have a rough understanding 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 would be 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 the area of development.

Requirements analysis and specification: - The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to

document them properly. This phase consists of two distinct activities, namely

- Requirements gathering and analysis
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

Design: - The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

- **Traditional design approach** -Traditional design consists of two different activities; first a structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.
- **Object-oriented design approach** -In this technique, various objects that occur in the problem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and unit testing:-The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

Integration and system testing: -Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document.

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

- α – testing: It is the system testing performed by the development team.
- β –testing: It is the system testing performed by a friendly set of customers.
- Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed,

specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

Maintenance: -Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratios. Maintenance involves performing any one or more of the following three kinds of activities:

- Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.

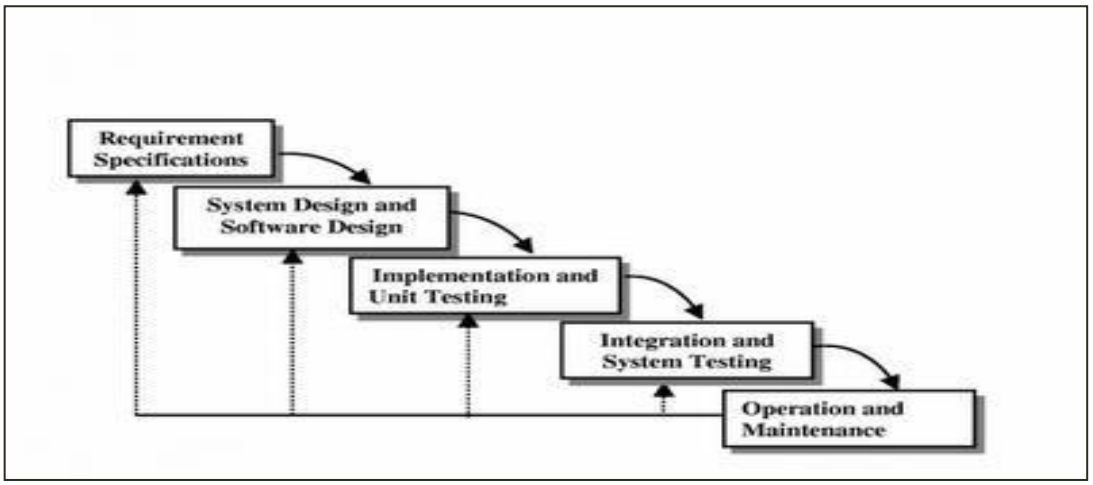
- Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.
- Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

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 the life 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.

ITERATIVE WATERFALL MODEL

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.



Iterative Waterfall Model

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

PRTOTYPING MODEL

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

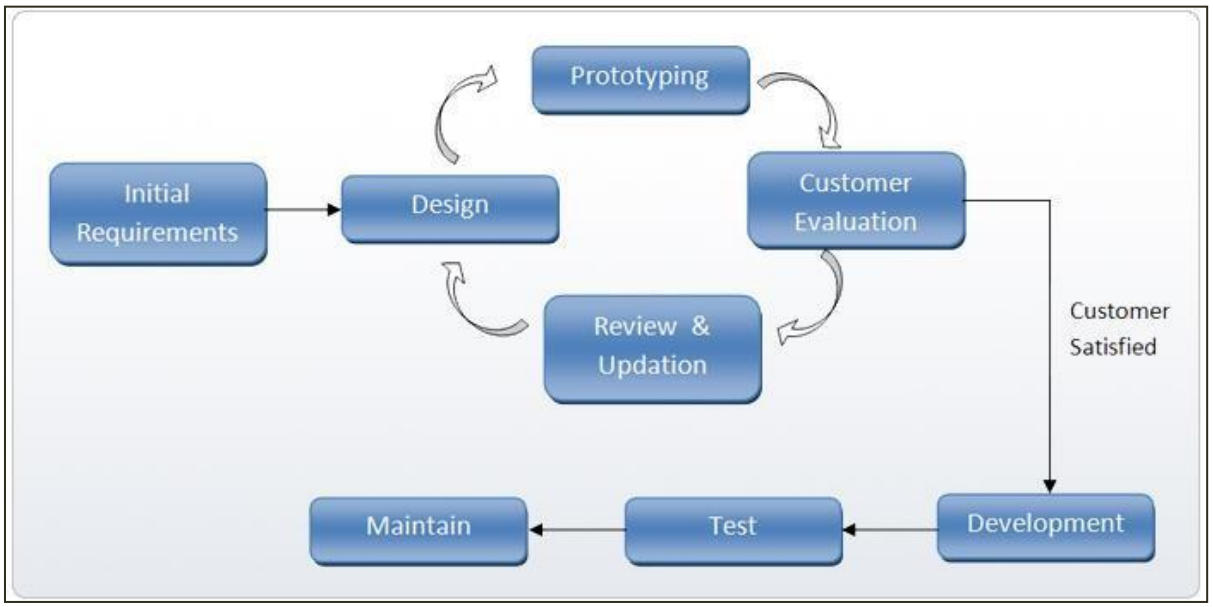
Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate

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.

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

- User requirements are not complete
- Technical issues are not clear



EVOLUTIONARY MODEL

It is also called successive versions model or incremental model. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

- Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.
- Also used in object oriented software development because the system can

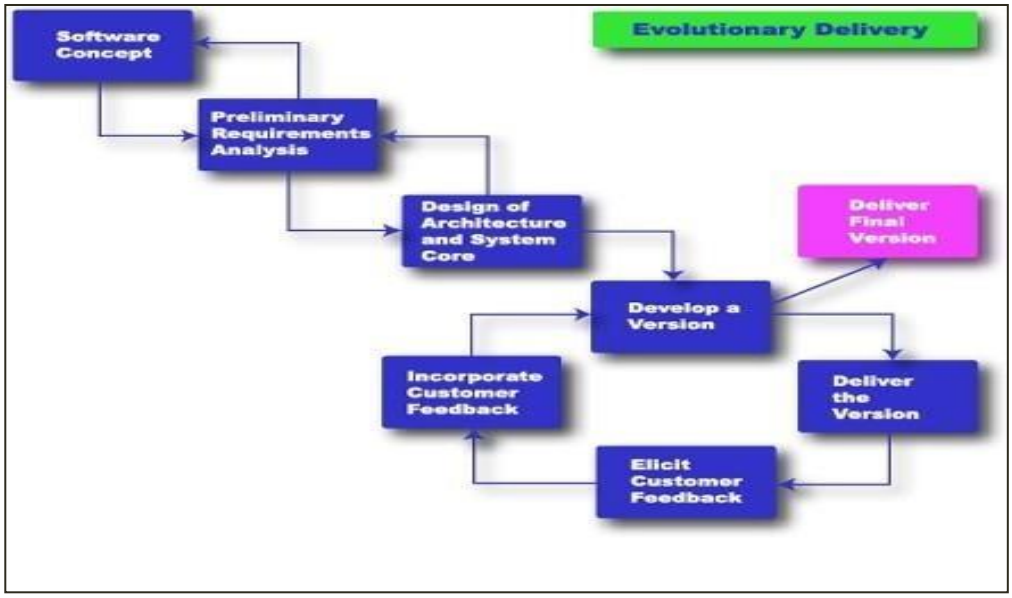
be easily portioned into units in terms of objects.

Advantages:

- User gets a chance to experiment partially developed system
- Reduce the error because the core modules get tested thoroughly.

Disadvantages:

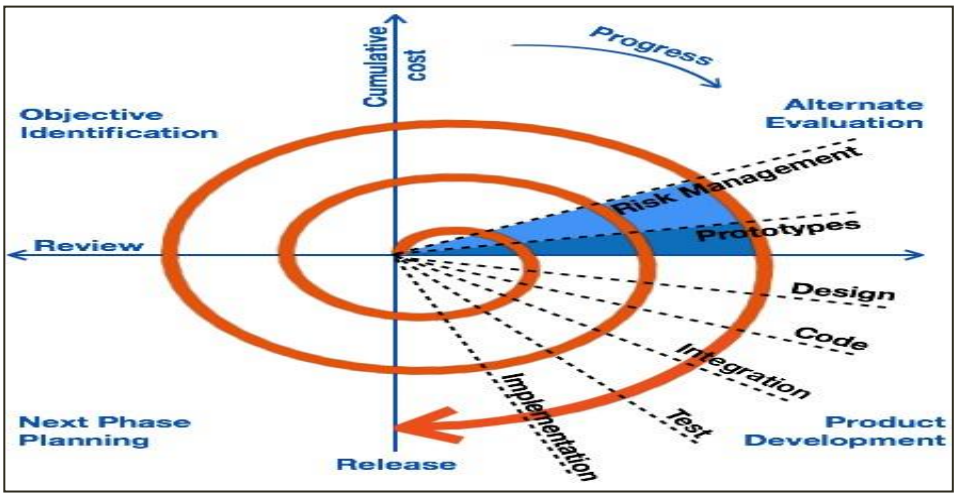
- It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.



Evolutionary Model

SPIRAL MODEL

The Spiral model of software development is shown in bellow figure. 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 bellow figure. The following activities are carried out during each phase of a spiral model.



Spiral Model

First quadrant (Objective Setting)

- During the first quadrant, it is needed to identify the objectives of the 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 be developed.

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 cannot 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 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.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is 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.

RAD MODEL:

Rapid application development(RAD) is a software development methodology that uses minimal planning in favor of rapid prototyping. A prototype is a working model that is functionally equivalent to a component of the product.

In the RAD model, the functional modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery. Since there is no detailed preplanning, it makes it easier to incorporate the changes within the development process.

RAD projects follow iterative and incremental model and have small teams comprising of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype.

The most important aspect for this model to be successful is to make sure that the prototypes developed are reusable.

RAD Model Design

RAD model distributes the analysis, design, build and test phases into a series of short, iterative development cycles.

Following are the various phases of the RAD Model:

1. **Business Modeling**

The business model for the product under development is designed in terms of flow of information and the distribution of information between various business channels. A complete business analysis is performed to find the vital information for business, how it can be obtained, how and when is the information processed and what are the factors driving successful flow of information.

2. **Data Modeling**

The information gathered in the Business Modelling phase is reviewed and analyzed to form sets of data objects vital for the business. The attributes of all data sets is identified and defined. The relation between these data objects are established and defined in detail in relevance to the business model.

3. **Process Modeling**

The data object sets defined in the Data Modeling phase are converted to establish the business information flow needed to achieve specific business objectives as per the business model. The process model for any changes or enhancements to the data object sets is defined in this phase. Process descriptions for adding, deleting, retrieving or modifying a data object are given.

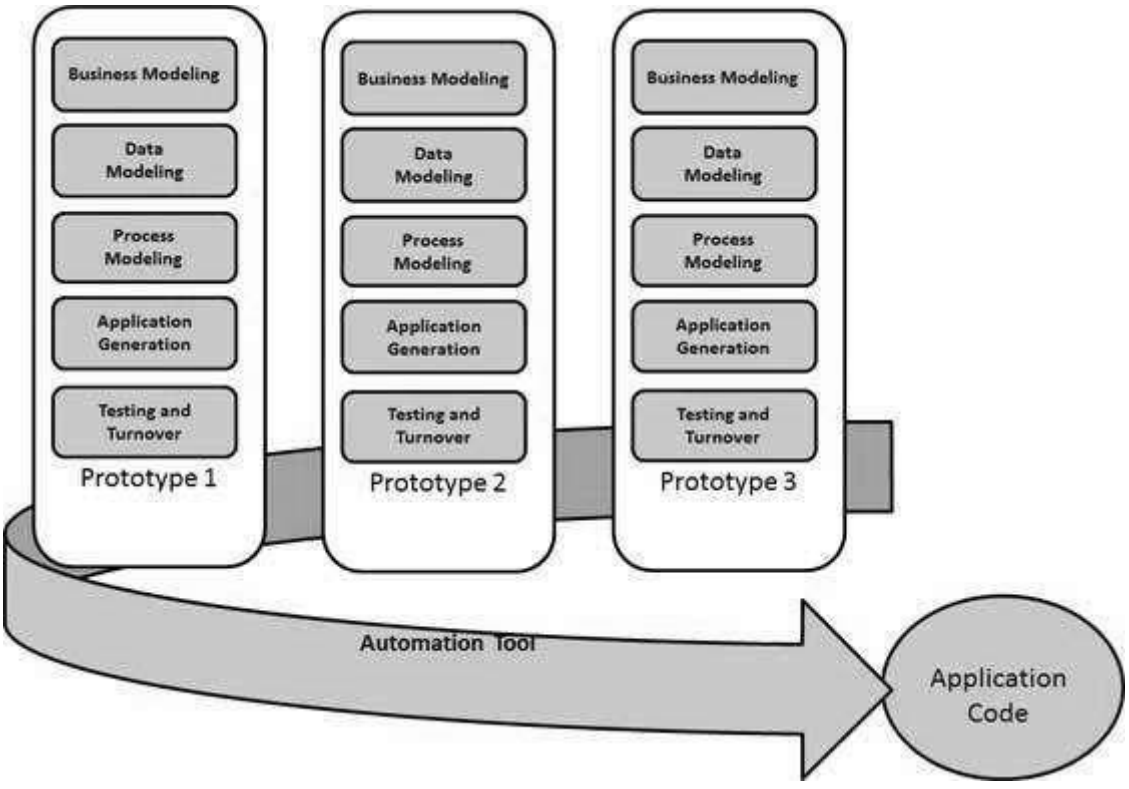
4. Application Generation

The actual system is built and coding is done by using automation tools to convert process and data models into actual prototypes.

5. Testing and Turnover

The overall testing time is reduced in the RAD model as the prototypes are independently tested during every iteration. However, the data flow and the interfaces between all the components need to be thoroughly tested with complete test coverage. Since most of the programming components have already been tested, it reduces the risk of any major issues.

The following illustration describes the RAD Model in detail.



RAD Model Vs Traditional SDLC

The traditional SDLC follows a rigid process models with high emphasis on requirement analysis and gathering before the coding starts. It puts pressure on the customer to sign off the requirements before the project starts and the customer doesn't get the feel of the product as there is no working build available for a long time.

The customer may need some changes after he gets to see the software. However, the change process is quite rigid and it may not be feasible to incorporate major changes in the product in the traditional SDLC.

The RAD model focuses on iterative and incremental delivery of working models to the customer. This results in rapid delivery to the customer and customer involvement during the complete development cycle of product reducing the risk of non-conformance with the actual user requirements.

RAD Model - Application

RAD model can be applied successfully to the projects in which clear modularization is possible. If the project cannot be broken into modules, RAD may fail.

The following pointers describe the typical scenarios where RAD can be used:

- RAD should be used only when a system can be modularized to be delivered in an incremental manner.
- It should be used if there is a high availability of designers for modeling.
- It should be used only if the budget permits use of automated code generating tools.
- RAD SDLC model should be chosen only if domain experts are available with relevant business knowledge.
- Should be used where the requirements change during the project and working prototypes are to be presented to customer in small iterations of 2-3 months.

RAD Model - Pros and Cons

RAD model enables rapid delivery as it reduces the overall development time due to the reusability of the components and parallel development. RAD works well only if high skilled engineers are available and the customer is also committed to achieve the targeted prototype in the given time frame. If there is commitment lacking on either side the model may fail.

The advantages of the RAD Model are as follows:

- Changing requirements can be accommodated.
- Progress can be measured.
- Iteration time can be short with use of powerful RAD tools.

- Productivity with fewer people in a short time.
- Reduced development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.

The disadvantages of the RAD Model are as follows:

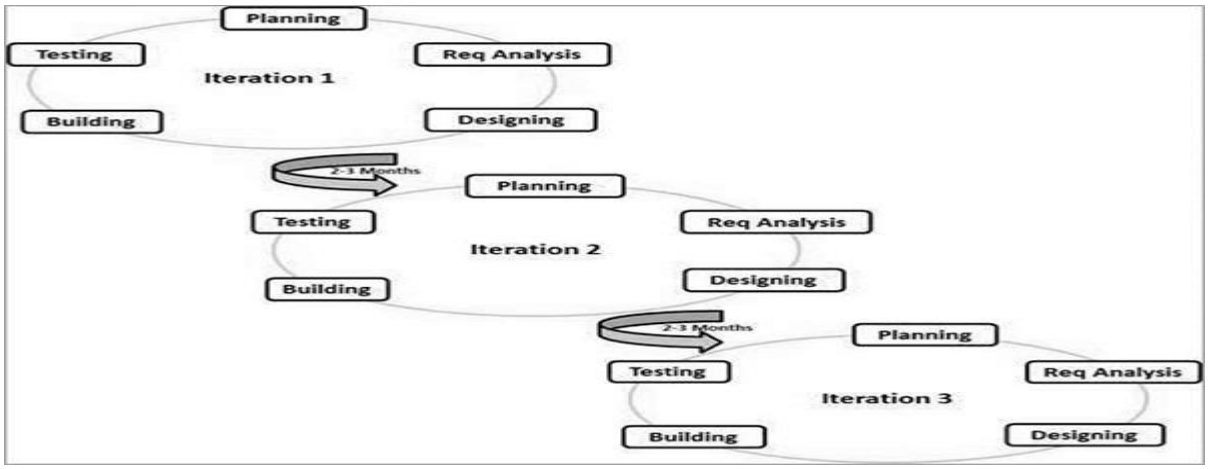
- Dependency on technically strong team members for identifying business requirements.
- Only system that can be modularized can be built using RAD.
- Requires highly skilled developers/designers.
- High dependency on modeling skills.
- Inapplicable to cheaper projects as cost of modeling and automated code generation is very high.
- Management complexity is more.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for project requiring shorter development times.

AGILE MODEL:

Agile model believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements. In Agile, the tasks are divided to time boxes (small time frames) to deliver specific features for a release.

Iterative approach is taken and working software build is delivered after each iteration. Each build is incremental in terms of features; the final build holds all the features required by the customer.

Here is a graphical illustration of the Agile Model −



The Agile thought process had started early in the software development and started becoming popular with time due to its flexibility and adaptability.

The most popular Agile methods include Rational Unified Process (1994), Scrum (1995), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995). These are now collectively referred to as **Agile Methodologies**, after the Agile Manifesto was published in 2001.

Following are the Agile Manifesto principles:

- **Individuals and interactions** − In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
- **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
- **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
- **Responding to change** − Agile Development is focused on quick responses to change and continuous development.

Agile Model - Pros and Cons

Agile methods are being widely accepted in the software world recently. However, this method may not always be suitable for all products. Here are some pros and cons of the Agile model.

The advantages of the Agile Model are as follows:

- Is a very realistic approach to software development.
- Promotes teamwork and cross training.
- Functionality can be developed rapidly and demonstrated.
- Resource requirements are minimum.
- Suitable for fixed or changing requirements
- Delivers early partial working solutions.
- Good model for environments that change steadily.
- Minimal rules, documentation easily employed.
- Enables concurrent development and delivery within an overall planned context.
- Little or no planning required.
- Easy to manage.
- Gives flexibility to developers.

The disadvantages of the Agile Model are as follows:

- Not suitable for handling complex dependencies.
- More risk of sustainability, maintainability and extensibility.

- An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is a very high individual dependency, since there is minimum documentation generated.
- Transfer of technology to new team members may be quite challenging due to lack of documentation.

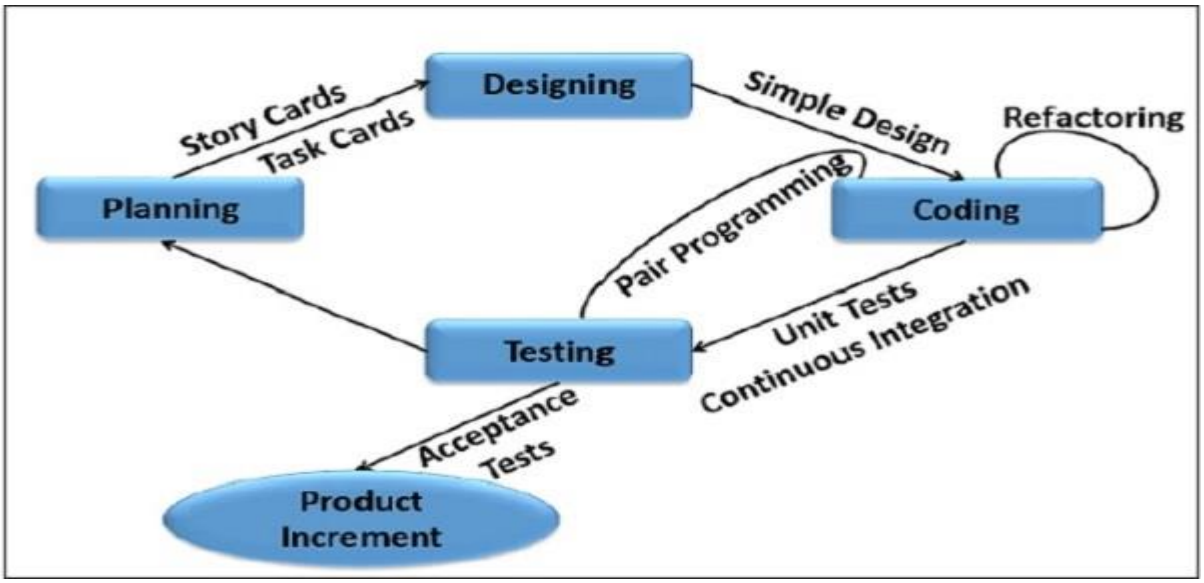
EXTREME PROGRAMMING:

XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop a software.

e**X**treme **P**rogramming (XP) was conceived and developed to address the specific needs of software development by small teams in the face of vague and changing requirements.

Extreme Programming is one of the Agile software development methodologies. It provides values and principles to guide the team behavior. The team is expected to self-organize. Extreme Programming provides specific core practices where:

- Each practice is simple and self-complete.
- Combination of practices produces more complex and emergent behavior.



Extreme Programming Advantages

Extreme Programming solves the following problems often faced in the software development projects −

- **Slipped schedules** − and achievable development cycles ensure timely deliveries.
- **Cancelled projects** − Focus on continuous customer involvement ensures transparency with the customer and immediate resolution of any issues.
- **Costs incurred in changes** − Extensive and ongoing testing makes sure the changes do not break the existing functionality. A running working system always ensures sufficient time for accommodating changes such that the current operations are not affected.
- **Production and post-delivery defects: Emphasis is on** − the unit tests to detect and fix the defects early.
- **Misunderstanding the business and/or domain** − Making the customer a part of the team ensures constant communication and clarifications.
- **Business changes** − Changes are considered to be inevitable and are accommodated at any point of time.
- **Staff turnover** − Intensive team collaboration ensures enthusiasm and good will. Cohesion of multi-disciplines fosters the team spirit.

MODULE-2

REQUIREMENT GATHERING AND ANALYSIS

1. Requirement Gathering:

The following are some of the well-known requirements gathering techniques −

Brainstorming

- Brainstorming is used in requirement gathering to get as many ideas as possible from group of people. Generally used to identify possible solutions to problems, and clarify details of opportunities.

Document Analysis

- Reviewing the documentation of an existing system can help when creating AS–IS process document, as well as driving gap analysis for scoping of migration projects. In an ideal world, we would even be reviewing the requirements that drove creation of the existing system – a starting point for documenting current requirements. Nuggets of information are often buried in existing documents that help us ask questions as part of validating requirement completeness.

Focus Group

- A focus group is a gathering of people who are representative of the users or customers of a product to get feedback. The feedback can be gathered about needs/opportunities/ problems to identify requirements, or can be gathered to validate and refine already elicited requirements. This form of market research is distinct from brainstorming in that it is a managed process with specific participants.

Interface analysis

- Interfaces for a software product can be human or machine. Integration with external systems and devices is just another interface. User centric design approaches are very effective at making sure that we create usable software. Interface analysis – reviewing the touch points with other external systems is important to make sure we don't overlook requirements that aren't immediately visible to users.

Interview

- Interviews of stakeholders and users are critical to creating the great software. Without understanding the goals and expectations of the users and stakeholders, we are very unlikely to satisfy them. We also have to recognize the perspective of each interviewee, so that, we can properly weigh and

address their inputs. Listening is the skill that helps a great analyst to get more value from an interview than an average analyst.

Observation

- By observing users, an analyst can identify a process flow, steps, pain points and opportunities for improvement. Observations can be passive or active (asking questions while observing). Passive observation is better for getting feedback on a prototype (to refine requirements), where active observation is more effective at getting an understanding of an existing business process. Either approach can be used.

Prototyping

- Prototyping is a relatively modern technique for gathering requirements. In this approach, you gather preliminary requirements that you use to build an initial version of the solution - a prototype. You show this to the client, who then gives you additional requirements. You change the application and cycle around with the client again. This repetitive process continues until the product meets the critical mass of business needs or for an agreed number of iterations.

Requirement Workshops

- Workshops can be very effective for gathering requirements. More structured than a brainstorming session, involved parties collaborate to document requirements. One way to capture the collaboration is with creation of domain-model artifacts (like static diagrams, activity diagrams). A workshop will be more effective with two analysts than with one.

Reverse Engineering

- When a migration project does not have access to sufficient documentation of the existing system, reverse engineering will identify what the system does. It will not identify what the system should do, and will not identify when the system does the wrong thing.

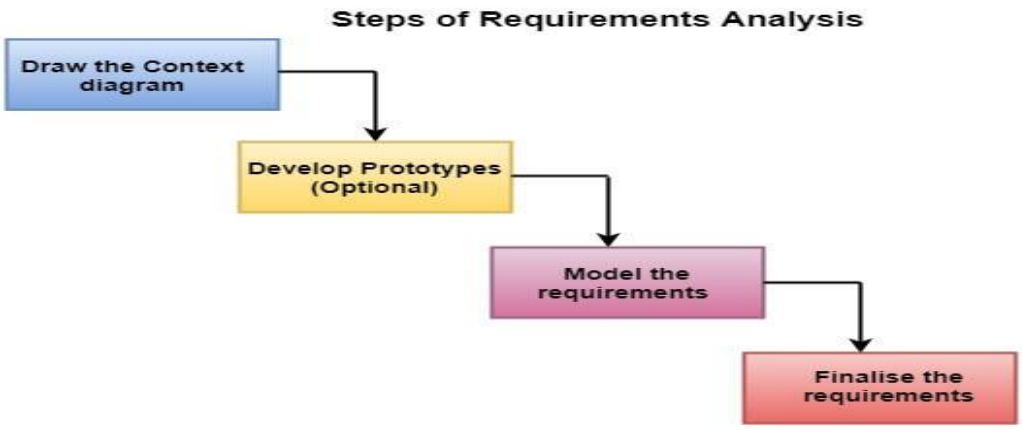
Survey/Questionnaire

- When collecting information from many people – too many to interview with budget and time constraints – a survey or questionnaire can be used. The survey can force users to select from choices, rate something ("Agree Strongly, agree…"), or have open ended questions allowing free-form responses. Survey design is hard – questions can bias the respondents.

2. Requirement Analysis:

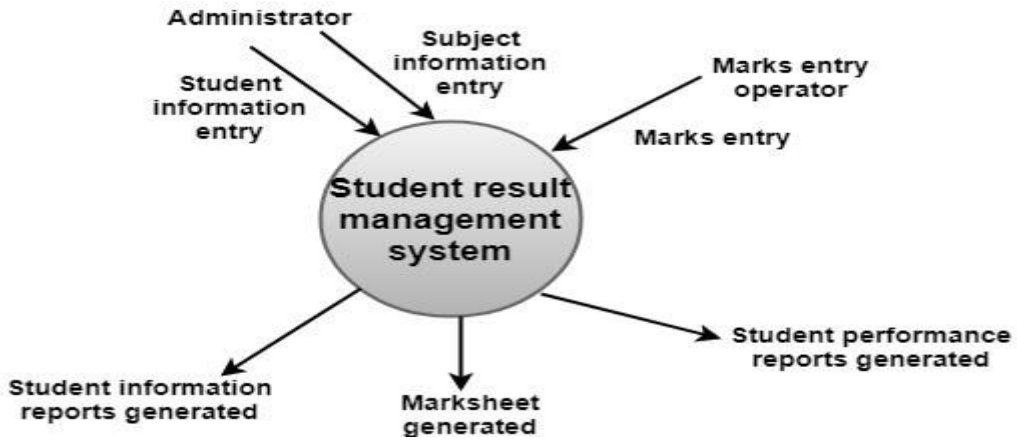
Requirement analysis is significant and essential activity after elicitation. We analyze, refine, and scrutinize the gathered requirements to make consistent and unambiguous requirements. This activity reviews all requirements and may provide a graphical view of the entire system. After the completion of the analysis, it is expected that the understandability of the project may improve significantly. Here, we may also use the interaction with the customer to clarify points of confusion and to understand which requirements are more important than others.

The various steps of requirement analysis are shown in fig:



(i)Draw the context diagram:

The context diagram is a simple model that defines the boundaries and interfaces of the proposed systems with the external world. It identifies the entities outside the proposed system that interact with the system. The context diagram of student result management system is given below:



(ii) Development of a Prototype (optional):

One effective way to find out what the customer wants is to construct a prototype, something that looks and preferably acts as part of the system they say they want.

We can use their feedback to modify the prototype until the customer is satisfied continuously. Hence, the prototype helps the client to visualize the proposed system and increase the understanding of the requirements. When developers and users are not sure about some of the elements, a prototype may help both the parties to take a final decision.

Some projects are developed for the general market. In such cases, the prototype should be shown to some representative sample of the population of potential purchasers. Even though a person who tries out a prototype may not buy the final system, but their feedback may allow us to make the product more attractive to others.

The prototype should be built quickly and at a relatively low cost. Hence it will always have limitations and would not be acceptable in the final system. This is an optional activity.

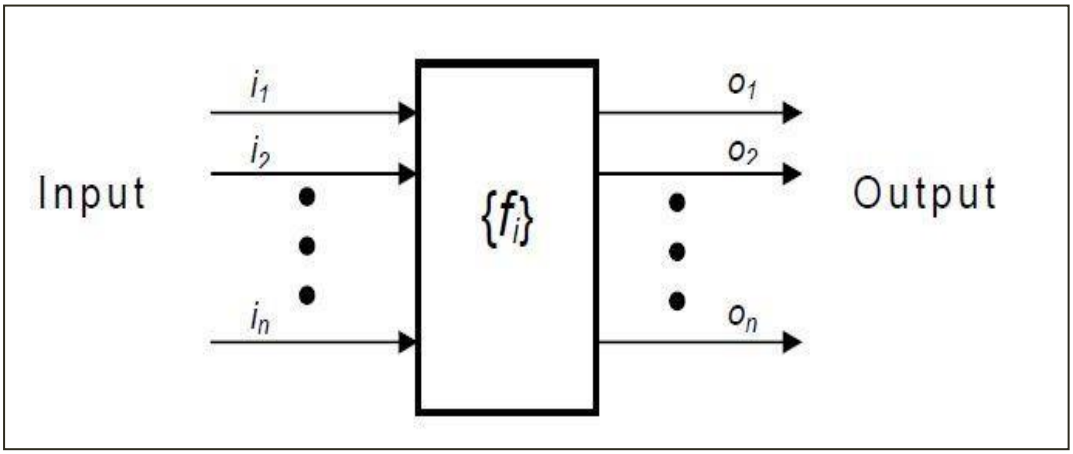
(iii) Model the requirements: This process usually consists of various graphical representations of the functions, data entities, external entities, and the relationships between them. The graphical view may help to find incorrect, inconsistent, missing, and superfluous requirements. Such models include the Data Flow diagram, Entity-Relationship diagram, Data Dictionaries, State-transition diagrams, etc.

(iv) Finalise the requirements: After modeling the requirements, we will have a better understanding of the system behavior. The inconsistencies and ambiguities have been identified and corrected. The flow of data amongst various modules has been analyzed. Elicitation and analyze activities have provided better insight into the system. Now we finalize the analyzed requirements, and the next step is to document these requirements in a prescribed format.

FUNCTIONAL AND NON-FUNCTIONAL REQUIREMENTS:

Functional requirements:-

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions $\{f\}$. The functional view of the system is shown in the bellow figure. Each function f of the system can be considered as a transformation of a set of input data (i_i) to the corresponding set of output data (o). The user can get some meaningful piece of work done using a high-level function.



View of a system performing a set of functions

Nonfunctional requirements:-

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

Goals of implementation:-

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

Identifying functional requirements from a problem description

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high- level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

Example: - Consider the case of the library System, where **–**

F1: Search Book function

Input: an author's name

Output: details of the author's books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

Documenting functional requirements

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

Example: - Withdraw Cash from ATM

R1: withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash; otherwise it generates an error message.

R1.1 select withdraw amount option

Input: "withdraw amount" option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user's account if sufficient balance is available, otherwise an error message displayed

SOFTWARE REQUIREMENT SPECIFICATION (SRS):

A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

Qualities of SRS:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

Types of Requirements:

The below diagram depicts the various types of requirements that are captured during SRS.

Parts of a SRS document:

The important parts of SRS document are:

i. Functional requirements of the system

ii. Non-functional requirements of the system, and

iii. Goals of implementation

Properties of a good SRS document:

The important properties of a good SRS document are the following:

- **Concise.** The SRS document should be concise and at the same time unambiguous, consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.

- **Structured.** It should be well-structured. A well-structured document is easy to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

- **Black-box view.** It should only specify what the system should do and refrain from stating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.
- **Conceptual integrity.** It should show conceptual integrity so that the reader can easily understand it.

- **Response to undesired events.** It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

- **Verifiable.** All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to

determine whether or not requirements have been met in an implementation.

Problems without a SRS document:

The important problems that an organization would face if it does not develop a SRS document are as follows:

- Without developing the SRS document, the system would not be implemented according to customer needs.
- Software developers would not know whether what they are developing is what exactly required by the customer.
- Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
- It will be very much difficult for user document writers to write the users' manuals properly without understanding the SRS document.

IEEE 830 GUIDELINES:

This is a standard format of writing the SRS document. Most developers use the standard and of course, there are small variations to the standard; but if we know the IEEE 830 standard should be able to have small variations of this. Now let us look at what are the sections of the IEEE 830 standard title that is name of the software table of contents and then, there are 3 main sections; one is the Introduction, Overall Description and the Specific Requirements and then, there are the Appendices and Index.

The introduction should have a purpose; subsection called as Purpose, Scope, Definition, Acronym, Abbreviations, References and Overview. The purpose should say that what the system will solve. So, general introduction and describe who are the users. Scope very overall idea about what the system will do; what it will not do, very crisply written and overall description of what is expected of the system and what it should not do.

Then, there are the various terminologies; the definitions, acronyms, abbreviations that will be used in the SRS document. But, often variation to the standard, these are put under the appendices. Similarly, a setup documents that are referred in the SRS documents. These references can also be put under a separate appendices and then, an overview of the document or organization of the document; how the SRS is organized and that is about the introduction. Now, let us look at the overall description. This is the section 2. The sub sections here are the Product Perspective, Product Functions, User Characteristics, Constraints, Assumptions and Dependencies. In the product perspective, we write here the business case that is how it will help the customer; how it will benefit how it will be used and so on and even met write about the external interfaces. Briefly the overview of what type of users, hardware, software etcetera will be used and then, any constraints in the memory are operational constraints site constraints and so on.

The product functions this not really the functional requirement specification; but a brief overview or a summer summarize a summary of the functional capabilities. For example, in a library, we might just write very briefly saying that the library software should serve the users for their issuing of books returning for the librarian to create members manage books and so on and for the accountant to manage the financial aspects of the library. The user characteristics are the technical skills of each user class. This is also required because finally, the user interface development we will need the user characteristics. If it is software is to be used by factory workers as well and programmers, we need to have different user interfaces. So, need to identify the user characteristics and how much what are the technical skills how will their convergent with the computers and so on. The constraints here, what are the platforms that will be used? The database that will be used; the network development standards and so on.

And then, if there are any assumptions here dependencies need to mention that and then let us look at the third section. This is possibly the most important section. Here we have the external interfaces functions; this is the major section here. This contains the functionalities the functional requirement. The non-functional requirement Logical Database Requirement, Design Constraints, Quality Attributes and any Object Oriented Models; in this section has to be written very carefully and in sufficient detail. So, that the designers can have all the information they need to carry out the development work and for the testers to be able to write the test cases. It is a very important section. While writing the functional requirement, we should make sure that we have as far as possible all the input or stimulus to the system; output that is produced by the system and the functionalities.

Now, first let us look at the external interface. Here, we have to write about all the user interface maybe you can give a GUI screenshots, the different file formats that may be needed. It should actually there are some part that were mentioned in the section 2; external interface. It should actually elaborate that and should not duplicate information there. This is the main section the functional requirements. This will contain the detailed specification of each functionality or use case including collaboration and other diagrams we will see these diagrams later.

Then, the Performance Requirements; the Logical Database Requirement; what are the different data that will be stored in the database the relationship? The Design Constraints are standard complaints. For example, we may say that I have to use UML 2.0 and have to use let us say (Refer Time: 16:08) for writing the code. The quality attributes and then any object oriented models like class diagram, state collaboration diagram, activity diagram etcetera can be include.

But then, the section 3 that we just saw now is just one example and IEE 830 specifies that there can be different organization of the section 3 based on the modes of the software that is software maybe having a expert mode, a novies mode and so on. If there are modes are there then, we might have to organize it such that the modes the software is written in terms of the modes. Sorry, the SRS document is written in terms of the modes. Similarly, depending on different user Classes, Concepts, Features, Stimuli; there can be slightly different organization of the section three.

DECISION TABLES:

A decision table is used to represent the complex processing logic in a tabular or a matrix form. The upper rows of the table specify the variables or conditions to be evaluated. The lower rows of the table specify the actions to be taken when the corresponding conditions are satisfied. A column in a table is called a rule. A rule implies that if a condition is true, then the corresponding action is to be executed.

Example: -

Consider the previously discussed LMS example. The following decision table (bellow fig) shows how to represent the LMS problem in a tabular form. Here the table is divided into two parts, the upper part shows the conditions and the lower part shows what actions are taken. Each column of the table is a rule.

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

Decision table for LMS

From the above table you can easily understand that, if the valid selection condition is false then the action taken for this condition is 'display error message'. Similarly, the actions taken for other conditions can be inferred from the table.

DECISION TREE

A decision tree gives a graphic view of the processing logic involved in decision making and the corresponding actions taken. The edges of a decision tree represent conditions and the leaf nodes represent the actions to be performed depending on the outcome of testing the condition.

Example: -

Consider Library Membership Automation Software (LMS) where it should support the following three options**:**

- **New member**
- **Renewal**
- **Cancel membership**

New member option-

Decision: When the 'new member' option is selected, the software asks details about the member like the member's name, address, phone number etc.

Action: If proper information is entered then a membership record for the member is created and a bill is printed for the annual membership charge plus the security deposit payable.

Renewal option-

Decision: If the 'renewal' option is chosen, the LMS asks for the member's name and his membership number to check whether he is a valid member or not.

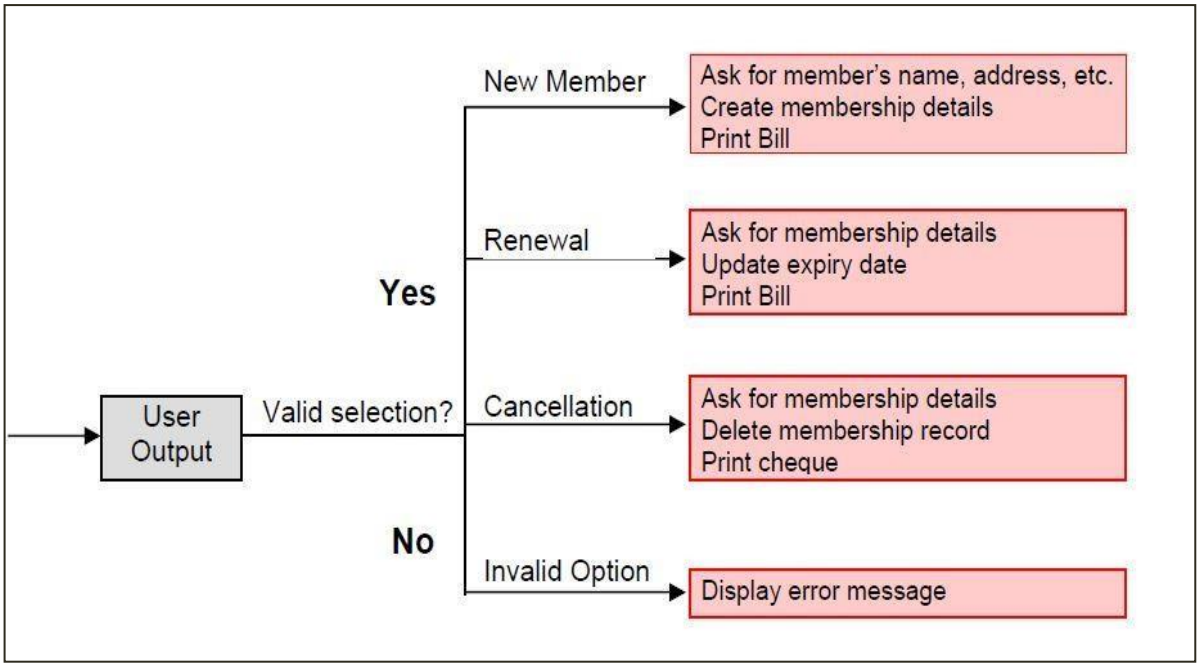
Action: If the membership is valid then membership expiry date is updated and the annual membership bill is printed, otherwise an error message is displayed.

Cancel membership option-

Decision: If the 'cancel membership' option is selected, then the software asks for member's name and his membership number.

Action: The membership is cancelled, a cheque for the balance amount due to the member is printed and finally the membership record is deleted from the database.

The following tree (fig) shows the graphical representation of the above example.



Decision Tree of LMS

SOFTWARE PROJECT MANAGEMENT

RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER:

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

PROJECT PLANNING:

Once a project is found to be feasible, software project managers undertake

project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

Estimating the following attributes of the project:

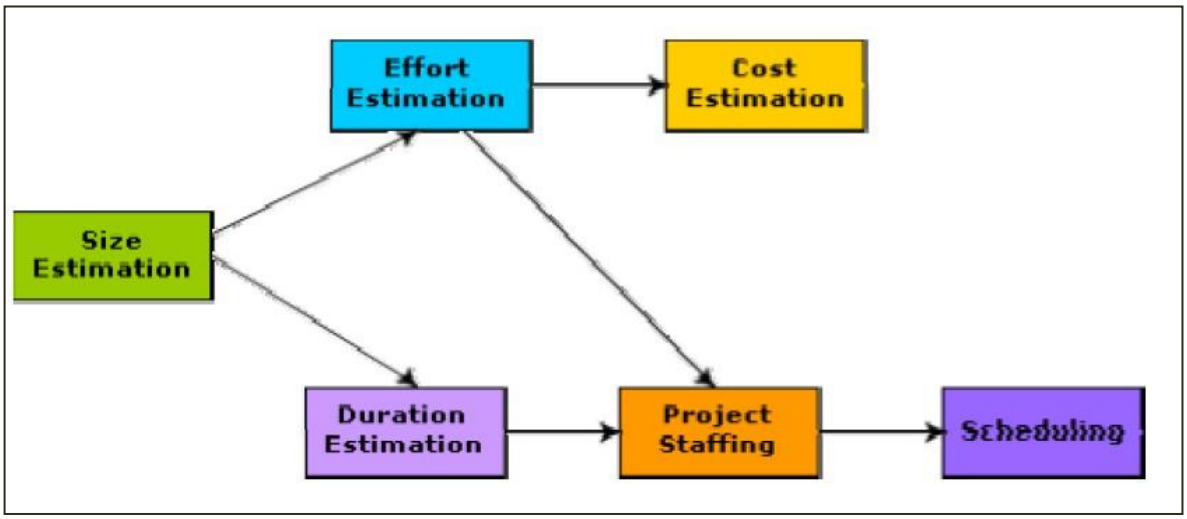
- **Project size**: What will be problem complexity in terms of the effort and time required to develop the product?
- **Cost**: How much is it going to cost to develop the project?
- **Duration**: How long is it going to take to complete development?
- **Effort**: How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

- Scheduling manpower and other resources.
- Staff organization and staffing plans.
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

Precedence ordering among project planning activities

Different project related estimates done by a project manager have already been discussed. Bellow fig shows the order in which important project planning activities may be undertaken. It can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.



Precedence ordering among planning activities

Sliding Window Planning

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However, project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases. After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

Software Project Management Plan (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below. This list can be used as a possible organization of the SPMP document.

Organization of the Software Project Management Plan (SPMP) Document

1. **Introduction**

   (a) Objectives
   (b) Major Functions
   (c) Performance Issues
   (d) Management and Technical Constraints

2. **Project Estimates**

   (a) Historical Data Used
   (b) Estimation Techniques Used
   (c) Effort, Resource, Cost, and Project Duration Estimates

3. **Schedule**

   (a) Work Breakdown Structure
   (b) Task Network Representation
   (c) Gantt Chart Representation
   (d) PERT Chart Representation

4. **Project Resources**

(a) People
(b) Hardware and Software
(c) Special Resources

5. **Staff Organization**

(a) Team Structure
(b) Management Reporting

6. **Risk Management Plan**

(a) Risk Analysis
(b) Risk Identification
(c) Risk Estimation
(d) Risk Abatement Procedures

7. **Project Tracking and Control Plan**

8. **Miscellaneous Plans**

(a) Process Tailoring
(b) Quality Assurance Plan
(c) Configuration Management Plan
(d) Validation and Verification
(e) System Testing Plan
(f) Delivery, Installation, and Maintenance Plan

METRICS FOR PROJECT SIZE ESTIMATION:

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

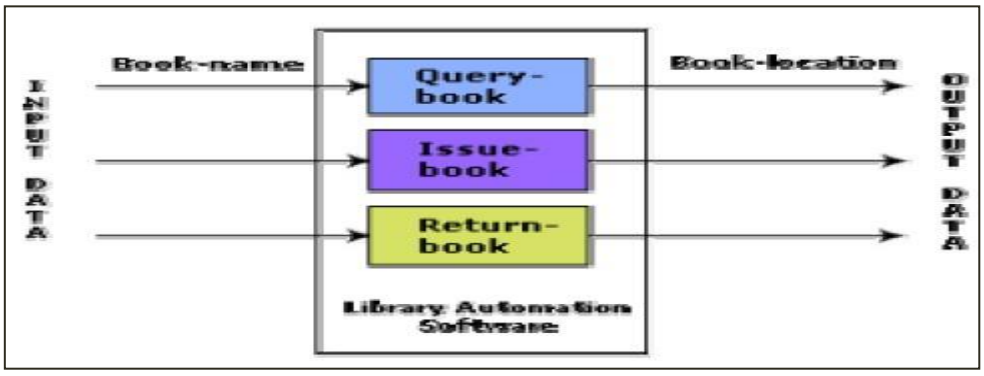
Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into submodules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed. The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the issue book feature (bellow figure) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.



System function as a map of input data to output data

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

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

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance.

Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input. For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

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

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, FP=UFP*TCF.

Shortcomings of function point (FP) metric

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.

- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.

- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.

- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.

- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

Feature Point Metric

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the

software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

Empirical Estimation Techniques

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: **Expert judgment technique** and **Delphi cost estimation.**

Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

Delphi Cost Estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the

summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

COCOMO MODELS:

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 embedded systems 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:

Effort = a1 x (KLOC)^{a2} PM

Tdev = b1 x (Effort)^{b2}b Months

Where

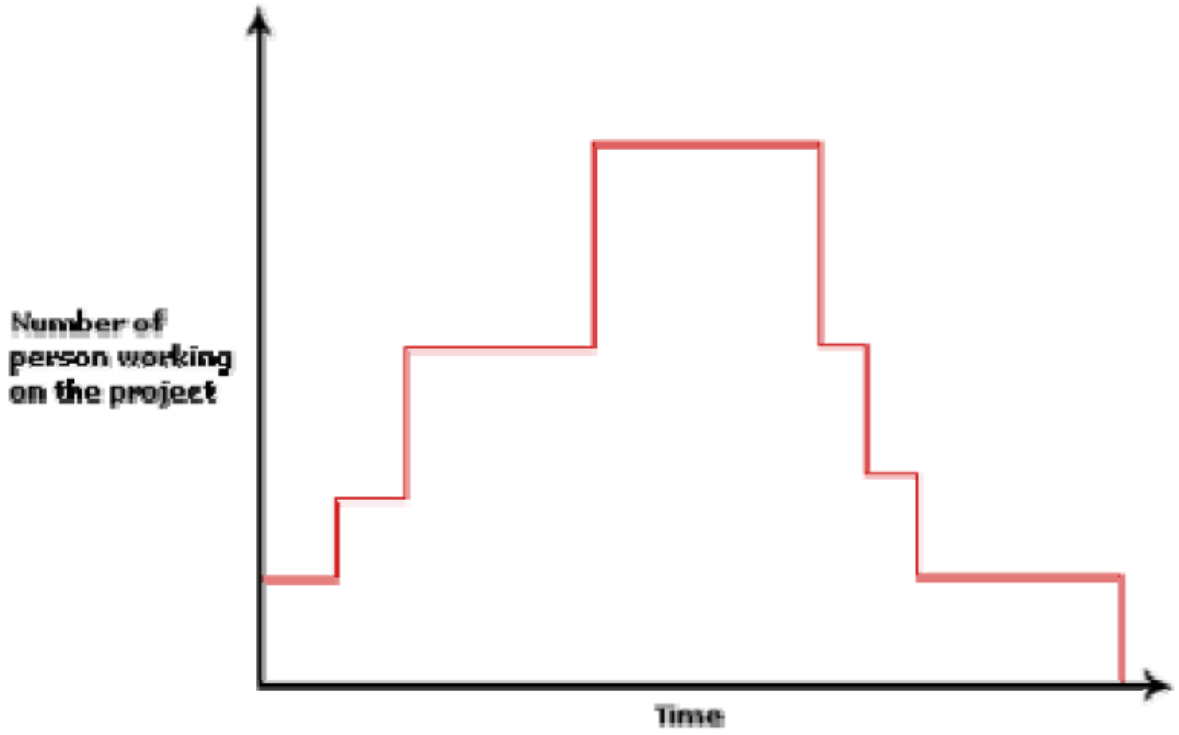
• KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

• a1, a2, b1, b2 are constants for each category of software products,

• Tdev is the estimated time to develop the software, expressed in months,

• Effort is the total effort required to develop the software product, expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot. 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.



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 a1, a2, b1, b2 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

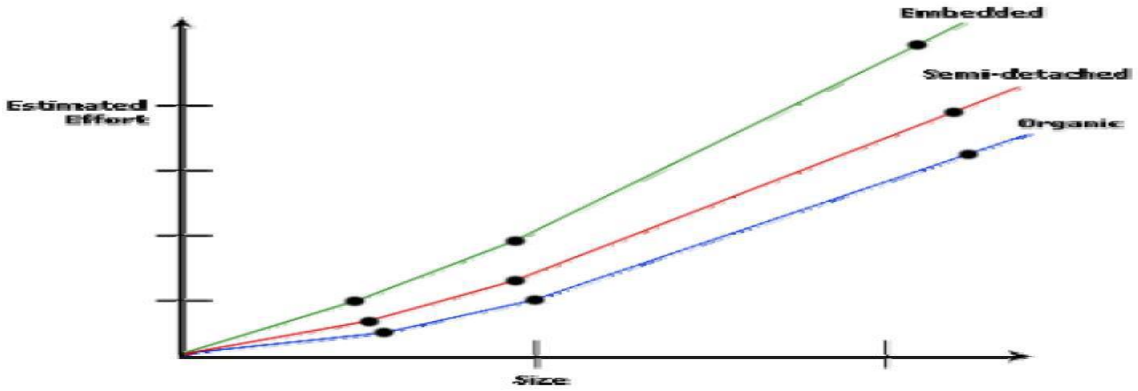
For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : **Tdev = 2.5(*Effort*)$^{0.38}$ Months**

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

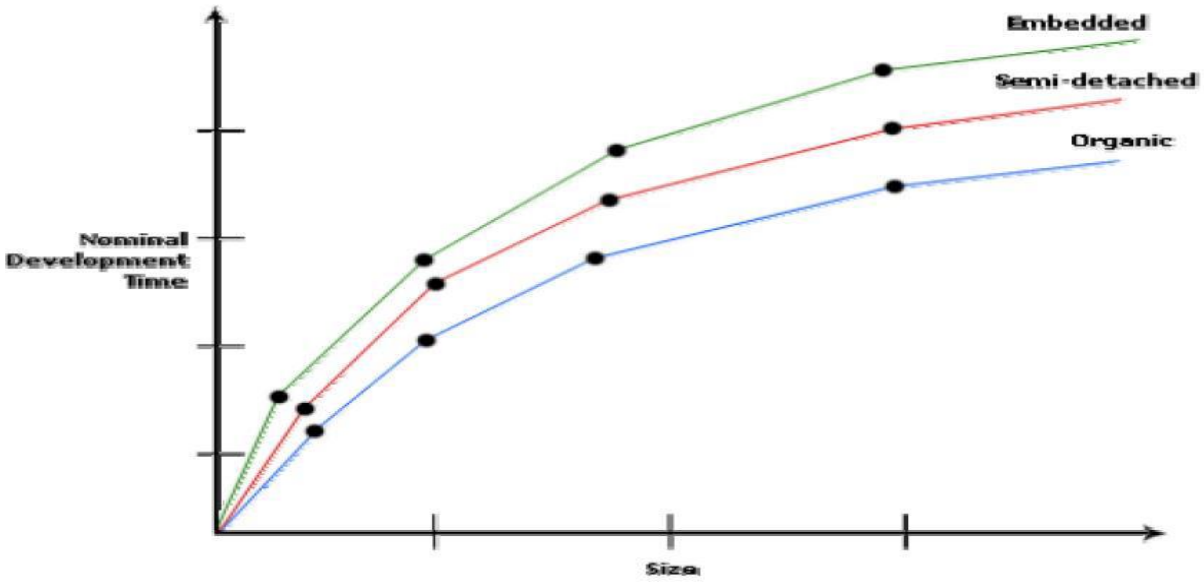
Embedded : **Tdev = 2.5(*Effort*)$^{0.32}$ Months**

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Bellow figure shows a plot of estimated effort versus product size. From bellow figure, we can observe that the effort is somewhat super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.



Effort versus product size

The development time versus the product size in KLOC is plotted in bellow figure. From figure, it can be observed that the development time is a sub linear 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 figure, 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.



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 org 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 develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **$2.4 \times (32)^{1.05}$** = **91 PM**

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

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 a set 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 rate these 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 the following 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 software development.

COMPLETE COCOMO MODEL

A major shortcoming of both the basic and intermediate COCOMO models is that they 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 subsystems may 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. The cost 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.

PROJECT SCHEDULING:

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines

the duration of the project.

The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different engineers. The work breakdown structure formalism helps the manager to breakdown the tasks systematically after the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. In general, the task dependencies define a partial ordering among tasks, i.e. each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities is represented in the form of an activity network.

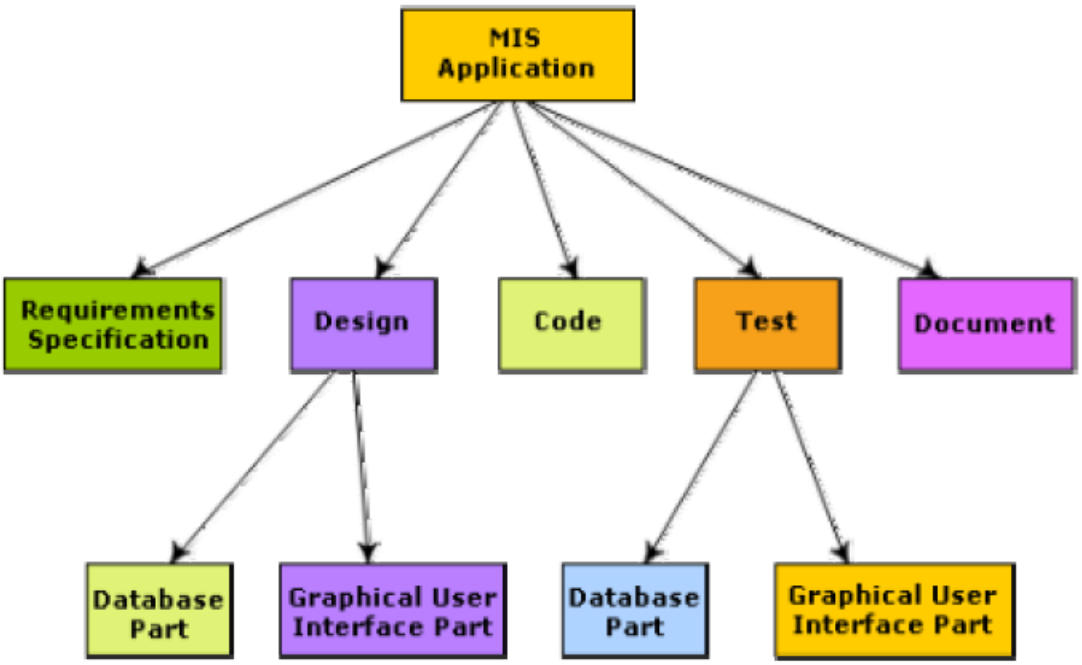
Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT chart representation is developed. The PERT chart representation is suitable for program monitoring and control. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

Work Breakdown Structure

Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem. The root of the tree is labeled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. Bellow fegure represents the WBS of a MIS (Management Information System) software.

While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower). Therefore, to be able to complete a project in the least amount

of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

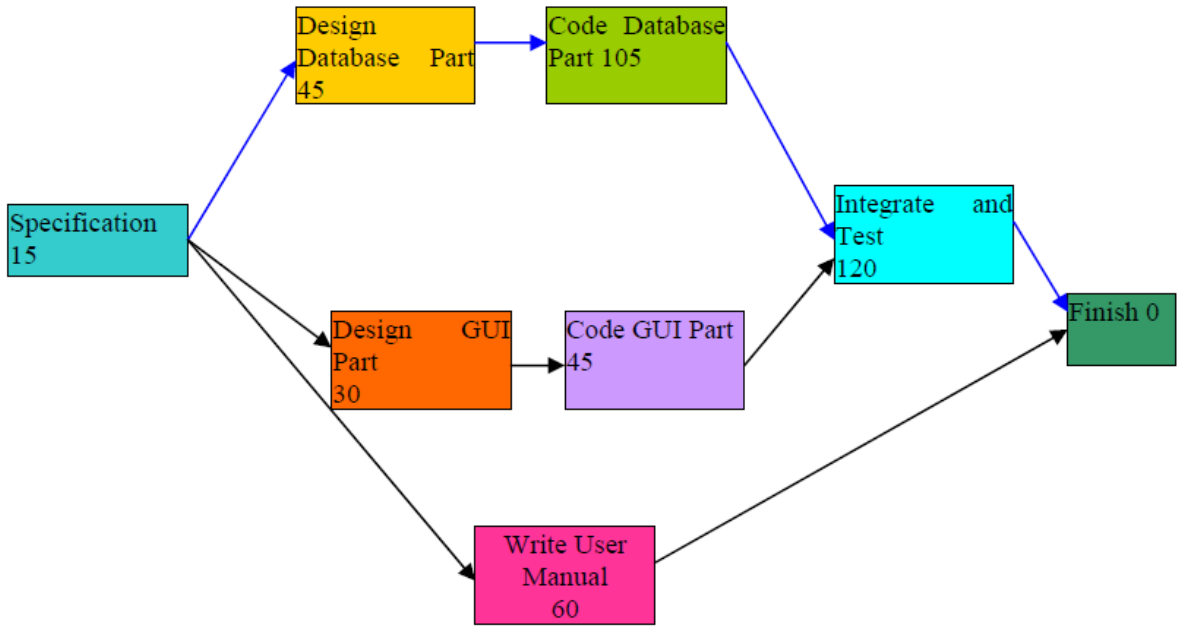


Work breakdown structure of an MIS problem

Activity networks and critical path method WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies. Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however is not a good idea, because software engineers often resent such unilateral decisions. A possible alternative is to let engineer himself estimate the time for an activity he can assigned to. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the engineers to do a better and faster job. However, careful experiments have shown that unrealistically aggressive schedules not only cause engineers to compromise on intangible quality aspects, but also are a cause for

schedule delays. A good way to achieve accurately in estimation of the task durations without creating undue schedule pressures is to have people set their own schedules.



Activity network representation of the MIS problem

Critical Path Method (CPM)

From the activity network representation following analysis can be made. The minimum time (MT) to complete the project is the maximum of all paths from start to finish. The earliest start (ES) time of a task is the maximum of all paths from the start to the task. The latest start time is the difference between MT and the maximum of all paths from this task to the finish. The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task. The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT. The slack time (ST) is LS – EF and equivalently can be written as LF – EF. The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the "flexibility" in starting and completion of tasks. A critical task is one with a zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path. These parameters for different tasks for the MIS problem are shown in the following table.

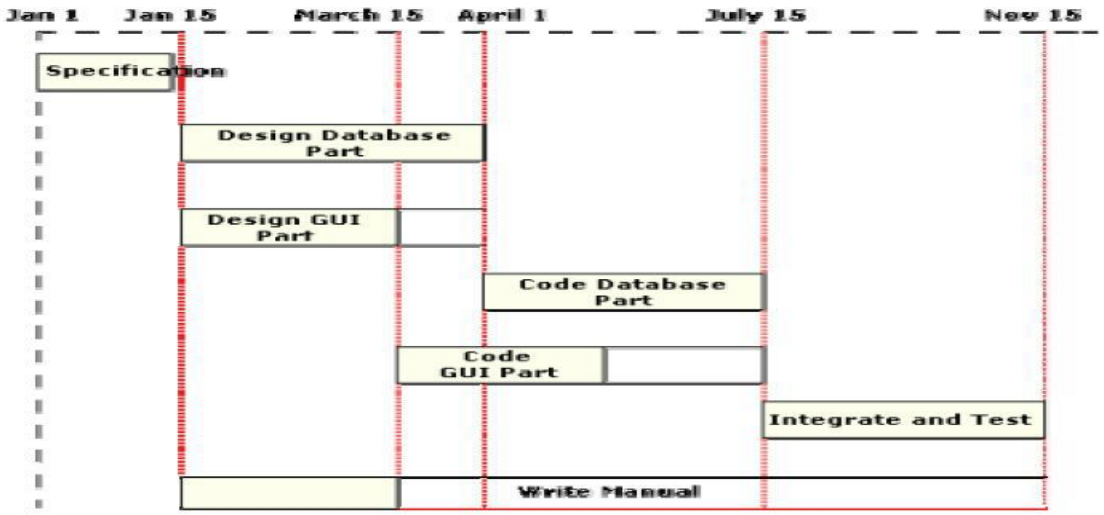
Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code database	60	165	60	165	0
Code GUI part	45	90	120	165	75
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT. The critical path in above figure is shown with a blue arrow.

Gantt Chart

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of above fig is shown in the bellow fig.

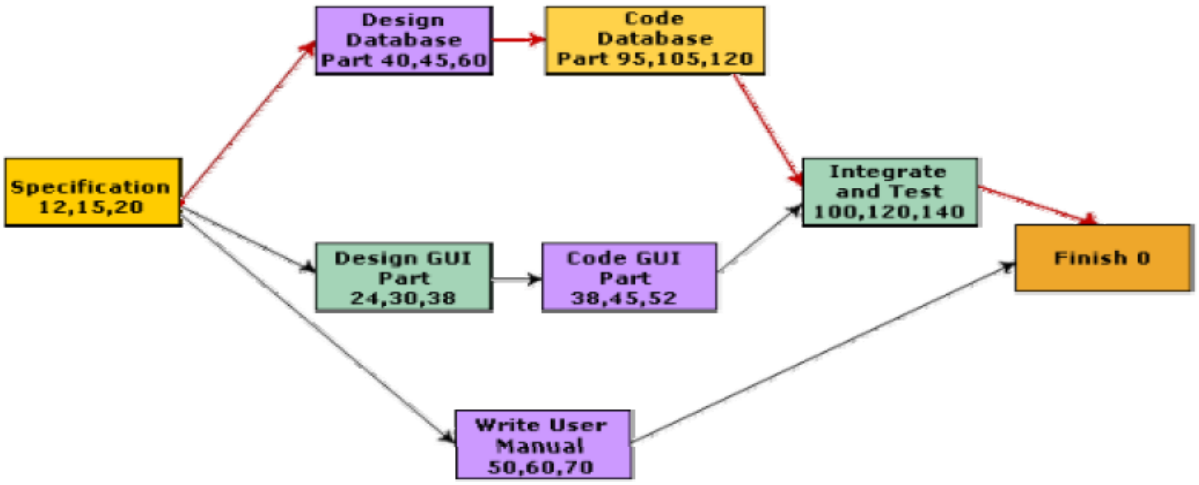


Gantt chart representation of the MIS problem

PERT Chart

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem of above fig is shown in bellow fig. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited.

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.



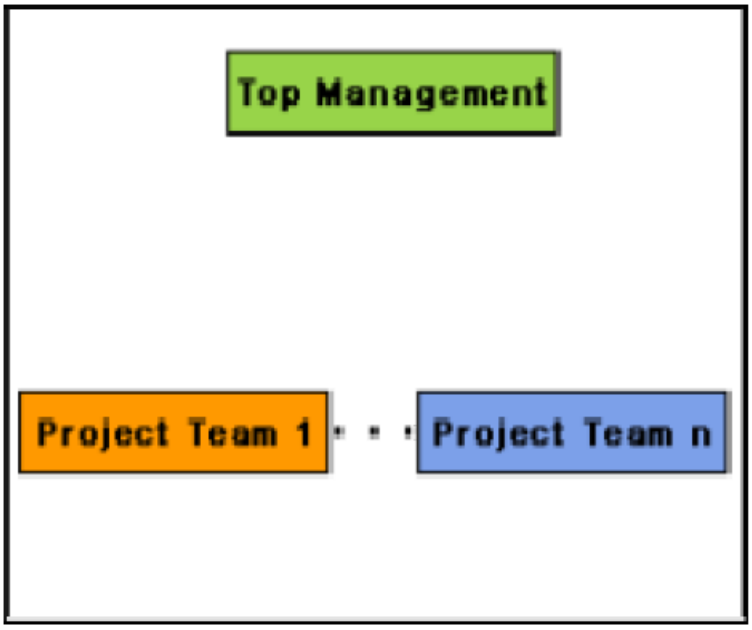
PERT chart representation of the MIS problem

ORGANIZATION STRUCTURE

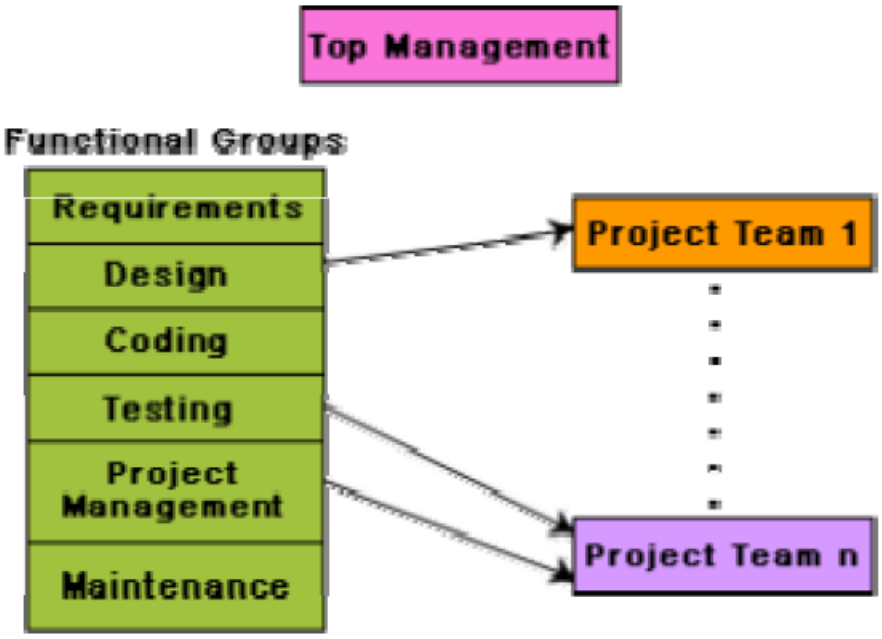
Usually every software development organization handles several projects at any time. Software organizations assign different teams of engineers to handle different software projects. Each type of organization structure has its own advantages and disadvantages so the issue “how is the organization as a whole structured?” must be taken into consideration so that each software project can be finished before its deadline.

Functional format vs. project format

There are essentially two broad ways in which a software development organization can be structured: functional format and project format. In the project format, the project development staffs are divided based on the project for which they work. In the functional format, the development staffs are divided based on the functional group to which they belong. The different projects borrow engineers from the required functional groups for specific phases to be undertaken in the project and return them to the functional group upon the completion of the phase.



Project Organization



Functional Organization
Schematic representation of the functional and project organization

In the functional format, different teams of programmers perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the project evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood by the subsequent teams working on the project. This requires good quality documentation to be produced after every activity.

In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. Obviously, the functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams.

Advantages of functional organization over project organization

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional organization are:

- Ease of staffing
- Production of good quality documents

• Job specialization
• Efficient handling of the problems associated with manpower turnover.

The functional organization allows the engineers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organization also provides an efficient solution to the staffing problem. We have already seen that the staffing pattern should approximately follow the Rayleigh distribution for efficient utilization of the personnel by minimizing their wait times. The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organization. A project organization structure forces the manager to take in almost a constant number of engineers for the entire duration of his project. This results in engineers idling in the initial phase of the software development and are under tremendous pressure in the later phase of the development. A further advantage of the functional organization is that it is more effective in handling the problem of manpower turnover. This is because engineers can be brought in from the functional pool when needed. Also, this organization mandates production of good quality documents, so new engineers can quickly get used to the work already done.

Unsuitability of functional format in small organizations

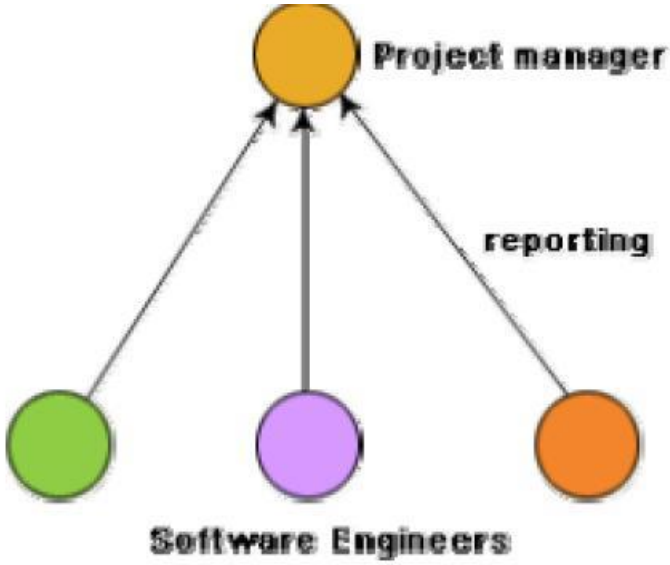
In spite of several advantages of the functional organization, it is not very popular in the software industry. The apparent paradox is not difficult to explain. The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, coder, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organizations to fill in slots for some roles such as maintenance, testing, and coding groups. Also, another problem with the functional organization is that if an organization handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects. Also, for obvious reasons the functional format is not suitable for small organizations handling just one or two projects.

Team Structures

Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized. There are mainly three formal team structures: chief programmer, democratic, and the mixed team organizations although several other variations to these structures are possible. Problems of different complexities and sizes often require different team structures for chief solution.

Chief Programmer Team

In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in bellow figure. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

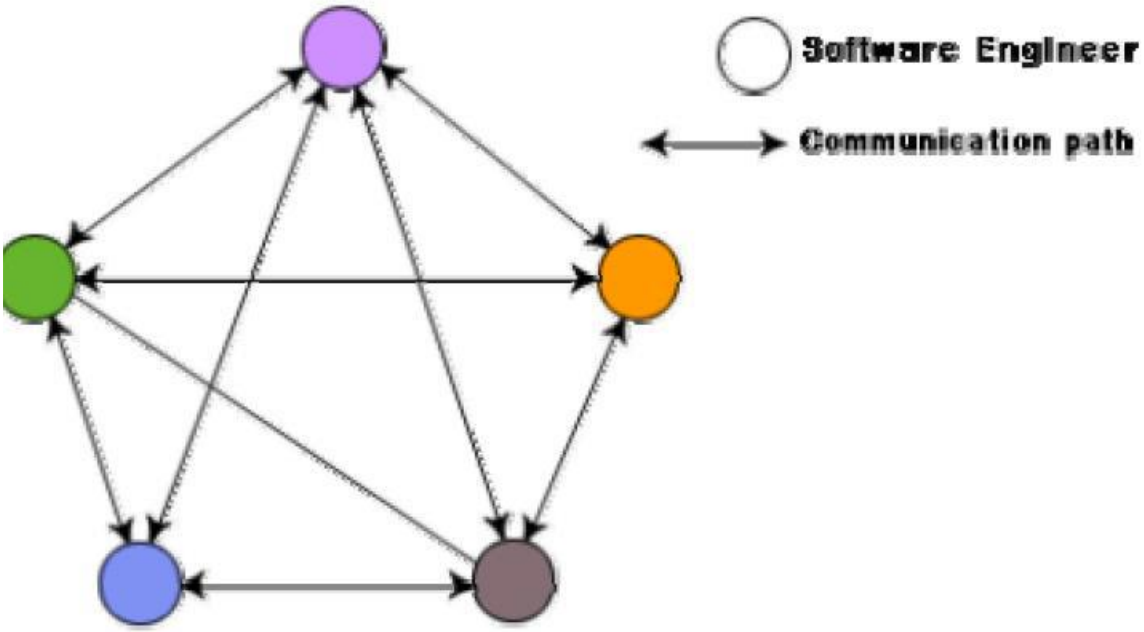


Chief programmer team structure

The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can work out a satisfactory design and ask the programmers to code different modules of his design solution. For example, suppose an organization has successfully completed many simple MIS projects. Then, for a similar MIS project, chief programmer team structure can be adopted. The chief programmer team structure works well when the task is within the intellectual grasp of a single individual. However, even for simple and well-understood problems, an organization must be selective in adopting the chief programmer structure. The chief programmer team structure should not be used unless the importance of early project completion outweighs other factors such as team morale, personal developments, life-cycle cost etc.

Democratic Team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy. Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

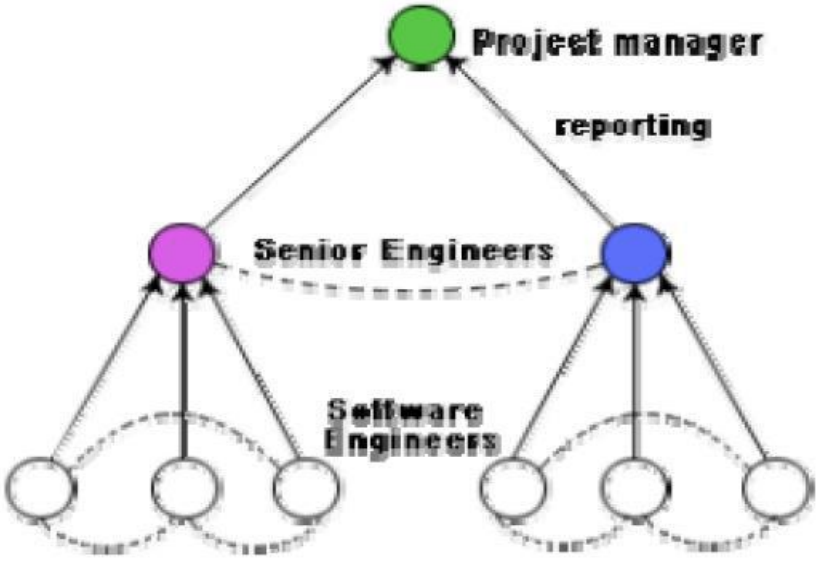


Democratic team structure

The democratic organization leads to higher morale and job satisfaction. Consequently, it suffers from less man-power turnover. Also, democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic. The democratic team organization encourages egoless programming as programmers can share and review one another's work.

Mixed Control Team Organization

The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization. The mixed control team organization is shown pictorially in bellow fig. This team organization incorporates both hierarchical reporting and democratic set up. In bellow fig., the democratic connections are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organization is suitable for large team sizes. The democratic arrangement at the senior engineers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organization is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.



Mixed team structure

Egoless Programming Technique

Ordinarily, the human psychology makes an individual take pride in everything he creates using original thinking. Software development requires original thinking too, although of a different type. The human psychology makes one emotionally involved with his creation and hinders him from objective examination of his creations. Just like temperamental artists, programmers find it extremely difficult to locate bugs in their own programs or flaws in their own design. Therefore, the best way to find problems in a design or code is to have someone review it. Often, having to explain one's program to someone else gives a person enough objectivity to find out what might have gone wrong. This observation is the basic idea behind code walk throughs. An application of this, is to encourage a democratic team to think that the design, code, and other deliverables to belong to the entire group. This is called egoless programming technique.

CHARACTERISTICS OF A GOOD SOFTWARE ENGINEER

The attributes that good software engineers should possess are as follows:

- Exposure to systematic techniques, i.e. familiarity with software engineering principles.
- Good technical knowledge of the project areas (Domain knowledge).
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.

- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team
- Discipline, etc.

STAFFING:

When tasks are defined and schedules are estimated, the planning effort has sufficient information to begin staffing plans and organizing a team into units to address the development problem. The comprehensive staffing plan identifies the required skills and schedules the right people to be brought onto the project at appropriate times and released from the project when their tasks are complete. Selection of individuals to fill position in the staffing plan is a very important step. Errors in staffing can lead to cost increases and schedule slips just as readily as errors in requirements, design, or coding.

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:

1. 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 the product being developed is an important reason why many software projects suffer from the risk of schedule slippage.

2. 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 due to the development team's insufficient knowledge about the project.

3. 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 (denoted as 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 require different 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:

1. **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.
2. **Transfer the risk-** This strategy involves getting the risky component developed by a third party, buying insurance cover, etc.
3. **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 reducing the risk. More formally,

risk leverage = (risk exposure before reduction – risk exposure after reduction) / (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 of a software engineering task is produced and gets successfully reviewed. Milestones need not be placed for every activity. An approximate rule of thumb is to 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 throughout 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 a new 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 an old 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.

1. **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.

2. **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.

3. **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. When an 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).

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

5. **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 every version 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 configuration management 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 system should be kept track of.
- Configuration control ensures that changes to a system happen smoothly.

Configuration Identification

The project manager normally classifies the objects associated with a software development effort into three main categories: controlled, pre controlled, and uncontrolled. Controlled objects are those which are already put under configuration control. One must follow some formal procedures to change them. Pre controlled 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 pre controlled objects. Typical controllable objects include:

- Requirements specification document
- Design documents

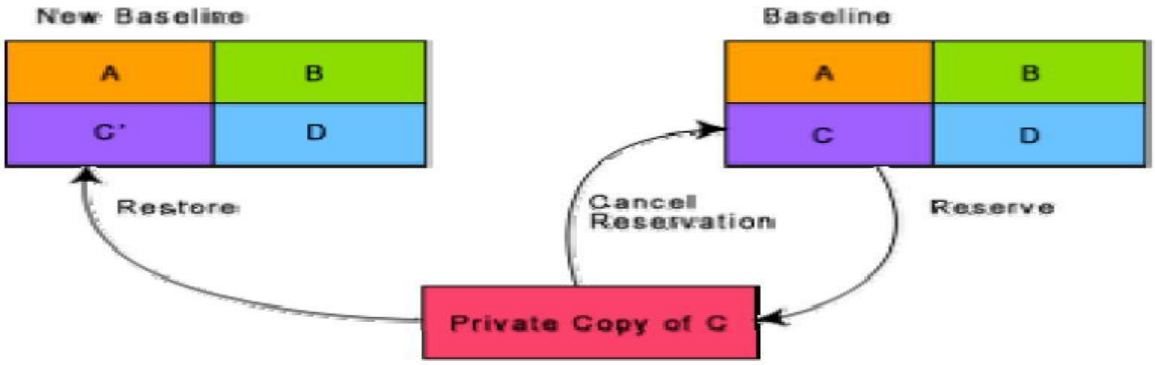
Tools used to build the system, such as compilers, linkers, lexical analyzers, parsers, etc.

- 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 to unreasonably 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 most directly affects the day-to-day operations of developers. The configuration control system 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 bellow figure. Configuration management tools allow only one person to reserve a module at a time. Once an object is reserved, it does not allow anyone else to reserve this module until the reserved module is restored as shown in bellow figure. Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.



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 module to 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. 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.

MODULE-3

STRUCTURED ANALYSIS & DESIGN

OVERVIEW OF DESIGN PROCESS:

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fullfill the requirements mentioned in SRS.

Software Design Levels:

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

- **High-level Design -** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.

- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

HIGH-LEVEL DESIGN

High-level design (HLD) explains the architecture that would be used for developing a software product. The architecture diagram provides an overview of an entire system, identifying the main components that would be developed for the product and their interfaces. The HLD uses possibly nontechnical to mildly technical terms that should be understandable to the administrators of the system. In contrast, low-level design further exposes the logical detailed design of each of these elements for programmers.

A high-level design provides an overview of a system, product, service or process. Such an overview helps supporting components be compatible to others. The highest-level design should briefly describe all platforms, systems, products, services and processes that it depends on and include any important changes that need to be made to them.

In addition, there should be brief consideration of all significant commercial, legal, environmental, security, safety and technical risks, issues and assumptions.

The idea is to mention every work area briefly, clearly delegating the ownership of more detailed design activity whilst also encouraging effective collaboration between the various project teams. Today, most high-level designs require contributions from a number of experts, representing many distinct professional disciplines.

Finally, every type of end-user should be identified in the high-level design and each contributing design should give due consideration to customer experience.

DETAILED DESIGN

It is the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation. Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

COHESION AND COUPLING

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely :

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization.

Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.
- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely:

- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

MODULARITY

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

LAYERING

A layered software design is one in which when the call relationships among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In the layered design, the modules are arranged in the hierarchy of layers.

In such design, a module can only invoke functions of the modules in the layer immediately below it. The higher layer module can be considered similar to management who can invoke lower layer module to get a certain task done.

Layered design makes the work easily understandable, since the layered design easily reflect the relationships among different layers and clears which layer invokes the other layers. This is also helpful in debugging in case of error. When failure is detected then it is obvious that only lower layer can possibly be source of the error.

FUNCTION–ORIENTED SOFTWARE DESIGN

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions changes data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

STRUCTURED ANALYSIS USING DFD

Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way.

It is a systematic approach, which uses graphical tools that analyze and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user.

It has following attributes:

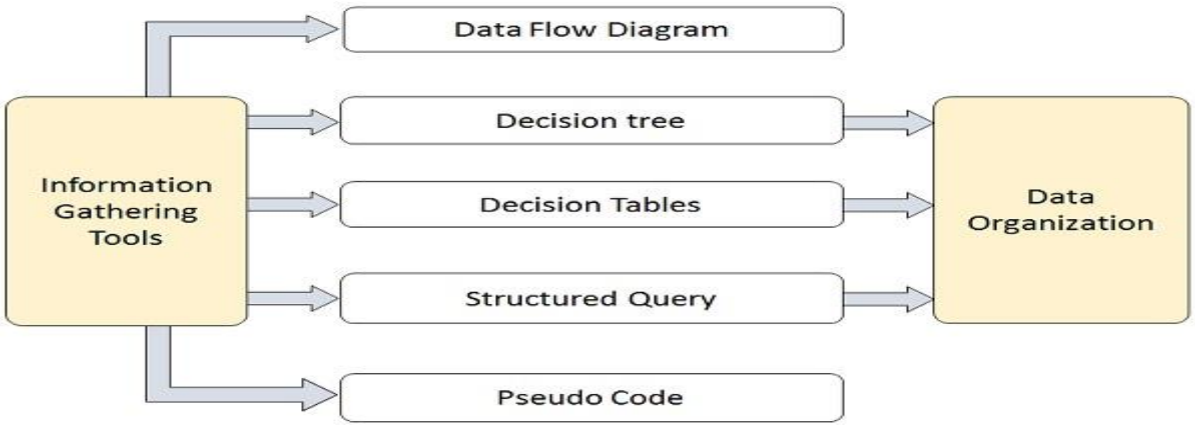
- It is graphic which specifies the presentation of application.
- It divides the processes so that it gives a clear picture of system flow.
- It is logical rather than physical i.e., the elements of system do not depend on vendor or hardware.
- It is an approach that works from high-level overviews to lower-level details.

Structured Analysis Tools

During Structured Analysis, various tools and techniques are used for system development. They are :

- Data Flow Diagrams
- Data Dictionary

- Decision Trees
- Decision Tables
- Structured English
- Pseudocode



Data Flow Diagrams (DFD) or Bubble Chart

It is a technique developed by Larry Constantine to express the requirements of system in a graphical form.

- It shows the flow of data between various functions of system and specifies how the current system is implemented.
- It is an initial stage of design phase that functionally divides the requirement specifications down to the lowest level of detail.
- Its graphical nature makes it a good communication tool between user and analyst or analyst and system designer.
- It gives an overview of what data a system processes, what transformations are performed, what data are stored, what results are produced and where they flow.

Basic Elements of DFD

DFD is easy to understand and quite effective when the required design is not clear and the user wants a notational language for communication. However, it requires a large number of iterations for obtaining the most accurate and complete solution.

The following table shows the symbols used in designing a DFD and their significance:

SYMBOL NAME	SYMBOL	MEANING
Square		Source or Destination of Data
Arrow		Data flow
Cercle		Process transforming data flow
Open Rectangle		Data Store

Types of DFD

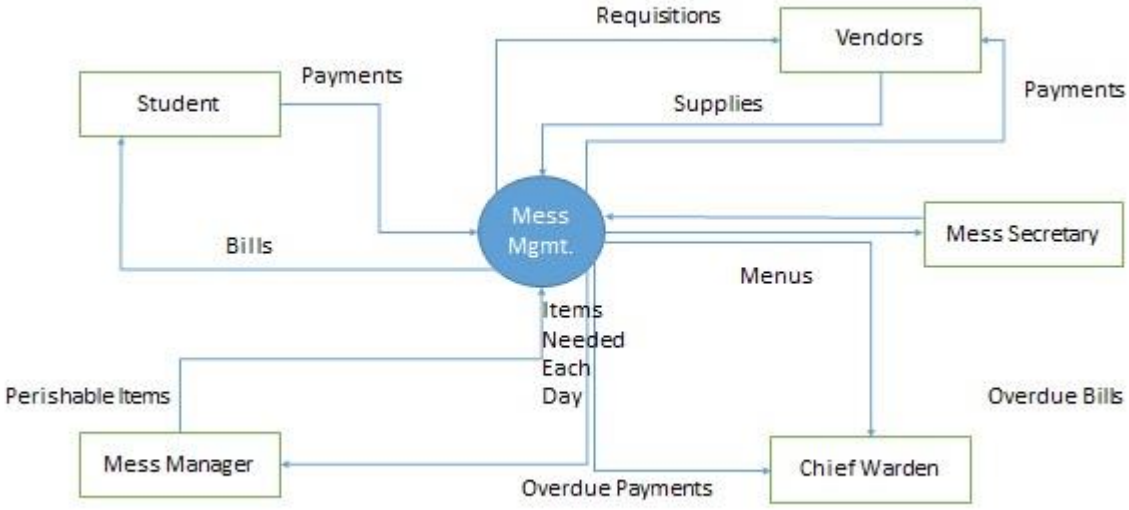
DFDs are of two types: Physical DFD and Logical DFD. The following table lists the points that differentiate a physical DFD from a logical DFD.

Physical DFD	Logical DFD
It is implementation dependent. It shows which functions are performed.	It is implementation independent. It focuses only on the flow of data between processes.
It provides low level details of hardware, software, files, and people.	It explains events of systems and data required by each event.
It depicts how the current system operates and how a system will be implemented.	It shows how business operates; not how the system can be implemented.

Context Diagram

A context diagram helps in understanding the entire system by one DFD which gives the overview of a system. It starts with mentioning major processes with little details and then goes onto giving more details of the processes with the top-down approach.

The context diagram of mess management is shown below.



STRUCTURED DESIGN USING 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 the following:

- **Rectangular boxes:** Represents a module.
- **Module invocation arrows:** Control is passed from one module to another 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:

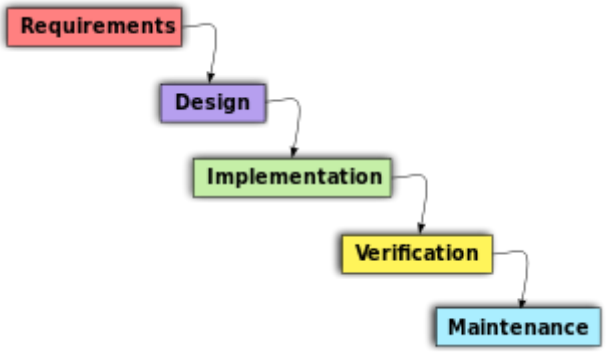
- It is usually difficult to identify the different modules of the software from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

BASIC CONCEPTS OF OBJECT ORIENTED ANALYSIS & DESIGN

It's a structured method for analyzing, designing a system by applying the object-orientated concepts, and develop a set of graphical system models during the development life cycle of the software.

OOAD In The SDLC

The software life cycle is typically divided up into stages going from abstract descriptions of the problem to designs then to code and testing and finally to deployment.



In analysis developers work with users and domain experts to define what the system is supposed to do. Implementation details are supposed to be mostly or totally ignored at this phase.

The goal of the analysis phase is to create a model of the system regardless of constraints such as appropriate technology. This is typically done via use cases and *abstract* definition of the most important objects using conceptual model.

The design phase refines the analysis model and applies the needed technology and other implementation constrains.

It focuses on describing the objects, their attributes, behavior, and interactions. The design model should have all the details required so that programmers can implement the design in code.

Object-Oriented Analysis

1. **Elicit** *requirements:* Define what does the software need to do, and what's the problem the software trying to solve.
2. **Specify** *requirements*: Describe the requirements, usually, using use cases (and scenarios) or user stories.
3. **Conceptual** *model*: Identify the important objects, refine them, and define their relationships and behavior and draw them in a simple diagram.

Object-Oriented Design

The analysis phase identifies the objects, their relationship, and behavior using the conceptual model (an **abstract** definition for the objects).

While in design phase, we describe these objects (by creating class diagram from conceptual diagram — usually mapping conceptual model to class diagram), their attributes, behavior, and interactions.

In addition to applying the software design principles and patterns which will be covered in later tutorials.

The input for object-oriented design is provided by the output of object-oriented analysis. But, analysis and design may occur in parallel, and the results of one activity can be used by the other.

In the object-oriented design, we:

1. **Describe the classes** and their relationships using class diagram.
2. **Describe the interaction** between the objects using sequence diagram.
3. **Apply** software design principles and design patterns.

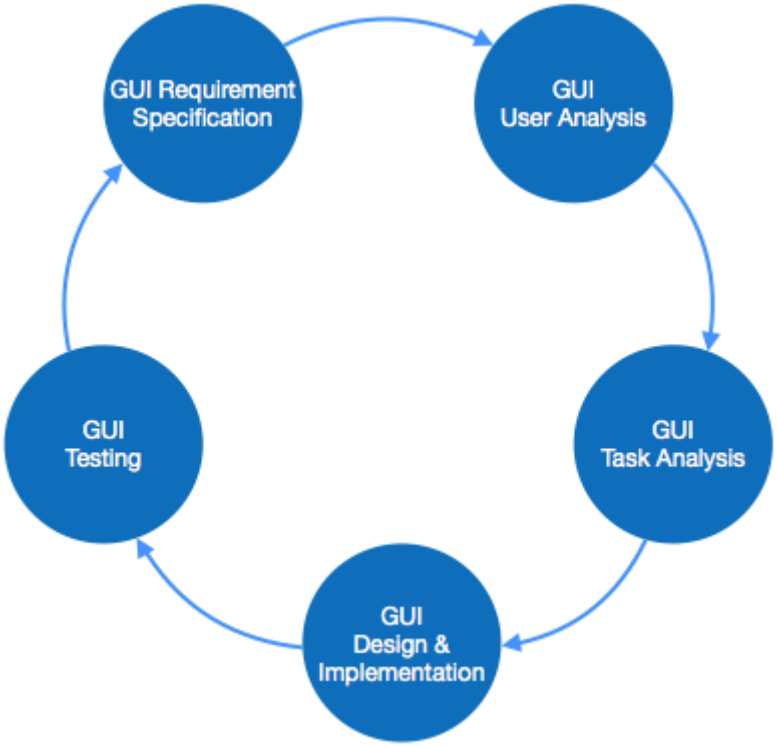
A class diagram gives a visual representation of the classes you need. And here is where you get to be really specific about object-oriented principles like inheritance and polymorphism.

Describing the interactions between those objects lets you better understand the responsibilities of the different objects, the behaviors they need to have.

USER INTERFACE DESIGN

There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.

A model used for GUI design and development should fulfill these GUI specific steps.



- **GUI Requirement Gathering** - The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
- **User Analysis** - The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
- **Task Analysis** - Designers have to analyze what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
- **GUI Design & implementation** - Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.
- **Testing** - GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

COMMAND LANGUAGE

A command language is a type of interpreted language using a command line structure. Command languages are typically not compiled but are interpreted on the fly. A prominent example is the MS-DOS computer system that controlled earlier personal computers where a command line structure was used to generate user-driven processes.

COMMAND LINE INTERFACE (CLI)

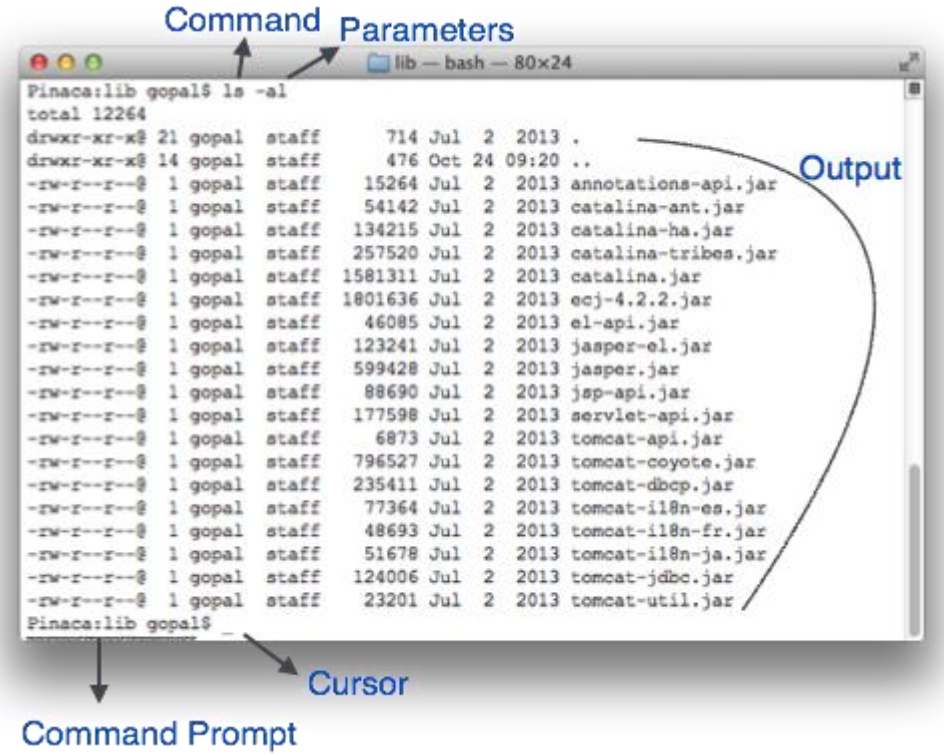
CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

CLI Elements



A text-based command line interface can have the following elements:

- **Command Prompt** - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
- **Cursor** - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
- **Command** - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

MENU AND ICONIC INTERFACES

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

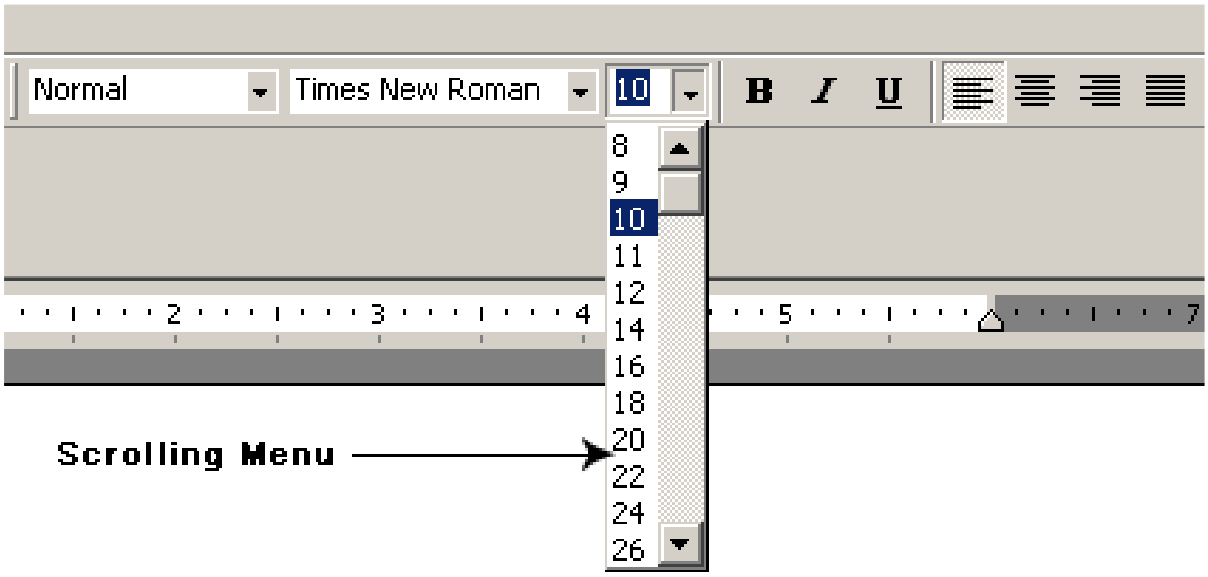
However, experienced users find a menu-based user interface to be slower than a command language-based interface because an experienced user can type fast and can get speed advantage by composing different primitive commands to express complex commands. Composing commands in a menu-based interface is not possible. This is because of the fact that actions involving logical connectives (and, or, etc.) are awkward to specify in a menu-based system. Also, if the number of choices is large, it is difficult to select from the menu. In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms.

When the menu choices are large, they can be structured as the following way:

Scrolling menu

When a full choice list can not be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs. This is important since the user cannot see all the commands at any one time. An example situation where a scrolling menu is frequently used is font size selection in a document processor (as shown in bellow figure). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up and down to find the size he is looking for. However, if the commands do not have any definite ordering relation,

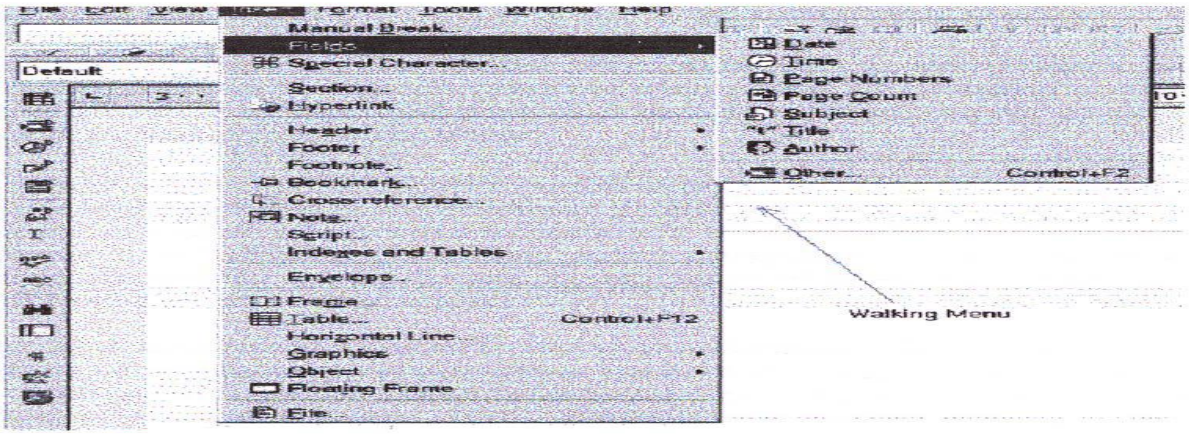
then the user would have to in the worst case, scroll through all the commands to find the exact command he is looking for, making this organization inefficient.



Font size selection using scrolling menu

Walking menu

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



Example of walking menu

Hierarchical menu

In this technique, the menu items are organized in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus in this case, one can consider the menu and its various sub-menus to form a hierarchical tree-like structure. Walking menu can be considered to be a form of hierarchical menu which is practicable when the tree is shallow. Hierarchical menu can be used to manage large number of choices, but the users are likely to face navigational problems because they might lose track of where they are in the menu tree. This probably is the main reason why this type of interface is very rarely used.

Types of menus and their features

Three main types of menus are scrolling menu, walking menu, and hierarchical menu. The features of scrolling menu, walking menu, and hierarchical menu have been discussed earlier.

Iconic interface

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

Example of an iconic interface

Above figure shows an iconic interface. Here, the user is presented with a set of icons at the top of the frame for performing various activities. On clicking on any of the icons, either the user is prompted with a sub menu or the desired activity is performed.

CODING AND SOFTWARE TESTING TECHNIQUES

CODING

The purpose of the coding phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested.

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 names is most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without 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 supporting documents 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.

UNIT TESTING

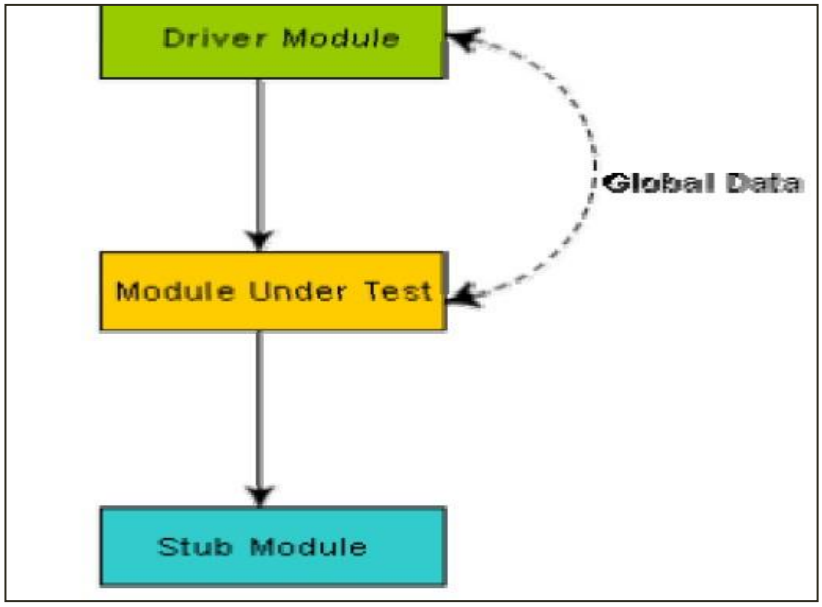
Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

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 test the module:

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

Modules are 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 bellow figure. A stub procedure is a dummy procedure that has the same I/O parameters as the given 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.



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:

- 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.
- 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}.

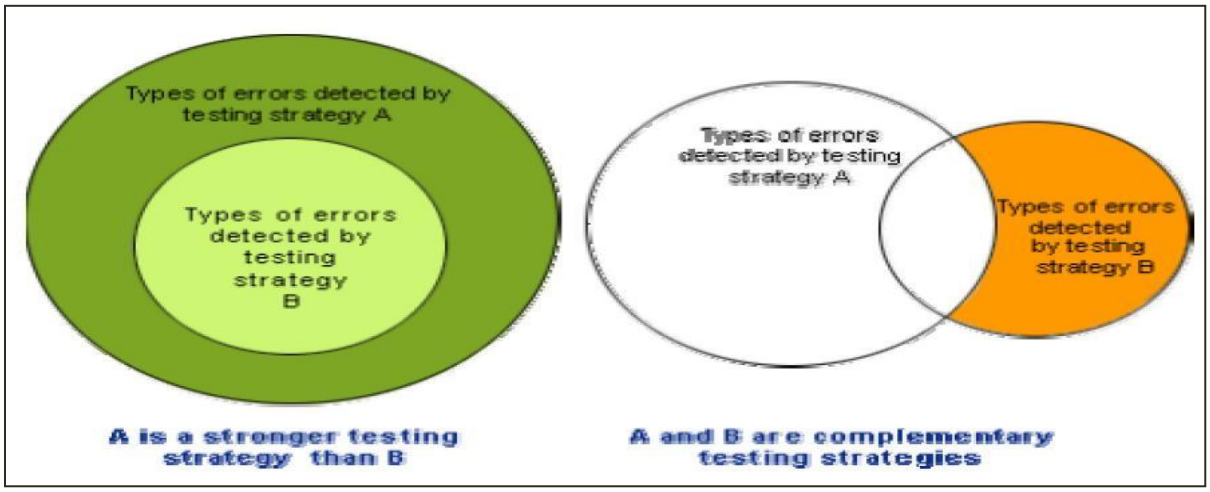
Boundary Value Analysis

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 < instead of <=, 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}.

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 bellow given figure.



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 can be {(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.

CYCLOMATIC COMPLEXITY MEASURE

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. The answers 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.

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 of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability.

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.

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 which would run all the test cases automatically.

DEBUGGING TECHNIQUES

Need for Debugging

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

Debugging Approaches

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

1. **Brute Force Method:**

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

2. **Backtracking:**

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

3. **Cause Elimination Method:**

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

4. **Program Slicing:**

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Debugging Guidelines

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

INTEGRATION TESTING

The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using

an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested. An important factor that guides the integration plan is the module dependency graph. The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart the integration plan can be developed.

Integration test approaches

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

- Big bang approach
- Bottom- up approach
- Top-down approach
- Mixed-approach

Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

Bottom-Up Integration Testing

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well- defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom- up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

Top-Down Integration Testing

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested. Top-down integration testing

approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

Mixed Integration Testing

A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top- down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

SYSTEM TESTING

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

- **Alpha Testing.** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- **Beta testing.** Beta testing is the system testing performed by a select group of friendly customers.
- **Acceptance Testing.** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the SRS document. Broadly, these tests can be classified into functionality and performance tests. The functionality test tests the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance test tests the conformance of the system with the nonfunctional requirements of the system.

REGRESSION TESTING

This type of testing is required when the system being tested is an upgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

MODULE-4

BASIC CONCEPTS IN 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 the corresponding instruction is 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.

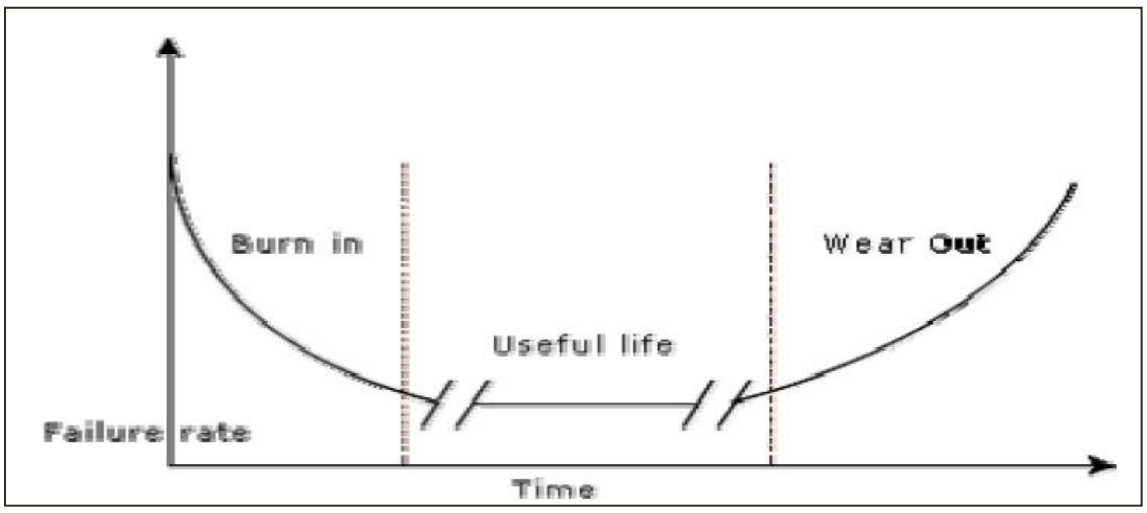
Reasons for software reliability being difficult to measure

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

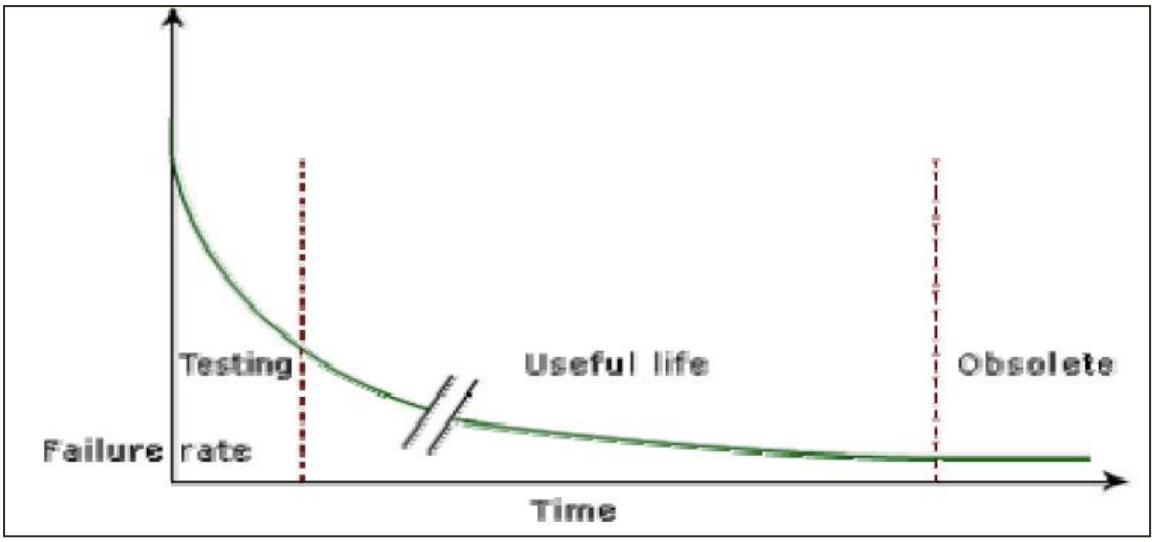
- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.
- Hardware reliability vs. software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

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 decrease if 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 bellow two figures. For hardware products, it can be 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 the failure rate increases. This gives the plot of hardware reliability over time its characteristics "bath tub" shape. On the other hand, for software the failure rate is 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. This error 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.



Hardware product



Software product
Change in failure rate of a product

CONCEPT OF SOFTWARE RELIABILITY

Most important and dynamic characteristic of software is its reliability. Informally, the reliability of a software system is a measure of how well it provides the services expected of it by its users but a useful formal definition of reliability is much harder to express. Software reliability metrics such as 'mean time between failures' may be used but they do not take into account the subjective nature appropriate and useful.

Users do not consider all services to be of equal importance and a system might be viewed as unreliable if it ever fails to provide some critical service. For example, say a system was used to control braking on an aircraft but failed to work a single set of very rare conditions. If the aircraft crashed because these failure conditions occurred, pilots of similar aircraft would (reasonably) regard the software as unreliable.

On the other hand, say a comparable software system provided some visual indication of its actions to the pilot. Assume this failed once per month in a predictable way without the main system function being affected and other indicators showed that the control system was working normally. In spite of frequent failure, pilots would not consider that software as unreliable as the system, which caused the catastrophic failure.

Reliability is a dynamic system characteristic, which is a function of the number of software failures. A software failure is an execution event where the software behaves in an unexpected way. This is not the same as a software fault, which is a static program characteristic. Software faults cause software failures when the faulty code is executed with

a particular set of inputs. Faults do not always manifest themselves as failures so the reliability depends on how the software is used. It is not possible to produce a single, universal statement of the software reliability.

Software faults are not just program defects. Unexpected behaviour can occur in circumstances where the software conforms to its requirements themselves are complete. Omissions in software documentations can also lead to unexpected behaviour, although the software may not contain defects.

Reliability depends on how the software is used, so it cannot be specified absolutely. Each user uses a program in different ways so faults, which affect the reliability of the system for one user, may never manifest themselves under a different mode of working. Reliability can only be accurately specified if the normal software operational profile is also specified.

As reliability is related to the probability of an error occurring in operational use, a program may contain known faults but may still never be seen select an erroneous input; the program always appears to be reliable. Furthermore, experienced users may 'work around' known software faults and deliberately avoid using features, which they know to be faulty. Repairing the faults in these features may make no practical difference to the reliability as perceived by these users.

For example, the word processor used to write this book has an automatic hyphenation capability. This facility is used when text columns are short and any faults might manifest themselves when users produce multi-column documents. I never use hyphenation so, from my viewpoint, faults in the hyphenation code do not affect the reliability of the word.

ERROR, FAULT AND FAILURE

So far, we have used the intuitive meaning of the term error to refer to problems in requirements, design, or code. Sometimes error, fault, and failure are used interchangeably, and sometimes they refer to different concepts. Let us start by defining these concepts clearly. We follow the IEEE definitions for these terms.

The term **error** is used in two different ways. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is, error refers to the difference between the actual output of a software and the correct output. In this interpretation, error is essentially a measure of the difference between the actual and the ideal. Error is also used to refer to human action those results in software containing a defect or fault. This definition is quite general and encompasses all the phases.

Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is synonymous with the commonly used term bug. The term error is also often used to refer to defects (taking a variation of the second

definition of error). In this book, we will continue to use the terms in the manner commonly used, and no explicit distinction will be made between errors and faults, unless necessary. It should be noted that the only " faults that a software has are "design faults"; there is no wear and tear in software.

Failure is the inability of a system or component to perform a required function, according to its specifications. A software failure occurs if the behaviour of the software is different from the specified behaviour. Failures may be caused due to functional or performance reasons. A failure is produced only when there is a fault in the system. However, presence of a fault does not guarantee a failure. In other words, faults have the potential to cause failures and their presence is a necessary but not a sufficient condition for failure to occur. Definition does not imply that a failure must be observed. It is possible that a failure may occur but not be detected.

There are some implications of these definitions. Presence of an error (in the state) implies that a failure must have occurred, and the observance of a failure implies that a fault must be present in the system. However, the presence of a fault does not imply that a failure must occur. The presence of a fault in a system only implies that the fault has a potential to cause a failure to occur. Whether a fault actually manifests itself in certain time duration depends on many factors. This means that if we observe the behaviour of a system for some time and we do not observe any errors, we cannot say anything about the presence or absence of faults in the system. If, on the other hand, we observe some failure in this duration, we can say that there are some faults in the system.

HARDWARE RELIABILITY VS. SOFTWARE RELIABILITY

Reliability behaviour 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 decrease if 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 below. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed.

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 performance measurement. 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.

1. **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 the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.

2. **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. An MTTF of 500 means that one failure can be expected in every 500 time unit.
- MTTF is relevant for system with long transaction i.e when processing takes long time

3. **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.

4. **Mean Time Between Failure (MTBF)**

- MTTF and MTTR can be combined to get the MTBF metric: MTBF = MTTF + MTTR.
- MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.

5. **Probability of Failure on Demand (POFOD)**

- 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 0.001would mean that 1 out of every 1000 service requests would result in a failure.

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 are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

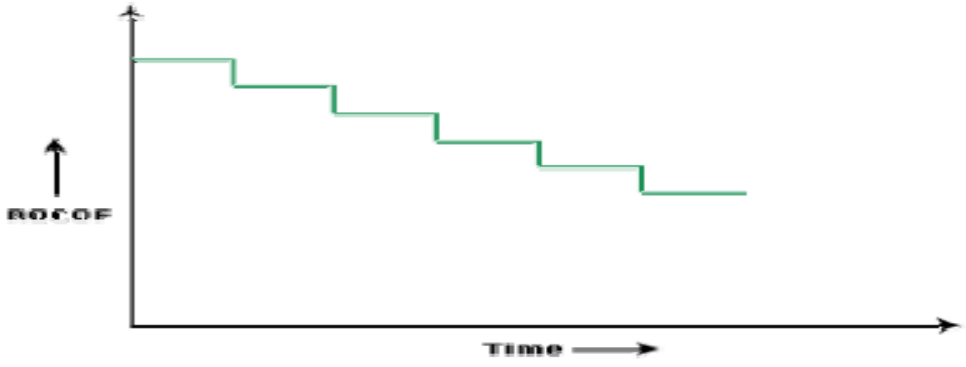
$$\text{AVAILABILITY} = \frac{MTTF}{(MTTF + MTTR)} \times 100$$

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 modelling 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. However, this simple 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 reliabilitygrowth.

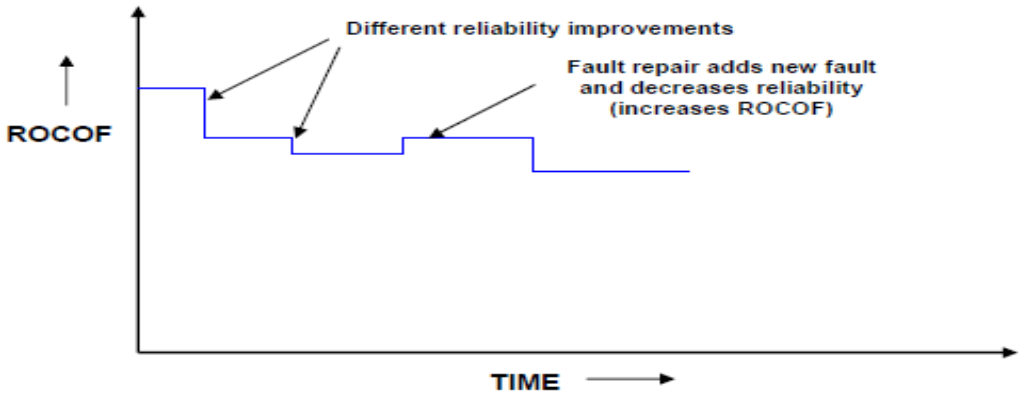


Step function model of reliability growth

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 .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.



Random-step function model of reliability growth

SOFTWARE QUALITY, SEI CMM AND ISO-9001

Software quality:

A quality product is defined in terms of its fitness of purpose robustness. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document.

The modern view of a quality associates with a software product several quality factors such as the following:

- **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc. It should platform independent.
- **Usability:** A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product easy to handle.
- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.
- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.
- **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

SEI Capability Maturity Model:

- SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.
- SEI CMM can be used two ways: capability evaluation and software process assessment.
    1. **Capability evaluation:**
       Capability evaluation provides a way to assess the software process capability of an organization.
    2. **Software process assessment:**
       Software process assessment is used by an organization with the objective to improve its process capability.
- SEI CMM classifies software development industries into the following five maturity levels.

1. **Level 1: Initial.**

A software development organization at this level is characterized by ad hoc activities. Very few or no processes are defined and followed.

2. **Level 2: Repeatable.**

At this level, the basic project management practices such as tracking cost and schedule are established.

3. **Level 3: Defined.**

At this level the processes for both management and development activities are defined and documented.

4. **Level 4: Managed.**

At this level, the focus is on software metrics.

5. **Level 5: Optimizing.**

At this stage, process and product metrics are collected.

Personal software process:

- Personal Software Process (PSP) is a scaled down version of the industrial software process.
- PSP is suitable for individual use. SEI CMM does not tell software developers how to analyze, design, code, test, or document software products, but assumes that engineers use effective personal practices.
- PSP recognizes that the process for individual use is different from that for a team.

Six sigma:

- The purpose of Six Sigma is to improve processes to do things better, faster, and at lower cost.
- It can be used to improve every facet of business, from production, to human resources, to order entry, to technical support.
- Six Sigma can be used for any activity that is concerned with cost, timeliness, and quality of results.

Sub-methodologies: DMAIC and DMADV.

The Six Sigma DMAIC process (defines, measure, analyze, improve, control) is an improvement system for existing processes failing below specification and looking for incremental improvement.

The Six Sigma DMADV process (define, measure, analyze, design, verify) is an improvement system used to develop new processes or products at Six Sigma quality levels.

ISO 9000 certification:

- ISO certification serves as a reference for contract between independent parties.
- The ISO 9000 standard specifies the guidelines for maintaining a quality system.
- ISO 9000 specifies a set of guidelines for repeatable and high quality product development.

Types of ISO 9000 quality standards:

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003.

- **ISO 9001:**
  ISO 9001 applies to the organizations that engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.
- **ISO 9002:**
  ISO 9002 applies to those organizations which do not design products but are only involved in production.
- **ISO 9003:**
  ISO 9003 applies to organizations that are involved only in installation and testing of the products.

Salient features of ISO 9001 certification:

The salient features of ISO 9001 are as follows:

- All documents concerned with the development of a software product should be properly managed, authorized, and controlled.
- Important documents should be independently checked and reviewed for effectiveness and correctness.
- The product should be tested against specification.
- Several organizational aspects should be addressed e.g., management reporting of the quality team

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 up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

TYPES OF SOFTWARE MAINTENANCE

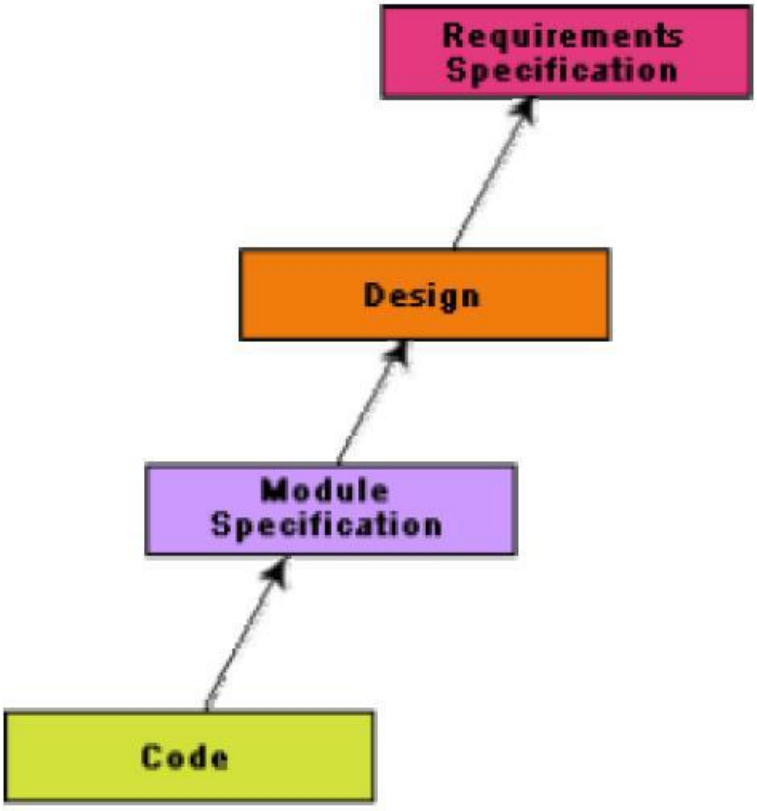
There are basically three types of software maintenance. These are:

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

SOFTWARE REVERSE ENGINEERING

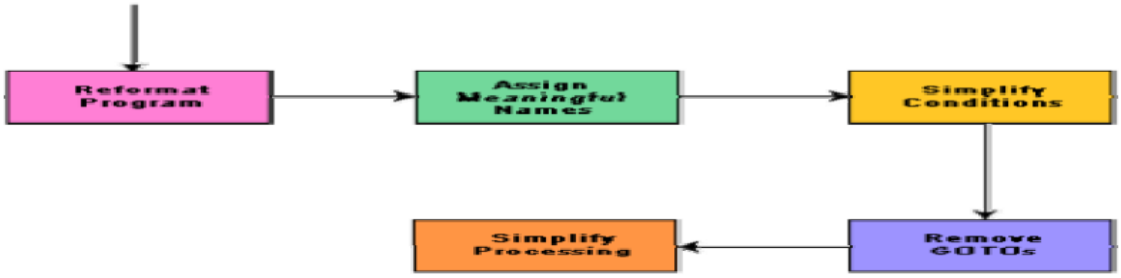
Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents 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 bellow figure. A program can be reformatted using any of the several available pretty printer 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 conditional statements or whenever appropriate by case statements.



A process model for reverse engineering

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



Cosmetic changes carried out before reverse engineering

SOFTWARE REENGINEERING

Software reengineering is a combination of two consecutive processes i.e. software reverse engineering and software forward engineering.

SOFTWARE REUSE

Advantages of software reuse

Software products are expensive. Software project managers are worried about the high cost of software development and are desperately look for ways to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality.

Artifacts that can be reused

It is important to know about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

- Requirements specification
- Design
- Code
- Test cases
- Knowledge

Pros and cons of knowledge reuse

Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts i.e. requirements specification, design, code, test cases, reuse of knowledge occurs automatically without any conscious effort in this direction. However, two major difficulties with unplanned reuse of knowledge are that a developer experienced in one type of software product might be included in a team developing a different type of software. Also, it is difficult to remember the details of the potentially reusable development knowledge. A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

Easiness of reuse of mathematical functions

The routines of mathematical libraries are being reused very successfully by almost every programmer. No one in his right mind would think of writing a routine to compute sine or cosine. Reuse of commonly used mathematical functions is easy. Several interesting aspects emerge. Cosine means the same to all. Everyone has clear ideas about what kind of

argument should cosine take, the type of processing to be carried out and the results returned. Secondly, mathematical libraries have a small interface. For example, cosine requires only one parameter. Also, the data formats of the parameters are standardized.

Basic issues in any reuse program

The following are some of the basic issues that must be clearly understood for starting any reuse program.

- Component creation
- Component indexing and storing
- Component search
- Component understanding
- Component adaptation
- Repository maintenance

1. **Component creation-** For component creation, the reusable components have to be first identified. Selection of the right kind of components having potential for reuse is important. Domain analysis is a promising technique which can be used to create reusable components.

2. **Component indexing and storing-** Indexing requires classification of the reusable components so that they can be easily searched when looking for a component for reuse. The components need to be stored in a Relational Database Management System (RDBMS) or an Object-Oriented Database System (ODBMS) for efficient access when the number of components becomes large.
3. **Component searching-** The programmers need to search for right components matching their requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.
4. **Component understanding-** The programmers need a precise and sufficiently complete understanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.
5. **Component**
6. **adaptation-** Often, the components may need adaptation before they can be reused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.
7. **Repository maintenance-** A component repository once is created requires continuous maintenance. New components, as and when created have to be entered into the repository.

The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

A client is basically a consumer of services and server is a provider of services as shown in fig.1. 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.

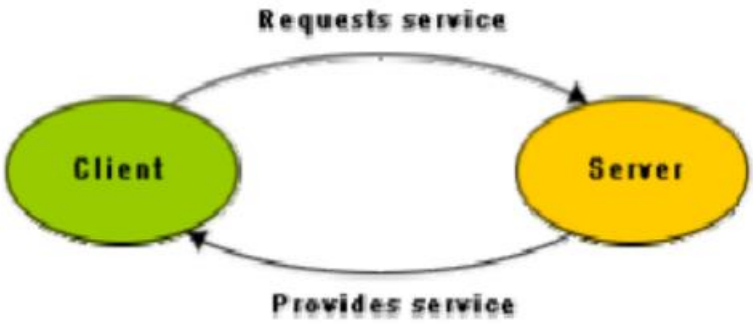


Fig. 1: Client-server model

Example: A man was visiting his friend's town in his car. The man had a handheld computer (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. 2.

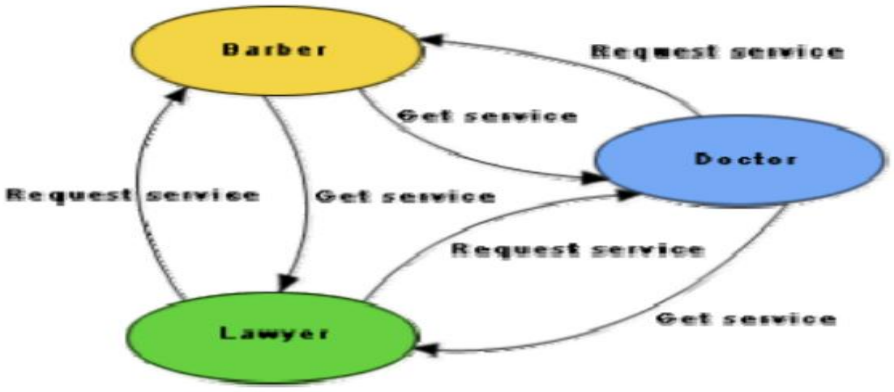


Fig.2 Client and server as roles

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 to monolithic 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 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, a developer essentially integrates pre-built components purchased off-theshelf. 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 can be switched to a redundant server.

There are many other advantages of client-server software. For example, we can locate 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 the communication 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.

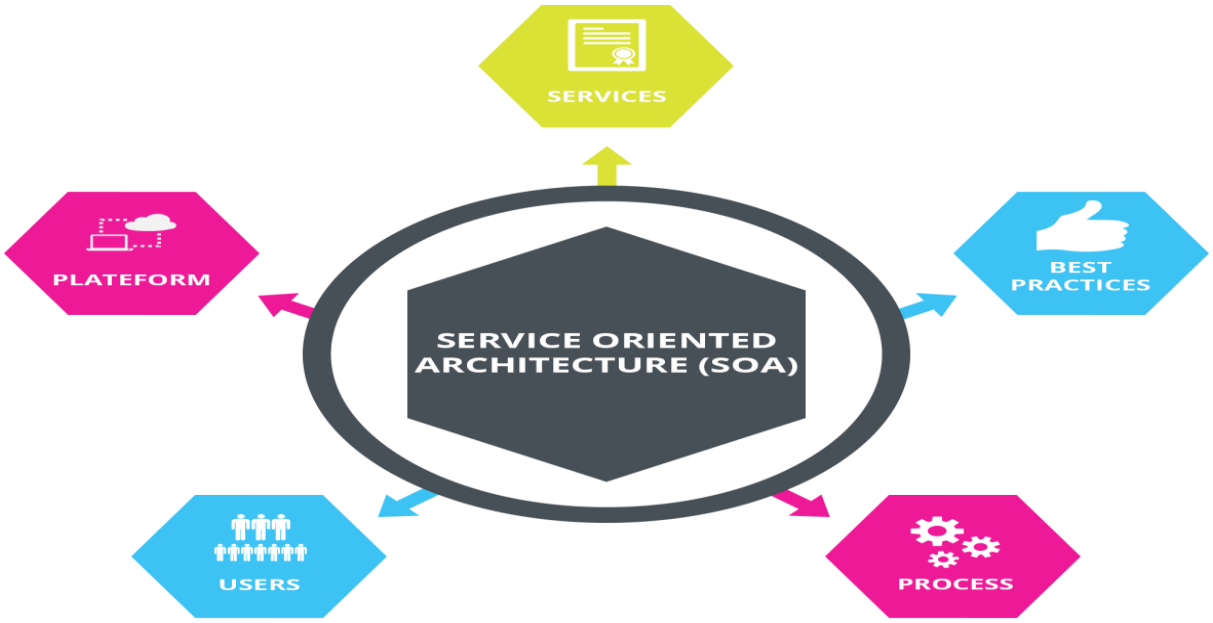
• Compatibility – Clients and servers may not be compatible to each other. Since the client and server components may be manufactured by different vendors, they may not be compatible with respect to data types, language, etc.

• Inconsistency – Replication of servers is a problem as it can make data inconsistent.

SERVICE-ORIENTED ARCHITECTURE (SOA)

Service-Oriented Architecture (SOA) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. Its principles are independent of vendors and other technologies.

In **service oriented architecture**, a number of services communicate with each other, in one of two ways: through passing data or through two or more services coordinating an activity. This is just one definition of Service-Oriented Architecture. An article on Wikipedia goes into much more detail.



Characteristics Of Service-Oriented Architecture

While the defining concepts of Service-Oriented Architecture vary from company to company, there are six key tenets that overarch the broad concept of Service-Oriented Architecture. These core values include:

- Business value
- Strategic goals
- Intrinsic inter-operability
- Shared services
- Flexibility
- Evolutionary refinement

Each of these core values can be seen on a continuum from older format distributed computing to Service-Oriented Architecture to cloud computing (something that is often seen as an offshoot of Service-Oriented Architecture).

Service-Oriented Architecture Patterns

There are three roles in each of the Service-Oriented Architecture building blocks: service provider; service broker, service registry, service repository; and service requester/consumer.

The service provider works in conjunction with the service registry, debating the whys and hows of the services being offered, such as security, availability, what to charge, and more. This role also determines the service category and if there need to be any trading agreements.

The service broker makes information regarding the service available to those requesting it. The scope of the broker is determined by whoever implements it.

The service requester locates entries in the broker registry and then binds them to the service provider. They may or may not be able to access multiple services; that depends on the capability of the service requester.

Implementing Service-Oriented Architecture

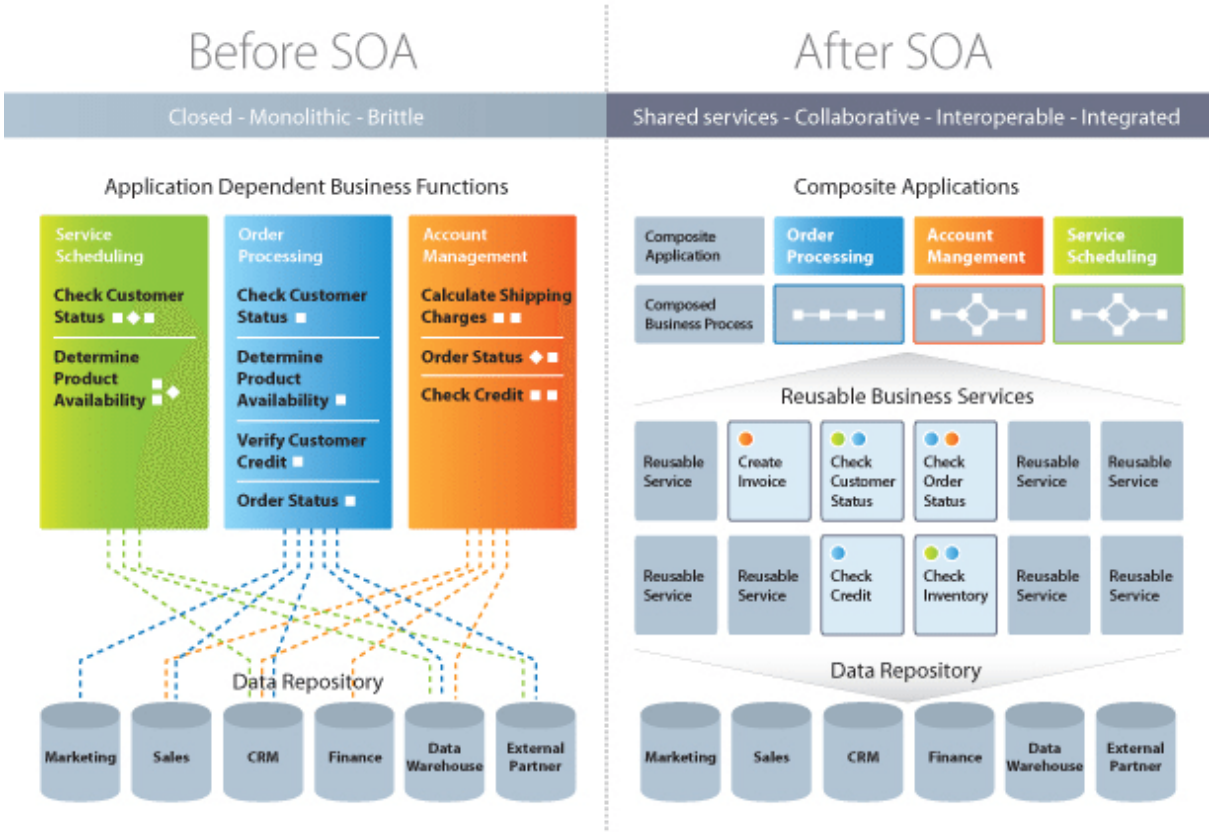
When it comes to implementing service-oriented architecture (SOA), there is a wide range of technologies that can be used, depending on what your end goal is and what you're trying to accomplish.

Typically, Service-Oriented Architecture is implemented with web services, which makes the "functional building blocks accessible over standard internet protocols."

An example of a web service standard is SOAP, which stands for Simple Object Access Protocol. In a nutshell, SOAP "is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. Although SOAP wasn't well-received at first, since 2003 it has gained more popularity and is becoming more widely used and accepted. Other options for implementing Service-Oriented Architecture include Jini, COBRA, or REST.

It's important to note that architectures can "operate independently of specific technologies," which means they can be implemented in a variety of ways, including messaging, such as ActiveMQ; Apache Thrift; and SORCER.

Why Service-Oriented Architecture Is Important



There are many benefits to service-oriented architecture, especially in a web service based business. We'll outline a few of those benefits here, in brief:

- **Use Service-Oriented Architecture to create reusable code:** Not only does this cut down on time spent on the development process, but there's no reason to reinvent the coding wheel every time you need to create a new service or process. Service-Oriented Architecture also allows for using multiple coding languages because everything runs through a central interface.

- **Use Service-Oriented Architecture to promote interaction:** With Service-Oriented Architecture, a standard form of communication is put in place, allowing the various systems and platforms to function independent of each other. With this interaction, Service-Oriented Architecture is also able to work around firewalls, allowing "companies to share services that are vital to operations."

- **Use Service-Oriented Architecture for scalability:** It's important to be able to scale a business to meet the needs of the client, however certain dependencies can get in the way of that scalability. Using Service-Oriented Architecture cuts back on the client-service interaction, which allows for greater scalability.

- **Use Service-Oriented Architecture to reduce costs:** With Service-Oriented Architecture, it's possible to reduce costs while still "maintaining a desired level of output." Using Service-Oriented Architecture allows businesses to limit the amount of analysis required when developing custom solutions.

SOFTWARE AS A SERVICE (SAAS)

Software as a service (SaaS) is a software distribution model in which a third-party provider hosts applications and makes them available to customers over the Internet. SaaS is one of three main categories of cloud computing, alongside infrastructure as a service (IaaS) and platform as a service (PaaS). SaaS is closely related to the application service provider (ASP) and on demand computing software delivery models. The hosted application management model of SaaS is similar to ASP, where the provider hosts the customer's software and delivers it to approved end users over the internet. In the software on demand SaaS model, the provider gives customers network-based access to a single copy of an application that the provider created specifically for SaaS distribution. The application's

source code is the same for all customers and when new features or functionalities are rolled out, they are rolled out to all customers. Depending upon the service level agreement (SLA), the customer's data for each model may be stored locally, in the cloud or both locally and in the cloud. Organizations can integrate SaaS applications with other software using application programming interfaces (APIs). For example, a business can write its own software tools and use the SaaS provider's APIs to integrate those tools with the SaaS offering. There are SaaS applications for fundamental business technologies, such as email, sales management, customer relationship management (CRM), financial management, human resource management (HRM), billing and collaboration. Leading SaaS providers include Salesforce, Oracle, SAP, Intuit and Microsoft.

S aaS applications are used by a range of IT professionals and business users, as well as C-level executives.

Advantages

SaaS removes the need for organizations to install and run applications on their own computers or in their own data centers. This eliminates the expense of hardware acquisition, provisioning and maintenance, as well as software licensing, installation and support. Other benefits of the SaaS model include:

1. **Flexible payments**: Rather than purchasing software to install, or additional hardware to support it, customers subscribe to a SaaS offering. Generally, they pay for this service on a monthly basis using a pay-as-you-go model. Transitioning costs to a recurring operating expense allows many businesses to exercise better and more predictable budgeting. Users can also terminate SaaS offerings at any time to stop those recurring costs.

2. **Scalable usage**: Cloud services like SaaS offer high vertical scalability, which gives customers the option to access more, or fewer, services or features on-demand.

3. **Automatic updates:** Rather than purchasing new software, customers can rely on a SaaS provider to automatically perform updates and patch management. This further reduces the burden on in-house IT staff.

4. **Accessibility and persistence:** Since SaaS applications are delivered over the Internet, users can access them from any Internet-enabled device and location.

Disadvantages

But SaaS also poses some potential disadvantages. Businesses must rely on outside vendors to provide the software, keep that software up and running, track and report accurate billing and facilitate a secure environment for the business' data. Providers that experience service disruptions, impose unwanted changes to service offerings, experience a security breach or any other issue can have a profound effect on the customers' ability to use those SaaS offerings. As a result, users should understand their SaaS provider's service-level agreement, and make sure it is enforced.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX