OBJECT ORIENTED

SOFTWARE ENGINEERING

FOR THE STUDENTS OF BACHELOR IN COMPUTER ENGINEERING

Compiled by: Er. Shiva Ram Dam

TABLE OF CONTENTS

Object Oriented Software Engineering

Chapter 1

INTRODUCTION

Er. Shiva Ram Dam

Shivaram.dam@pec.edu.np

1. INTRODUCTION

1.1 Software and Programs

Software is:

- A set of instruction or computer program that when executed provide desired functions and performance.
- Data structure that enable the programs to adequately manipulate information.
- Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

In broad sense, Software is not just the programs but also all associated documentation and configuration data which is needed to make these programs operate correctly.

A software usually consists of a number of separate programs, configuration files which are to set up these programs, system documentation which explains how to use the system, and for software products and websites for users to download recent product information.

In conclusion, we can define software as a **collection of programs, configuration files, documentation, user manual, update facilities and support.** It is the product that software professionals build and then support over the long term.

Characteristics of Software:

a) Software is developed or engineered, but not manufactured.

Software is a design of strategies, instruction which finally perform the designed, instructed tasks. And a design can only be developed, not manufactured.

b) Software does not "wear out"

Software is not susceptible to the environmental melodies and it does not suffer from any effects with time.

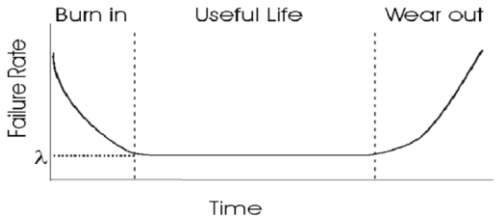


Fig 1: Hardware Reliability Curve

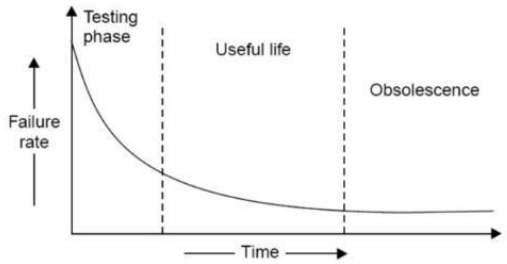


Fig 2: Software Reliability Curve

c) **Most softwares are custom-built**, rather than being assembled from existing components.

Programs VS Software

Program	Software
1. Usually small in size.	1. Large
2. Author himself is sole user.	2. Large number of users
3. Single developer.	3. Team of developers.
4. Lacks proper documentation.	4. Well documented and user manual prepared.
5. Adhoc (i.e. not systematic / unplanned) development.	5. Systematic development.
6. Database not part of program.	6. Database and program are parts of software.
7. Programs are instructions that when executed provide desired feature and functions.	7. Software is the collection of data structures, programs and other documentation to enable manipulation of data.

1.2 Software Engineering

Software Engineering is the field of Computer science that deals with building of software systems, which are so large and complex and are built by a team(s) and usually exist in multiple versions.

The IEEE defines software engineering as:

The study of the approaches and application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Software Engineering Layers:

Software Engineering is viewed as a layered technology which rests on an organizational commitment to quality. Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently in real machines.

Fig. Software Engineering layers

1) **A quality Focus:**
   - Main principle of Software Engineering is Quality Focus.
   - An engineering approach must have a focus on quality.
   - Total Quality Management (TQM), Six Sigma, ISO 9001, ISO 9000-3, CAPABILITY MATURITY MODEL (CMM), CMMI & similar approaches encourages a continuous process improvement culture
2) **Process:**
   - It is a foundation of Software Engineering
   - It is the glue the holds the technology layers
   - It defines a framework with activities for effective delivery of software engineering technology
3) **Methods:**
   - It provides technical how-to's for building software
   - It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support
4) **Tools:**
   - Software Engineering Tools allows automation of activities which helps to perform systematic activities. A system for the support of software development, called computer-aided software engineering (CASE). Examples: Testing Tools, Bug/Issue Tracking Tools etc…
   - Computer-aided software engineering (CASE) is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products
   - CASE tools automate many of the activities involved in various life cycle phases

Role of Software Engineering in System Design:

Software is what enables us to use computer hardware effectively and is needed for our modern life. A software system is often a component of a much larger system. The software engineering activity is therefore a part of a much larger system design activity in which the requirements of the software are balanced against the requirements of other parts of system being designed.

For an example, a telephone switching system consists of computers, telephone lines and cables, telephones, satellites and finally software to control the various other components. It is the combination of all these components that is expected to meet the requirements of the whole system.

Power plant or traffic monitoring system, banking system, hospital administration system are other examples of systems that exhibit the need to view the software as a component of a larger system.

In today's world, softwares are being increasingly embedded in various systems, from television set to airplanes. An dealing with such system requires a software engineer to know about the requirements of these systems and their operational process.

With the development of technology, requirements are increasing and is becoming more and more complex. To deal with the increasing complex nature of software SE plays an important role. Some of the major roles of SE are noted below:

- SE researches, designs and develops software systems to meet with client requirements. Once the system has been fully designed, Software engineers then test, debug and maintain the system.
- SE translates vague (unclear) requirements and derives into precise specification.
- SE develops model for the system or application and understands the behavior and performance of the system.
- SE provides methods to schedule work, operate systems at various levels and obtain the necessary details of the software.
- SE promotes interpersonal and communication skills and management skills.

1.3 Types of Software

Classification on the basis of use of software:

1. System Software
2. Application Software
3. Utility software

Classification on the basis of use of software:

1. Generic software
2. Tailored software

Software Application Domains:

Today seven broad categories of computer software present continuing challenges for software engineers:

1. **System software:**

   These are programs written to serve other programs. Some system softwares (e.g. compilers, editors, file management utilities) processes complex but determinate information structures. Other system applications (e.g. OS, drivers, Networking softwares, telecommunication softwares) process largely indeterminate data.

2. **Application software**

   These are stand-alone programs that solve a specific business need (e.g. business operations or management, technical decision making). In addition to conventional data processing applications, application software is used to control business functions in a real time (e.g. Point-of-Sale transaction processing, Real-time manufacturing process control).

3. **Engineering/Scientific software**

   These are also called “number crunching” algorithm. Such application are used in astronomy, volcanology, space shuttle orbital dynamics, automotive stress analysis, molecular biology,

automated manufacturing and so on. In modern engineering, CAD, system simulation and other interactive applications are emerging in real-time systems,

4. **Embedded software**

   These are small software that resides inside a product (or system) that are used to implement and control features and functions for the end users. For e.g. keypad control for microwave oven, digital function in an automobile such a s fuel control, dashboard display and braking systems.

5. **Product-line software**

   They provide a specific capability for use by many different customers. For e.g. word processing, spreadsheets, computer graphics, multimedia, database management, etc.

6. **Web applications**

   These are network-centric software category. Web apps are evolving into sophisticated computing environments to integrate with corporate databases and business applications.

7. **Artificial Intelligence software**

   Applications within this area include robotics, Expert systems, pattern recognition, artificial neural networks, theorem proving and game playing.

1.4 Brief history of Software Engineering

With the advent of computer systems in 1940s and as more and more system evolved, the concept of software engineering also evolved. The term "Software Engineering" was first coined in a NATO report in 1968. It discussed about the group work for software engineering in more systematic way.

In the 1950s, software development was less emphasized. Software development was assigned to third party.

Programmers moved for machine level to high level. To preserve the privacy, hardware manufacturers started to develop their own software themselves.

In 1980s, the software cost of a system had risen to 80% and many experts pronounced the field "in crisis" because the software development face the problem of not been delivered on time, over budgeted, unmaintainable due to poor design and documentation, and unreliable due to poor system analysis and testing.

It was required to apply engineering approach for management, team organization, tools, theories, methodologies and techniques which finally gave rise to "Software Engineering".

1.5 Software Process and Framework

Software Process:

When you walk to build a product or system, it is important to go through a series of predictable steps: a road map that helps you to create a timely, high quality result. The road map that you follow is called a "Software Process". These may involve the development of software from the scratch although it is

increasingly the case that new software is developed by extending and modifying existing systems. Thus, a *software process* is a collection of activities, actions, and tasks that are performed when some software product is to be created.

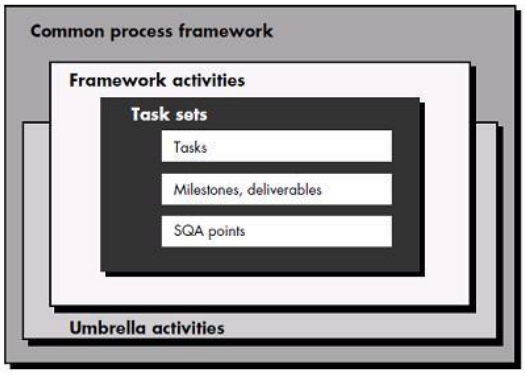
In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

Although there are many different software processes, a generic process framework for software engineering encompasses five activities:

a) Communication	Understand objectives of the project, define software features and functions.
b) Planning	Initial study of: technical tasks to be conducted, risk that are likely, work schedule, etc.
c) Modeling	Create a sketch or models using Use-Case diagrams, DFD, ERD, etc.
d) Construction	Actual coding, testing, debugging and integration
e) Deployment	Delivery to customer and get feedback

Software Process Framework:

Framework is a Standard way to build and deploy applications. **Software Process Framework** is a foundation of complete software engineering process. Software process framework includes all set of umbrella activities. It also includes number of framework activities that are applicable to all software projects.



1. INTRODUCTION

A generic process framework encompasses five activities which are given below one by one:

1. **Communication:**
   In this activity, heavy communication with customers and other stakeholders, requirement gathering is done.
2. **Planning:**
   In this activity, we discuss the technical related tasks, work schedule, risks, required resources etc.
3. **Modeling:**
   Modelling is about building representations of things in the 'real world'. In modelling activity, a product's model is created in order to better understanding and requirements.
4. **Construction:**
   In software engineering, construction is the application of set of procedures that are needed to assemble the product. In this activity, we generate the code and test the product in order to make better product.
5. **Deployment:**
   In this activity, a complete or non-complete products or software are represented to the customers to evaluate and give feedback. on the basis of their feedback we modify the products for supply better product.

Umbrella activities

Umbrella Activities are those activities to be performed through the entire Software Process. Umbrella activities are a set of steps or procedure that the software engineering team follows to maintain the progress, quality, change and risks of the overall development tasks. These steps of umbrella activities will evolve through the phases of generic view of software development.

Software engineering process framework activities are complemented by a number of umbrella activities.

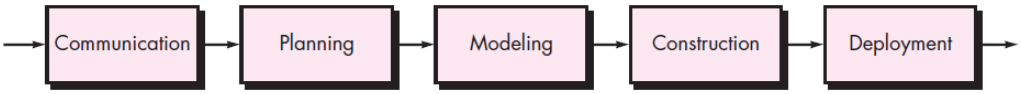
1. **Software project tracking and control:** allows the team to assess progress against the project plan and take necessary action to maintain schedule.
2. **Risk Management:** Assesses the risks that may affect the outcome of the project or the quality. Software quality assurance: defines and conducts the activities required to ensure software quality.
3. **Formal Technical Review:** meeting conducted by technical staff to uncover and remove errors before they propagate to the next action.
4. **Measurement:** defines and collects process, project, and product measures that assist the team in delivering software that meets customers' needs.
5. **Software configuration management:** Manages the effect of change throughout the software process
6. **Re-usability management:** defines criteria for work product reuse.
7. **Work product preparation and production:** encompasses the activities required to create work products such as models, documents, etc.

Software Process Flow:

Process flow describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

a) Linear Process Flow

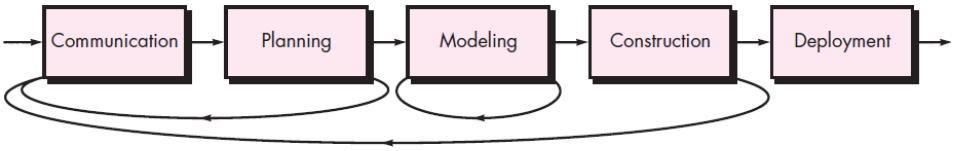
A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.



(a) Linear process flow

b) Iterative Process Flow

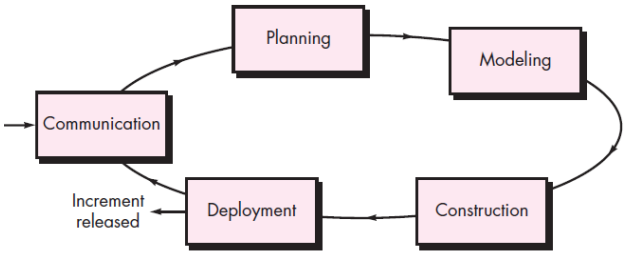
An iterative process flow repeats one or more of the activities before proceeding to the next.



(b) Iterative process flow

c) Evolutionary Process Flow

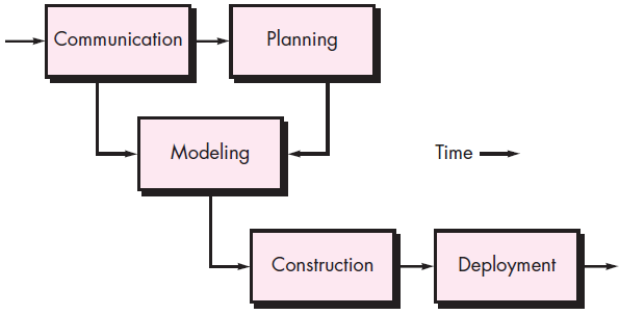
An evolutionary process flow executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software.



(c) Evolutionary process flow

d) Parallel Process Flow

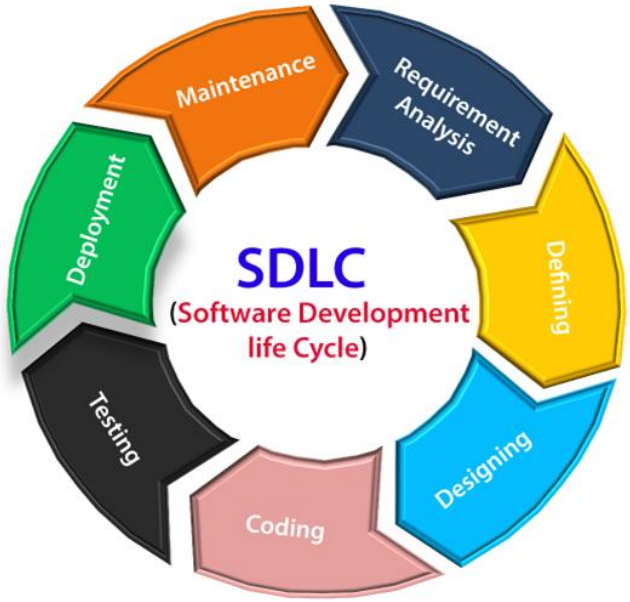
A parallel process flow executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).



(d) Parallel process flow

1.6 Software Development Life Cycle (SDLC)

SDLC Cycle represents the process of developing software. SDLC framework includes the following steps:



Stage 1: Planning and requirement analysis

Requirement Analysis is the most important and necessary stage in SDLC. The senior members of the team perform it with inputs from all the stakeholders and domain experts or SMEs in the industry.

Planning for the quality assurance requirements and identifications of the risks associated with the projects is also done at this stage.

Business analyst and Project organizer set up a meeting with the client to gather all the data like what the customer wants to build, who will be the end user, what is the objective of the product. Before creating a product, a core understanding or knowledge of the product is very necessary.

For Example, A client wants to have an application which concerns money transactions. In this method, the requirement has to be precise like what kind of operations will be done, how it will be done, in which currency it will be done, etc.

Once the required function is done, an analysis is complete with auditing the feasibility of the growth of a product. In case of any ambiguity, a signal is set up for further discussion.

Once the requirement is understood, the SRS (Software Requirement Specification) document is created. The developers should thoroughly follow this document and also should be reviewed by the customer for future reference.

Stage 2: Defining Requirements

Once the requirement analysis is done, the next stage is to certainly represent and document the software requirements and get them accepted from the project stakeholders.

This is accomplished through "SRS"- Software Requirement Specification document which contains all the product requirements to be constructed and developed during the project life cycle.

Stage 3: Designing the Software

The next phase is about to bring down all the knowledge of requirements, analysis, and design of the software project. This phase is the product of the last two, like inputs from the customer and requirement gathering.

Stage 4: Developing the project

In this phase of SDLC, the actual development begins, and the programming is built. The implementation of design begins concerning writing code. Developers have to follow the coding guidelines described by their management and programming tools like compilers, interpreters, debuggers, etc. are used to develop and implement the code.

Stage 5: Testing

After the code is generated, it is tested against the requirements to make sure that the products are solving the needs addressed and gathered during the requirements stage.

During this stage, unit testing, integration testing, system testing, acceptance testing are done.

Stage 6: Deployment

Once the software is certified, and no bugs or errors are stated, then it is deployed.

Then based on the assessment, the software may be released as it is or with suggested enhancement in the object segment.

After the software is deployed, then its maintenance begins.

Stage 7: Maintenance

Once when the client starts using the developed systems, then the real issues come up and requirements to be solved from time to time. This procedure where the care is taken for the developed product is known as maintenance.

1.7 Perspective Process Models

The SDLC illustrates us the general prescriptive process model for the development of software product. Prescriptive (Decided) or traditional process models were originally proposed to bring order to the vague of software development. They are called “Prescriptive” because they prescribe a set of process elements such as framework activities, software engineering actions, tasks, and work products, quality assurance, and change control mechanism for each project. History indicates that these traditional models have brought a certain amount of useful structure to software engineering work & have provided a reasonably effective road map for software team.

Some of the perspective process models are :

- Linear Sequential Model
- Prototyping Model
- RAD Model
- Evolutionary Model
- Incremental Model
- Spiral Model
- WIN-WIN Spiral Model
- Concurrent Development Model
- Component Based Development
- Formal Methods Models
- Fourth Generation Technology Models

1.7.1 Waterfall Model

- This model was first introduced by Dr. Winston W. Royce in a paper published in 1970.
- The waterfall model is a classical model used in system development life cycle to create a system with a linear and sequential approach.
- It is termed as waterfall because the model develops systematically from one phase to another in a downward fashion.
- This model is divided into different phases and the output of one phase is used as the input of the next phase.
- Every phase has to be completed before the next phase starts and there is no overlapping of the phases.

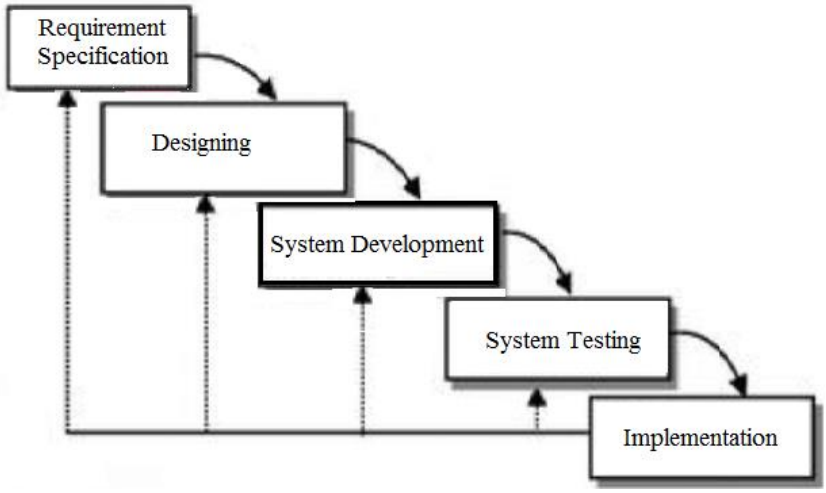


Figure: The Waterfall Model

1. **Requirement Analysis and Specification**

   This phase identifies and documents the exact requirements of the system through feasibility study. This is done by combination of customers, developer and organization.

2. **Design**

   From the specified requirement, software engineers develop a design. It can be split into two phases: High Level Design and Detailed Design.

   The High-level design deals with the overall module structure and organization. Then, it is refined by designing each module (i.e. detailed design). Screen layout, business rules, process diagrams, DFD, ERD, etc. are used in this phase.

3. **System Development**

   - Produces the actual code that will be developed to the customers.
   - Also develops prototypes, modules, test drivers, etc.
   - Testing is done at program modules and at various levels.
   - Includes unit testing, black-box testing, white-box testing, etc.

4. **System Testing**

   All modules are tested individually in the previous phase are then integrated to make a complete system, and performance test is done on the whole system.

5. **Implementation**

   Once the system phases all the tests, it is delivered to the customers and enters the maintenance phase. Any modifications made to the system, after initial delivery, are usually attributed to this phase.

Advantages:

- Easy to understand and implement
- Suitable for simple project
- Well understood milestones.
- Process and results are well documented.

Disadvantages:

- Difficult to trace back
- No working software is produced until late during the life cycle.
- Not suitable for large product
- Risk analysis is not performed.
- Cannot accommodate changing requirements.
- Product may be outdated at the end.

When to use it?

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

Major Drawback:

The major drawback of the Waterfall model is we move to the next stage only when the previous one is finished and there is no chance to go back if something is found wrong in later stages.

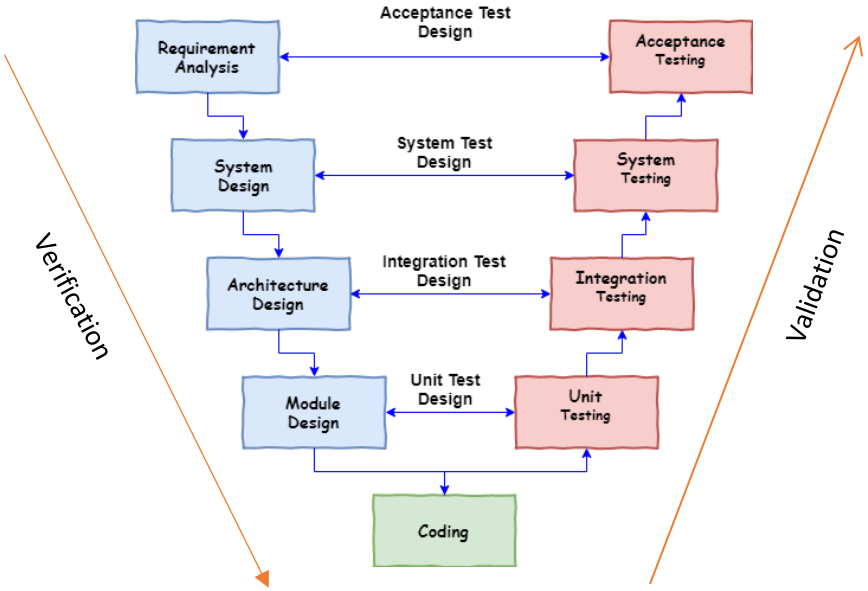
More Reference:

https://www.toolsqa.com/software-testing/waterfall-model/

1.7.2 The V Model

The major drawback of the Waterfall model is we move to the next stage only when the previous one is finished and there is no chance to go back if something is found wrong in later stages. V-model provides means of testing of software at each stage in reverse manner.

The V-Model is SDLC model where execution of processes happens in a sequential manner in V-shape. It is also known as Verification and Validation Model. V-Model is an extension of the Waterfall Model and is based on association of a testing phase for each corresponding development stage. This means that for every single phase in the development cycle there is a directly associated testing phase. This is a highly disciplined model and next phase starts only after completion of the previous phase.



The left-hand side shows the development activities and the right-hand side shows the testing activities. In the development phase, both verification and validation are performed along with the actual development activities. Verification and Validation phases are joined by coding phase in V-shape. Thus it is called V-Model. This demonstrates that testing can be done in all phase of development activities as well.

Design Phases:

1. ***Business Requirement Analysis:*** In this first phase, the product requirements are understood from the customer perspective. The users are interviewed and a document called the *User Requirements Document* is generated. The user requirements document will typically describe the system's functional, interface, performance, data, security and other requirements as expected by the user. The user's carefully review this document as this document would serve as the guideline for the system designers in the system design phase.
2. ***System Design:*** In the phase, system developers analyze and understand the business of the proposed system by studying the user requirements document. They figure out possibilities and techniques by which the user requirements can be implemented. System design comprises of understanding and detailing the complete hardware and communication setup for the product under development. System test plan is developed based on the system design.
3. ***Architecture Design:*** This is also referred to as *High Level Design (HLD)***.** This phase focuses on system architecture and design. It provides overview of solution, platform, system, product and

service/process. Usually more than one technical approach is proposed and based on the technical and financial feasibility the final decision is taken. An integration test plan is created in order to test the pieces of the software systems ability to work together.

4. ***Module Design:*** The module design phase can also be referred to as low-level design. In this phase the actual software components are designed. It defines the actual logic for each and every component of the system. The designed system is broken up into smaller units or modules and each of them is explained so that the programmer can start coding directly. It is important that the design is compatible with the other modules in the system architecture and the other external systems.

Validation Phases:

In the V-Model, each stage of verification phase has a corresponding stage in the validation phase. The following are the typical phases of validation in the V-Model.

1. ***Unit Testing:*** Unit tests designed in the module design phase are executed on the code during this validation phase. Unit testing is the testing at code level and helps eliminate bugs at an early stage. A unit is the smallest entity which can independently exist, e.g. a program module. Unit testing verifies that the smallest entity can function correctly when isolated from the rest of the codes/units.
2. ***Integration Testing:*** Integration testing is associated with the architectural design phase. These tests verify that units created and tested independently can coexist and communicate among themselves within the system.
3. ***System Testing:*** System Tests Plans are developed during System Design Phase. System Test Plans are composed by client's business team. System Test ensures that expectations from application developed are met. The whole application is tested for its functionality, interdependency and communication. User acceptance testing: Acceptance testing is associated with the business requirement analysis phase and involves testing the product in user environment. UAT verifies that delivered system meets user's requirement and system is ready for use in real time.
4. ***User Acceptance Testing (UAT):*** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user's requirement and system is ready for use in real world

When to use?

- Where requirements are clearly defined and fixed.
- The V-Model is used when ample technical resources are available with technical expertise.
- Project is short.

Advantages:

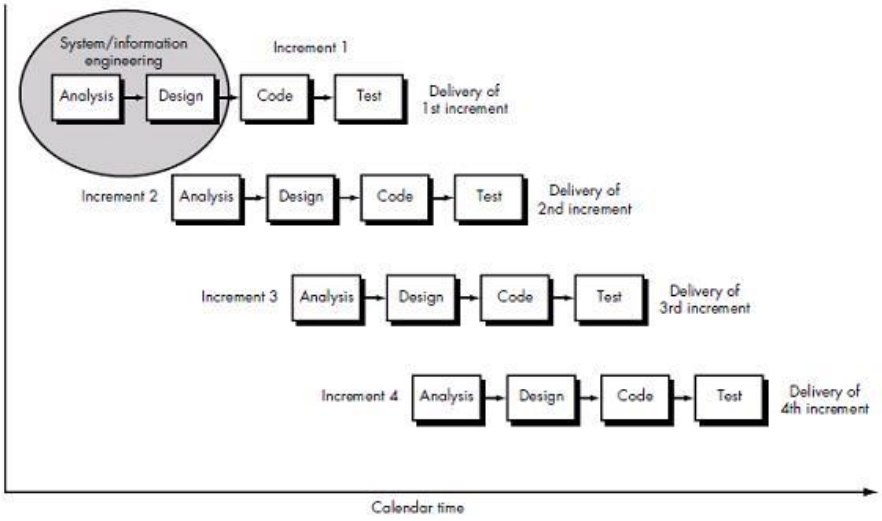
- This is a highly disciplined model and Phases are completed one at a time.
- V-Model is used for small projects where project requirements are clear.

- Simple and easy to understand and use.
- This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
- It enables project management to track progress accurately.

Disadvantages:

- High risk and uncertainty.
- It is not a good for complex and object-oriented projects.
- It is not suitable for projects where requirements are not clear and contains high risk of changing.
- This model does not support iteration of phases.
- It does not easily handle concurrent events.

1.7.3 Incremental Model



The Incremental Model is a method of software development where the product is designed, implemented and tested incrementally. Little more is added each time until the product is finished. It involves both development and maintenance. This is also known as the Iterative Model.

The basic idea of this model is that the software should be developed in increments, where each increment adds functional capability to the system until the full system is implemented.

The product is decomposed into a number of components, each of which is designed and built separately. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed modules. Each module passes through the requirements, design, implementation and testing phases. The first module is often a core product where the basic requirements are addressed, and supplementary features are added in the next increments. Once the core

product is analyzed by the client, there is plan development for the next increment. Each subsequent release of the module adds function to the previous release. The process continues until the complete system is achieved.

For example: word processing software developed using this model might deliver:

- file management, editing and printing in the first increment,
- most sophisticated editing and document production capabilities in the second increment,
- Spelling and grammar checking in the third increment and so on.

Advantages:

- Incremental Model allows partial utilization of the product and avoids a long development time.
- Supports changing environment
- Generates working software quickly and early during the software life cycle.
- This model is more flexible and less costly to change scope and requirements.
- It is easier to test and debug as smaller changes are made during each iteration.
- In this model, the customer can respond to each built. During the life cycle, software is produced early which facilitates customer evaluation and feedback
- As testing is done after each iteration, faulty elements of the software can be quickly identified because few changes are made within any single iteration.

Disadvantages:

- As additional functional is added to the product at every stage, problems may arise related to system architecture which was not evident in earlier stages.
- It needs good planning and design at every step, more management attention is required.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall, more resources may be required.
- End of project may not be known, which is a risk.

When to use Iterative Enhancement model?

The below are the cases when we need to use Iterative Enhancement Model:

- Some functionalities or requested enhancements may evolve with time.
- There is a time to the market constraint.
- New technology is being adapted.
- There are some high-risk features and goals which may change in the future.

1.7.4 Evolutionary Model

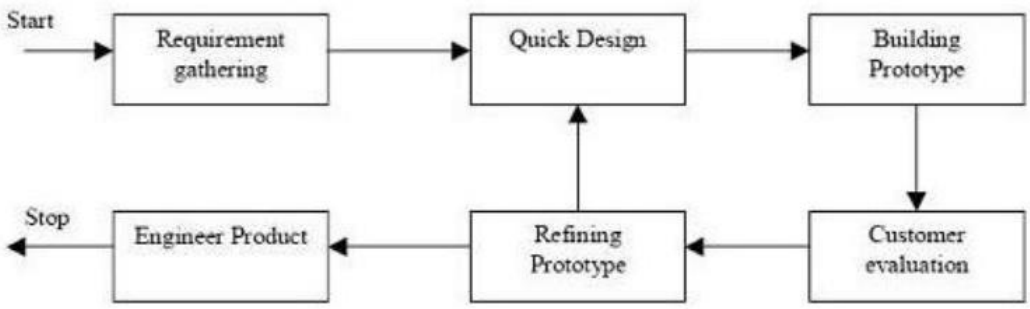
Evolutionary models are iterative which is much suitable for new systems where no clear idea of the requirements, inputs and outputs parameters. It produces an increasingly more complete version of the software with each iteration. The evolutionary process models are appropriate in following situation:

- No clear idea about the product for both customers and developers.
- Good communication between the customers and developers
- Sufficient time for the software development.
- Less chance of outsourcing and availability of re-usable components.

The two common evolutionary process models are:

1. Prototyping model
2. Spiral Model

1. Prototype model



Prototyping Model

- In this model, a prototype (an early approximation of a final system or product) is built, tested, and then reworked as necessary until an acceptable prototype is finally achieved. Then, the complete system is developed.
- The analyst works with users to determine the initial or basic requirements for the system. The analyst then quickly builds a prototype and gives it to the user. The analyst uses the feedback to improve the prototype and takes the newer version back to users. The process is repeated until the user is relatively satisfied.

Advantages:

- Users are actively involved in the development.
- Users get better understanding of the system being developed.
- Errors can be detected much earlier.
- Missing functionality can be identified quickly.
- Reduces project risk and less documentation.

Disadvantages:

- May increase the complexity of the system since the user may demand more functionality,
- Integration can be very difficult.
- Frequent improvement and rebuilding of software.
- Slower process.

When to use Prototyping model?

- When requirement is not clear.
- Useful in development of systems having high level of user interactions such as online systems.

2. Spiral Model

This Spiral model, proposed by Barry Boehm (1985), is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each iteration around the spiral.

The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in each iteration/ spiral.

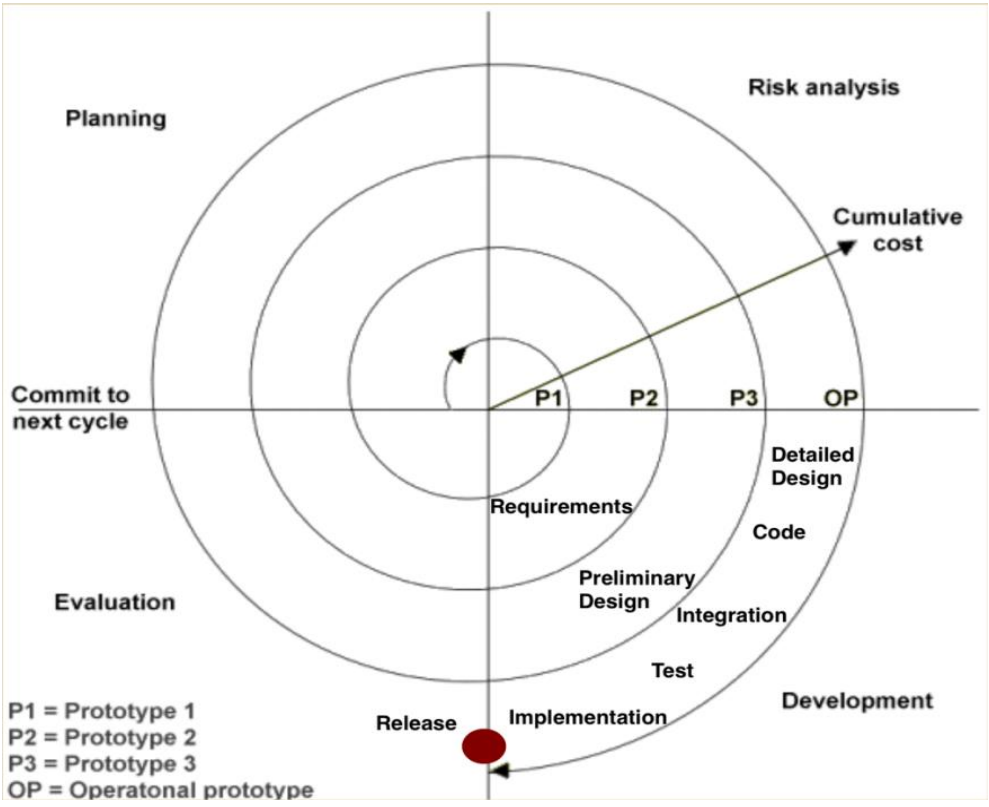


Figure: The Spiral Model

i) Planning:

This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.

ii) Risk Analysis:

Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. A prototype is produced at the end of the risk analysis phase. If any risk is found during the risk analysis then alternate solutions are suggested and implemented.

iii) Software Engineering / Development

This phase refers to production of the actual software product at every spiral. Actual development and testing of the software takes place in this phase. It includes testing, coding and deploying software at the customer site. A POC (Proof of Concept) is developed in this phase to get customer feedback. In this phase software is developed, along with testing at the end of the phase. Hence in this phase the development and testing is done.

iv) Evaluation:

This phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

Advantages:

- Risk monitoring is one of the core parts which makes it pretty attractive, especially when you manage large and expensive projects. Moreover, such approach makes your project more transparent because, by design, each spiral must be reviewed and analyzed
- Customer can see the working product at the early stages of software development lifecycle
- Different changes can be added at the late life cycle stages
- Project can be separated into several parts, and riskier of them can be developed earlier which decreases management difficulties
- Project estimates in terms of schedule, costs become more and more realistic as the project moves forward, and loops in spiral get completed
- Strong documentation control

Disadvantages:

- Since risk monitoring requires additional resources, this model can be pretty costly to use. Each spiral requires specific expertise, which makes the management process more complex. That's why this SDLC model is not suitable for small projects
- A large number of intermediate stages. As a result, a vast amount of documentation

- Time management may be difficult. Usually, the end date of a project is not known at the first stages

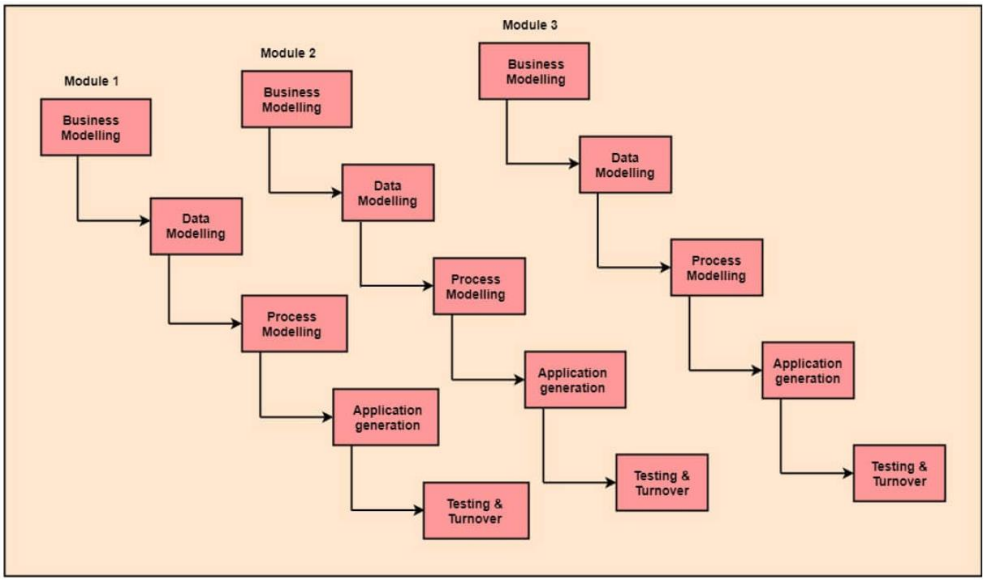
When to use Spiral model:

Spiral Model can be used in the below scenario:

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- Significant changes are expected (research and exploration)

1.7.5 The RAD Model

RAD or Rapid Application Development process is an adoption of the waterfall model; it targets at developing software in a short span of time. RAD follow the iterative approach. In RAD model the components or functions are developed in parallel as if they were mini projects. The developments are time boxed, delivered and then assembled into a working prototype. This can quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements. Development of each module involves the various basic steps as in waterfall model i.e. analyzing, designing, coding and then testing.



The various phases of RAD are as follows:

1. **Business Modelling:**

The information flow among business functions is defined by answering questions like what data drives the business process, what data is generated, who generates it, where does the information go, who process it and so on.

2. **Data Modelling:**

   The data collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified, and the relation between these data objects (entities) is defined.

3. **Process Modelling:**

   The information object defined in the data modeling phase are transformed to achieve the data flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

4. **Application Generation:**

   Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.

5. **Testing & Turnover:**

   Many of the programming components have already been tested since RAD emphasis reuse. This reduces the overall testing time. But the new part must be tested, and all interfaces must be fully exercised.

Advantages –

- Use of reusable components helps to reduce the cycle time of the project.
- Feedback from the customer is available at initial stages.
- Reduced costs as fewer developers are required.
- Use of powerful development tools results in better quality products in comparatively shorter time spans.
- The progress and development of the project can be measured through the various stages.
- It is easier to accommodate changing requirements due to the short iteration time spans.

Disadvantages –

- The use of powerful and efficient tools requires highly skilled professionals.
- The absence of reusable components can lead to failure of the project.
- The team leader must work closely with the developers and customers to close the project in time.
- The systems which cannot be modularized suitably cannot use this model.
- Customer involvement is required throughout the life cycle.
- It is not meant for small scale projects as for such cases, the cost of using automated tools and techniques may exceed the entire budget of the project.

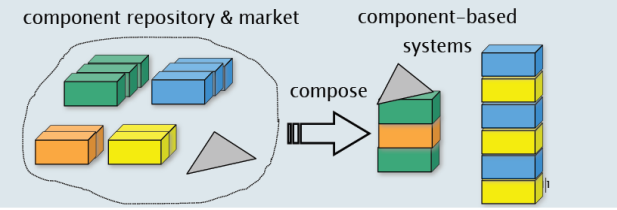
Applications –

- This model should be used for a system with known requirements and requiring short development time.
- It is also suitable for projects where requirements can be modularized and reusable components are also available for development.
- The model can also be used when already existing system components can be used in developing a new system with minimum changes.
- This model can only be used if the teams consist of domain experts. This is because relevant knowledge and ability to use powerful techniques is a necessity.
- The model should be chosen when the budget permits the use of automated tools and techniques required.

1.8 Specialized Process Models

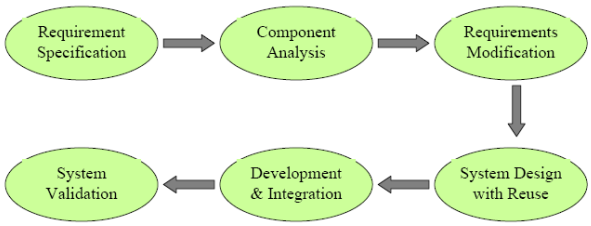
Specialized process model is applied when a specialized or narrowly defined software engineering approach is chosen. These models take many characteristics of one or more of the traditional models.

1.8.1 Component Based Development



The Component-based development (CBD) model incorporates many of the characteristics of Spiral model. CBD is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing. CBD techniques involve procedures for developing software systems by choosing ideal off-the-shelf components and then assembling them using a well-defined software architecture. With the systematic reuse of coarse-grained components, CBD intends to deliver better quality and output.

- Emphasizes the design and construction of computer-based systems using software "components".
- The process relies on reusable software components.
- Similar to the characteristics of the spiral model.



1. **Requirement Specification:**

   Requirement specification and validation steps are similar to the other processes.

2. **Component Analysis**

   During this stage try to find the software components need for the implementation once the requirements are specified.

3. **Requirements Modification**

   Analyze the discovered software components to find out whether it is able to achieve the specified requirements.

4. **System Design with Reuse**

   The frame work of the system is designed to get the maximum use of discovered components. New software may have to design if the reusable components are not available.

5. **Development and integration**

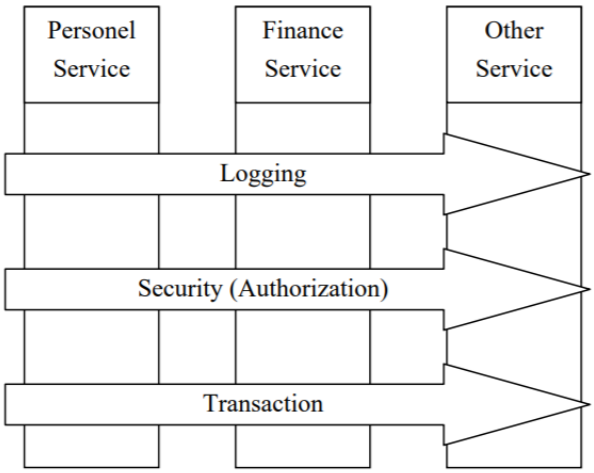
   Software that cannot be discovered is developed, and the reusable components are integrated to create the new system. The integration process, may be part of the development process rather than a separate activity.

Advantages of CBD:

1. Minimized SDLC time: Recycling of pre-fabricated components
2. Improved efficiency: Developers concentrate on application development
3. Improved quality: Component developers can permit additional time to ensure quality
4. Minimized expenditures

1.8.2 Aspect Oriented Software Development

Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects. The aspect is customer requirements solely defined for their software system. The aspects may be logging, security, transaction, memory management, integrity and so on. Components may provide or require one or more "aspect details" relating to a particular aspect.



AOSD aims to address crosscutting concerns by providing means for systematic identification, separation, representation and composition. Crosscutting concerns are encapsulated in separate modules, known as aspects, so that localization can be promoted. This results in better support for modularization hence reducing development, maintenance and evolution costs.

1.8.3 Formal Methods Model

The Formal Methods Model is an approach to Software Engineering that applies mathematical methods or techniques to the process of developing complex software systems. The approach uses a formal specification language to define each characteristic of the system. The language is very particular, and employs a unique syntax whose components includes objects, relations, and rules. When used together, these components can validate the correctness of each characteristic. Think of the process like balancing a series of equations. At each step in the series, if the right-side of the equation doesn't equal the left, there is a problem that must be addressed.

Formal methods translate software requirements into a more formal representation by applying the notation and heuristics of sets to define the data invariant, states, and operations for a system function. The formal methods used during the development process provide a mechanism for eliminating problems, which are difficult to overcome using other software process models. The software engineer creates formal specifications for this model. These methods minimize specification errors and this result in fewer errors when the user begins using the system.

Generally, the formal method comprises two approaches, namely, property based and model-based. The property-based specification describes the operations performed on the system. The model-based specification utilizes the tools of set theory, function theory, and logic to develop an abstract model of the system.

Advantages	Disadvantages
• Discovers ambiguity, incompleteness, and inconsistency in the software. • Offers defect-free software.	• Time consuming and expensive. • Difficult to use this model as a communication mechanism for non-technical personnel.

• Incrementally grows in effective solution after each iteration. • This model does not involve high complexity rate. • Formal specification language semantics verify self-consistency.	• Extensive training is required since only few developers have the essential knowledge to implement this model.

1.8.4 Cleanroom Engineering

Cleanroom Engineering is a software development philosophy that is based on avoiding software defects by using formal methods of development and a rigorous inspection. The objective of this approach to software development is zero-defective software.

Cleanroom software engineering makes use of a specialized version of the incremental software model. A "pipeline of software increments" is developed by small independent software teams. As each increment is certified, it is integrated into the whole. Hence, functionality of the system grows with time.

The clean-room approach to software development is based on 5 key strategies:

1. Formal Specification
2. Incremental development
3. Structural programming
4. Static Verification
5. Statistical testing of system

1. **Formal Specification**

   The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.

2. **Incremental development**

   The software is partitioned into increments which are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.

3. **Structural programming**

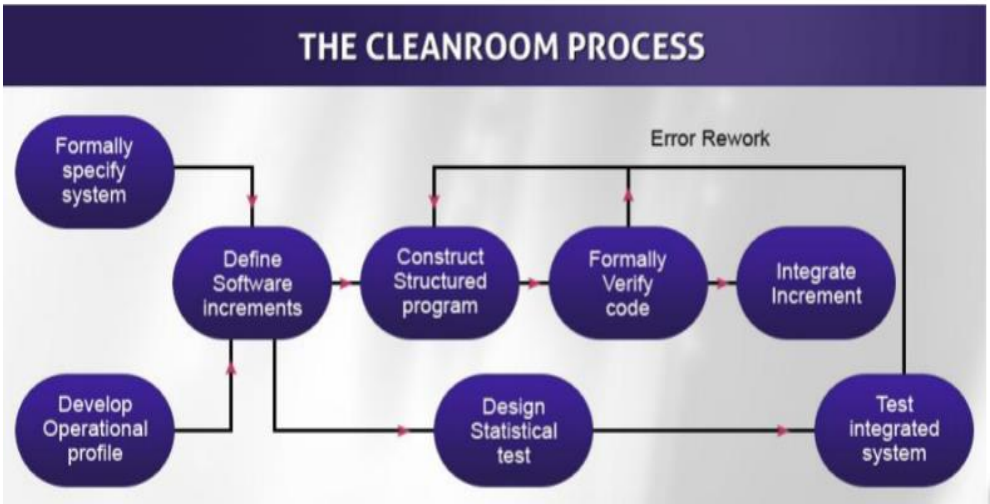
   A limited number of constructs are used and the aim is to apply correctness-preserving transformations to the specification to create the program code.

4. **Static Verification**

   The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.

5. **Statistical testing of system**

The integrated software increment is tested statistically, to determine its reliability. These statistical tests are based on an operational profile which is developed in parallel with the system specification.



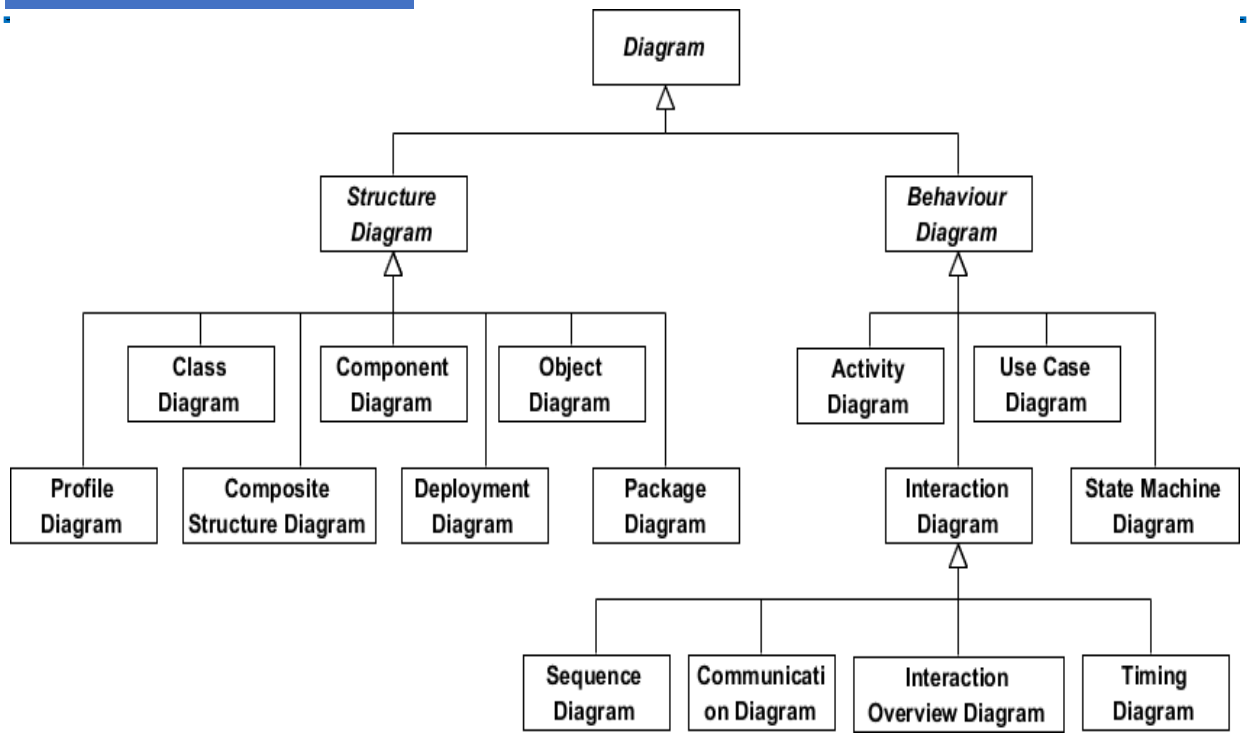
1.9 The Unified Process and UML

The Unified Modeling Language (UML):

UML is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object-oriented software and the software development process. The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

UML is not a programming language but tools that can be used to generate code in various languages using UML diagrams.

1. INTRODUCTION



Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/

Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other. Behavior diagrams show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

Unified Process Model:

Unified process (UP) is an architecture-centric, use-case driven, iterative and incremental development process that leverages unified modeling language and is compliant with the system process engineering metamodel. Unified process can be applied to different software systems with different levels of technical and managerial complexity across various domains and organizational cultures. UP is also referred to as the unified software development process.

The *phases* of unified process are similar to the generic framework activities as shown in figure:

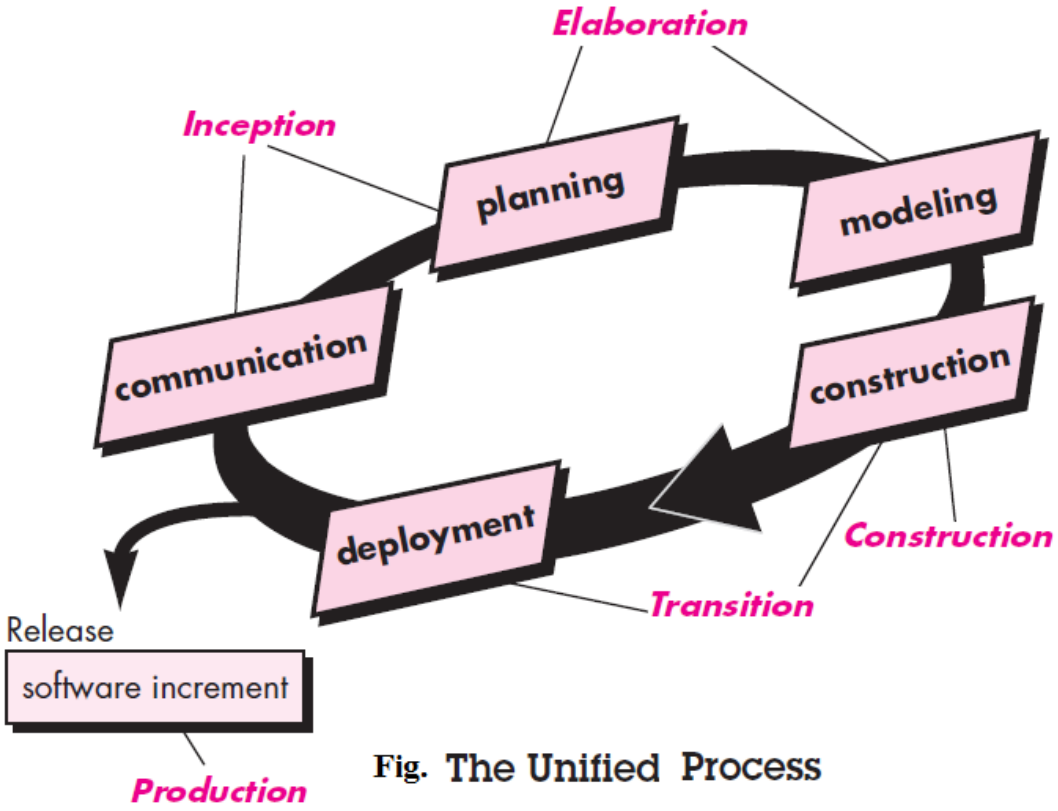


Fig. The Unified Process

This process divides the development process into four phases:

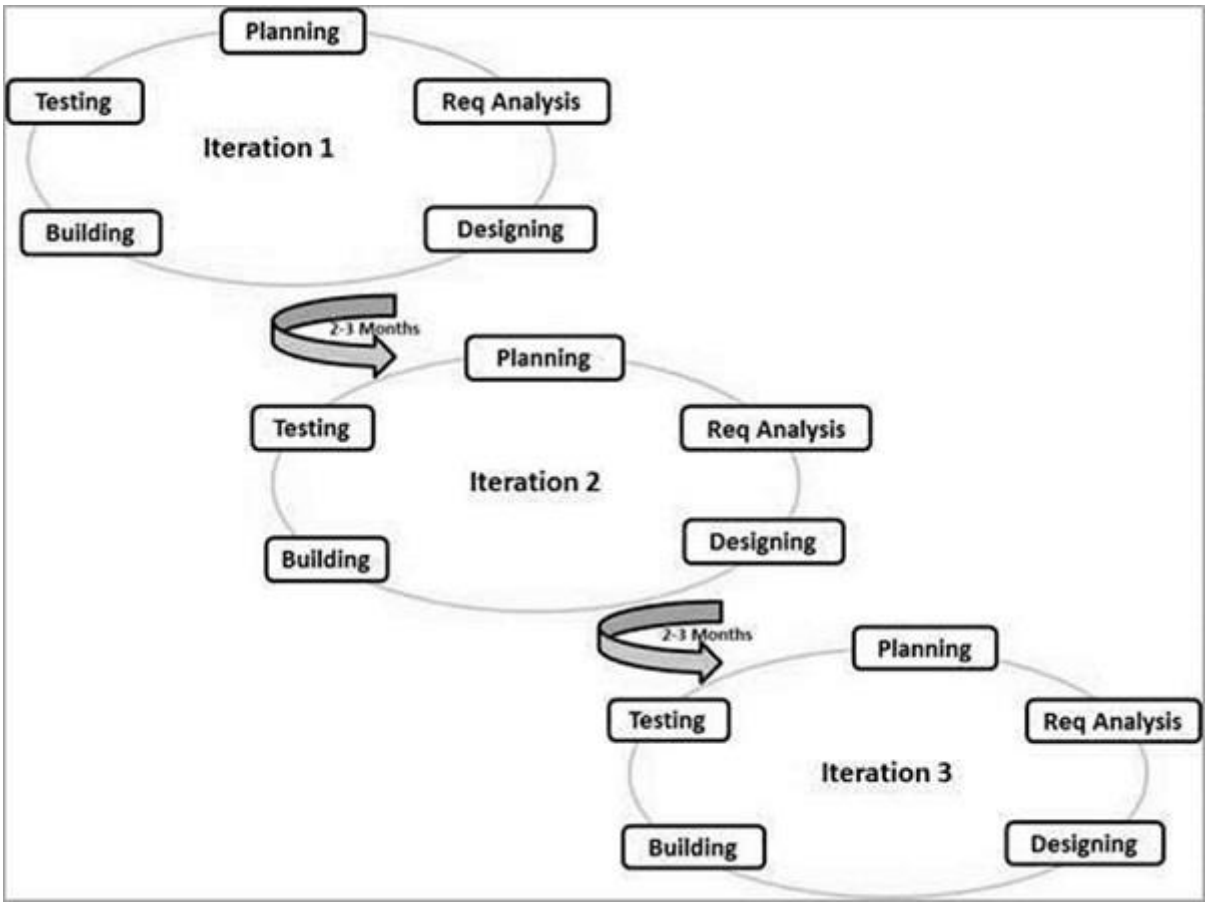
1. **Inception**: The inception phase is similar to the requirements collection and analysis stage of the waterfall model of software development. In this phase, requirements are collected from the customer and analyze the project's feasibility, its cost, risks, and profits.
2. **Elaboration**: In this phase, the activities undertaken in the inception phase are expanded. The major goals of this phase include creating fully functional requirements (use-cases) and creating a detailed architecture for fulfillment of the requirements.
3. **Construction**: In this phase, actual code is written and the features are implemented for each iteration. The first iteration of the software is rolled out depending on the key use-cases that make up the core functionalities of the software system.
4. **Transition**: In this phase, next iterations are rolled out to the customer and fix bugs for previous releases. Finally, builds of the software is deployed to the customer.

Agile Development

Agile software engineering combines a philosophy and a set of development guidelines to develop the software project as customer need. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design, and active and continuous communication between developers and customers. The change management is a primary goal of this philosophy. The changes may be from employ turnover, change of customer need, technology change, stakeholder decision change etc.

Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it. The basic framework activities—communication, planning, modeling, construction, and deployment—are remains the same for agile development but they transform into a minimal task set that pushes the project team toward construction and delivery.

In a nutshell, it is a process for managing a project characterized by constant iteration and collaboration in order to more fully answer a customer's needs. Below gives the graphical illustration of Agile Model:



Source: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm

1.9.1 Agility Principles

The Agile Alliance defines 12 agility principles for those who want to achieve agility:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Not every agile process model applies these 12 principles with equal weight, and some models choose to ignore (or at least downplay) the importance of one or more of the principles. However, the principles define an *agile spirit* that is maintained in each of the process models.

1.9.2 Agile Process

Agile is a process by which a team can manage a project by breaking it up into several stages and involving constant collaboration with stakeholders and continuous improvement and iteration at every stage. The Agile methodology begins with clients describing how the end product will be used and what problem it will solve. This clarifies the customer's expectations to the project team. Once the work begins, team cycle through a process of planning, executing, and evaluating — which might just change the final deliverable to fit the customer's needs better. Continuous collaboration is key, both among team members and with project stakeholders, to make fully-informed decisions.

The agile software development emphasizes on four core values.

1. Individual and team interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

1.9.3 Agile Models

Every organization is unique and faces different internal factors (i.e. organization size and stakeholders) and external factors (i.e. customers and regulations). To help meet the varying needs of different organizations, there are various agile methodologies. Some of the widely used Agile models are: Extreme Programming, Scrum, Kanban, Adaptive Software Development, Dynamic System Development Methods. Feature Driven Development, Lean Software Development.

i. Extreme Programming (XP)

Extreme programming (XP) is one of the most important software development framework of Agile models. It is used to improve software quality and responsive to customer requirements. The extreme programming model recommends taking the best practices that have worked well in the past in program development projects to extreme levels.

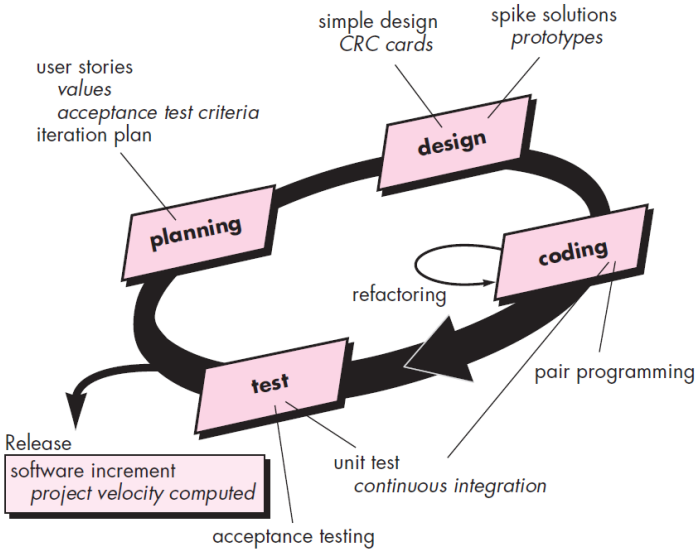


Fig. The Extreme Programming process

Basic principles of Extreme programming:

XP is based on the frequent iteration through which the developers implement User Stories. User stories are simple and informal statements of the customer about the functionalities needed. A User story is a conventional description by the user about a feature of the required system. It does not mention finer details such as the different scenarios that can occur.

On the basis of User stories, the project team proposes Metaphors. Metaphors are a common vision of how the system would work. The development team may decide to build a Spike for some feature. A Spike is a very simple program that is constructed to explore the suitability of a solution being proposed. It can be considered similar to a prototype.

A key concept of "Pair Programming" is done during the coding activity. Code review detects and corrects errors efficiently. XP recommends two people work together at one computer workstation to provide better solution for the problem. It suggests pair programming as coding and reviewing of written code carried out

by ~~a pair of programmers who switch their works between them~~ every hour. Refactoring is always considered. Refactoring is the technique of improving code without changing functionality.

XP model gives high importance on testing and considers it be the primary factor to develop a fault-free software. XP suggests test-driven development (TDD) to continually write and execute test cases. In the TDD approach test cases are written even before any code is written.

Incremental development is very good because customer feedback is gained and based on this development team come up with new increments every few days after each iteration.

ii. Adaptive Software Development

Adaptive Software Development (ASD) is a direct outgrowth of an earlier agile framework, Rapid Application Development (RAD). It aims to enable teams to quickly and effectively adapt to changing requirements or market needs by evolving their products with lightweight planning and continuous learning. The ASD approach encourages teams to develop according to a three-phase process: speculate, collaborate, learn.

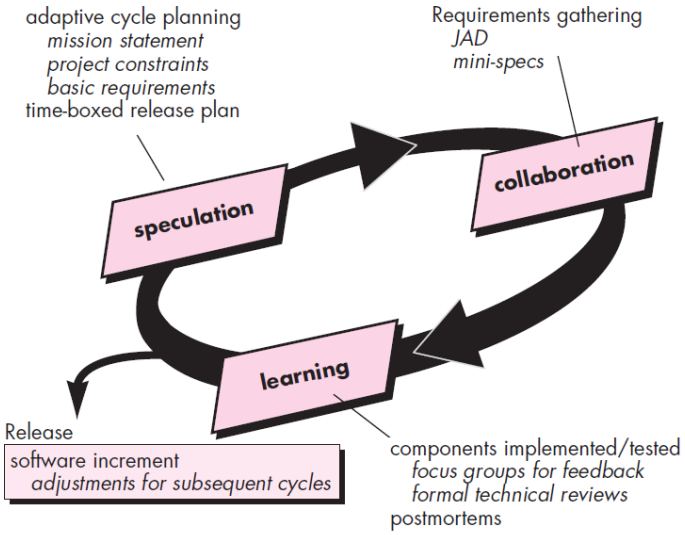


Fig. Adaptive software development

1. **Speculation:**
   During this phase project is initiated and planning is conducted. The project plan uses project initiation information like project requirements, user needs, customer mission statement etc, to define set of release cycles that the project wants.

2. **Collaboration:**
   It is the difficult part of ASD as it needs the workers to be motivated. It collaborates communication and teamwork but emphasizes individualism as individual creativity plays a major role in creative thinking. People working together must trust each others to criticize without animosity and assist without resentment.

3. **Learning:**
   The workers may have a overestimate of their own understanding of the technology which may not lead to the desired result. Learning helps the workers to increase their level of understanding over the project. Learning process is of 3 ways: Focus groups, Technical reviews, Project postmortem

iii. Scrum

Scrum is the type of Agile framework. It is a framework within which people can address complex adaptive problem while productivity and creativity of delivering product is at highest possible values. Scrum uses Iterative process.

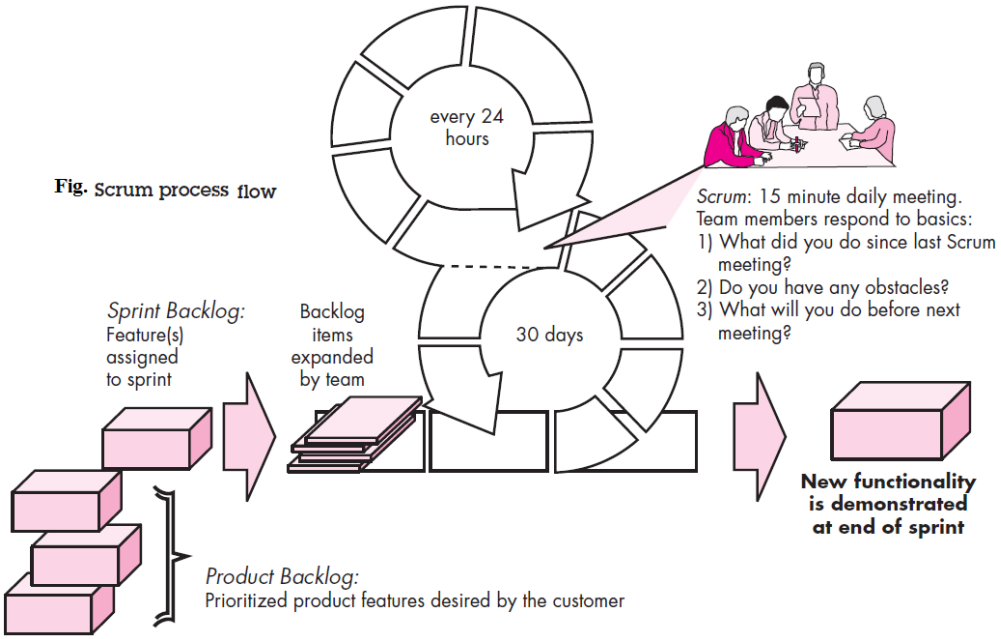


Fig. Scrum process flow

Fig: Scrum Life Cycle

Product Backlog:

It is the master list of work that needs to get done maintained by the product owner or product manager. Items can be added to the backlog at any time. The product backlog is constantly revisited, re-prioritized and maintained by the Product Owner.

Sprint Backlog:

It is the list of items, user stories, or bug fixes, selected by the development team for implementation in the current sprint cycle. Before each sprint, in the sprint planning meeting, the team chooses which items it will work on for the sprint from the product backlog.

Scrum Meetings:

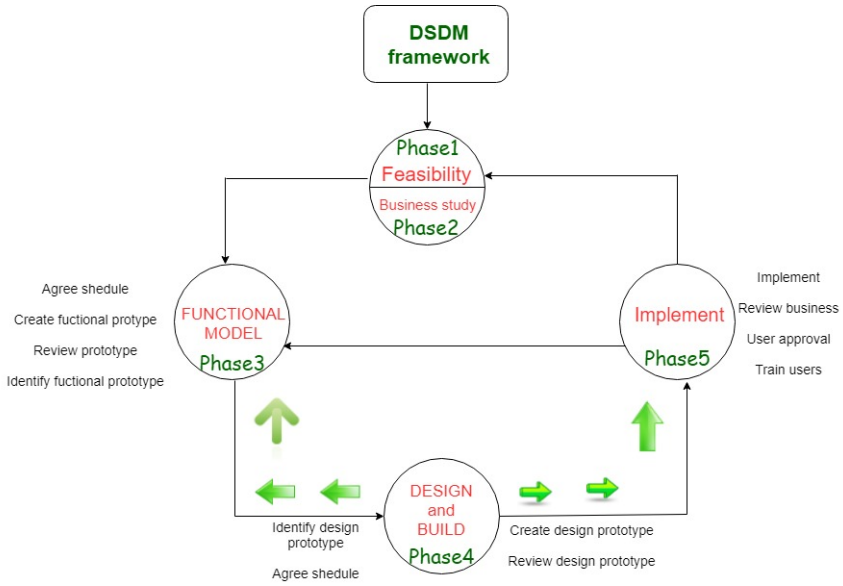
These are short (typically 15 minutes) meetings held daily by the Scrum team. A team leader, called a Scrum master, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure.

Demos:

Demos deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.

iv. Dynamic System Development Method (DSDM)

DSDM is an Agile method that focuses on the full project lifecycle. It is an iterative, incremental approach that is largely based on the Rapid Application Development (RAD) methodology. The method provides a five-phase framework consisting of:



Dynamic Systems Development Method life cycle

1. ***Feasibility study:***
   Feasibility study establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.
2. ***Business study:***
   Business study establishes the functional and information requirements that will allow the application to provide business value.
3. ***Functional model iteration:***
   produces a set of incremental prototypes that demonstrate functionality for the customer. The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.
4. ***Design and build iteration:***
   revisits prototypes built during *functional model iteration* to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.
5. ***Implementation:***
   provides the operational environment of the design model using any programming language.

v. Feature Driven Development

Feature Driven Development (FDD) is an iterative agile software development model. More specifically, FDD organizes workflow based on which features need to be developed next.

In the context of FDD, a feature "is a client-valued function that can be implemented in two weeks or less".

By releasing new features in an incremental fashion, developers are able to prioritize client requests, respond to requests in a timely manner and keep clients satisfied. In order to achieve this, developers map out what features they are capable of creating, break complex requests into a series of smaller feature sets and and then create a plan for how to complete each goal over time.

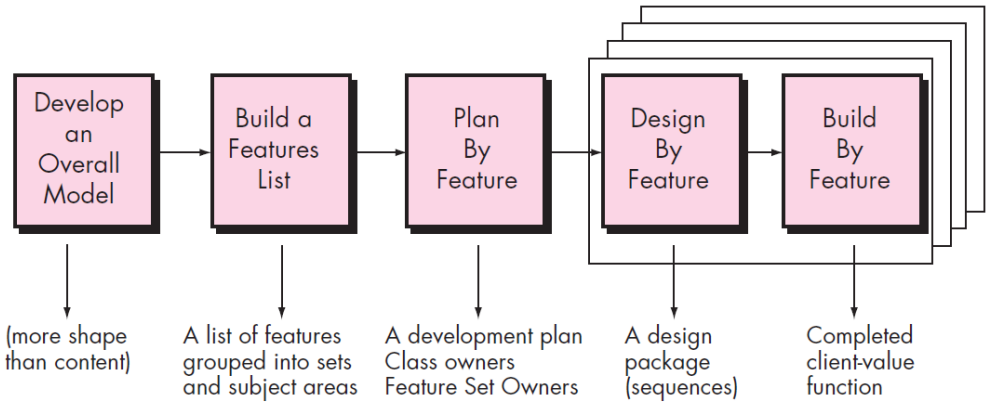


Fig. Feature Driven Development

In FDD, 'best practice' means following five stages of activity:

1. **Develop an overall model:**

   Here, teams should focus more on the shape and overall scope of the product than the detailed content.

2. **Build a feature list:**

   Next, teams should use the overall model to identify which features will be required.

3. **Plan by feature:**

   The planning phase is essential in FDD. Here, teams should allocate reasonable estimates to each feature, assign them to a team member and work out what needs to happen for these deadlines to be met. For ultimate success, all team members should take part in this process — so everyone is aligned with the plan of action.

4. **Design by feature:**

   Now it's time to get started! As FDD is an agile practice, teams should design concurrently and collaboratively.

5. **Build by feature:**

   Again, team members should work on their individual build responsibilities at the same time — visual designers on the UI, programmers on coded components, etc. When everything is ready to be pulled together, it's sent to QA for testing. Then, the next feature can be tackled.

vi. Lean Software Development

The concept of lean, in general, was pioneered by Toyota, the Japanese automotive giant, as a mean to reduce waste in manufacturing; but soon grew explosively popular and was adapted, among others, to various competencies in IT business.

Lean Software Development (LSD) is an agile framework based on optimizing development time and resources, eliminating waste, and ultimately delivering only what the product needs. The Lean approach is also often referred to as the Minimum Viable Product (MVP) strategy, in which a team releases a bare-minimum version of its product to the market, learns from users what they like, don't like and want to be added, and then iterates based on this feedback. It intends to speed up development time, by focusing only on the necessary deliverables, getting a product to market before adding new features.

"Lean development means removing all things unnecessary"

As the title suggests, it aims at removing all things unnecessary, all the fat, from software engineering. Its principles resonate largely with those of Agile development; some even consider the two methodologies to be inseparable. Others, on the other hand, admitting to the shared philosophy, still state that lean is more convergeable in terms of development strategy.

Overall, there are 7 principles to Lean software development, each aiming to quicken delivery and bring higher value to end-user:

1. Eliminating Waste
2. Building Quality In
3. Amplifying Knowledge
4. Delaying Commitment
5. Delivering Fast
6. Respecting people
7. Optimizing the whole thing

Refer: https://perfectial.com/blog/lean-software-development/

Exam questions:

1. What is lean software development? Mention the lean principles and explain how each of these principles can be adapted to software process with examples. [2019 Spring]
2. Describe the V process model with its advantages and disadvantages. [2019 Fall]
3. What do you mean by agile development? Why is it important? Explain the scrum software development in details. [2019 Fall]
4. Mention and explain some of the agility principle defined by agile alliance. [2018 Spring]
5. Define Spiral model with its advantages and disadvantages. [2018 Spring]
6. What do you mean by Agile development? Differentiate between Scrum and Lean software development. [2018 Fall]

7. What is the significance of Software Engineering? With diagram, explain Scrum process model for software development. [2017 Spring]
8. What do you mean by agile development? Explain the agile process and agility principles. [2017 Fall]
9. Define Software Engineering and its significant role over the system design. [2016 Spring, 2015 Spring]
10. Define agile software development with its advantages and disadvantages. [2016 Spring, 2015 Spring]
11. What is software engineering? Define the characteristics of system software. [2016 Fall]
12. Explain how both the waterfall model of the software process and prototyping model can be accommodated in the spiral process model. [2016 Fall]
13. Explain Agility principles of Software development. Which software development model is best suited for a risk driven software development? [2015 Fall]
14. Can Spiral model be used for all types of project? Give an example of development project for which spiral model is not appropriate. [2015 Fall]
15. Why is Agile process necessary? Differentiate between Scrum and Extreme programming. [2014 Spring]
16. A pharmaceutical company wants to develop an Inventory Control System for its internal use. The requirements are well understood and scope is well constrained. However, the project is required to be delivered within short period of time (as soon as possible). Propose a life cycle model for this scenario and provide reason(s) for justification of your answer. [2014 fall]
17. Describe the relationship and role of software engineering over other computer science areas. [2014 fall]
18. What are software process models? Differentiate between Iterative and Non-iterative Software process models. [2013 Fall]
19. A government office wants to develop a document management software system for its internal use. The development of the system is costly and it cannot afford for the fully developed system in a single fiscal year. They want to spend certain portion of the total budget of the software in this year and additional budget will be allocated in the upcoming years. Also the requirements are not well specified. Propose a life cycle model for this scenario and provide reason(s) for justification of our answer. [2013 spring]
20. Write short notes on:
    a) Spiral Model [2017 Spring]
    b) Cleanroom Engineering [2017 Fall, 2014 Spring]
    c) Agile Process [2016 Fall]
    d) RAD model [2014 Spring]

***

Object Oriented Software Engineering

Chapter 2

PLANNING SOFTWARE PROJECTS

Er. Shiva Ram Dam

Shivaram.dam@pec.edu.np

2.1 Project Management concepts

Building computer software is a complex undertaking, particularly if it involves many people working over a relatively long time. That's why software projects need to be managed. The management activities vary among people involved in a software project. A software engineer manages her day-to-day activities, planning, monitoring, and controlling technical tasks. Project managers plan, monitor, and control the work of a team of software engineers. Senior managers coordinate the interface between the business and software professionals.

Software project management is an umbrella activity within software engineering. It begins before any technical activity is initiated and continues throughout the definition, development and lifetime support of computer software. The project management activity encompasses measurement and metrics, estimation and scheduling, risk analysis, tracking, and control. The goals of software project management are effective team, focusing their attention on customer needs and product quality.

2.1.1 Management Spectrum

In software engineering, the management spectrum describes the management of a software project. The management of a software project starts from requirement analysis and finishes based on the nature of the product, it may or may not end because almost all software products faces changes and requires support. It is about turning the project from plan to reality. It focuses on the four P's; people, product, process and project. Here, the manager of the project has to control all these P's to have a smooth flow in the project progress and to reach the goal.

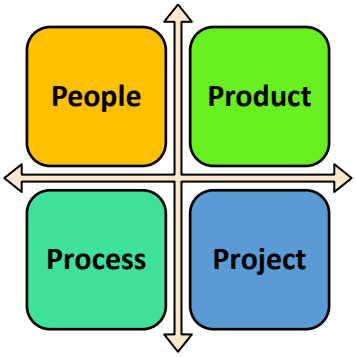
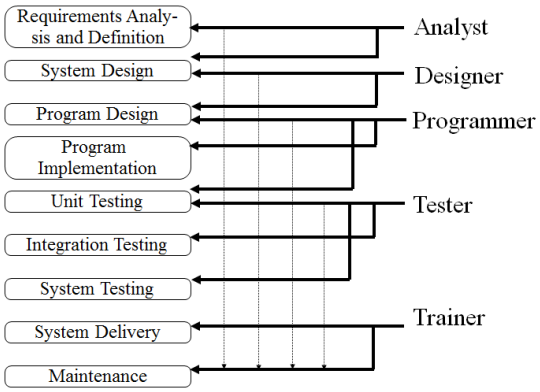


Figure: The 4 P's

a) The People

The people are the primary key factor for successful organization. For the successful software production environment, any organization must perform the proper staffing, communication and coordination, work environment, performance management, training, compensation (or reward), competency analysis and development, career development, workgroup development, team/culture development, and others.

The people involve: The Senior managers, Project managers, Analysts, Designers, Software developers, testers and customers.



b) The Product

Product is any software that has to be developed. To develop successfully, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified. Without this information, it is impossible to define reasonable and accurate estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks or a manageable project schedule that provides a meaningful indication of progress.

c) The Process

It is the set of framework activities and software engineering tasks to get the job done. The project manager is responsible for deciding what process to follow for doing the project and may use appropriate SDLC model.

A software process provides the framework from which a comprehensive plan for software development can be established. A number of different tasks sets— tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities overlay the software process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

d) The Project

The project is the complete software project that includes requirement analysis, development, delivery, maintenance and updates. The project manager of a project or sub-project is responsible for managing the people, product and process. The responsibilities or activities of software project manager would be a long list but that has to be followed to avoid project failure.

2.1.2 W5HH Principle

The W5HH principle in software management exists to help project managers guide objectives, timelines, responsibilities, management styles, and resources.

Created by software engineer Barry Boehm, the **purpose behind the W5HH principle** is to work through the objectives of a software project, the project timeline, team member responsibilities, management styles,

and necessary resources. In an article he wrote on the topic, Boehm stated that an organizing principle is needed that works for any size project. So he developed W5HH as a guiding principle for software projects.

The W5HH principle may have a funny-sounding name, but it too is designed for practicality. The **W5HH principle** outlines a series of questions that can help project managers more efficiently manage software projects. Each letter in W5HH stands for a question in the series of questions to help a project manager lead.

W5HH	The Question	What It Means
Why?	Why is the system being developed?	This focuses a team on the business reasons for developing the software.
What?	What will be done?	This is the guiding principle in determining the tasks that need to be completed.
When?	When will it be completed?	This includes important milestones and the timeline for the project.
Who?	Who is responsible for each function?	This is where you determine which team member takes on which responsibilities. You may also identify external stakeholders with a claim in the project.
Where?	Where are they organizationally located?	This step gives you time to determine what other stakeholders have a role in the project and where they are found.
How?	How will the job be done technically and managerially?	In this step, a strategy for developing the software and managing the project is concluded upon.
How Much?	How much of each resource is needed?	The goal of this step is to figure out the amount of resources necessary to complete the project.

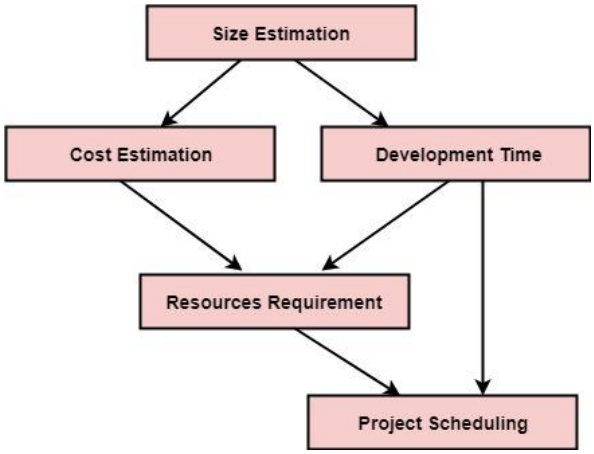
2.2 Project Planning Process

Software project management begins with a set of activities that are collectively called *project planning.* The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. Software project planning encompasses five major activities—estimation, scheduling, risk analysis, quality management planning, and change management planning. The schedule slippage, cost overrun, poor quality, and high maintenance costs for software may cause due to the lack of planning. Thus, planning attempt to define the best case and worst case scenario so that project outcome can be bounded. Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks. Therefore, the plan must be

adapted and updated as the project proceeds because "*The more you know, the better you estimate. Therefore, update your estimates as the project progresses.*"

The **estimation** attempt to determine how much money, effort, resources, and time it will take to build a specific software-based system or product. Estimation begins with a description of the **scope** of the problem. The problem is then decomposed into a set of smaller problems, and each of these is estimated using historical data and experience as guides. Problem complexity and risk are considered before a final estimate is made. After completion of estimation, **project scheduling** is started that defines software engineering tasks and milestones, identifies who is responsible for conducting each task, and specifies the inter-task dependencies that may have a strong bearing on progress.

Software Project planning starts before technical work start. The various steps of planning activities are:



2.3 Software Scope

Software scope describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a consequence of using the software; and the performance, constraints, interfaces, and reliability that *bound* the system.

Scope is defined using one of two techniques:

- A narrative description of software scope is developed after communication with all stakeholders.
- A set of use cases is developed by end users.

In general sense, the software scope indicates the acceptance limit of software by end user or environment. The software scope must be unambiguous and understandable at the management and technical level. Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: "Can we build software to meet this scope? Is the project feasible?"

2.4 Feasibility

The feasibility explains the acceptance of the software in different condition of technology, finance, time, resources, and process of operation, which all are explained below:

1. **Technical Feasibility**
   The software product must be technically feasible by the use of present and near future hardware and techniques. Also, it must be feasible to afford the technical person for new techniques or features required on the software product.

2. **Schedule (Time) Feasibility**
   The software product must be built in time, lunch in market in proper time stamp for competition, and financially beneficial. The overrun of software time will reduce the level of believe on customer for future task, may be loss the market price and popularity by incoming of same type of software from the competitor, and production cost may be much more.

3. **Financial Feasibility**
   The software product must be cost effective so that its user or users can purchase it. Sometime the selling of large copy of a software product in low price will be more beneficial than that of high cost selling in the prospective of popularity and benefit.

4. **Operational Feasibility**
   The user must know about the operation of the software product to use it in efficient way. Thus, the process of operation must be predefined in many ways such as training, meeting, presentation demo etc.

5. **Resource Feasibility**
   The software development company must have the software and hardware resources to build the quality software product as the customer demand. Also, the product must be well operating on the end user machine after delivery.

2.5 Resources

Project resources simply mean resources that are required for successful development and completion of project. These resources can be capital, people, material, tool, or supplies that are helpful to carry out certain tasks in project.

The three major categories of software engineering resources are—people, reusable software components, and the development environment (hardware and software tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time

when the resource will be required, and duration of time that the resource will be applied.

1. **Human Resources**

   Human are the primary resources that evaluate software scope and select the skills required to complete development. They specify both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, and client-server).

   The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required.

For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified.

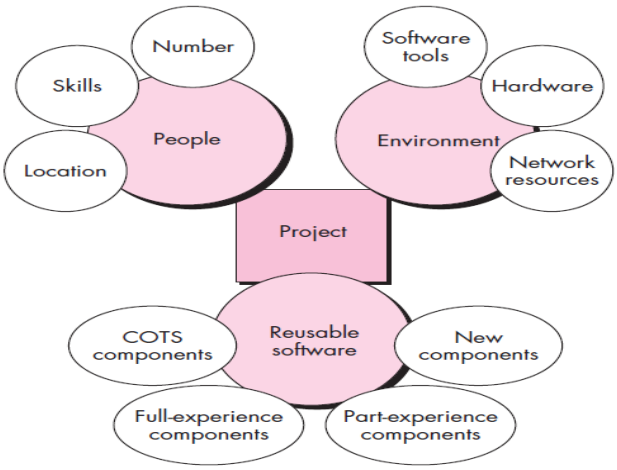
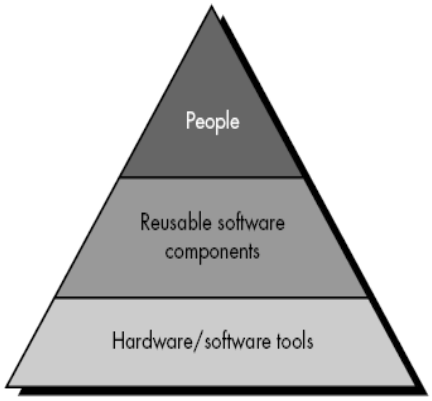


Fig. Project resources



2. Reusable Software Resources

Component-based software engineering (CBSE) emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan suggests four software resource categories that should be considered as planning proceeds:

i) **Off-the-shelf components:** Existing software that can be acquired from a third party or from a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

ii) **Full-experience components:** Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low risk.

iii) **Partial-experience components:** Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

iv) **New components:** Software components must be built by the software team specifically for the needs of the current project.

3. Environmental Resources

The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools

(software) required to produce the work products that are an outcome of good software engineering practice. When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a robotic device may require a specific robot as part of the validation test step.

2.6 Software Project Estimation

Software is the most expensive element of all computer system. Inappropriate estimation can make a big difference between the profit and loss. Software cost and effort estimation will never be an exact science. Too many variables- human, technical, environmental, political- can affect the cost and effort for software development. However, software project estimation can be done after considering these aspects:

A general project estimation can be done by:

i. Delaying estimation until late in the project.
ii. Estimation projects based on similar projects in the past.
iii. Using simple relatively decomposition technique.
iv. Using one or more empirical models for software cost and effort estimation.

Delayed Estimation and Estimation based on past projects:

The first two techniques are not efficient because estimation should be done earlier as possible. And not all the past experiences resemble the same criteria and might not fulfill the requirement of the current project.

Decomposition Technique:

Decomposition techniques works on "Divide and Conquer" approach in software project estimation. By decomposition a project is divided into different components and related software engineering activities. Cost and effort estimation can be performed step by step on each component.

Software cost estimation is a form to solve the problems. Most of the times problem to be solved is too complex to be solve in a single step. Then the problem is decomposed into number of components in order to achieve an accurate cost estimate.

There are two approaches in decomposition technique:

i. Problem based estimation: Problem decomposed. LOC and FP
ii. Process based estimation: Process is decomposed into a relatively small set of tasks and effort required to accomplish each task is estimated.

Empirical models:

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP. The structure of empirical estimation models is a formula, derived from data collected from past software projects, that uses software size to estimate effort. Eg: COCOMO

2.5.1 LOC Based Estimation

- LOC is a software metric used to measure the size of a software program by counting the no. of lines in the text of the program's source code.
- In LOC, lines used for commenting the code and the header lines should be ignored.
- The software is divided into different modules and the LOC is estimated for each module.
- The LOC count cannot be accurately computed before the actual coding. Hence, this method is less used.
- For eg:

```
#include <stdio.h>
int main()
{
    printf("Hello World");
    return 0;
}
```

Here, the line of code (LOC) =5

Logical line of code (LLOC) =2

Advantages	**Disadvantages**
• Simple to measure	• Cannot measure the size of specification. • Takes no account of functionality or complexity. • Bad software design may cause more LOC. • Language dependent.

For example: If

- Estimated total LOC on any project = 33,200
- Organizational average productivity = 620 LOC/pm
- Labor rate per month = $8000

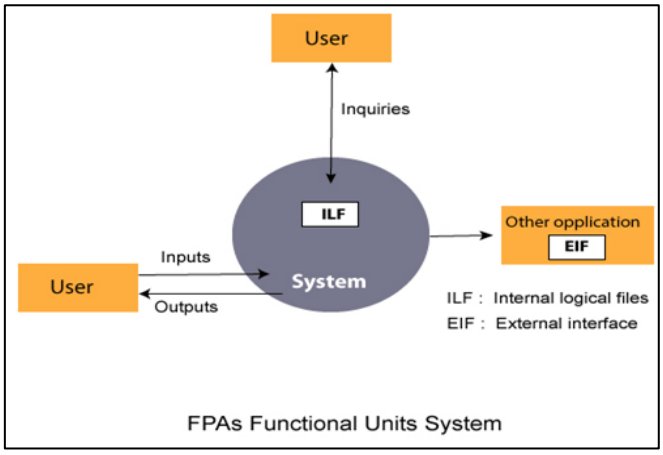
Then,

- Cost per LOC = ($8000/m)/(620LOC/pm) = $13
- Total project cost = Total LOC x Cost per LOC = 33,200 x $13 = $4,31,600
- Estimated effort = Total LOC / Average productivity

  = (33,200 LOC) / (620 LOC/person month) = 54 persons

2.5.2 FP Based Estimation

- An FP (Functional Point) is a unit of measurement to express the amount of business functionality that a software provides to a user. It measures functionality from user's point of view.
- In such method, the software product is directly dependent on the number of functions or features it supports.
- The approach is to identify and count a number of unique function types:
    - External inputs (eg: file names)
    - External outputs (eg: reports, message)
    - Queries (interactive inputs needing a response)
    - External files or interface (files shared with other software systems)
    - Internal files (invisible outside the system)



FPAs Functional Units System

Advantages	Disadvantages
• Not restricted to code • Language independent • More accurate than estimated LOC	• Subjective counting • Hard to automate and difficult to compute. • Ignores quality of output.

For example:

If FP = 375 FP, average productivity = 6.5 FP/PM and labor rate = $8000 per month then

- Cost per FP = ($8000) / (6.5 FP/PM) = $1230
- Total project cost = 375 x $1230 = $461250
- Estimated effort = Total FP / Average productivity = 375 FP / (6.5 FP/PM) = 58 persons

Computation of FP:

Mathematically;

FP= Count Total x CAF

Where, Count Total= sum of all FP entries

$CAF = 0.65 + 0.01 \times \sum F_i$

CAF→ Complexity Adjustment Factor

The F_i (i=1 to 14) is the value adjusted factor based on response to 14 questions.

Numerical: Given the following values, Compute FP when all complexity adjustment factors and weighting factors are average.

Information Domain Value	Count	Weighting Factor		
		Simple	Average	Complex
External Inputs (EI)	50	3	4	6
External Outputs (EO)	40	4	5	7
External Inquiries (EQ)	35	3	4	6
Internal Logical Files (IFL)	6	7	10	15
External Interface Files (EIF)	4	5	7	10

Solution: FP is given by :

$$FP = \text{Count Total} \times CAF$$

Now , Count Total =50*4 + 40*5 + 35*4 + 6*10 + 4*7 =628

$$CAF = 0.65 + 0.01 \times \sum F_i$$

$$= 0.65 + 0.01 \times (14 \times 3) = 1.07$$ *(Here 14 is taken for 14 questions to be answered)*

Hence FP= 628 x 1.07 =672

Reference for further study: *https://www.javatpoint.com/software-engineering-functional-point-fp-analysis*
https://www.youtube.com/watch?v=nN379biPGxE

2.5.3 COCOMO II Model

The constructive cost model (COCOMO) is an algorithmic software cost estimation model developed by Barry Boehm and published in 1981 on his book named Software Engineering Economics. The model uses a basic regression formula, with some parameters that are derived from historical project data and current project characteristics.

With reference to the COCOMO, the COCOMO-II was developed in 1997 and finally published in 2000 in the book Software Cost Estimation with COCOMO-II. Thus, COCOMO-II is the successor of COCOMO-81 and is better suited for estimating software development projects. It provides more support for modern software development processes and an updated project database. The need for new model came as software development technology moved from mainframe and overnight batch processing to desktop development, cost reusability and the use of off-the-self software components.

COCOMO II is tuned to modern software life cycles. The original COCOMO model has been very successful, but it doesn't apply to newer software development practices as well as it does to traditional practices. COCOMO II targets modern software projects, and will continue to evolve over the next few years.

COCOMO II has three different models:

i) The Application Composition Model

Suitable for projects built with modern GUI-builder tools. It estimates the size and efforts based on new Object Points. It supports estimation of prototyping. Applicable in where software can be quickly developed. Eg: GUI, database manager, etc.

Object points are calculated using the counts of: screens (at UI), reports and 3GL components.

Each object instance (e.g. a screen or report) is classified into one of the three complexity levels (i.e. simple, medium or difficult) using criteria suggested by Boehm.

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Table: Complexity Weighting for Object types

Once complexity is determined, the no. of screens, reports and components are weighted according to above table. The object point count is then determined by multiplying the original number of object instances by the weighting factor (in above table) and summing to obtain total object point count, The percentage of reuse is adjusted and the object point count is adjusted as:

New object points (NOP) = (object points) x [(100 - % reuse) / 100]

Productivity rate is given by:

$$\text{PROD} = \frac{\text{NOP}}{\text{person-month}}$$

Estimate of project effort is given by:

$$\text{Estimated effort} = \frac{\text{NOP}}{\text{PROD}}$$

ii) The Early Design Model

We can use this model to get rough estimates of a project's cost and duration before we've determined it's entire architecture (i.e. when we less know about it). It uses a small set of new Cost Drivers, and new estimating equations. Based on Unadjusted Function Points or KSLOC.

iii) The Post-Architecture Model

This is the most detailed COCOMO II model. We'll use it after we've developed our project's overall architecture. It has new cost drivers, new line counting rules, and new equations.

Software Download: *"calico for SystemStar 3.03"* : http://www.softstarsystems.com/calico.htm

References video for COCOMO II **steps for computation** : https://www.youtube.com/watch?v=KL1gK-H3Qvk

The COCOMO I

- It is the previous version of COCOMO II.
- COCOMO consists of a hierarchy of three increasingly detailed and accurate forms namely: Basic COCOMO, Intermediate COCOMO, and Detailed COCOMO
- The COCOMO model assumes every project is developed in one of the three models:
  a) *Organic mode:* The project requires little innovation
  b) *Semi-organic mode:* Intermediate between organic and embedded mode.
  c) *Embedded mode:* Requires a great deal of innovation

i) The Basic COCOMO:

The BASIC COCOCMO model computes software development effort and cost as a function of program size expressed in Estimated Lines of Code (KLOC). It is good for quick and rough estimation of software costs. Its accuracy is limited due to it does not consider the cost drivers.

Various equations in this model are:

$E = a * KLOC^{b}$

$T = c * E^{d}$

Where

E is effort applied in person-months

T is the development time

Project	a_b	b_b	c_b	d_b
Organic mode	2.4	1.05	2.5	0.38
Semidetached mode	3.0	1.12	2.5	0.35
Embedded mode	3.6	1.20	2.5	0.32

Table: Basic COCOCMO coefficients

Example:

The size of organic software is estimated to be 32,000 LOC. The average salary for software engineering is Rs. 15000/- per month. What will be effort and time for the completion of the project?

Solution:

- Effort applied = 2.4 x $(32)^{1.05}$ PM = 91.33 PM (Since: 32000 LOC = 32KLOC)
- Time = 2.5 x $(91.33)^{0.38}$ Month = 13.899 Months
- Cost = Time x Average salary per month = 13.899 x 15000 = Rs. 208480.85
- People required = (Effort applied) / (development time) = 6.57 = 7 persons

ii) Intermediate COCOMO

It is an extension of Basic COCOMO model that computes software development effort by adding a set of "Cost Drivers" that will determine the effort and duration of the project, such as assessment of personnel and hardware.

Equation used in this model is:

$$E = a * KLOC^{b} + EAF$$

Where

E is effort applied in person-month

EAF is effort adjustment factors

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table: Intermediate COCOCMO coefficients

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

Table: Intermediate COCOCMO EAF coefficients

iii) Detailed COCOMO

It is an extension of the Intermediate model that adds effort multipliers for each phase of the project to determine the cost-drivers impact on each step. Detailed COCOMO used the same equation for estimation as the Intermediate model, but for each development phases.

2.5.4 Estimation of Object-Oriented Projects

Conventional software project estimation techniques require estimates of lines of code (LOC) or function points (FP) as the primary driver for estimation. Because an overriding goal for OO projects should be reuse, LOC estimates make little sense. FP estimates can be used effectively because the information domain counts that are required are readily obtainable from the problem statement. FP analysis may provide value for estimating OO projects, but the FP measure does not provide enough granularity for the schedule and effort adjustments that are required as we iterate through the recursive/parallel paradigm.

Lorenz and Kidd suggest the following approach:

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using the requirements model, develop use cases and determine a count. Recognize that the number of use cases may change as the project progresses.
3. From the requirements model, determine the number of key classes (called analysis classes).
4. Categorize the type of interface for the application and develop a multiplier for support classes: Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.

Interface Type	Multiplier
No GUI	2.0
Text-based user interface	2.25
GUI	2.5
Complex GUI	3.0

5. Multiply the total number of classes (key + support) by the average number of work units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
6. Cross-check the class-based estimate by multiplying the average number of work units per use case.

2.5.5 Make/Buy Decisions and Outsourcing

In many software application areas, it is often more cost effective to acquire rather than develop computer software. Software engineering managers are faced with a make/buy decision that can be further complicated by a number of acquisition options:

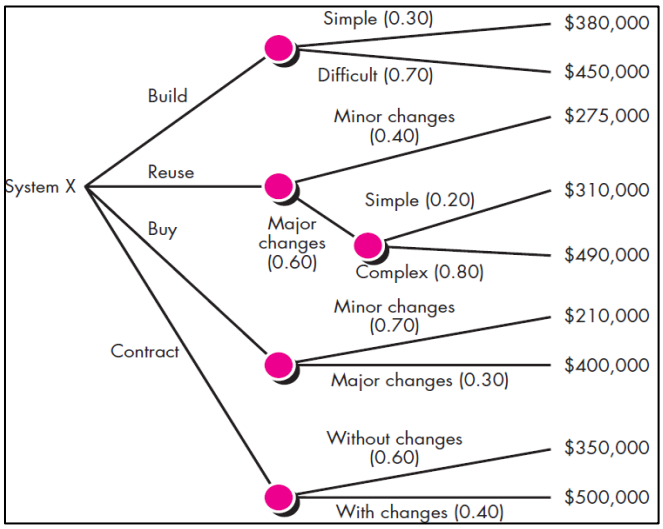
- software may be purchased (or licensed) off-the-shelf,
- "full-experience" or "partial experience" software components may be acquired and then modified and integrated to meet specific needs, or
- software may be custom built by an outside contractor to meet the purchaser's specifications

The make/buy decision is made based on the following conditions:

- Will the delivery date of the software product be sooner than that for internally developed software?
- Will the cost of acquisition plus the cost of customization be less than the cost of developing the software internally?
- Will the cost of outside support (e.g., a maintenance contract) be less than the cost of internal support?

Creating a decision tree

A decision tree is used to support the make/buy decision in similar to the given figure:



Now the expected value for cost, computed along any branch of the decision tree is:

$$\text{Expected cost} = \Sigma\ (\text{path probability})_i \times (\text{estimated path cost})_i$$

where i is the decision tree path.

$$\text{Expected cost}_{build} = 0.30\ (\$380K) + 0.70\ (\$450K) = \$429K$$
$$\text{Expected cost}_{reuse} = 0.40\ (\$275K) + 0.60\ [0.20\ (\$310K) + 0.80\ (\$490K)] = \$382K$$
$$\text{Expected cost}_{buy} = 0.70\ (\$210K) + 0.30\ (\$400K) = \$267K$$
$$\text{Expected cost}_{contract} = 0.60\ (\$350K) + 0.40\ (\$500K) = \$410K$$

From above calculation, decision can be made for buy option. However, it is not sufficient condition. Availability, experience of developer or vendor or contractor are some other criteria that affect the make/buy decision.

Outsourcing:

In concept, outsourcing is extremely simple. Software engineering activities are contracted to a third party who does the work at lower cost and, hopefully, higher quality. Software work conducted within a company is reduced to a contract management activity.

- On the positive side, cost savings can usually be achieved by reducing the number of software people and the facilities (e.g., computers, infrastructure) that support them.
- On the negative side, a company loses some control over the software that it needs. Since software is a technology that differentiates its systems, services, and products, a company runs the risk of putting the fate of its competitiveness into the hands of a third party.

2.7 Project Scheduling and Time Line charts

In order to build a complex system, many software engineering tasks occur in parallel, and the result of work performed during one task may have a profound effect on work to be conducted in another task. These interdependencies are very difficult to understand without a schedule. It's also virtually impossible to assess progress on a moderate or large software project without a detailed schedule.

Thus, after selection of appropriate process model, identification of the software engineering tasks that have to be performed, estimation of the amount of work and the number of people, fixing the deadline, and consideration of the risks, the next process on software engineering is the project scheduling. *Software project scheduling* is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. The schedule may evolve over time according to the changes appeared on the project.

Program evaluation and review technique (PERT) and the *critical path method* (CPM) are two project scheduling methods that can be applied to software development. Both PERT and CPM provide quantitative tools that allow us: to determine the critical path and estimate time for individual tasks.

Time Line chart or Gantt chart:

A Gantt chart, commonly used in project management, is one of the most popular and useful ways of showing activities (tasks or events) displayed against time. On the left of the chart is a list of the activities and along the top is a suitable time scale. Each activity is represented by a bar; the position and length of the bar reflects the start date, duration and end date of the activity. This allows us to see at a glance.

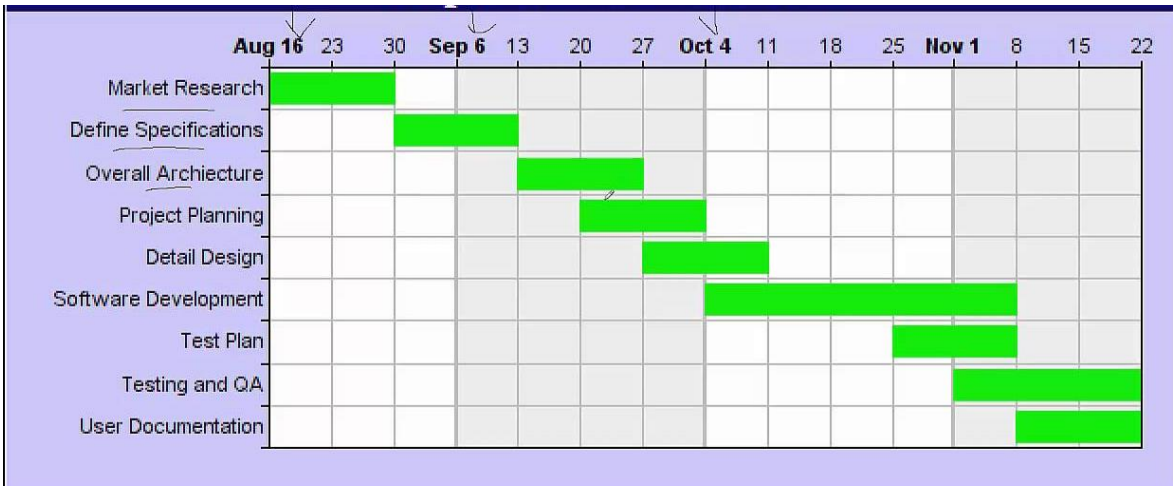


Figure: A Simple Gantt Chart

PERT Chart

Project Evaluation and Review Technique (PERT) chart, sometimes called a PERT diagram, is a project management tool used to schedule, organize and coordinate tasks within a project. It provides a graphical representation of a project's timeline that allows project managers to break down each individual task in the project for analysis.

PERT charts should be used when a project manager needs to:

- Determine the project's critical path in order to guarantee all deadlines are met.
- Display the various interdependencies of tasks.
- Estimate the amount of time needed to complete the project.
- Prepare for more complex and larger projects.

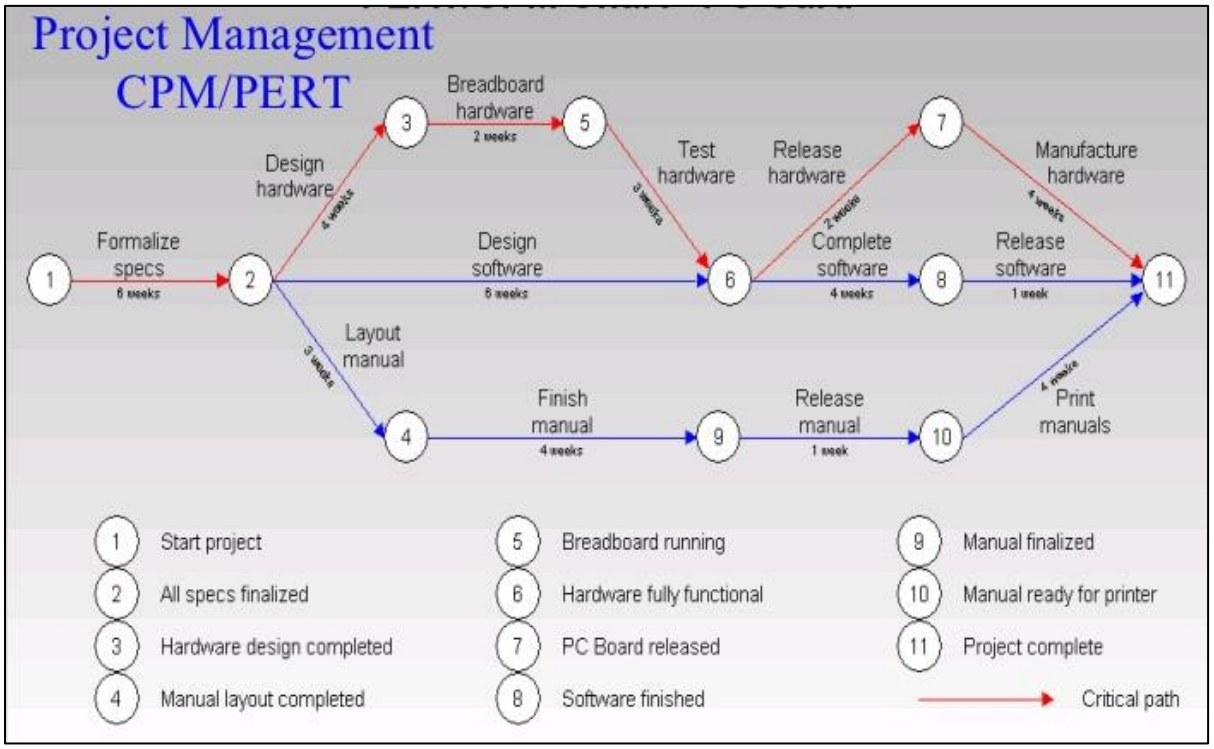


Figure: PERT chart

Reference for PERT: https://www.youtube.com/watch?v=ieNsHl9kk7g
Reference Video for Critical Path calculation: https://www.youtube.com/watch?v=uCR2x-VyxLg
https://www.youtube.com/watch?v=4oDLMs11Exs

Critical Path:

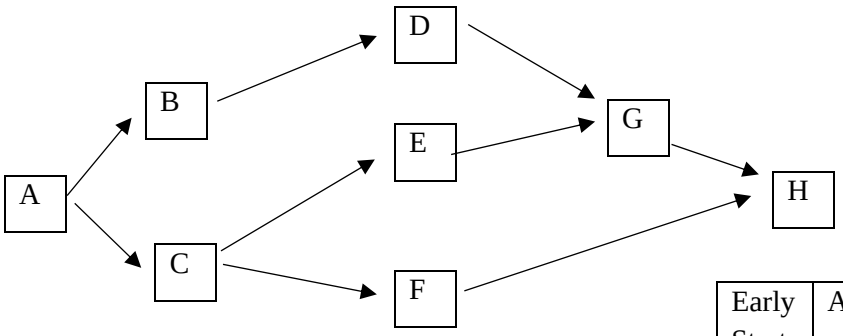
The critical path is the sequence of activities with the longest duration. A delay in any of these activities will result in a delay for the whole project.

Numerical example:

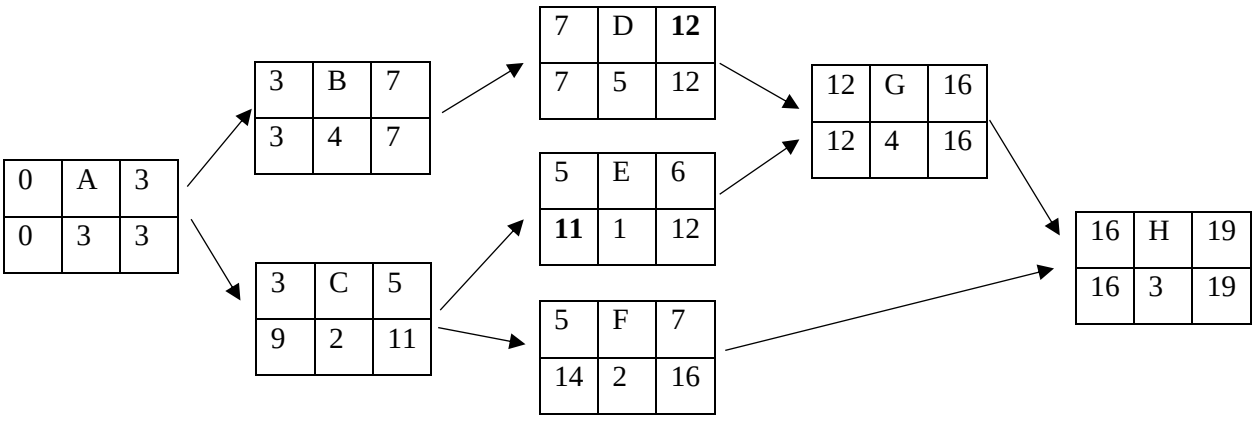
Calculate the project duration and Critical path from the below:

Activity	Predecessor	Duration (days)
A	-	3
B	A	4
C	A	2
D	B	5
E	C	1
F	C	2
G	D,E	4
H	F,G	3

Solution:



Early Start	Activity	Early Finish
Late Start	Duration	Late Finish



Here, Total project duration is 19 days.

Critical path is: A-B-D-G-H

2.8 Basics of Risk Management

What is risk?

It is a potential problem that may or may not occur in the future. It is the uncertainty, a irrelevant future occurrence or happening.

Effects of risks:

- Project diversion from its actual performance
- Unable to meet proper time-lines and thus overall success is affected.
- Quality degradation

Since risks are the factors for project failure and quality degradation, measures should be taken to avoid them. Thus, the overall process of identifying its impact and making plans in advance is called the risk analysis and management.

2.7.1 Reactive and Proactive Risk Handling Strategies

Generally, the Risk Analysis and Management procedures follow two major strategy: Reactive and Proactive.

a) Reactive Risk Strategy

Reactive Risk strategy is concerned with the process of finding solutions once the problem exists, the software team does nothing about risk until something goes wrong. The software engineer or team never worry about the problems until they happen, This is often called a *fire-fighting mode*.

b) Proactive Risk Strategy

In proactive software engineer starts thinking about possible risks in a project before they occur**.** A proactive strategy begins long before technical work is initiated where potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but all risks cannot be avoided, so exercise to make them minimal.

2.7.2 Software Risks

Risk concerns the future happening that always involves two characteristics:

(a) *uncertainty*—the risk may or may not happen; that is, and

(b) *loss*—if the risk becomes a reality, unwanted losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

1) Project risk: threaten the project plan.

Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

2) **Technical risks:** threaten the quality and timeliness

Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than expected.

3) **Business risks:** threaten the feasibility

It includes the top five business risks which are

- o *Market risk:* system that no one really wants
- o *Strategic risk:* product no longer fits into the overall business strategy
- o *Sales risk:* the sales force doesn't understand how to sell the product
- o *Management risk:* losing the support of senior management
- o *Budget risks:* losing budgetary or personnel commitment

4) **Known risks**

Risk that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

5) **Predictable risks**

These risks are identified from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

6) **Unpredictable risks**

They can and do occur, but they are extremely difficult to identify in advance.

2.7.3 Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). Risks can be removed or avoided only if they can be identified. By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary. The different categories of risk (project, technical, business, known, predictable, and unpredictable) can be further divided as:

- ***Generic risks:*** Generic risks are those that are common to all. They are a potential threat to every software project.
- ***Product-specific risks****:* Product-specific risks are those that are associated only to certain products. They can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built. To identify product-specific risks, the project plan and the software statement of scope are examined, and analyze the special characteristics of any product that may threaten the project plan.

One method for identifying risks is to create a **risk item checklist**; we need to identify the following area from where risk will appear:

1. ***Product size***—risks associated with the overall size of the software to be built or modified.
2. ***Business impact***—risks associated with constraints imposed by management or the marketplace.
3. ***Stakeholder characteristics***—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
4. ***Process definition—***risks associated with the degree to which the software process has been defined and is followed by the development organization.
5. ***Development environment***—risks associated with the availability and quality of the tools to be used to build the product.
6. ***Technology to be built***—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
7. ***Staff size and experience***—risks associated with the overall technical and project experience of the software engineers who will do the work.

Risk components:

The risk components defined by U.S. air force are defined in the following manner:

1. ***Performance risk***—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
2. ***Cost risk***—the degree of uncertainty that the project budget will be maintained.
3. ***Support risk***—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
4. ***Schedule risk***—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk component is divided into one of four categories—

1. negligible,
2. marginal,
3. critical, or
4. catastrophic.

2.7.4 Risk Projection and Risk Table

Risk Projection (***aka Risk Estimation)*** is the process of rating the risk based upon its likelihood or probability of occurrence and the consequence of the problem associated with the risk. It is carried out by the project planner along with other managers and technical staffs. During risk estimation, the first step is the prioritizing risks and hence we can allocate resources where they will have the most impact. The process of categorization of various possible risks that may appears in in project is called ***risk assessment***. The analysis of nature, scope and time are main component of risk assessment.

This process involves different activities like:

- Establishing a scale for the likelihood of a risk.
- Determining a scale for the likelihood (probability) of a risk.
- Determining and listing out the consequences of the risk.

Risk Table:

A risk table provides a simple technique for risk projection. The risk table can be implemented as a spreadsheet model. This enables easy manipulation and sorting of the entries. Here, at first all risks are listed in the first column of the table. This can be accomplished with the help of the risk item checklists reference. Each risk is categorized in the second column. The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually.

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ Σ Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Risk Category:
PS = Project size risk
BU = Business risk
TE = Technology risk
DE = Development risk
ST = Stakeholder risk
CU =Customer risk

Fig. RISK TABLE

Finally, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization. After that the table is sorted and defines a cutoff line. The *cutoff line* (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are reevaluated to accomplish second-order prioritization. The column labeled RMMM contains a pointer into a *risk mitigation, monitoring, and management plan.*

2.7.5 Risk Refinement

During early stages of project planning, a risk may be stated quite generally. But as time passes, we become more familiar learn about the project and risk. It might be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor and manage.

One way to do this is to represent the risk in *condition-transition-consequence* (CTC) format. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

For example:

- **Sub-condition 1.** Certain reusable components were developed by a third party with no knowledge of internal design standards.
- **Sub-condition 2.** The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
- **Sub-condition 3.** Certain reusable components have been implemented in a language that is not supported on the target environment.

2.7.6 RMMM and RMMM Plan

All of the risk analysis activities assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

Risk Mitigation is a problem avoidance activity, Risk Monitoring is a project tracking activity, Risk Management includes contingency plans that risk will occur. The goal of the risk mitigation, monitoring and management plan is to identify as many potential risks as possible

1. Risk Mitigation (Risk Avoidance)

The risk mitigation is proactive approach to adopting always the best strategy to reduce future possibilities of risk. Related to risk planning, through risk mitigation, the team develops strategies to reduce the possibility or the loss impact of a risk. Risk mitigation produces a situation in which the risk items are eliminated or otherwise resolved.

For example, assume that high staff turnover is noted as a project risk after evaluating the high-risk probability and impact of risk. Now, to mitigate this risk, we would develop a strategy for reducing turnover by taking the following possible steps:

- Meet with current staff to determine causes for turnover (poor working conditions, low pay, and competitive job market).
- Mitigate those causes that are under control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is “up to speed”).
- Assign a backup staff member for every critical technologist.

2. Risk Monitoring

After risks are identified, analyzed, and prioritized, and actions are established, it is essential that the team regularly monitor the progress of the product and the resolution of the risk items, taking corrective action when necessary. This monitoring can be done as part of the team project management activities or via explicit risk management activities. Often teams regularly monitor their "Top 10 risks." Risks need to be revisited at regular intervals for the team to reevaluate each risk to determine when new circumstances caused its probability and/or impact to change. At each interval, some risks may be added to the list and others taken away. Risks need to be reprioritized to see which are moved "above the line" and need to have action plans and which move "below the line" and no longer need action plans. A key to successful risk management is that proactive actions are owned by individuals and are monitored.

In the case of **high staff turnover**, the following factors must be monitored:

- The general attitude of team members based on project pressures,
- The degree to which the team has jelled (i.e. begun to work well)
- Interpersonal relationships among team members,
- Potential problems with compensation and benefits, and
- The availability of jobs within the company and outside

In addition to monitoring these, other risk monitoring parameters are effectiveness of risk mitigation steps, documentation and if a newcomer were forced to join the software team somewhere in the middle of the project.

3. Risk Management

The risk management process can be broken down into two interrelated phases, risk assessment and risk control, as outlined in Figure 1. These phases are further broken down. Risk assessment involves risk identification, risk analysis, and risk prioritization. Risk control involves risk planning, risk mitigation, and risk monitoring. It is essential that risk management be done iteratively, throughout the project, as a part of the team's project management routine.

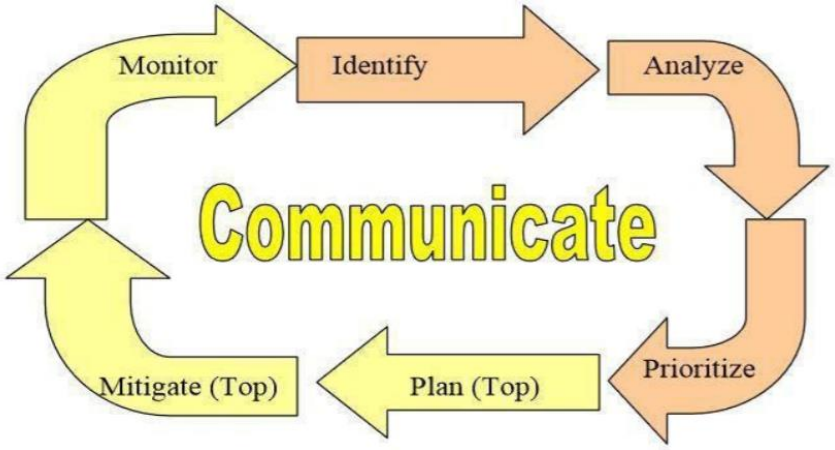


Fig: The Risk Management Process

Continuing the **example**, the project is well under way and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, to "get up to speed", newcomers enabled, and those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents or Wikis," and/or meeting with other team members who will remain on the project.

The RMMM plan

The RMMM plan is a document in which all the risk analysis activities are described. The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan. Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS). In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. The format of the RIS is illustrated in figure. Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence.

Risk information sheet

Risk ID: P02-4-32	Date: 5/9/09	Prob: 80%	Impact: high

Description:
Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Refinement/context:
Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards.
Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.

Mitigation/monitoring:
1. Contact third party to determine conformance with design standards.
2. Press for interface standards completion; consider component structure when deciding on interface protocol.
3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.

Management/contingency plan/trigger:
RE computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly.
Trigger: Mitigation steps unproductive as of 7/1/09.

Current status:
5/12/09: Mitigation steps initiated.

Originator: D. Gagne	Assigned: B. Laster

2.9 Software Maintenance

Maintenance corrects defects, adapts the software to meet a changing environment, and enhances functionality to meet the evolving needs of customers. The software maintenance begins almost immediately with in a day or week or month to adopt the challenges. At an organizational level, maintenance is performed by support staff that are part of the software engineering organization.

Most of software organization spend more money and time for maintaining existing programs. It is not unusual for a software organization to expend as much as 60 to 70 percent of all resources on software maintenance. Another software maintenance problem is the mobility of software people. It is likely that the software team (or person) that did the original work is no longer around. Worse, other generations of software people have modified the system and moved on. And today, there may be no one left who has any direct knowledge of the legacy system.

To reduce the software maintenance problem, we have to use the well-defined coding standards and conventions, leading to source code that is self-documenting and understandable. A variety of quality assurance techniques also uncover potential maintenance problems before the software is released.

Board Exam Questions:

1. What is risk management? What are the different types of risks and how can we identify risk? Explain. [2019 spring]
2. What is Project management? State and explain the concepts (The 4 P's) of project management. [2019 spring, 2014 spring]]
3. What is software scope? Effective software project management focuses on the four P's. Explain all. [2019 Fall]
4. Explain different feasibility associated with software. Explain how human resources and reusable software resources aid in software management activities. [2019 Fall]
5. What is W5HH principle? Explain the acceptance of the software in different condition in terms of types of feasibility. [2018 spring]
6. Explain briefly the need of risk identification during software development process. Also explain RMMM and RMMM plan. [2018 spring]
7. What are software risks? Explain briefly the different types of software risks. [2018 Fall, 2014 fall]
8. Explain the project management concepts. How do you estimate a software project? [2018 Fall]
9. What do you mean by software scoping? With example, explain decision-tree in order to understand the norms of Make/Buy decision. [2017 spring]
10. How do you estimate the software project? Explain any one of them with example. [2017 fall]
11. What do you mean by RMMM plan? Describe the reactive and proactive risk handling strategies in brief. [2017 Fall]
12. What are the project planning processes? Explain the W5HH principle in brief. [2017 fall]

13. Explain briefly the need of risk identification during software development process. Also explain RMMM and RMMM plan. [2016 spring]
14. A software project started on June 2015 was supposed to be completed by August 2016. But the progress review at the end of December 2016 shows that only 20% of the tasks have been completed and the major reason in delay of the project was the people factor. Explain briefly the possible reasons that the people factor affects the software development process. [2016 spring, 2016 fall, 2014 fall, 2013 spring]
15. What is a risk in a software? How do you identify risk in software engineering? Explain risk mitigation, monitoring and management. [2016 fall]
16. Define risk. Explain how risk management is carried out int a software project. Explain all the steps. [2015 spring]
17. Explain why software system which is used in a real-world environment must change or become progressively less useful. [2015 spring]
18. What is risk management? What are different types of risks and how can we identify risk? Explain. [2015 fall]
19. Compare and contrast FP based estimation with COCOMO II Model.[2015 fall]
20. Why do we need to estimate the cost of software projects? Differentiate between Reverse and Forward engineering with relevant example. [2014 spring]
21. Define Requirement traceability and explain why it is relevant to the maintenance of software systems. [2014 fall]
22. Why is it important to measure software? If estimated total LOC on ay project is 33200, organizational average productivity is 620 LOC/pm and labor rate per month is Rs. 8000, then calculate the total project cost and estimated effort. [2013 spring]
23. Suppose you are the project manager of an organization, Now, identify the most remarkable risk of your company making a risk table to prioritize them. [2013 spring]
24. Differentiate between LOC and FP techniques of cost estimation with suitable examples. [2013 fall]
25. Explain the role of managerial and technical manpower in project. [2013 fall]
26. What are the characteristics of risks? How can a project manager handle risk efficiently? [2013 fall]
27. Describe different types of feasibility study that needs to be performed before undertaking project. [2013 fall]
28. Write short notes on:
    a) People CMM [2019 Fall]
    b) COCOMO model [2018 Fall, 2017 fall]
    c) Software project estimation [2017 fall]
    d) Estimation of object-oriented projects. [2015 fall]
    e) Outsourcing [2014 fall]

***

Object Oriented Software Engineering

Chapter 3

SOFTWARE MODELING

Er. Shiva Ram Dam

Shivaram.dam@pec.edu.np

3.1 Software Engineering Principles and Practice

Software engineering practice is a broad array of ***principles***, ***concepts***, ***methods***, and ***tools*** that we must consider as software is planned and developed. Principles that guide practice establish a foundation from which software engineering is conducted. The software process provides everyone involved in the creation of a computer-based system or product with a road map for getting to a successful destination. Principles instructs us on how to drive, where to slow down, and where to speed up.

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the ***process level***, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the ***level of practice***, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community.

Principles That Guide Practice

Software engineering practice has a single overriding goal—to deliver on-time, high quality, operational software that contains functions and features that meet the needs of all stakeholders. To achieve this goal, we should adopt a set of core principles that guide all technical work. These principles have merit regardless of the analysis and design methods, the construction techniques (e.g., programming language, automated tools), or the verification and validation approach chosen. The following set of core principles are fundamental to the practice of software engineering:

- **Principle 1. *Divide and conquer.*** A large problem is easier to solve if it is subdivided into a collection of elements (or *concerns*). Ideally, each concern delivers distinct functionality that can be developed, and in some cases validated, independently of other concerns.
- **Principle 2. *Understand the use of abstraction.*** An abstraction is a simplification of some complex element of a system used to communicate meaning in a single phrase. In analysis and design work, a software team normally begins with models that represent high levels of abstraction (e.g., a spreadsheet) and slowly refines those models into lower levels of abstraction (e.g., a *column* or the *SUM* function).
- **Principle 3. Strive for consistency.** Whether it's creating a requirements model, developing a software design, generating source code, or creating test cases, the principle of consistency suggests that a familiar context makes software easier to use. As an ***example***, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.
- **Principle 4. *Focus on the transfer of information.*** Software is about information transfer—from a database to an end user, legacy system to a WebApp, operating system to an application, etc. In every case of information flows, there are opportunities for error, or omission, or ambiguity. This principle focus on pay special attention to the analysis, design, construction, and testing of interfaces.
- **Principle 5. *Build software that exhibits effective modularity.*** Any complex system can be divided into modules (components), each module should focus exclusively on one well-constrained aspect of the system—it should be cohesive in its function and/or constrained in the content it represents. Additionally, modules should be interconnected in a relatively simple manner—each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.

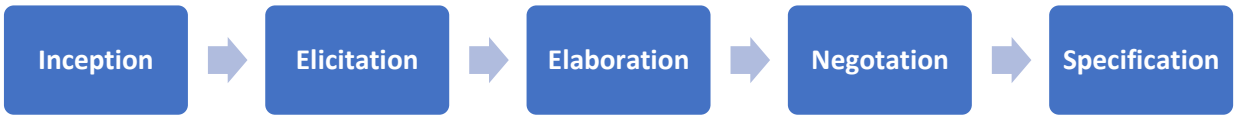
- **Principle 6.** ***Look for patterns.*** The goal of patterns within the software community is to create a body of knowledge for future planning which defines our understanding of good architectures that meet the needs of their users.
- **Principle 7.** ***When possible, represent the problem and its solution from a number of different perspectives.*** When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered.
- **Principle 8.** ***Remember that someone will maintain the software.*** Over the long term, software will be corrected as defects are uncovered, adapted as its environment changes, and enhanced as stakeholders request more capabilities. These maintenance activities can be facilitated if solid software engineering practice is applied throughout the software process.

3.2 Requirement Engineering

The process of gathering the software requirements from the client, analyze and document them is known as Requirement Engineering. The goal of requirement engineering is to develop and maintain sophisticated and descriptive Software Requirement Specification (SRS) document.

Requirements engineering involves set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end users will interact with the software. In more detail manner, requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system.

The steps for requirement engineering can be illustrated as follows:



1. **Inception:** A task that defines the scope and nature of the problem to be solved.
2. **Elicitation:** A task that helps stakeholders define what is required. Problems of scope, problems of understanding, and problems of volatility are three problems that are encountered as elicitation occurs.
3. **Elaboration:** The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.
4. **Negotiation:** what are the priorities, what is essential, when is it required?
5. **Specification:** The term *specification* means different things to different people. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

3.3 Requirement Elicitation/Gathering

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation—a requirements gathering activity that makes use of facilitated meetings, quality function deployment (QFD), and the development of usage scenarios. There are some problems encountered as elicitation occurs.

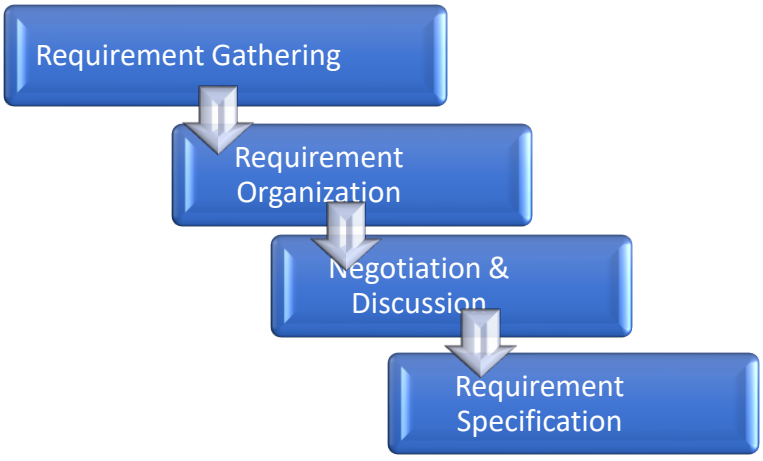
- **Problems of scope:** The problems due to ill-defined system boundary, unnecessary technical detail specified by customer etc.
- **Problems of understanding:** The problems due to customers are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, etc.
- **Problems of volatility:** The requirements change over time.

To help overcome these problems, **collaborating requirements gathering** technique is useful, which have following basic guidelines:

- Meetings are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

Requirement Elicitation is the process to find out the requirements for an intended software system by communicating with client, end-users, system users and other stakeholders in the software system development. Various techniques used for requirement elicitation are: interviews, brainstorming, surveys, questionaries, use-case approach, task-analysis, domain-analysis, prototyping, observations and so on.

Requirement Elicitation involves the following process:



1. **Requirement Gathering**

   The developers discuss with the client and end-users and know their expectations for the software.

2. **Requirement Organization**

   The developers prioritize and arrange the requirements in order of importance, urgency and convenience.

3. **Negotiation & Discussion**

   If requirements are ambiguous (having conflicts), it is then negotiated and discussed with the stakeholders. Unrealistic requirements are compromised reasonably.

4. **Requirement Specification**

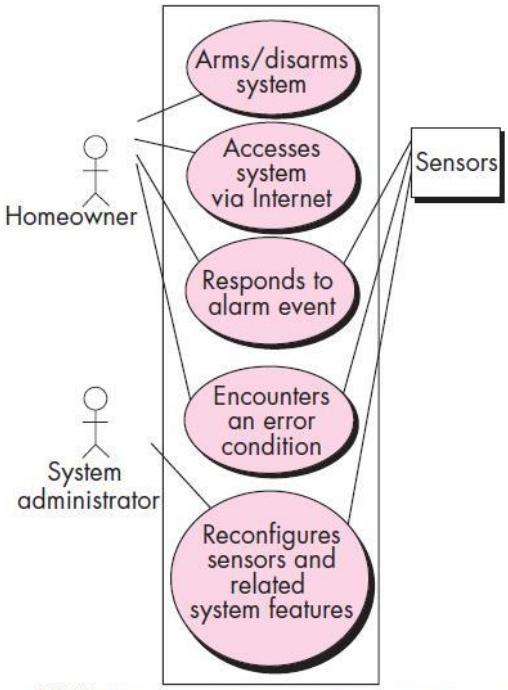
   All formal and informal, functional and non-functional requirements are documented, and are made available for next phase processing.

Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD concentrates on maximizing customer satisfaction from the software engineering process.

Sometimes, the developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called use cases, provide a description of how the system will be used.

The general work products produced as a consequence of requirements elicitation will be: a statement of need and feasibility, a list of customers, users, and other stakeholders who participated, description of the system's technical environment, any prototypes developed to better define requirements, etc.



UML use case diagram for *SafeHome*

3.4 Developing requirement Model

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as we learn more about the system to be built, and other stakeholders understand more about what they really require. For that reason, the analysis model is a snapshot of requirements at any given time.

There are many different ways to look at the requirements for a computer-based system. The most common elements of the requirements model are described below:

- **Scenario-based elements:** The system is described from the user's point of view using a scenario-based approach. For example, basic use cases and their corresponding use-case diagrams, and UML activity diagram.
- **Class-based elements:** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes—a collection of things that have similar attributes and common behaviors. For example, a UML class diagram
- **Behavioral elements:** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior. The *state diagram* is one method for representing the behavior of a system by depicting its states and the events that cause the system to change state. A *state* is any externally observable mode of behavior. In addition, the state diagram indicates actions (e.g., process activation) taken as a consequence of a particular event.
- **Flow-oriented elements:** Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms. For example, data flow diagram (DFD) and control flow diagram (CFD)

3.5 Requirement Validation

Requirement Validation is the process of examining the specification to ensure that:

- all system requirements have been stated clearly
- inconsistency and errors have been deleted and corrected
- work-products confirm to the standards established for the process, the project and the product,

Each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is to be built. *Requirements validation* examines the specification to ensure that all system requirements have been stated unambiguously, inconsistencies are removed, errors have been detected and corrected and the work products are ready for the process, the project, and the product. The primary requirements validation mechanism is the *formal technical review.*

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?

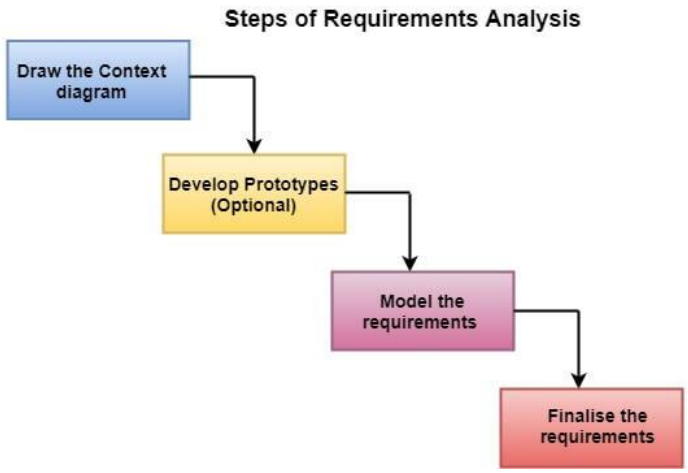
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?

3.6 Requirement Analysis

Requirements Analysis is the process of defining the expectations of the users for an application that is to be built or modified. It involves all the tasks that are conducted to identify the needs of different stakeholders. Therefore, requirements analysis means to analyze, document, validate and manage software or system requirements.

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 figure below:



(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.

(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.

(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) Finalize 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:

Broadly, software requirements can be categorized into two categories:

a) Functional requirements

Requirements which are related to functional aspect of software falls into this category. Examples include:

- Search option given to search from various invoices,
- Mailing facility,
- Report generation, etc.

b) Non-functional requirements

Non-functional requirements are those which are not related to functional aspect of software. Examples include:

- Security,
- Storage,
- Configuration,
- Cost,
- Disaster recovery,
- Flexibility,etc.

3.6.1 Domain Analysis

Domain analysis is the process by which a software engineer learns background information. He or she has to learn sufficient information so as to be able to understand the problem and make good decisions during requirements analysis and other stages of the software engineering process. The word 'domain' in this case means the general field of business or technology in which the customers expect to be using the software.

Some domains might be very broad, such as 'airline reservations', 'medical diagnosis', and 'financial analysis'. Others are narrower, such as 'the manufacturing of paint' or 'scheduling meetings'. People who work in a domain and who have a deep knowledge of it (or part of it), are called *domain experts*. Many of these people may become customers or users.

To perform domain analysis, you gather information from whatever sources of information are available: these include the domain experts; any books about the domain; any existing software and its documentation, and any other documents he or she can find. The interviewing, brainstorming and use case analysis techniques, and also Object-oriented modelling can help with domain analysis.

Benefits of performing Domain Analysis:

- Faster development
- Better system
- Anticipation of extensions

(Reference: http://www.site.uottawa.ca/~laganier/seg2500/domain)

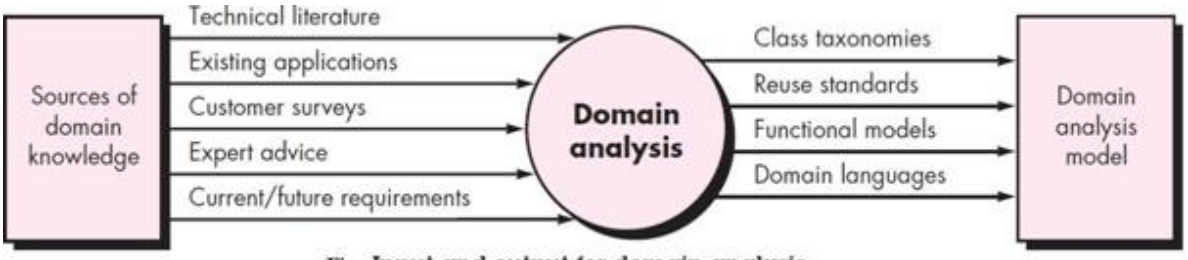


Fig. Input and output for domain analysis

3.6.2 Requirement Modeling

Requirements modeling in software engineering is essentially the planning stage of a software application or system. Generally, the process will begin when a business or an entity approaches a software development team to create an application or system from scratch or update an existing one. Requirements modeling comprises several stages, or '**patterns**': scenario-based modeling, data modeling, flow-oriented modeling, class-based modeling and behavioral modeling. Each of these stages/patterns examines the same problem from a different perspective.

Technically, there is no 'right way' of going through these stages, but generally, the process would begin with scenario-based modeling and complete with behavioral modeling.

Software engineering basically have two approaches of requirements modeling, called *structured analysis*, and *object-oriented analysis*. *Structured analysis* considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system. *Object-oriented analysis* focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly object oriented.

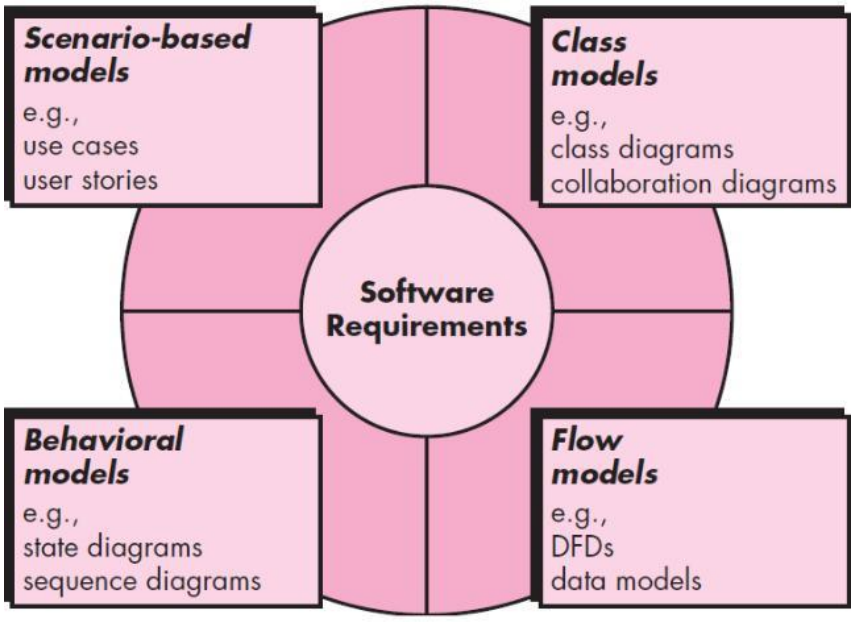


Figure: Elements of the analysis model

- Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.
- Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined.
- Behavioral elements depict how external events change the state of the system or the classes that reside within it.
- Finally, flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

The negotiation among customer, users, and stakeholder is also important because if they cannot agree on defined requirements then the risk of failure is very high. Thus, the best negotiation strives for a "Win-Win" result.

3.7 Scenario Based Modeling

This is typically the first stage of requirements modeling where top-level requirements are collected; although there is no hard and fast rule to initiate modeling with this phase. This is the phase where top-level use-cases are identified and described. The use case diagrams give overview of interactions of various actors with the system. This is preliminary user interactions with system, which shows how uses are going to use the system, which sort out any high-level ambiguities.

If you understand how end users (and other actors) want to interact with a system, your software team will be better able to properly characterize requirements and build meaningful analysis and design models. Hence, requirements modeling with UML begins with the creation of scenarios in the form of use cases, activity diagrams, and swim-lane diagrams.

3.7.1 Use-case Modeling

A use-case model is a model of how different types of users interact with the system to solve a problem. As such, it describes the goals of the users, the interactions between the users and the system, and the required behavior of the system in satisfying these goals.

Writing use cases is an excellent technique to understand and describe requirements. Informally, they are stories of using a system to meet goals. The essence is discovering and recording functional requirements by writing stories of using a system to help fulfill various stakeholder goals.

Use Case Texts and Use Case Diagram

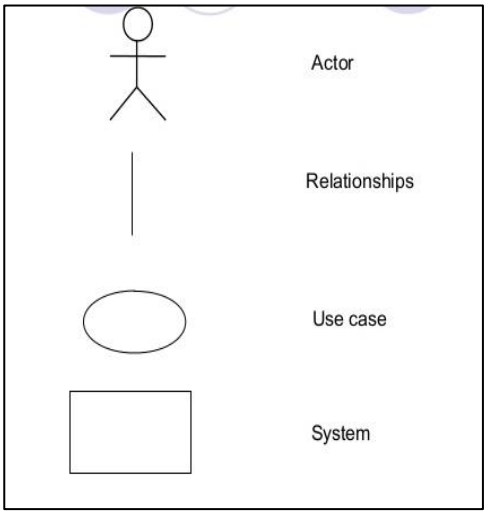
Use case Texts are the written description of how users will perform tasks on the system. The text provides the details and context; the diagram visually summarizes the interaction.

Use-case Diagram:

Use-case diagrams are the system based description of how the system will be used and how the actors (i.e. people, device, program, etc.) will be dealing with the system to be developed. It models the system from the end-users point of views. So, the actor should be identified first and their roles are also defined.

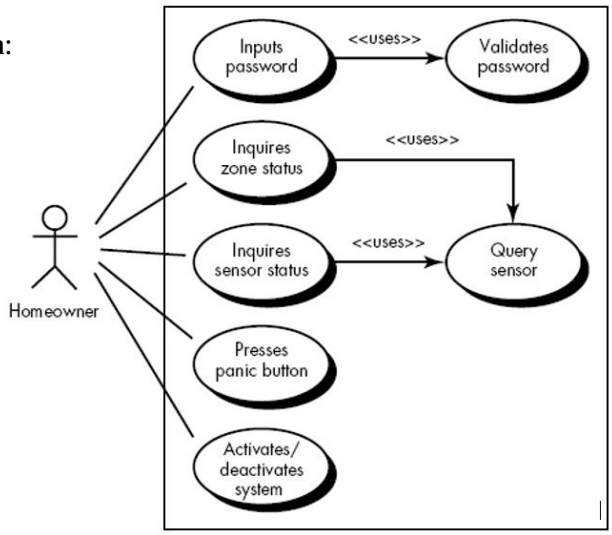
The purposes of Use-Case are:

- To gather requirements of a system
- To get on outside view of a system
- Identify external and internal factor influencing the system.
- Show the interaction among requirements and actors.



Symbols used in Use-case diagram

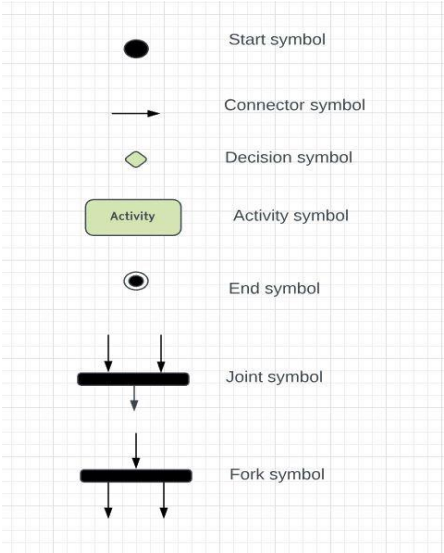
Sample Use-case Diagram for Safe-Home System:



3.7.2 Activity Diagram

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system. The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario.

Similar to theflowchart, an activity diagram uses *rounded rectangles* to imply a specific system function, a*rrows* to represent flow through the system, *decision diamonds* to depict a branching decision (each arrow emanating from the diamond is labeled), and *solid horizontal lines* to indicate that parallel activities are occurring.



Example:

Given the problem description related to the workflow for processing an order, let's model the description in visual representation using an activity diagram:

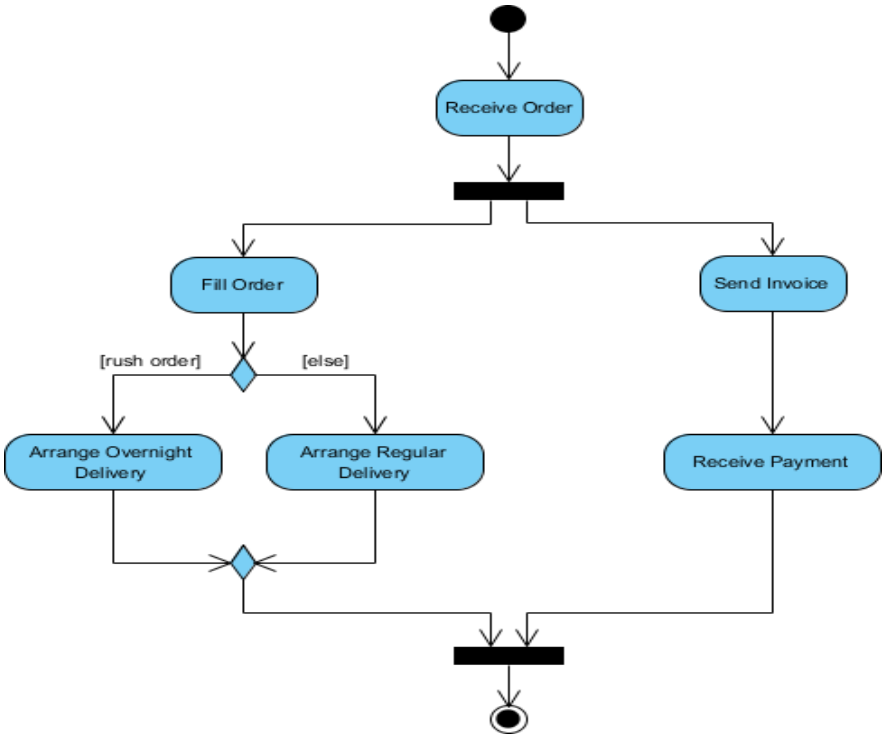
Process Order - Problem Description

Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.

On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.

Finally the parallel activities combine to close the order.

The activity diagram example below visualizes the flow in graphical form.



Reference: *https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/*
https://sceweb.sce.uhcl.edu/helm/WEB-TOC-UML/Tutorial/uml_activity_diagrams.html

3.8 Data Modeling

Data modeling, sometimes also called information modeling, is the process of visually representing what data the application or system will use, and how it will flow. The resulting diagram or other visual representation is meant to be designed in a way that is as easy to understand as possible. The fundamental elements that a data model needs to include and describe are the **data objects**, more frequently called **'entities'**, the **attributes** of those objects/entities, and the **relationships** between the objects/entities. A data modeling is performed through the ERD.

Entity Relationship diagram (ERD)

The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships (*data object type hierarchies* and *associative data object*). ERD uses graphical notation, three basic elements in ER models are:

- Entities are the "things" about which we seek information (**Rectangle**).
- Attributes are the data we collect about the entities (**Oval**).
- Relationships provide the structure needed to draw information from multiple entities (Line and Diamond).

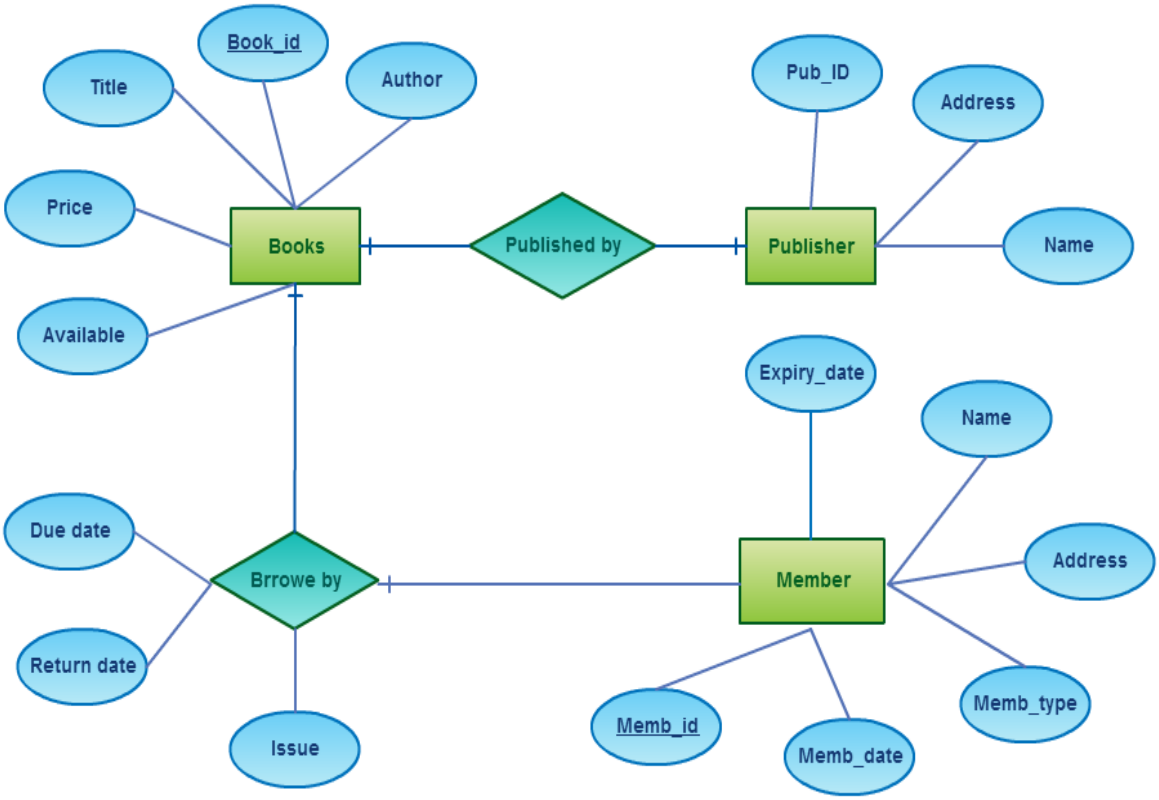


Figure: ERD of Library Management System

3.9 Class Based modeling

Class-based modeling represents the objects that the system will manipulate, the operations (also called methods or services) that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects,

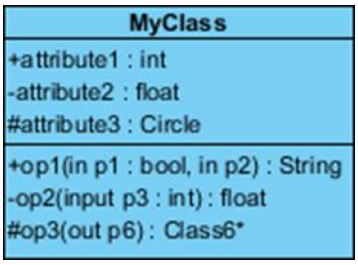
attributes, operations, class responsibility collaborator (CRC) models, collaboration diagrams, and packages.

3.9.1 Class Diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is **a** type of static structure diagram **t**hat describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. The class diagram depicts classes, which include both behaviors and states, with the relationships between the classes. The main building block of a class diagram is the *class*, which stores and manages information in the system. During analysis, classes refer to the people, places, events, and things about which the system will capture information. Later, during design and implementation, classes can refer to implementation-specific artifacts such as windows, forms, and other objects used to build the system.

A UML class diagram is made up of:

- A set of classes and
- A set of relationships between classes



Class Notation

A class notation consists of three parts:

1. **Class Name**
   - The name of the class appears in the first partition.

2. **Class Attributes**
   - Attributes are shown in the second partition.
   - The attribute type is shown after the colon.
   - Attributes map onto member variables (data members) in code.

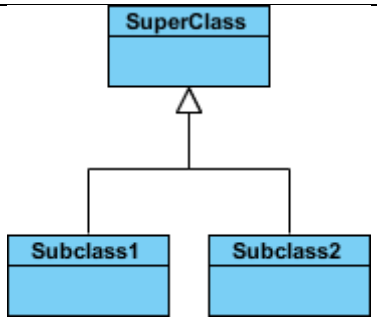
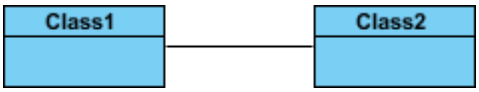
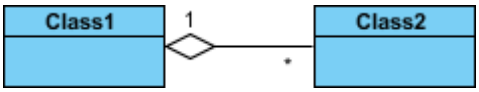
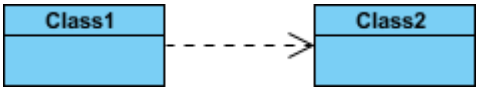
3. **Class Operations** (Methods)
   - Operations are shown in the third partition. They are services the class provides.
   - The return type of a method is shown after the colon at the end of the method signature.
   - The return type of method parameters is shown after the colon following the parameter name.
   - Operations map onto class methods in code

The graphical representation of the class - *MyClass* as shown above:

- *MyClass* has 3 attributes and 3 operations
- Parameter *p3* of *op2* is of type *int*
- *op2* returns a *float*
- *op3* returns a pointer (denoted by a *) to *Class6*

Class Relationships

A class may be involved in one or more relationships with other classes. A relationship can be one of the following types: (Refer to the figure on the right for the graphical representation of relationships).

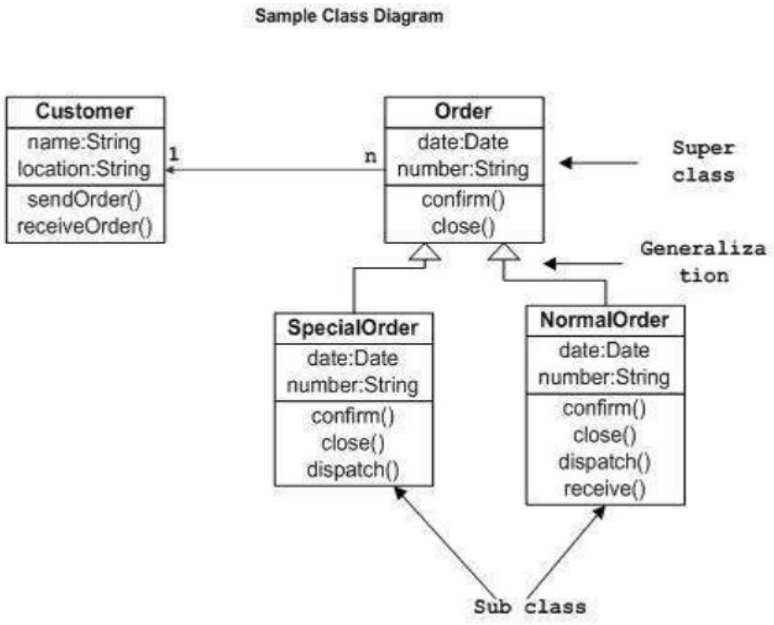
Relationship Type	Graphical Representation
1. Inheritance (or Generalization): • Represents an "is-a" relationship. • An abstract class name is shown in italics. • SubClass1 and SubClass2 are specializations of Super Class. • A solid line with a hollow arrowhead that point from the child to the parent class	
2. Simple Association: • A structural link between two peer classes. • There is an association between Class1 and Class2 • A solid line connecting two classes	
3. Aggregation: A special type of association. It represents a "part of" relationship. • Class2 is part of Class1. • Many instances (denoted by the *) of Class2 can be associated with Class1. • Objects of Class1 and Class2 have separate lifetimes. • A solid line with an unfilled diamond at the association end connected to the class of composite	
4. Composition: A special type of aggregation where parts are destroyed when the whole is destroyed. • Objects of Class2 live and die with Class1. • Class2 cannot stand by itself. • A solid line with a filled diamond at the association connected to the class of composite	
5. Dependency: • Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around). • Class1 depends on Class2 • A dashed line with an open arrow	

Reference: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

E.g: Order Management System

Class identification:

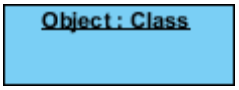
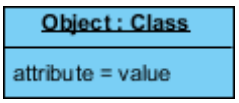
1. First of all *Order* and *Customer* are identified as two elements of the system, and they have one to many relationship because customers can have multiple orders.
2. We would keep *Order* class as an Abstract class, and it has two concrete Classes (Inheritance, relationship), i.e *Special_order* and *Normal_order*
3. The two inherited classes have all the properties of the O*rder* class. In addition, they have additional functions like *dispatch()* and *receive()*.

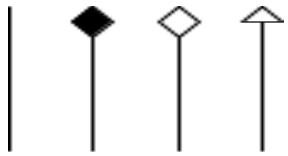


3.9.2 Object Diagram

Object is an instance of a class in a particular moment in runtime that can have its own state and data values. Likewise a static UML object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time, thus an object diagram encompasses objects and their relationships which may be considered a special case of a class diagram or a communication diagram.

Basic Object Diagram Symbols and Notations

Detail	Graphical Representation
1. Object Names: Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.	
2. Object Attributes: Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.	

3. Links: Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.	

E.g: Order Management System:

The following diagram is an example of an object diagram. It represents the Order management system which we have discussed in the earlier Class Diagram. The following diagram is an instance of the system at a particular time of purchase. It has the following objects.

- Customer
- Order
- SpecialOrder
- NormalOrder

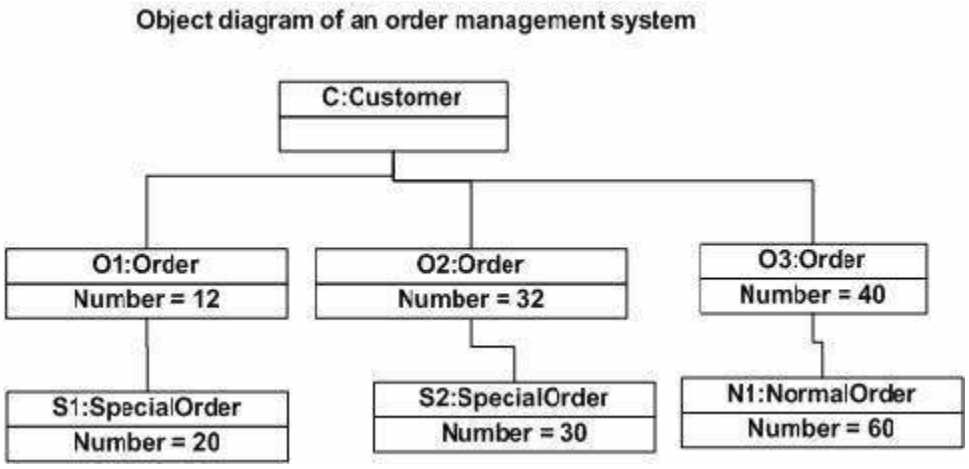
Now the *customer* object (C) is associated with three *order* objects (O1, O2, and O3). These *order* objects are associated with *specialorder* and *normalorder* objects (S1, S2, and N1). The customer has the following three orders with different numbers (12, 32 and 40) for the particular time considered.

The customer can increase the number of orders in future and in that scenario the object diagram will reflect that. If order, special order, and normal order objects are observed then you will find that they have some values.

For orders, the values are 12, 32, and 40 which implies that the objects have these values for a particular moment (here the particular time when the purchase is made is considered as the moment) when the instance is captured

The same is true for special order and normal order objects which have number of orders as 20, 30, and 60. If a different time of purchase is considered, then these values will change accordingly.

The following object diagram has been drawn considering all the points mentioned above



3.10 Flow Oriented Modeling

The flow-oriented modeling represents how data objects are transformed as they move through the system. Derived from structured analysis, flow models use the data flow diagram, a modeling notation that depicts how input is transformed into output as data objects move through the system. Each software function that transforms data is described by a process specification or narrative. In addition to data flow, this modeling element also depicts control flow.

Flow oriented modeling focuses on structured analysis and design, follows a top to down methodology and uses a graphical technique depicting information flows and the transformations that are applied as data moves from input to output. Data flow diagram (DFD) and Control flow diagram (CFD) are two common flow models.

3.10.1 Brief Introduction to Data Flow Diagram

A Data Flow Diagram (DFD) is a traditional way to visualize the information flows within a system. It shows how information enters and leaves the system, what changes the information and where information is stored. The purpose of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communications tool between a systems analyst and any person who plays a part in the system that acts as the starting point for redesigning a system.

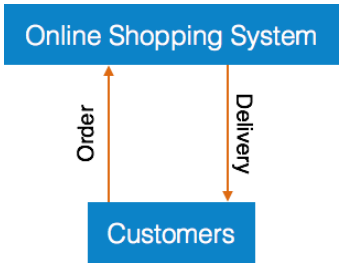
DFD Components

DFD can represent Source, destination, storage and flow of data using the following set of components -

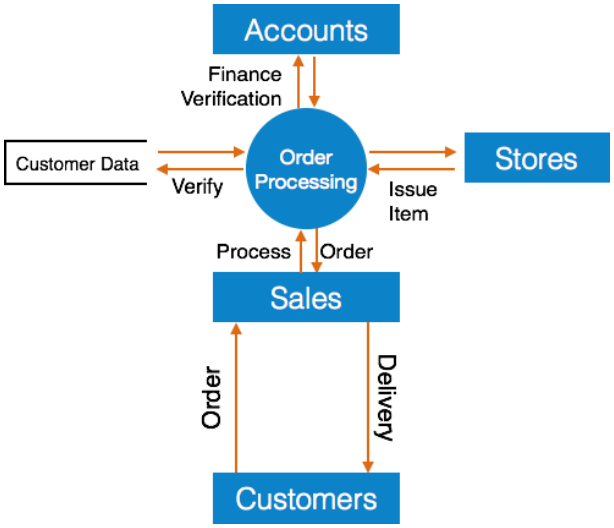
- **Entities** - Entities are source and destination of information data. Entities are represented by a rectangles with their respective names.
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

Levels of DFD

Level 0 - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



Level 1 - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.



Level 2 - At this level, DFD shows how data flows inside the modules mentioned in Level 1.

Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

Reference:	*https://www.tutorialspoint.com/software_engineering/software_analysis_design_tools.htm*
	https://www.javatpoint.com/software-engineering-data-flow-diagrams
	https://www.geeksforgeeks.org/levels-in-data-flow-diagrams-dfd/

3.10.2 Brief Introduction to Process Specification and Control Specification

Process Specification (PSPEC):

A process specification is used to specify requirements for one process on a data flow diagram, more precisely and in more detail than is possible using the data flow diagram alone.

If a system is represented using a single data flow diagram, then there should be a process specification for each one of the processes shown on the diagram.

The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm, mathematical equations, tables, or UML activity diagrams. The PSPEC is a "mini-specification" for each transform at the lowest refined level of a DFD.

A process specification should include the following components:

1. A *process name;*
2. A *process number*
3. A list of all the process's *inputs* and *outputs.*
4. All *assumptions* on inputs
5. All *error conditions*

Example:

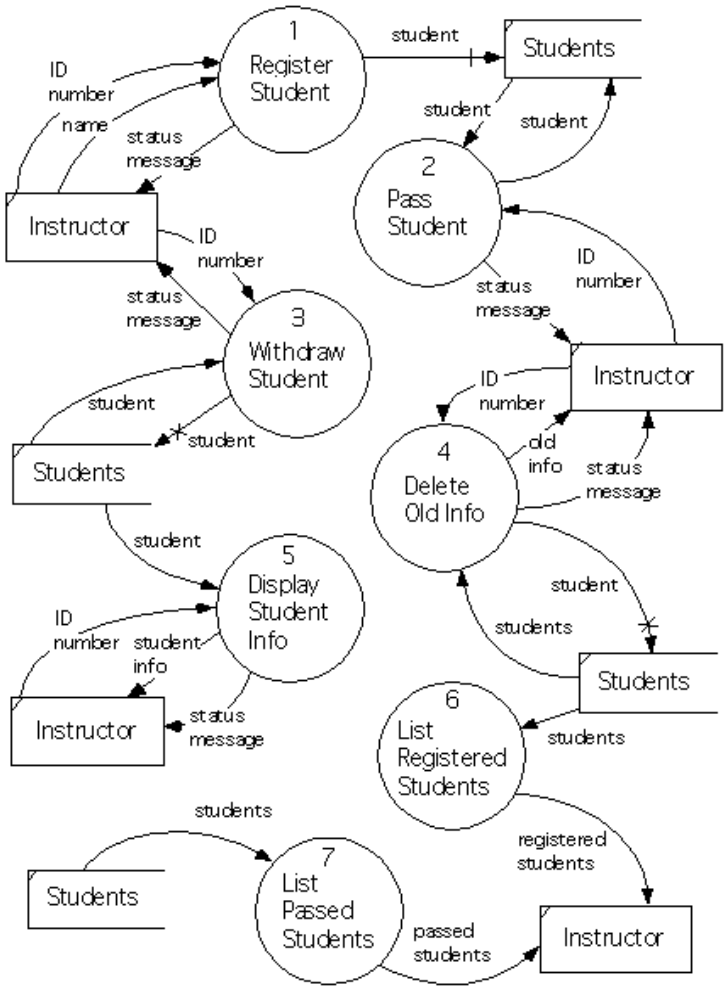


Figure: Student Information System

A process specification that corresponds to Process #1 in Student Information System might be as follows.

Process Name: Register Student
Process Number: 1
Inputs: ID number, name
Outputs: status message, student

Assumptions:

Data management subsystem is correct. That is, if it is provided with an instance of ``student" whose ID number is not already in use (and is asked to add it to the data table), then this will be added to the data table and no other change will be made. otherwise, an error message will be returned and no change will be made to the data table at all.

Error Conditions:

1. Either the ID number or name received from the Instructor (or both) is syntactically incorrect.
2. The data management subsystem reports an error on an attempt to add a new ``student," formed from the Instructor's input, to the system's data tables - effectively, signalling that the ID number is already in use.

Relationships between Inputs and Outputs:

1. ``student[ID number]" is the same as the input ``ID number"
2. ``student[first name]" is the same as ``name[first name]"
3. ``student[middle initial]" is the same as ``name[middle initial]"
4. ``student[last name]" is the same as ``name[last name]"
5. ``student[status]" is `registered'
6. ``status message" indicates that no errors were detected

Control Specification (CSPEC):

The Control Specifications (CSPEC) is used to indicate:

1. how the software behaves when an event or control signal is sensed and
2. which processes are invoked as a consequence of the occurrence of the event.

The control specification (CSPEC) contains a number of important modeling tools.

A control specification represents the behavior of the system (at the level from which it has been referenced) in two different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior.

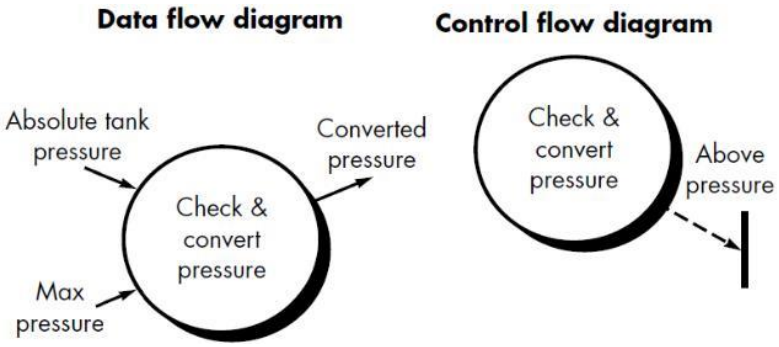
Input events						
Sensor event	0	0	0	0	1	0
Blink flag	0	0	1	1	0	0
Start/stop switch	0	1	0	0	0	0
Display action status						
Complete	0	0	0	1	0	0
In progress	0	0	1	0	0	0
Time out	0	0	0	0	0	1
Output						
Alarm signal	0	0	0	0	1	0
Process activation						
Monitor and control system	0	1	0	0	1	1
Activate/deactivate system	0	1	0	0	0	0
Display messages and status	1	0	1	1	1	1
Interact with user	1	0	0	1	0	1

Fig: Process activation table (PAT) for SafeHome security function

3.10.3 Brief Introduction to Control Flow Diagram

A **control flow diagram** helps us understand the detail of a process. It shows us where control starts and ends and where it may branch off in another direction, given certain situations.

The dashed arrow is once again used to represent control or event flow. Therefore, a *control flow diagram* contains the same processes as the DFD, but shows control flow, rather than data flow. The CFD shows how events move through a system. The *control specification* (CSPEC) indicates how software behaves as a consequence of events and what processes come into play to manage events.



Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various processes to be activated. The interrelationship between the process and control models is shown schematically in figure above.

3.11 Behavioral Modeling

Behavioral models are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1. ***Data.*** Some data arrives that has to be processed by the system.
2. ***Events.*** Some event happens that triggers system processing. Events may have associated data but this is not always the case.

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, our bookstore system will accept information about orders made by a customer, calculate the costs of these orders, and using another system, it will generate an invoice to be sent to that customer.

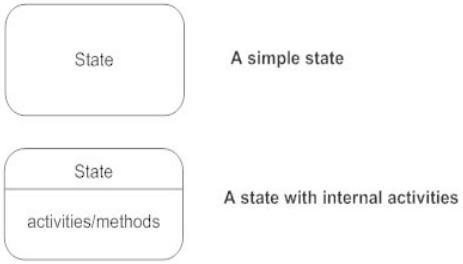
3.11.1 State Chart Diagram

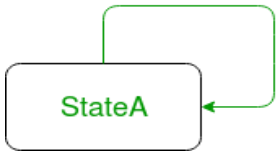
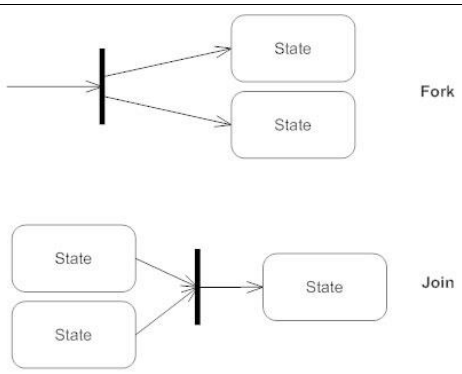
State chart diagram is one of the UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. State chart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

State chart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of State chart diagram is to model lifetime of an object from creation to termination.

State chart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Basic State Chart Diagram Symbols and Notations

Details	**Graphical notation**
States: States represent situations during the life of an object. You can easily illustrate a state by using a rectangle with rounded corners.	
Transition: A solid arrow represents the path between different states of an object. Label the transition with the event that triggered it and the action that results from it. A state can have a transition that points back to itself.	

Self transition: We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases	
Initial State: A filled circle followed by an arrow represents the object's initial state.	
Final State: An arrow pointing to a filled circle nested inside another circle represents the object's final state.	
Synchronization and Splitting of Control: A short heavy bar with two transitions entering it represents a synchronization of control. The first bar is often called a fork where a single transition splits into concurrent multiple transitions. The second bar is called a join, where the concurrent transitions reduce back to one.	

Examples:

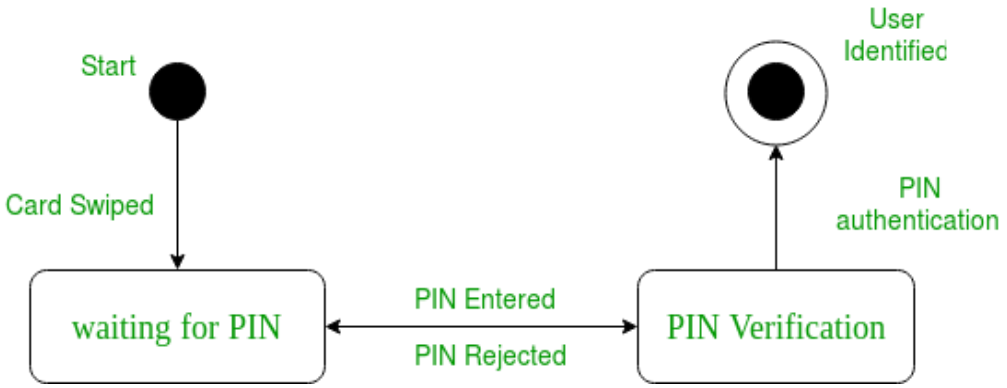


Figure: A state diagram for User Verification

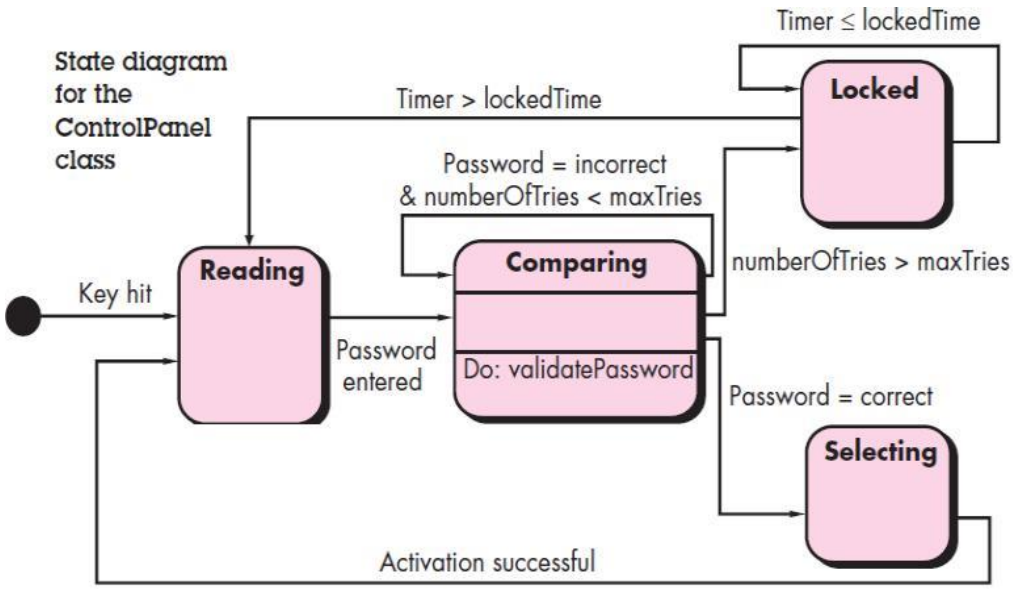


Figure: Sate Diagram for the ControlPanel class in SafeHome System

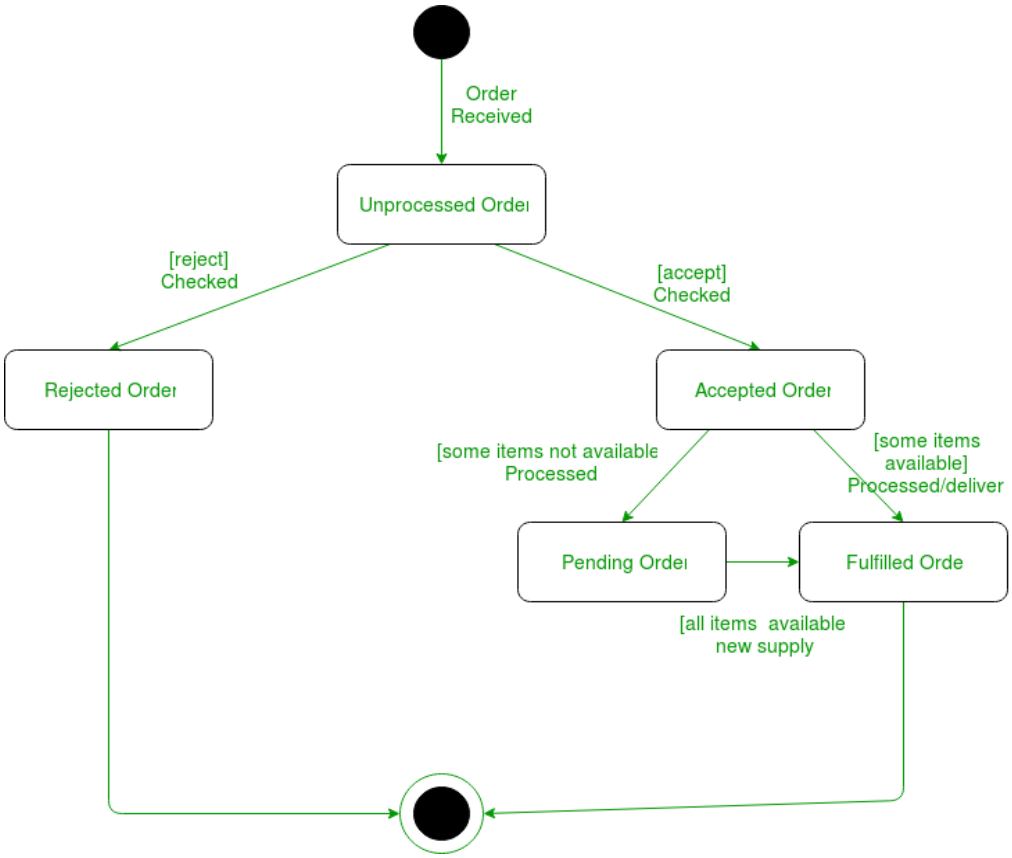
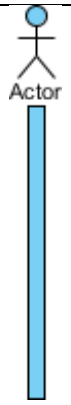
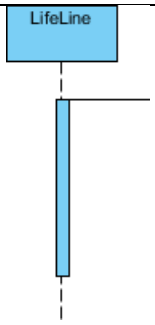
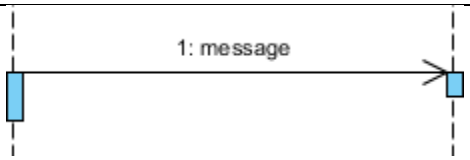


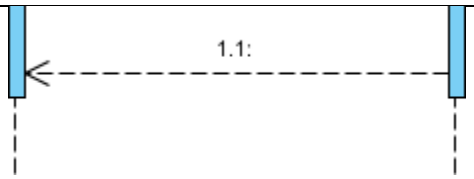
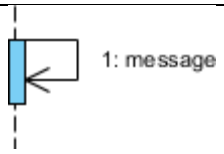
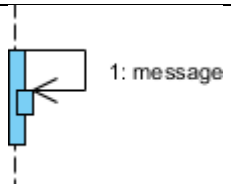
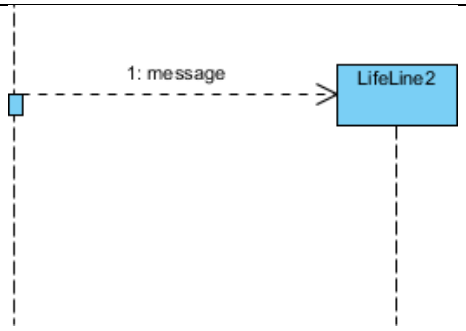
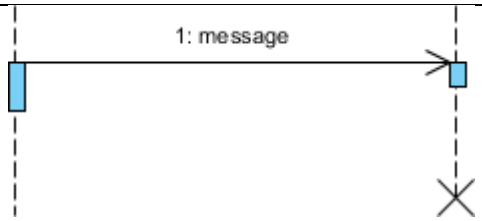
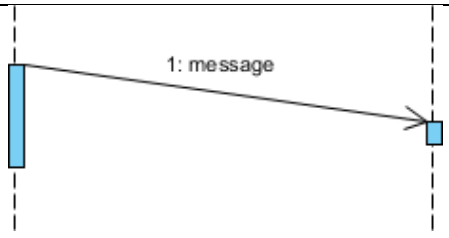
Figure: A State Diagram for an online order

3.11.2 Sequence Diagram

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

Sequence Diagram Notation

Notation Description	Visual Representation
Actor • a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data) • external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). • represent roles played by human users, external hardware, or other subjects. Note that: • An actor does not necessarily represent a specific physical entity but merely a particular role of some entity • A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.	
Lifeline • A lifeline represents an individual participant in the Interaction.	
Activations • A thin rectangle on a lifeline) represents the period during which an element is performing an operation. • The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively	
Call Message • A message defines a particular communication between Lifelines of an Interaction. • Call message is a kind of message that represents an invocation of operation of target lifeline.	

Return Message • A message defines a particular communication between Lifelines of an Interaction. • Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.	
Self Message • A message defines a particular communication between Lifelines of an Interaction. • Self message is a kind of message that represents the invocation of message of the same lifeline.	
Recursive Message • A message defines a particular communication between Lifelines of an Interaction. • Recursive message is a kind of message that represents the invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from.	
Create Message • A message defines a particular communication between Lifelines of an Interaction. • Create message is a kind of message that represents the instantiation of (target) lifeline.	
Destroy Message • A message defines a particular communication between Lifelines of an Interaction. • Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.	
Duration Message • A message defines a particular communication between Lifelines of an Interaction. • Duration message shows the distance between two time instants for a message invocation.	
Note • A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.	

Reference: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

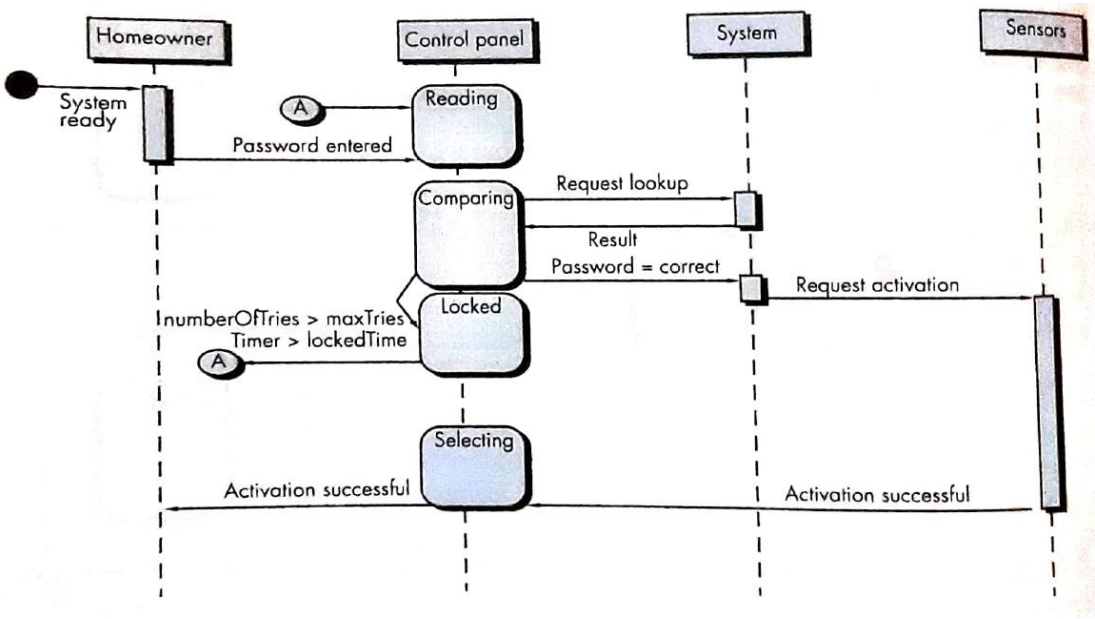
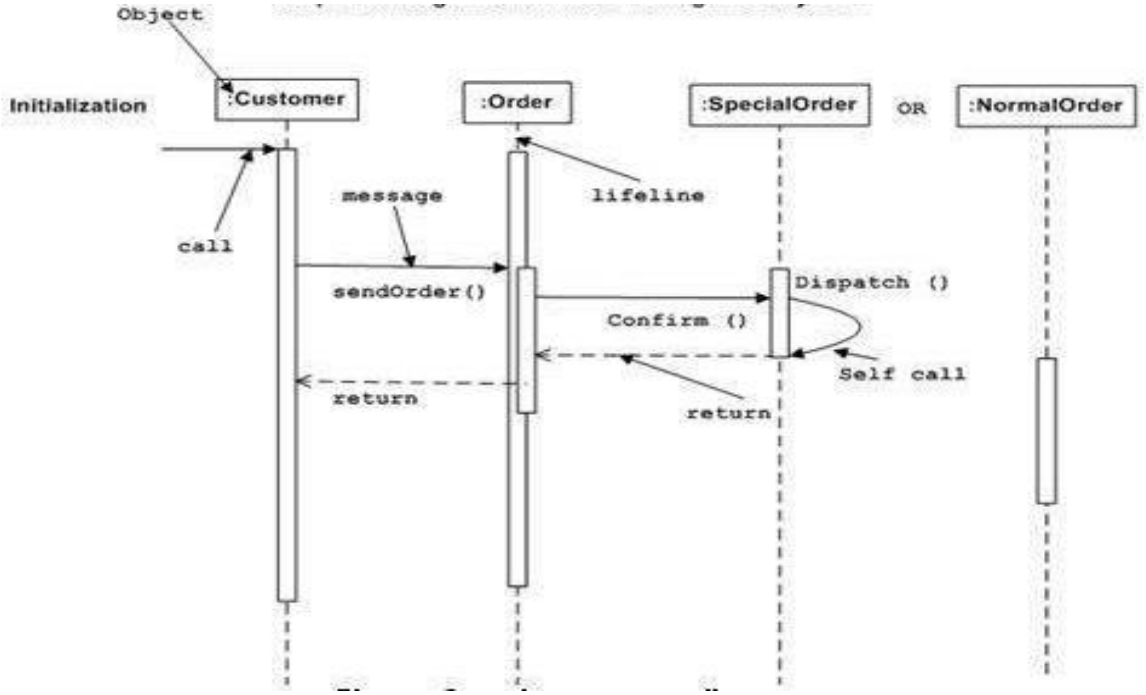


Figure: Sequence Diagram for the SafeHome Security function



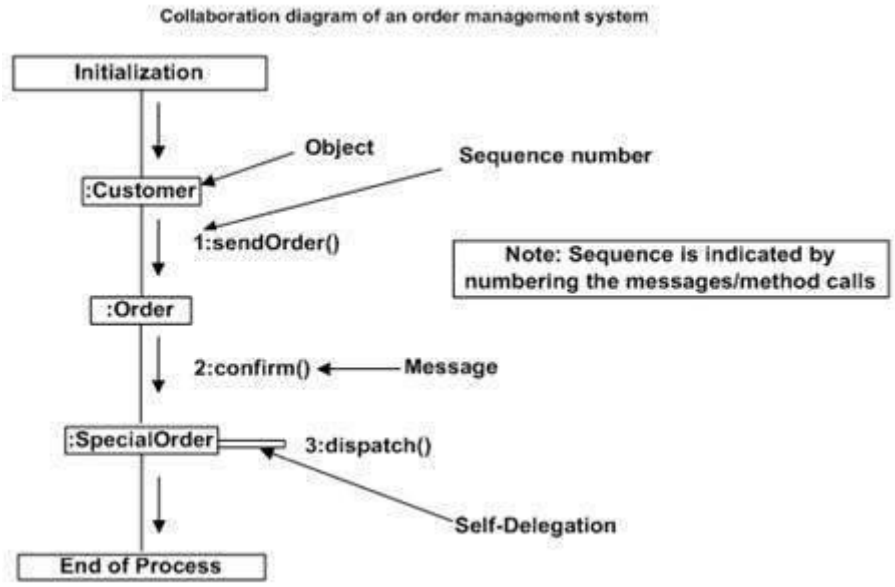
Reference: https://www.tutorialspoint.com/uml/uml_interaction_diagram.htm

Figure: Sequence Diagram of Order Management System

3.11.3 Communication or Collaboration Diagram

Collaboration diagrams (known as Communication Diagram in UML 2.x) are used to show how objects interact to perform the behavior of a particular use case, or a part of a use case. Along with sequence diagrams, collaboration are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case. They are the primary source of information used to determining class responsibilities and interfaces.

Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.



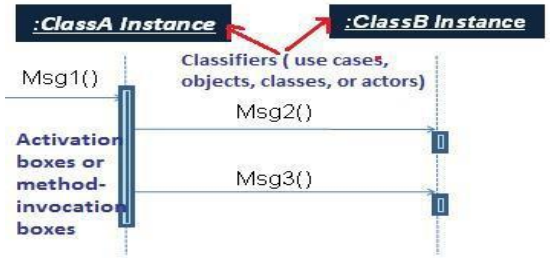
Reference: https://www.tutorialspoint.com/uml/uml_interaction_diagram.htm

Figure: Collaboration Diagram for Order Management System

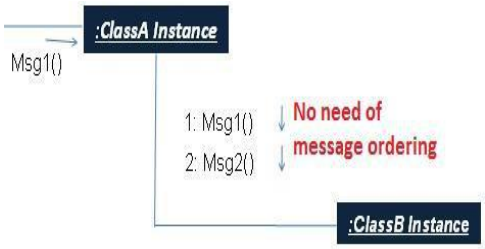
Sequence Diagram Vs Collaboration Diagram:

Sequence Diagram	Collaboration Diagram
1. It focuses on the dynamic aspect of an object. 2. It defines more clearly the chronological order of messages with respect to time on object from other object. 3. It only shows the message of one instant of time between the objects. 4. It has no provision to show all possible the relationship on an object.	1. It focuses on the structural aspect of one object to other. 2. It can define the total possible messages passed from one object to another but the order of message is not important. 3. It can show the total responsibility of an object. 4. It can show one or many possible relation of on objects.

5. For example:



5. For example:



3.12 Design Process

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction- a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Design is the place where software quality is established. The design process moves from a "big picture" view of software to a narrower view that defines the detail required to implement a system. The process begins by focusing on architecture. Subsystems are defined; communication mechanisms among subsystems are established; components are identified, and a detailed description of each component is developed. In addition, external, internal, and user interfaces are designed.

Generic task set for Design

1. Examine the information domain model, and design appropriate data structures for data objects and their attributes.
2. Using the analysis model, select an architectural style that is appropriate for the software.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture: Be certain that each subsystem is functionally cohesive. Design subsystem interfaces. Allocate analysis classes or functions to each subsystem.
4. Create a set of design classes or components: Translate analysis class description into a design class. Check each design class against design criteria; consider inheritance issues. Define methods and messages associated with each design class. Evaluate and select design patterns for a design class or a subsystem. Review design classes and revise as required.
5. Design any interface required with external systems or devices.
6. Design the user interface: Review results of task analysis. Specify action sequence based on user scenarios. Create behavioral model of the interface. Define interface objects, control mechanisms. Review the interface design and revise as required.
7. Conduct component-level design. Specify all algorithms at a relatively low level of abstraction. Refine the interface of each component. Define component-level data structures. Review each component and correct all errors uncovered.
8. Develop a deployment model.

For more reference:

https://www.geeksforgeeks.org/software-engineering-software-design-process/#:~:text=The%20design%20phase%20of%20software,Architectural%20Design

3.13 Design Concepts

Concepts are defined as a principal idea or invention that comes in our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system or software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:

Figure: Software Design Concepts

1. **Abstraction - hide relevant data**

   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broadways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.

2. **Architecture - design a structure of something**

Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.

3. **Patterns - a repeated form**

The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

4. **Separation**

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behavior that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve. Separation of concerns is expressed in other related design concepts: modularity, aspects, functional independence, and refinement.

5. **Modularity - subdivide the system**

Modularity is the single attribute of software that allows a program to be intellectually manageable. Software is divided into separately named and addressable components, sometimes called *modules*, which are integrated to satisfy problem requirements. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

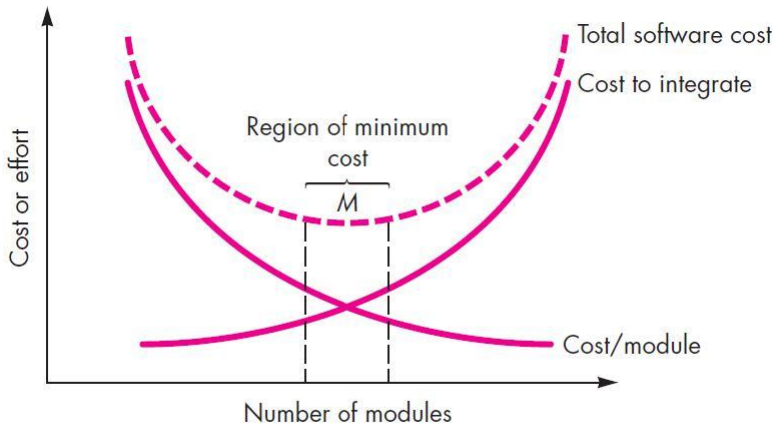


Fig. Modularity and software cost

6. **Information Hiding - hide the information**

Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and it cannot be accessed by any other modules.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

7. Functional Independence- Coupling and cohesion

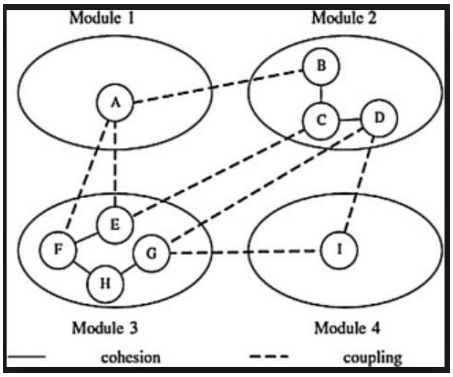
The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding. The functional independence is accessed using two criteria i.e Cohesion and coupling.

Cohesion:

Cohesion is an extension of the information hiding concept. A cohesive module performs a single task and it requires a small interaction with the other components in other parts of the program.

Coupling:

Coupling is an indication of interconnection between modules in a structure of software. Ideally, no coupling is considered to be the best. The lower the coupling, the better the program is.



8. Refinement - removes impurities

Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

9. Aspects – crosscutting concerns

An *aspect* is a representation of a crosscutting concern. A crosscutting concern is some characteristic of the system that applies across many different requirements. Consider two requirements, *A* and *B*. Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account".

An aspect is implemented as a separate module (component) rather than as software fragments that are "scattered" or "tangled" throughout many components. To accomplish this, the design architecture should support a mechanism for defining an aspect—a module that enables the concern to be implemented across all other concerns that it crosscuts.

10. Refactoring - reconstruct something

Refactoring simply means to reconstruct something in such a way that it does not affect the behavior or any other features. Refactoring in software design means to reconstruct the design to reduce and complexity and simplify it without affecting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't affect the behavior of the design and improves the internal structure".

For more reference: https://www.tutorialride.com/software-engineering/software-process-designing-concepts.htm
https://www.geeksforgeeks.org/introduction-of-software-design-process-set-2/
https://study.com/academy/lesson/design-concepts-in-software-engineering-types-examples.html

3.14 Design Model

The design model can be viewed in two different dimensions as illustrated in figure.

- The *process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process.
- The *abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.
- The dashed line indicates the boundary between the analysis and design models.

The elements of the design model use many of the same UML diagrams that were used in the analysis model. These diagrams are refined and elaborated as part of design.

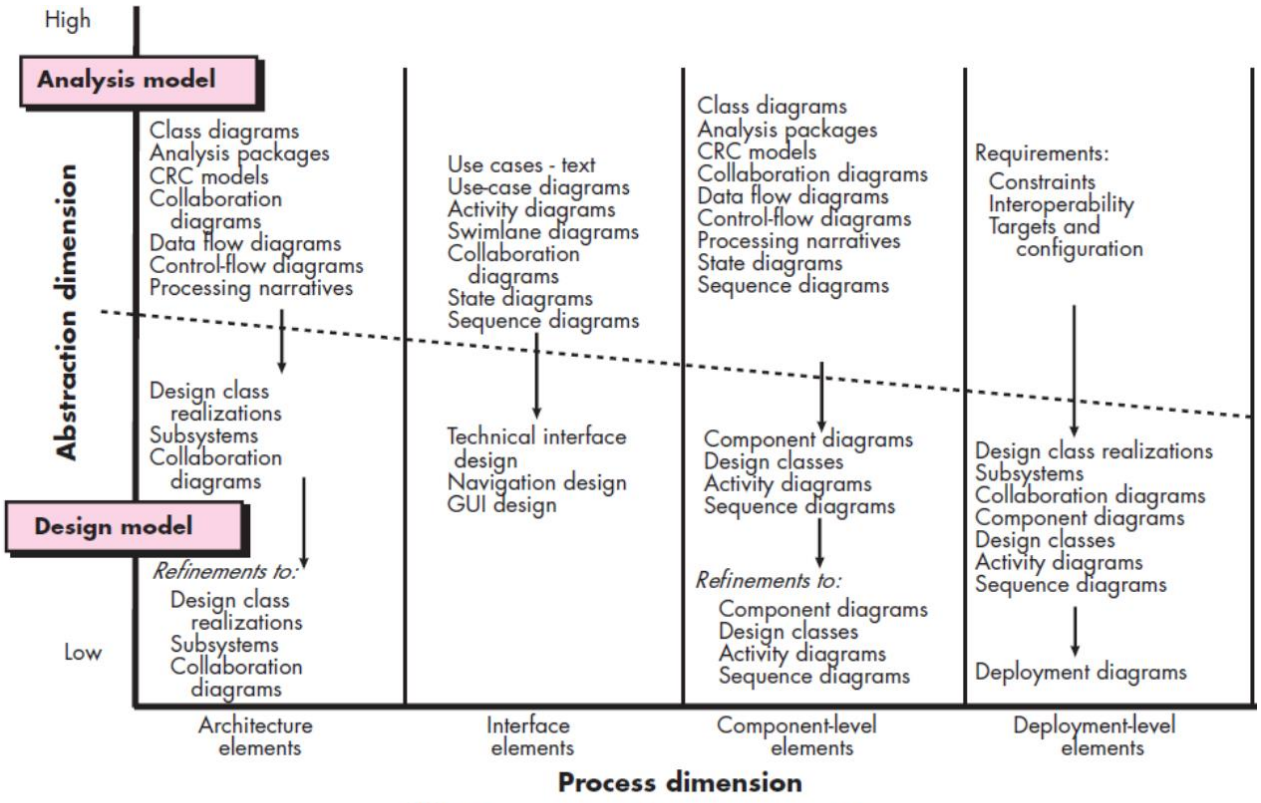


Fig. Dimensions of the design model

Object Oriented Analysis (OOA)	Object Oriented Design (OOD)
• **Analysis:** Analysis emphasizes an investigation of the problem and requirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used?	• **Design:** Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented.
• **Object-oriented analysis:** During **object-oriented analysis,** there is an emphasis on finding and describing the objects—or concepts—in the problem domain. o For example, in the case of the library information system, some of the concepts include Book, Library, and Patron. o Investigation of the requirements, concepts, and operations related to a system.	• **Object-oriented design:** During object-oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements. For example, in the library system, a Book software object may have a title, attribute and a getChapter () method o Designing a solution for this iteration in terms of collaborating software objects.
• **Essential use cases:** They are created during early stage of eliciting the requirements and independent of the design decisions. For example: ▪ **Actor action:** Cashier record identifier for each item. If there is more than one of the same, the cashier enters quantity. ▪ **System response:** Determine the item price and add the item info to the running sales transaction description price of the current item in item presented.	• **Real use cases:** Make the essential use case more concrete by specifying the particular technology and methods. They are developed only after the design decisions have been made. For example: o **Actor action:** For each item cashier types UPC. In the UPC input field of window then they press "enter item" button with the mouse or keyboard. o **System response:** Display item price and add the item information to the running sales transaction the description and price of current item are displayed.

Design Elements of Design Model

Following are the types of design elements:

1. Data design elements
2. Architectural design elements
3. Interface Design elements
4. Component level Design elements
5. Deployment-level Design elements

Reference for short explanation: https://www.tutorialride.com/software-engineering/software-design-model-elements.htm

3.14.1 Data Design (aka Data Architecting)

Data design creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

3.14.2 Architectural Design

As the various architecture of building construction, the software that is built for computer-based systems also exhibits one of many architectural styles. Each style describes a system category that encompasses

1. A set of components (e.g., a database, computational modules) that perform a function required by a system;
2. A set of connectors that enable "communication, coordination and cooperation" among components;
3. Constraints that define how components can be integrated to form the system; and
4. Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Patterns can be used in conjunction with an architectural style to shape the overall structure of a system. However, a pattern differs from a style in a number of fundamental ways: (1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety; (2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency); (3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).

Architectural Styles

Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:

a) Data-centric Architecture

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure illustrates a typical data-centered style.

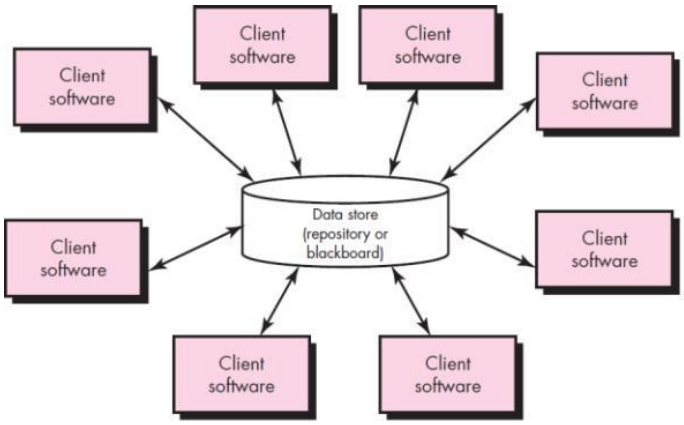


Fig. Data-centered architecture

b) Data-flow Architecture

This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form.

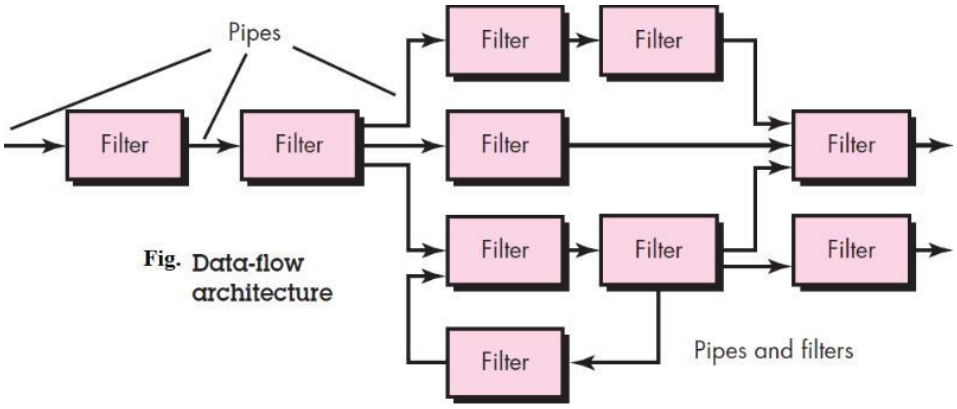


Fig. Data-flow architecture

c) Call and Return Architecture

This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub-styles exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure illustrates an architecture of this type.
- *Remote procedure call architectures.* The components of a main program/subprogram architecture are distributed across multiple computers on a network.

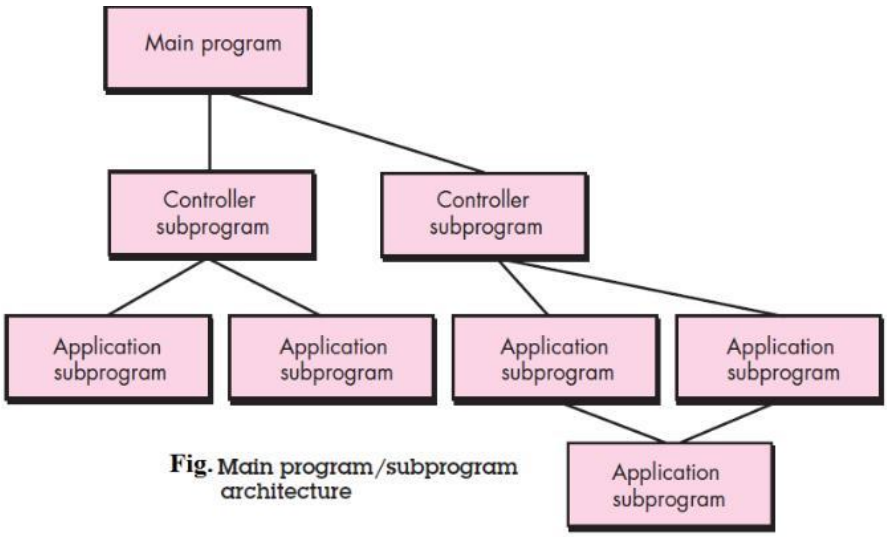


Fig. Main program/subprogram architecture

d) Object-oriented Architecture

The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication an coordination between components are accomplished via message passing.

e) Layered Architecture

A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

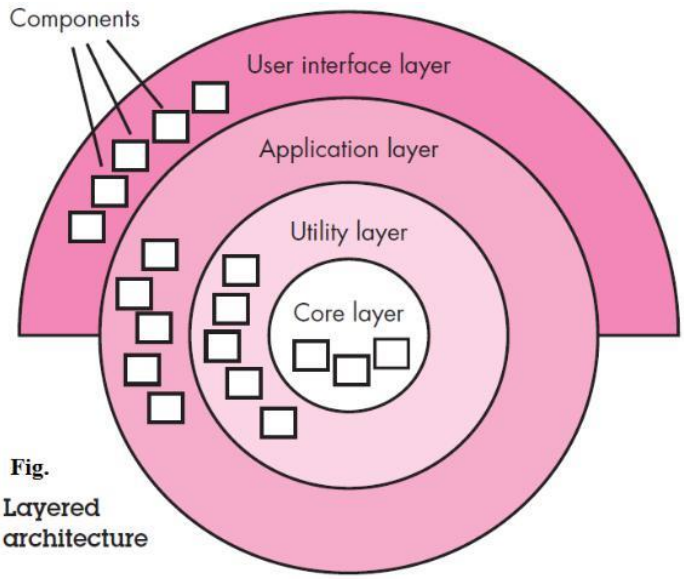


Fig. Layered architecture

3.14.3 Interface Design

The interface design for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture. There are three important elements of interface design:

(1) the user interface (UI);

(2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and

(3) internal interfaces between various design components.

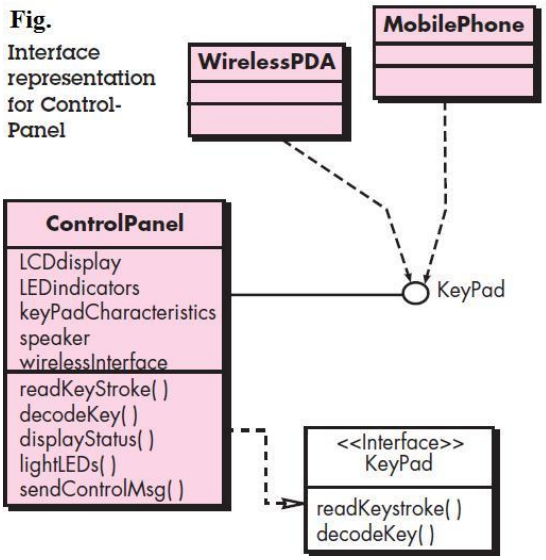
These interface design allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

1. UI Design

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype. Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

For example, the *SafeHome* security function makes use of a control panel that allows a homeowner to control certain aspects of the security function. In an advanced version of the system, control panel functions may be implemented via a wireless PDA or mobile phone. The interface representation for such system can be shown in figure.



2. Interface Design Steps

User interface design begins with the identification of user, task, and environmental requirements. Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step. Although many different user interface design models have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis, define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

3. Interface Design Evaluation

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users. The user interface evaluation cycle takes the form shown in figure.

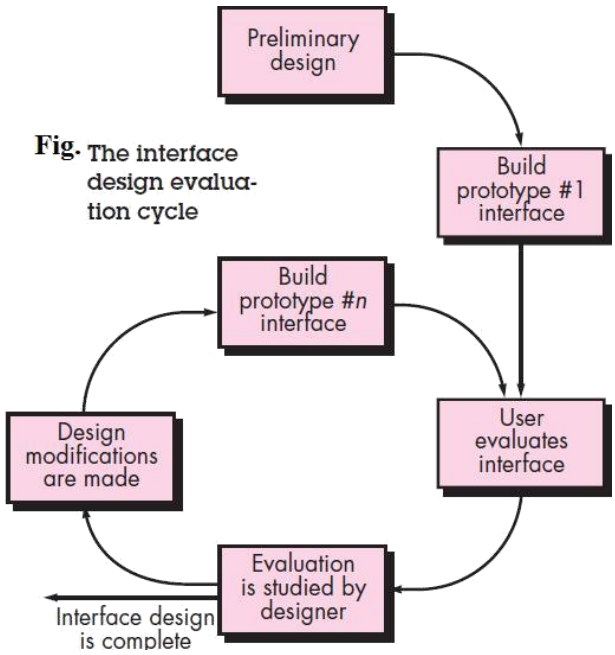


Fig. The interface design evaluation cycle

3.14.4 Component Level Design

A *component* is a modular building block for computer software that encapsulates implementation and exposes a set of interfaces. The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

The *component-level design* for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).

Component Diagram

The component diagram is used to model the physical aspects of an OO system. It shows the software components of a system & how they are related to each-other. Thus, they show the organization and dependencies between a set of components. Component diagrams are related to class diagrams in that a component *typically maps to one or more classes, interfaces, or collaborations.* Use component diagrams to model the *static implementation view* of a system. This involves modeling the physical things that reside on a node, such as executable, libraries, tables, files, and documents.

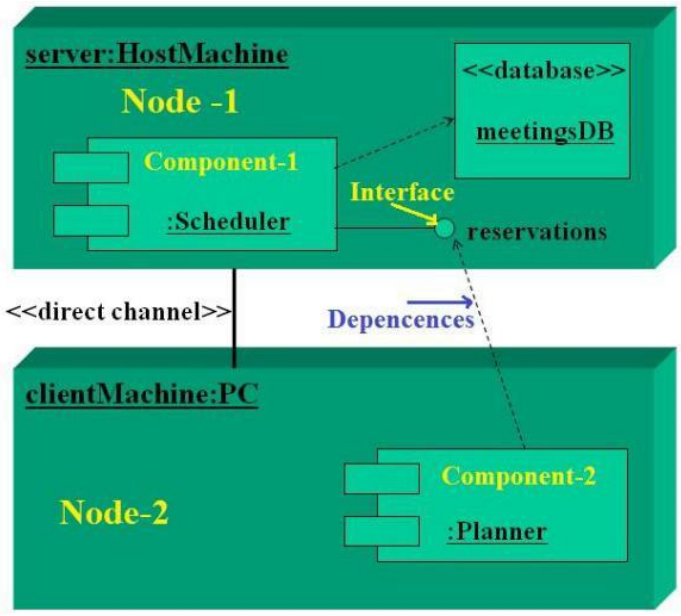
3.14.5 Deployment Level Design

Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software. Deployment diagrams begin in descriptor form, where the deployment environment is described in general terms. Later, instance form is used and elements of the configuration are explicitly described.

Deployment Diagram

The deployment diagram is also used to model the physical aspects of an OO system. A *deployment diagram* shows a *set of nodes and their relationships.* They show the configuration of *run-time processing* nodes and the components that live on them. Use deployment diagrams to model the *static deployment view* of a system or architecture.

This involves modeling the topology of the hardware on which the system executes. Deployment diagrams are *related to component diagrams* in that a node *typically encloses one or more components.*



3.15 Brief Introduction to Design Patterns

Design patterns provide a codified mechanism for describing problems and their solution in a way that allows the software engineering community to capture design knowledge for reuse. A pattern describes a problem, indicates the context enabling the user to understand the environment in which the problem resides, and lists a system of forces that indicate how the problem can be interpreted within its context and how the solution can be applied.

Coplien characterizes an effective design pattern in the following way:

- *It solves a problem*: Patterns capture solutions, not just abstract principles or strategies.
- *It is a proven concept*: Patterns capture solutions with a track record, not theories or speculation.
- *The solution isn't obvious*: Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.
- *It describes a relationship*: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- *The pattern has a significant human component (minimize human intervention).* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

Board Exam Questions:

1. Prepare level-1 DFD for the following food ordering system. [2019 Spring]

   A potential patient joins the doctors by submitting a patient application form. A new patient record is created and stored in the patient records store. A patient makes an appointment by providing their patient details. An appointment card is given to the patient after they have made the appointment. The appointment details are stored in the database.

   A receptionist makes a telephone appointment for a patient by entering a patient detail. A receptionist also cancels appointments for a patient by entering their cancelation details. Both processes update the appointment section of the database.

   A doctor will see a patient. When they see a patient a list of appointments and patients record will be sent to the doctor. A doctor may want to issue a prescription by entering prescription details into the system and a prescription be issued to the patient.

2. Draw a use case diagram from the given case study. [2019 Spring]

   In hospital, a patient goes to registration machine. He presses the ON button then the screen opens. He enters patient ID number. He books for the doctor for checkup. He checks the category of disease from given list, then he chooses the doctor's name from given doctor's name list, he enters time he wants to meet with doctor. For this registration, he needs to enter the amount. If the amount digit is ok it accepts the registration and prints the registration slip, otherwise it will give a signal of alarm "insufficient amount".

3. What is modularity? Differentiate between sequence and communication diagram with regards to the strength and weakness with example. [2019 spring]

4. What is state chart diagram? Draw a sequence diagram for the given scenario: [2019spring]

   A customer wants to draw money from his bank account. He enters his card into and ATM. The ATM machine prompts "Enter pin". The customer enters his PIN. The ATM (internally retrieves the bank account number from the card). The ATM encrypts the pin and the account number and sends it over to the bank. The bank verifies the encrypted account and pin number. If the pin number is correct, the ATM displays "enter amount", draw money from bank account and pays out the amount.

5. what is requirement analysis? Explain different elements of requirement analysis model. [2019 fall]

6. What do you mean by Scenario-based modeling and Behavioral modeling? Explain with examples. [2019 fall, 2018 fall]

7. Obtain 1-level DFD for the Food Ordering System: [2019 fall]

   Customer can place an *Order of the food*. The *Order Food* process receives the *Order*, forwards it to the *Kitchen*, store it in the *Order* data store, and store the updated *Inventory details* in the *Inventory* data store. The process also delivers a *Bill* to the Customer, *Manger* can receive *Reports* through the *Generate Reports* process, which takes *Inventory details* and *Orders* as input form the *Inventory* and *Order* data store respectively. *Manager* can also initiate the *Order Inventory* process by providing *Inventory Order*. The process forwards the *Inventory order* to the *Supplier* and stores the updated *Inventory details* in the *Inventory* data store.

8. What do you mean by design model? What are the elements of effective Interface design? Explain the evaluation of the user interface with evaluation cycle diagram. [2019 fall]
9. Demonstrate use of Scenario based testing for thread testing with suitable example. [2019 fall]
10. Draw a detailed use-case diagram for the following case study: [2018 spring]

    Case study:

    A customer visits online shopping portal. A customer may buy item or just visit the page and logout. The customer can select a segment, then a category and brand to get different products in the desired band. The customer cam select product for purchasing. The process can be repeated for more items. Once the customer finishes selecting the product/s, the cart cane be viewed. If the customer wants to edit the final cart it can be done here. For final payment, the customer has to login to portal. If the customer is visiting for the first time, he must register with the site, else the customer use login page to proceed. Final cart is submitted for payment and card details and address details are to be confirmed with customer. Customer is confirmed with the shipment id and delivery if goods within 15 days.
11. What is Sequence diagram? What are the elements used in sequence diagram? Explain each. [2018 spring]
12. A restaurant uses an Information system that takes customer orders, send the order to the kitchen monitors: the goods sold and inventory and generates reports for management. List functional and Non-functional requirements for this Restaurant Information System. [2018 fall]
13. Obtain the use case diagram for Library Management System. [2018 fall]
14. Draw the different levels of DFD for Safe Home System where any person can enter to the home on matching his/her password at the entrance door. [2018 fall]
15. Suppose that Pokhara University is going to develop Online Learning System (OLS). You have been asked by the university to mention four major requirements definition for OLS. List the four requirements definition and explain their requirement specifications. [2017 spring]
16. Draw the Context Level Data Flow Diagram (DFD) and Level 1 DFD for the above OLS. [2017 spring]
17. What do you mean by software design? Briefly explain different steps that represent a typical task set for component-level design. [2017 spring]
18. Briefly explain different categories of class. Describe the guidelines for allocating responsibilities to classes. [2017 spring]
19. Why User interface design is important in software development? Referencing a mobile application for smart agriculture, describe user interface design issues.
20. What do you mean by the functional and non-functional requirements? Give any four functional and non-functional requirements for computer based online hotel reservation system. [2017 fall]
21. Obtain the DFD for Library Management System [2017 fall]
22. What is UML diagram? Explain briefly about state chart diagram, sequence diagram and collaboration diagram with example. [2017 fall]

23. Draw the Class diagram and activity diagram for an automated teller machine. [2017 fall]
24. How eliciting requirement is difficult task? Explain the basic guidelines of collaborative requirements gathering. [2016 spring]
25. Explain how Use-case diagram and CRC modeling aids in object-oriented analysis. [2016 spring]
26. Obtain 1-level DFD for the following system of encashing cheque in a bank. A customer presents a cheque to a clerk. The clerk checks the ledger containing all account numbers and make sure whether the account number in the cheque is valid, whether adequate balance is there in the account to pay the cheque, and whether the clerk also debits customer's account by the amount specified on the cheque. If cash cannot be paid due to an error in the cheque, the cheque is returned. The token number is written on the top og the cheque and it is passed on to the cashier. The cashier calls out the token number, takes the customer's signature, pays cash, enters cash paid in ledger called day book, and file the cheque. [2016 spring, 2014 fall]
27. Design a class diagram for class Books which has following attributes and operations:

    Attributes: name, Author Name, ISSN Number, Price

    Operations: to take value for the above data, to search a book by its ISSN and to display the details of a book. [2016 spring]
28. Why system modeling is essential for software development? What are the difference between data an behavior modeling? [2016 fall]
29. Discuss the notations of UML diagram in detail. [2016 fall]
30. A simple system is to be developed to support the management of exercises completed by students taking a course. Students first meet with course tutor to register for a course, and then during the course they submit a number of exercises. Every course has a certain deadline assigned by the course tutor. Tutors can allow an exercise to be submitted late. At any point, a student can find out from the system the marks they have received for any exercises already completed. A student shall also be able to view any comments made by the tutor on certain exercise. The course tutor can also enter a mark for an exercise, and print out a summary of the marks gained by all students on course.

    Identify classes and draw a class diagram to model an efficient solution for the problem.
31. A customer presents a cheque to a clerk. The clerk checks the ledger containing all account numbers and make sure whether the account number in the cheque is valid, whether adequate balance is there in the account to pay the cheque, and whether the signature is authentic. Having done these, the clerk gives the customer a token. The clerk also debits customer's account by the amount specified on the cheque. If cash cannot be paid due to an error in the cheque, the cheque is returned. The token number is written on the top of the cheque and it is passed on to the cashier. The cashier calls out the token number, takes the customer's signature, pays cash, enters cash paid in ledger called day book, and file the cheque.

    Derive use-cases from the above scenario and model them into a Use-case diagram. [2016 fall]
32. What do you mean by object-oriented design model? Discuss concurrency and subsystem allocation. [2016 fall]
33. Obtain 1-level DFD for Movie Management System: [2015 spring]

A customer can book a ticket from the Internet or can directly buy the ticket in the Movie-hall itself. There can be multiple halls within one movie theatre. The ticket operator provides a ticket with hall's stamp after checking the booking information to the customer. The guard in each hall validates the ticket and provides access to the customer inside the hall. There is also provision of complementary food item which the café will provide in the break time of the movie.

34. From the same above-mentioned case, obtain a use-case diagram also. [2015 spring]
35. Define requirement engineering. What are various methods of requirement elicitation? Explain. [2015 spring]
36. What is difference between class diagram and object diagram? Draw class diagram for the following scenario: [2015 spring]

    A bank provides Debit card service for its customers only for Current account and saving account. Customers can perform: withdraw, transfer, balance query and pin change operations. Customers can use ATM of any location.

37. Obtain DFD for the following system: [2015 fall]

    A customer can book a ticket from the Internet or can directly buy the ticket in the Movie-hall itself. There can be multiple halls within one movie theatre. The ticket operator provides a ticket with hall's stamp after checking the booking information to the customer. The guard in each hall validates the ticket and provides access to the customer inside the hall. There is also provision of complementary food item which the café will provide in the break time of the movie.

38. Discuss the various steps in Interface design. How is it evaluated? [2015 fall]
39. Suppose you want to develop software for an alarm clock. The clock shows the time of day. Using buttons, the user can set the hours and minutes fields individually, and choose between 12 and 24-hour display. It is possible to set one or two alarms. When an alarm fires, it will sound some noise. The user can turn it off, or choose to snooze. It the user does not respond at all, the alarm turns off itself after 2 minutes. Snoozing means to turn off the sound, but the alarm will fire again after some minutes of delay. This snoozing time is pre adjusted. Draw use case for this system. [2016 fall]
40. What do you mean by design patterns? What is the importance of incorporating reuse in a project? List out its major advantages. [2015 fall]
41. Draw a UML class diagram representing the following elements from the problem domain for a hockey league. A hockey league s made up of at least four hockey teams. Each hockey team is composed of 6 to 12 players, and one player captains the team. A team has a name and a record. Players have a number and a position. Hockey team play games against each other. Each game has a score and a location. Teams are sometimes led by a coach. A coach has a level of accreditation and a number of years of experience, and can coach multiple teams. Coaches and players are people, and people have names and addresses. Draw a class diagram for this information, and be sure to label all associations with appropriate multiplicities. [2015 fall]
42. Obtain DFD for the following Mess management system: [2014 Spring]

    A hostel has 500 rooms and 4 messes. Currently, there are 1000 students in all in 2 seated rooms. They eat in any of the messes but can get rebate if they inform and do not eat for at least 4 consecutive days. Besides normal menu, extra items are also given to students when they ask for it. Such extras are entered in an extra book. At the end of the month, a bill is prepared based on the

normal daily rate and extras and given to each student. System for stores issue and control is maintained for daily use of perishables and non-perishables items and order to vendor and suppliers are also maintained as well.

43. Why is Software modeling essential? Which modeling do you prefer to use if you are allowed to choose between data modeling and class base modeling? [2014 Spring]

44. Draw ERD for the following situation: [2014 Spring]

    An accountant is a relationship between customer and bank. A customer has name. a bank has a branch. A customer may have several accountants of different types and balances.

45. Discuss about scenario based modeling. Differentiate between Use-Case modeling and Activity diagram. Provide examples as well. [2014 Spring]

47. A health clinic provides medical services to patients in a small town. Five doctors and three nurses work at the clinic. They consult with patients, prescribe medicines and carry out minor medical treatments. Patients with more serious conditions are referred to specialists at the local hospital. A medical information system is being designed for the use in the clinic. The system will manage information about employees (doctors, nurses and administrator), patients and their contact details, appointments and consultations, medicines and prescriptions, treatments given, and referrals.

    Produce a UML class diagram or use in constructing the system using an object-oriented programming language. Your diagram must include all applicable classes and relationships. There is no need to show the attributes and operations for each class. [2014 Fall]

48. Discuss briefly on how Use-case and CRC aids in object-oriented analysis. [2014 Fall]

49. Describe the concurrency and subsystem allocation for object-oriented design. [2014 Fall]

50. A library maintains a collection of books. The information about all book is kept in a database. Th information about users is kept in another database. A user of the library can borrow a book, return a book and reserve a book. Assume that there is no limit to the number of books a user can borrow, develop a data flow diagram for this problem. [2013 spring]

51. Discuss the need of UML for modern software engineering. Explain briefly about sequence diagram and collaboration diagram with example. [2013 spring]

52. Draw a sequence diagram of the following: [2013 spring]

    A valid user can login to the online shopping system. The user can order the items and as well as pay the Credit card. The items will be delivered later. The items within 15 days will be reimbursed.

53. Discuss the various types of Use-case for Safe Home System where any person can enter to the home on matching his/her password at the entrance door. [2013 spring]

54. What is Class Responsibility Collaborator model? How do you develop CRC model? Explain with example. [2013 spring]

55. Compare and contrast the OOA and OOD in detail. How these help to implementation? Explain. [2013 spring]

56. Explain the significance of the notations of the UML in object-oriented analysis and design. [2013 fall]

57. How can classes and objects and their relationship be identified for a particular system? [2013 fall]

58. How can the collaboration between objects be elaborated during the object-oriented design phase? [2013 fall]

59. Write short notes on:

a) Functional vs Non-functional requirements [2019 spring, 2019 fall]
b) Requirement engineering [2019 spring]
c) Data design [2018 fall]
d) Domain and reuse analysis [2016 spring]
e) Modularity [2016 spring]
f) Data modeling [2016 fall]
g) Function independence [2015 fall]
h) Deployment level design [2015 fall]
i) CRC modeling [2014 fall]
j) Polymorphism, Inheritance and Abstraction in OO concept [2014 fall]

***

Object Oriented Software Engineering

Chapter 4

QUALITY MANAGEMENT

&

TESTING

Er. Shiva Ram Dam

Shivaram.dam@pec.edu.np

4.1 Quality Concepts

Quality is a complex and multifaceted concept that can be described from five different points of view:

- *Transcendental view:* something that immediately recognizes, but cannot explicitly define.
- *User view:* If a product meets an end user's specific goals, it exhibits quality
- *Manufacturer's view:* Product conforms to the original specification.
- *Product view:* Quality can be tied to inherent characteristics (e.g., functions and features) of a product.
- *Value-based view:* How much a customer is willing to pay for a product?

Thus, Quality can be defined as: Degree of excellence or Fitness for purpose or Best for the customer's use and selling price or Total characteristics of an entity that bear on its ability to satisfy stated or implied needs (ISO).

Quality of design refers to the characteristics that designers specify for a product. The grade of materials, tolerances, and performance specifications all contribute to the quality of design. As higher-grade materials are used, tighter tolerances and greater levels of performance are specified. *Quality of conformance* focuses on the degree to which the implementation follows the design and the resulting system meets its requirements and performance goals.

Software Quality:

Software quality can be defined as: *An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.* The definition serves to emphasize three important points:

- **Effective software process**: establishes the infrastructure that supports any effort at building a high-quality software product.
- **Useful product:** delivers the content, functions, and features that the end user desires in a reliable, error-free way.
- **Added measurable values:** For producer: high-quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- **For end user:** (1) greater software product revenue, (2) better profitability when an application supports a business process, and/or (3) improved availability of information that is crucial or vital for the business.

To achieve high-quality software, four activities must occur: proven software engineering process and practice, solid project management, comprehensive quality control, and the presence of a quality assurance infrastructure.

4.1.1 Quality Factors

The various factors, which influence the quality of software, are termed as software quality factors. They can be broadly divided into two categories. The first category of the factors is of those that can be measured directly such as the number of logical errors, and the second category clubs those factors which can be

measured only indirectly. For example, maintainability but each of the factors is to be measured to check for the content and the quality control.

Several models of software quality factors and their categorization have been suggested over the years. The classic model of software quality factors, suggested by McCall, consists of 11 factors (McCall et al., 1977).

McCall's Factor Model

This model classifies all software requirements into 11 software quality factors. The 11 factors are grouped into three categories – product operation, product revision, and product transition factors.

- Product Operation category includes five software quality factors, which deal with the requirements that directly affect the daily operation of the software.
- Product Revision category deals with maintainability, flexibility and testability.
- Product Transition category deals with the adaptation of software to other environments and its interaction with other software systems.

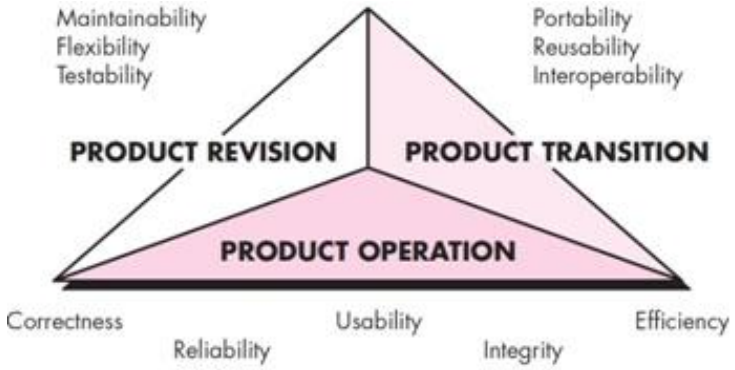


Fig. McCall's software quality factors

1. **Correctness**

   Correctness deals with the extent to which a program satisfies to specification (requirement) and fulfils the customer's mission or objectives. The requirements deal with the correctness of the software system.

2. **Reliability**

   Reliability requirements deal with service failure. They determine the maximum allowed failure rate of the software system, and can refer to the entire system or to one or more of its separate functions. Reliability deals with the extent to which a program can be expected to perform its intended function with required precision.

3. **Efficiency**

   It deals with the hardware resources needed to perform the different functions of the software system. It includes processing capabilities (given in MHz), its storage capacity (given in MB or GB) and the data communication capability (given in MBPS or GBPS).

4. **Integrity**

   This factor deals with the software system security, that is, to prevent access to unauthorized persons, also to distinguish between the group of people to be given read as well as write permit.

5. **Usability**

   Usability requirements deal with the staff resources needed to train a new employee and to operate the software system.

6. **Maintainability**

   This factor considers the efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of the corrections.

7. **Flexibility**

   This deals with the effort required to modify an operational program. This factor deals with the capabilities and efforts required to support adaptive maintenance activities of the software. These include adapting the current software to additional circumstances and customers without changing the software.

8. **Testability**

   Testability requirements deal with the testing of the software system as well as with its operation. It includes predefined intermediate results, log files, and also the automatic diagnostics performed by the software system prior to starting the system, to find out whether all components of the system are in working order and to obtain a report about the detected faults.

9. **Portability**

   This deals with transferring of a software from one hardware to another. Portability requirements tend to the adaptation of a software system to other environments consisting of different hardware, different operating systems, and so forth. The software should be possible to continue using the same basic software in diverse situations.

10. **Reusability**

    This factor deals with the use of software modules originally designed for one project in a new software project currently being developed. They may also enable future projects to make use of a given module or a group of modules of the currently developed software. The reuse of software is expected to save development resources, shorten the development period, and provide higher quality modules.

11. **Interoperability**

    Interoperability requirements focus on creating interfaces with other software systems or with other equipment firmware. For example, the firmware of the production machinery and testing equipment interfaces with the production control software.

Similarly, models consisting of 12 to 15 factors, were suggested by Deutsch and Willis (1988) and by Evans and Marciniak (1987). All these models do not differ substantially from McCall's model. The McCall factor model provides a practical, up-to-date method for classifying software requirements (Pressman, 2000).

The ISO 9126 standard Quality Factors:

The ISO 9126 standard was developed in an attempt to identify the key quality attributes for computer software. The standard identifies six key quality attributes:

1. Functionality
2. Reliability
3. Usability
4. Efficiency
5. Maintainability
6. Portability

4.1.2 Cost of Quality

We know that quality is important but it costs us time and money as well. These arguments seem reasonable. There is no question that quality has a cost, but lack of quality also has a cost. The question is this: "which costs should we be worried about". To answer this question, we must understand both the costs of achieving quality and the costs of low-quality software.

The software cost depends on *time and money*. The *cost of quality* includes all costs incurred in the pursuit of quality or in performing quality-related activities and the downstream costs of lack of quality. To understand these costs, an organization must collect metrics to provide a baseline for the current cost of quality, identify opportunities for reducing these costs, and provide a normalized basis of comparison. The cost of quality can be divided into costs associated with prevention, appraisal, and failure.

1. Prevention Costs:

Prevention costs include:

i. the cost of management activities required to plan and coordinate all quality control and quality assurance activities,
ii. the cost of added technical activities to develop complete requirements and design models,
iii. test planning costs, and
iv. the cost of all training associated with these activities.

2. Appraisal Costs:

Appraisal costs are related to activities to gain insight into product condition the "first time through" each process. Appraisal costs include:

i. Cost of conducting technical review for software engineering work products
ii. Cost of data collection and metrics evaluation.
iii. Cost of testing and debugging.

3. **Failure Costs**:

Failure costs are those that would appear just before or after shipping a product to customers, but if no errors appeared before. Failure costs may be subdivided into internal failure costs and external failure costs.

- *Internal failure costs*:

  These are incurred when you detect an error in a product prior to shipment. Cost required to perform rework (repair) to correct an error is an example of internal failure cost.

- *External failure costs*:

  These are associated with defects found after the product has been shipped to the customer. Examples of external failure costs are complaint resolution, product return and replacement, help line support, and labor costs associated with warranty work. A poor reputation and the resulting loss of business is another external failure cost.

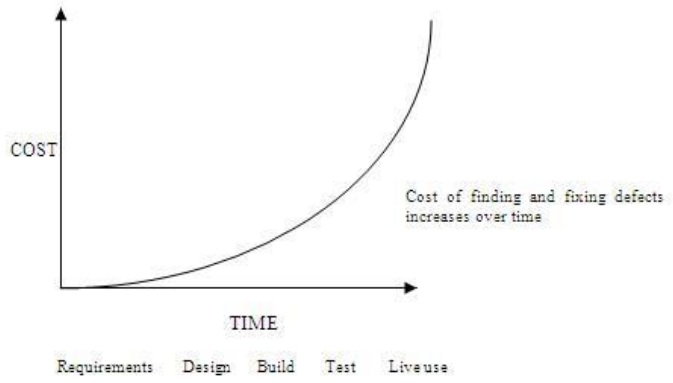
4.1.3 Cost Impact of Defects

The cost of defects can be measured by the impact of the defects and when we find them. Earlier the defect is found lesser is the cost of defect. For example if error is found in the requirement specifications during requirements gathering and analysis, then it is somewhat cheap to fix it. The correction to the requirement specification can be done and then it can be re-issued.

In the same way when defect or error is found in the design during design review then the design can be corrected and it can be re-issued. But if the error is not caught in the specifications and is not found till the user acceptance then the cost to fix those errors or defects will be way too expensive.

If the error is made and the consequent defect is detected in the requirements phase then it is relatively cheap to fix it.

Similarly if a requirement specification error is made and the consequent defect is found in the design phase then the design can be corrected and reissued with relatively little expense.



The same applies for construction phase. If however, a defect is introduced in the requirement specification and it is not detected until acceptance testing or even once the system has been implemented then it will be much more expensive to fix. This is because rework will be needed in the specification and design before

changes can be made in construction; because one defect in the requirements may well propagate into several places in the design and code; and because all the testing work done-to that point will need to be repeated in order to reach the confidence level in the software that we require.

It is quite often the case that defects detected at a very late stage, depending on how serious they are, are not corrected because the cost of doing so is too expensive.

4.1.4 Defect Amplification and Removal

The defect amplification model [IBM81] can be used to describe the detection and generation of errors during preliminary design, detail design and coding steps of the software engineering procedure. The model is illustrated schematically in figure below:

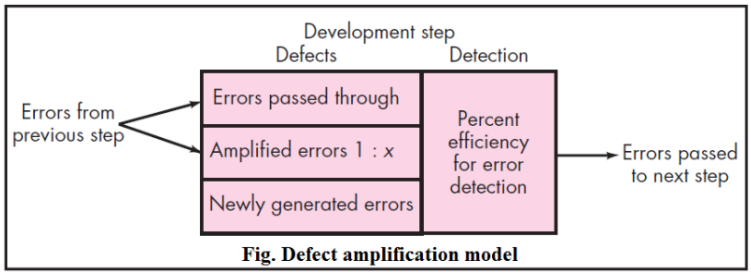
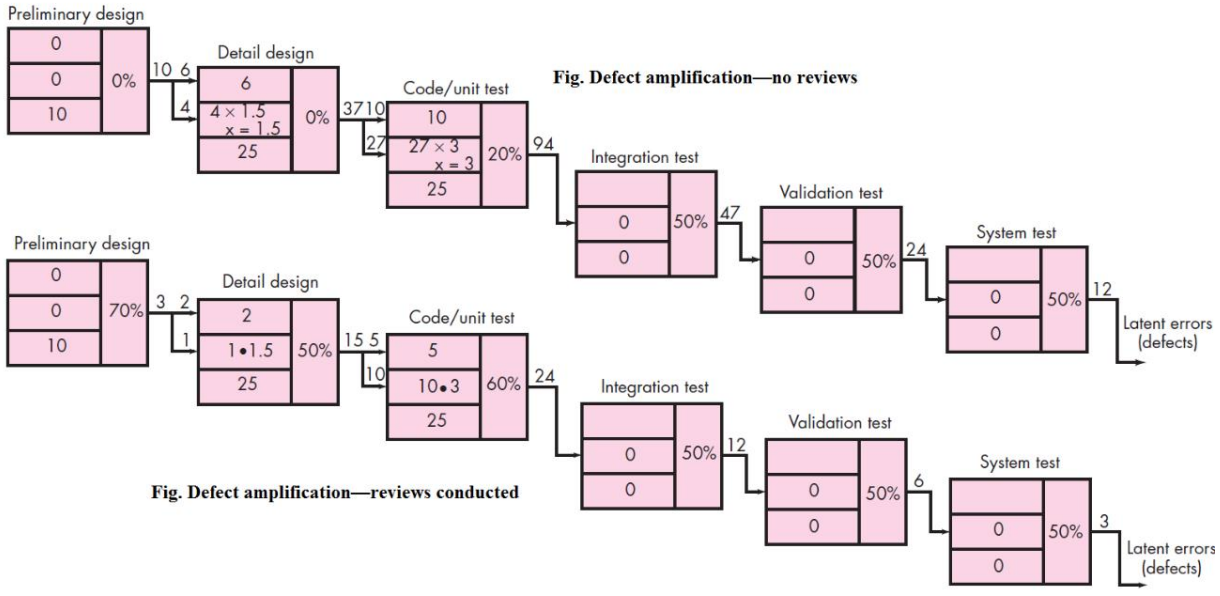


Fig. Defect amplification model

A box represents a software development step:

- During the step, errors may be inadvertently generated.
- Review may fail to uncover newly generated errors and errors from previous steps, resulting in some numbers of errors that are passed through.
- In some cases, error passed through from previous steps are amplified (amplification factor, x) by current work.

Box subdivisions represent each of these characteristics and the percentage of efficiency for detecting errors, a function of the thoroughness of the review.



The above figure illustrates a hypothetical example of defect amplification for a software process with no reviews and reviews are conducted.

The first figure represents the defect amplification with no reviews. Each test step is supposed to uncover and correct 50% of all incoming errors without introducing any new errors. An optimistic assumption, 10 pre-liminary design errors are amplified to 94 errors before testing commences. 12 latent defects are released to the area.

The second figure represents the defect amplification with reviews conducted. The figure considers the similar conditions except which code and design reviews are conducted as categorized of each development step. In this case 10 initial preliminary design errors are amplified to 24 errors before testing commences. Only 3 latent defects exist.

So, for the removal of the defects, review must be conducted in each step. The effort expended when reviews are used does increase early in the development of a software increment, but this early investment for reviews pays dividends because testing and corrective effort is reduced. As important, the deployment date for development with reviews is sooner than the deployment dates without reviews.

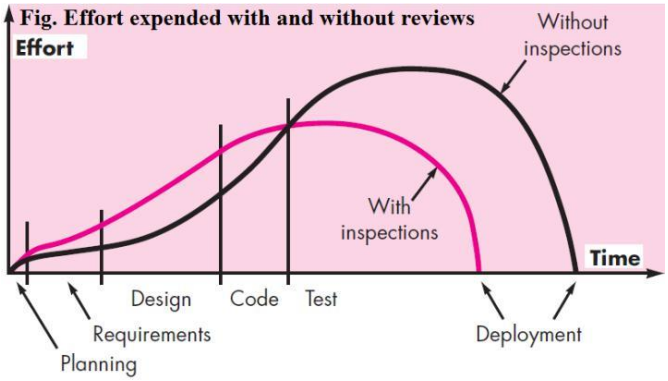


Fig. Effort expended with and without reviews

4.1.5 Software Review and FTR

Software Review:

Technical work needs reviewing for the same reason that pencils need erasers. Thus, software reviews are a "filter" for the software engineering process. That is, reviews are applied at various points during software development and serve to uncover errors and defects that can then be removed. Software reviews "purify" the software engineering activities (analysis, design, and coding). In general, six steps are employed for software review: planning, preparation, structuring the meeting, noting errors, making corrections (done outside the review), and verifying that corrections have been performed properly.

In other words, software review is the way of using the diversity of a group of people to:

- Point out needed improvement in the software
- Confirm those portions of the software in which improvement is either desired or not desired.
- Achieve more uniform technical or more predictable work.

Review can be either formal or informal. An Informal Review can be made at any place with small number of people involvement, whereas Formal Technical Review is very much effective.

Formal Technical Review (FTR):

A formal technical review (FTR) is a software quality control activity performed by software engineers (and others). The objectives of an FTR are:

- To uncover errors in function and logic.
- To verify that the software under review meets its requirements.
- To ensure that the software has been represented according to predefined standards.
- To achieve software that is developed in a uniform manner and
- To make projects more manageable.

In addition, the FTR serves as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation. The FTR also serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen.

Actually, FTR is a class of reviews that include walkthroughs, inspections, round robin reviews and other small group technical assessments of software. Each FTR is conducted as meeting and is considered successfully only if it is properly planned, controlled and attended.

The review meeting:

Each review meeting should be held considering the following constraints-

Involvement of people:

1. Between 3, 4 and 5 people should be involved in the review.
2. Advance preparation should occur but it should be very short that is at the most 2 hours of work for every person.
3. The short duration of the review meeting should be less than two hour.

At the end of the review, all attendees of FTR must decide what to do.

1. Accept the product without any modification.
2. Reject the project due to serious error (Once corrected, another app need to be reviewed), or
3. Accept the product provisional (minor errors are encountered and are should be corrected, but no additional review will be required).

Finally, decision is made, with all FTR attendees completing a sign-of indicating their participation in the review and their agreement with the findings of the review team.

Review reporting and record keeping :

Record keeping and reporting are other quality assurance activities. It is applied to record all issues that have been raised. At the end of review meeting, a review issue list is produced. A review summary report answers the following questions.

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

Review guidelines:

Guidelines for the conducting of formal technical reviews should be established in advance. These guidelines must be distributed to all reviewers, agreed upon, and then followed. A review that is unregistered can often be worse than a review that does not minimum set of guidelines for FTR.

1. Review the product, not the manufacture (producer).
2. Take written notes (record purpose)
3. Limit the number of participants and insists upon advance preparation.
4. Develop a checklist for each product that is likely to be reviewed.
5. Allocate resources and time schedule for FTRs in order to maintain time schedule.
6. Conduct meaningful training for all reviewers in order to make reviews effective.
7. Reviews earlier reviews which serve as the base for the current review being conducted.
8. Set an agenda and maintain it.
9. Separate the problem areas, but do not attempt to solve every problem notes.
10. Limit debate and rebuttal.

4.1.6 Quality Control and Assurance

Quality Control in Software Testing is a systematic set of processes used to ensure the quality of software products or services. The main purpose of the quality control process is ensuring that the software product meets the actual requirements by testing and reviewing its functional and non-functional requirements. Quality control is popularly abbreviated as QC.

Quality Assurance is popularly known as QA Testing, is defined as an activity to ensure that an organization is providing the best possible product or service to customers.

Quality Assurance (QA)	**Quality Control (QC)**
1. It is a procedure that focuses on providing assurance that quality requested will be achieved	1. It is a procedure that focuses on fulfilling the quality requested.
2. QA aims to prevent the defect	2. QC aims to identify and fix defects

3. It is a method to manage the quality- Verification	3. It is a method to verify the quality- Validation
4. It does not involve executing the program	4. It always involves executing a program
5. It's a Preventive technique	5. It's a Corrective technique
6. It's a Proactive measure	6. It's a Reactive measure
7. It is the procedure to create the deliverables	7. It is the procedure to verify that deliverables
8. QA involves in full software development life cycle	8. QC involves in full software testing life cycle
9. In order to meet the customer requirements, QA defines standards and methodologies	9. QC confirms that the standards are followed while working on the product
10. It is performed before Quality Control	10. It is performed only after QA activity is done
11. It is a Low-Level Activity, it can identify an error and mistakes which QC cannot	11. It is a High-Level Activity, it can identify an error that QA cannot
12. Its main motive is to prevent defects in the system. It is a less time-consuming activity	12. Its main motive is to identify defects or bugs in the system. It is a more time-consuming activity
13. QA ensures that everything is executed in the right way, and that is why it falls under verification activity	13. QC ensures that whatever we have done is as per the requirement, and that is why it falls under validation activity
14. It requires the involvement of the whole team	14. It requires the involvement of the Testing team
15. The statistical technique applied on QA is known as SPC or Statistical Process Control (SPC)	15. The statistical technique applied to QC is known as SQC or Statistical Quality Control

4.1.7 Software Quality Assurance (SQA) and its elements

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of software product standards, processes, and procedures. SQA is an umbrella activity that is applied in each step of software engineering. SQA encompasses procedures for the effective application of methods and tools, oversight of quality control activities such as technical reviews and software testing, procedures for change management, procedures for assuring compliance to standards, and measurement and reporting mechanisms.

Software Quality Assurance (SQA) is simply a way to assure quality in the software. It is the set of activities which ensure processes, procedures as well as standards suitable for the project and implemented correctly.

Software Quality Assurance is a process which works parallel to development of a software. It focuses on improving the process of development of software so that problems can be prevented before they become a major issue. Software Quality Assurance is a kind of an Umbrella activity that is applied throughout the software process.

Software Quality Assurance have:

1. A quality management approach
2. Formal technical reviews
3. Multi testing strategy
4. Effective software engineering technology
5. Measurement and reporting mechanism

Major Software Quality Assurance Activities:

1. **Creating an SQA Management Plan:**

   The foremost activity includes laying down a proper plan regarding how the SQA will be carried out in your project.

Along with what SQA approach you are going to follow, what engineering activities will be carried out, and it also includes ensuring that you have a right talent mix in your team.

2. **Setting the Checkpoints:**

   The SQA team sets up different checkpoints according to which it evaluates the quality of the project activities at each checkpoint/project stage. This ensures regular quality inspection and working as per the schedule.

3. **Apply software Engineering Techniques:**

   Applying some software engineering techniques aids a software designer in achieving high-quality specification. For gathering information, a designer may use techniques such as interviews and FAST (Functional Analysis System Technique).

   Later, based on the information gathered, the software designer can prepare the project estimation using techniques like WBS (work breakdown structure), SLOC (source line of codes), and FP(functional point) estimation.

4. **Executing Formal Technical Reviews:**

   An FTR is done to evaluate the quality and design of the prototype.

   In this process, a meeting is conducted with the technical staff to discuss regarding the actual quality requirements of the software and the design quality of the prototype. This activity helps in detecting errors in the early phase of SDLC and reduces rework effort in the later phases.

5. **Having a Multi- Testing Strategy:**

   By multi-testing strategy, we mean that one should not rely on any single testing approach, instead, multiple types of testing should be performed so that the software product can be tested well from all angles to ensure better quality.

6. **Enforcing Process Adherence:**

   This activity insists the need for process adherence during the software development process. The development process should also stick to the defined procedures.

Benefits of Software Quality Assurance (SQA):

- SQA produce high quality software.
- High quality application saves time and cost.
- SQA is beneficial for better reliability.
- SQA is beneficial in the condition of no maintenance for long time.
- High quality commercial software increase market share of company.
- Improving the process of creating software.
- Improves the quality of the software.

Disadvantage of SQA:

There are a number of disadvantages of quality assurance. Some of them include adding more resources, employing more workers to help maintain quality and so much more.

Elements of SQA:

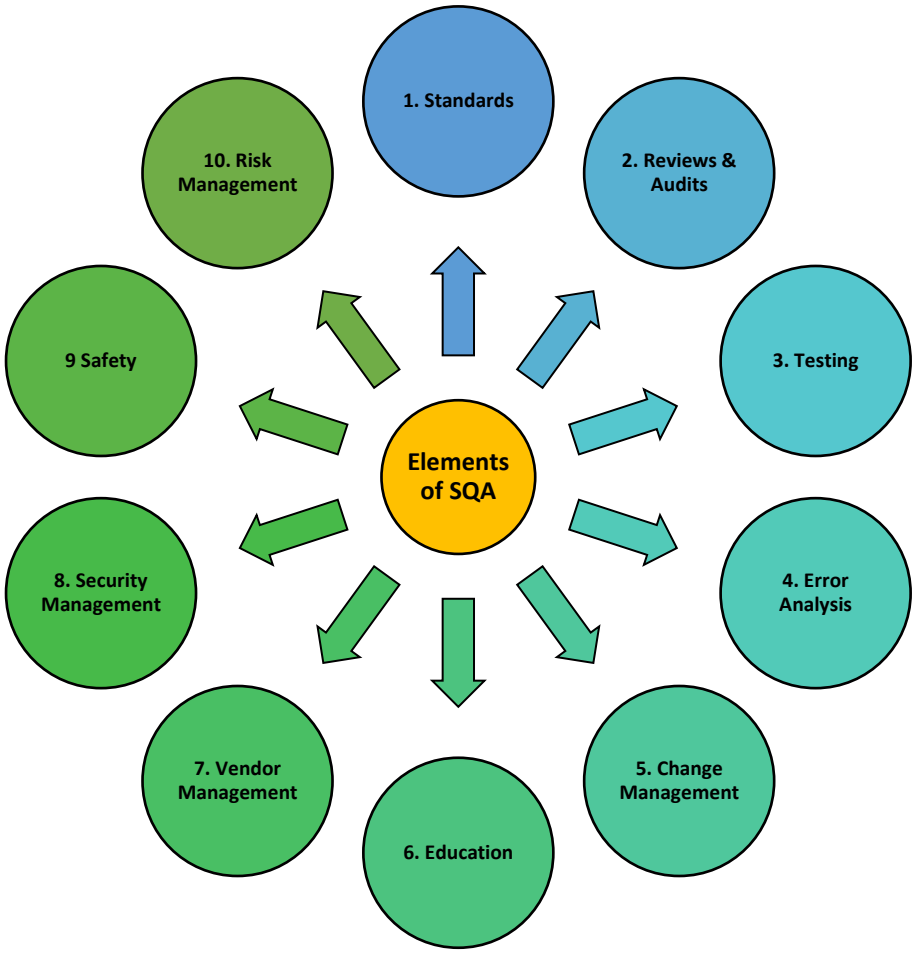


Figure: Elements of SQA

Software quality assurance encompasses a broad range of concerns and activities that focus on the management of software quality. These can be summarized in the following manner:

1. **Standards**

   We all know that organizations like IEEE, ISO etc. has lined-up a lot up software engineering standards which should be imposed by the customer and even embraced by software engineers while developing software. SQA team must ensure that standards established by the authorized organization are followed by the software engineers.

2. **Review & Audit**

The software is evaluated technically by the SQA team in order to discover an error in the software. While auditing a software the SQA team confirms that good quality guidelines are followed by the software engineers while developing the software.

3. **Testing**

The elemental goal of software testing is to identify the bug in the software. SQA team has the responsibility of planning the testing systematically and conducting it efficiently. This would raise the possibility of finding the bug in the software if any.

4. **Analyzing Error**

Software testing reveals bugs and errors in the software which are analyzed by the SQA team in order to discover how the bugs or errors are introduced in the software and also discover the possible methods required to eliminate those errors or bugs.

5. **Change Management**

The customer can ask for modifications between the development of the software. The changes are the most distracting aspect of any software project. If the implementation of the changes is not properly managed it will cause confusion which will affect the quality of the software. The SQA team takes care that change management is practiced while developing the software.

6. **Education**

It is the responsibility of the SQA organization to enlighten the software organizations by improving their software engineering practices.

7. **Vendor Management**

It is the responsibility of SQA to suggest software vendors, to accept the quality practices while developing the software. SQA must also incorporate these quality instructions in a contract with software vendors.

8. **Security Management**

With the increase in cybercrime, the government has regulated the software organization to incorporate policies to protect data at all level. It is the responsibility of SQA to verify whether the software organization are using appropriate technology to acquire software security.

9. **Safety**

The hidden defects of any human-rated software (aircraft applications) can lead to catastrophic events. So, it is the responsibility of SQA to evaluate the effect of software failure and introduce steps to eliminate the risk.

10. **Risk Management**

Though it is the job of software organization to analyze and reduce the risk in the software. But, it is the responsibility of SQA to make sure that risk management actions are properly conducted and the backup plan has been confirmed.

4.1.8 SQA Tasks, Goals and Metrics

Software quality assurance is composed of a variety of tasks associated with two different constituencies: the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, oversight, record keeping, analysis, and reporting. Software engineers address quality by applying solid technical methods and measures, conducting technical reviews, and performing well-planned software testing.

SQA Tasks

The SQA group is to assist the software team in achieving a high quality end product. The Software Engineering Institute recommends a set of SQA actions that address quality assurance planning, oversight, record keeping, analysis, and reporting. The SQA group generally possess the following tasks:

1. Prepares an SQA plan for a project.
2. Participates in the development of the project's software process description.
3. Reviews software engineering activities to verify compliance with the defined software process.
4. Audits designated software work products to verify compliance with those defined as part of the software process.
5. Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
6. Records any noncompliance and reports to senior management.

SQA Goal and Metrics

Goals	Attributes	Metrics
1. Requirement quality	Ambiguity	Number of ambiguous modifiers
	Completeness	Number of requirements
	Understandability	Number of sections/subsections
	Volatility	Number of changes per requirement
	Traceability	Number of requirements
2. Design quality	Architectural integrity	Existence of architectural model
	Component completeness	Complexity of procedural design,
	Interface complexity	Layout appropriateness
	Patterns	Number of patterns used
3. Code quality	Complexity	Cyclomatic complexity
	Maintainability	Design factors
	Understandability	Percent internal comments
	Reusability	Percent reused components
	Documentation	Readability index
4. Quality control effectiveness	Resource allocation	Staff hour percentage per activity
	Completion rate	Actual vs. budgeted completion time
	Review effectiveness	See review metrics
	Testing effectiveness	Origin of error

Table: Summary of SQA goals, attributes and metrics

1. Requirements Quality

SQA must ensure that the requirements specified by the user are properly reviewed by the software developers. Software developers have properly analyzed the correctness, completeness of specified requirements as it has a strong influence on the quality of software.

Attribute: Metric

(a) Ambiguity: There can be a number of ambiguous modifiers in the specified requirement.

(b) Completeness: There can be a number of requirements that are neither announced nor determined yet.

(c) Understandability: There can be a number of sections or subsections in the specified requirements that are not understood by developers.

(d) Volatility: Requirements can be volatile as the user can request a number of changes in the requirements. As the requirements are volatile and the user can request changes in it, the time requires to complete the process will also be changed.

(e) Traceability: There can be a number of requirements that cannot be traced to design code.

2. Design Quality

After analyzing the requirements specified by the user UML models are designed to visualize the way the system will be designed. The job of SQA is to evaluate whether the design model conforms the specified requirements to achieve quality.

Attribute: Metric

(a) Architectural Integrity: Developers must be able to assess the architectural elements to confirm the quality of the architectural model.

(b) Component Completeness: The components of the architectural model are completely defined or not.

(c) Interface Complexity: The number moves required to get the desired function.

(d) Patterns: The number of patterns used to design architectural model.

3. Code Quality

The code of the software and it's related descriptive information must obey the local coding standards. So, that it can be maintained for long in future. It is the job of SQA to observe the quality of code.

Attribute: Metric

(a) Complexity: The number of cyclomatic complexity in the code.

(b) Understandability: The understandability of code relies on the percent of internal comment in the code and the naming convention used while naming the variables.

(c) Reusability: The number of components in the code that can be reused.

4. Quality Control Effectiveness

Software developers should use minimal resources and try to achieve high quality. The job of SQA is to analyze whether the resources are allocated in an effective manner.

Attribute: Metric

(a) Resource Allocation: Time spent by staff per activity.

(b) Completion rate: Actual time for completion vs. Committed time for completion.

4.1.9 Statistical SQA and Six Sigma

Statistical SQA:

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects is collected and categorized.
2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the vital few).
4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

This relatively simple concept represents an important step towards the creation o: an adaptive software engineering process in which changes are made to improve those elements of the process that introduce error.

To illustrate this, assume that a software engineering organization collects information on defects for a period of one year. Some of the defects are uncovered as software is being developed. Others are encountered after the software has been released to its end-users. Although hundreds of different errors are uncovered, all can be tracked to one (or more) of the following causes:

- Incomplete or Erroneous Specifications (IBS)
- Misinterpretation of Customer Communication (MCC)
- Intentional Deviation from Specifications (IDS)
- Violation of Programming Standards (VPS)
- Error in Data Representation (EDR)
- Inconsistent Component Interface (ICI)
- Error in Design Logic (EDL)
- Incomplete or Erroneous Testing (IET)
- Inaccurate or Incomplete Documentation (IID)
- Error in Programming Language Translation of Design (PLT)
- Ambiguous or Inconsistent Human/Computer Interface (HCI)
- Miscellaneous (MIS)

To apply statistical SQA, Table is built as shown below:

	Total		Serious		Moderate		Minor	
Error	No.	%	No.	%	No.	%	No.	%
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totals	942	100%	128	100%	379	100%	435	100%

Table: Data collection for statistical SQA

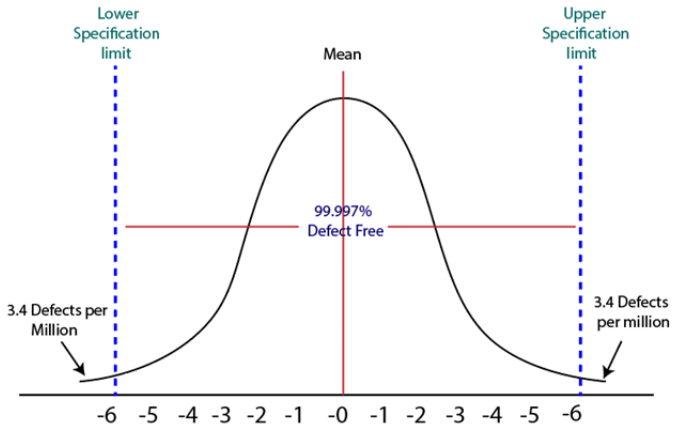
The table indicates that IES, MCC, and EDR are the vital few causes that account for 53 percent of all errors. It should be noted, however, that IES, EDR, PLT, and EDL would be selected as the vital few causes if only serious errors are considered. Once the vital few causes are determined, the software engineering organization can begin corrective action.

For example:

- To correct MCC, the software developer might implement facilitated application specification techniques to improve the quality of customer communication and specifications.
- To improve EDR, the developer might acquire CASE tools for data modeling and perform more stringent data design reviews.

Six Sigma:

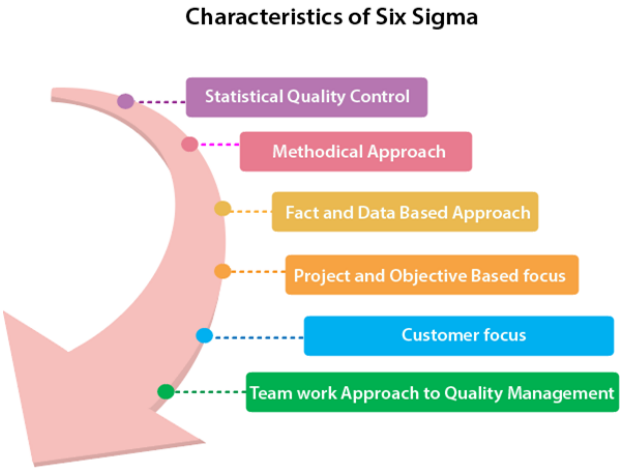
Six Sigma is the process of improving the quality of the output by identifying and eliminating the cause of defects and reduce variability in manufacturing and business processes. The maturity of a manufacturing process can be defined by a sigma rating indicating its percentage of defect-free products it creates. A six-sigma method is one in which 99.99966% of all the opportunities to produce some features of a component are statistically expected to be free of defects (3.4 defective features per million opportunities).



Six Sigma is the most widely used strategy for statistical quality assurance in industry today. Originally popularized by Motorola in the 1980s, the Six Sigma strategy "is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects' in manufacturing and service-related processes". The term Six Sigma is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high-quality standard.

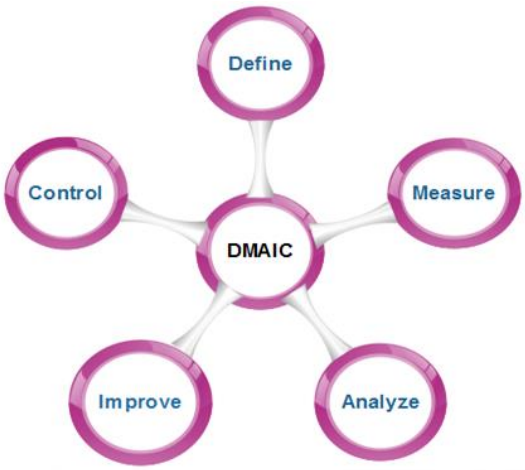
Characteristics of Six Sigma:

The Characteristics of Six Sigma are as follows:



1. **Statistical Quality Control:** Six Sigma is derived from the Greek Letter σ (Sigma) from the Greek alphabet, which is used to denote Standard Deviation in statistics. Standard Deviation is used to measure variance, which is an essential tool for measuring non-conformance as far as the quality of output is concerned.
2. **Methodical Approach:** The Six Sigma is not a merely quality improvement strategy in theory, as it features a well defined systematic approach of application in DMAIC and DMADV which can be used to improve the quality of production. DMAIC is an acronym for Design-Measure- Analyze-Improve-Control. The alternative method DMADV stands for Design-Measure- Analyze-Design-Verify.
3. **Fact and Data-Based Approach:** The statistical and methodical aspect of Six Sigma shows the scientific basis of the technique. This accentuates essential elements of the Six Sigma that is a fact and data-based.
4. **Project and Objective-Based Focus:** The Six Sigma process is implemented for an organization's project tailored to its specification and requirements. The process is flexed to suits the requirements and conditions in which the projects are operating to get the best results.
5. **Customer Focus:** The customer focus is fundamental to the Six Sigma approach. The quality improvement and control standards are based on specific customer requirements.
6. **Teamwork Approach to Quality Management:** The Six Sigma process requires organizations to get organized when it comes to controlling and improving quality. Six Sigma involving a lot of training depending on the role of an individual in the Quality Management team.

Six-Sigma Methodology: DMAIC



The Six Sigma methodology defines three core steps:

1. **Define** customer requirements and deliverables and project goals via well-defined methods of customer communication.
2. **Measure** the existing process and its output to determine current quality performance (collect defect metrics).
3. **Analyze** defect metrics and determine the vital few causes.

If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

4. **Improve** the process by eliminating the root causes of defects.
5. **Control** the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the **DMAIC** (define, measure, analyze, improve, and control) method.

4.2 Software Reliability

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time". For example: program *X* is estimated to have a reliability of 0.999 over eight elapsed processing hours.

The reliability of a computer program describes the acceptance limit of its overall quality. Unlike many other quality factors, the software reliability can be measured directly and estimated using historical and development data. The software failure determines the software reliability that may be frequently appeared on the program and can be corrected either frequently or after long time.

Software Reliability means **Operational reliability**. It is described as the ability of a system or component to perform its required functions under static conditions for a specific period. Simply, software reliability

is the probability of a failure-free operation of a program for a specified time in a specified environment. It is concerned with how well the software functions to meet requirements of the customer.

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the input are free of error.

Software Reliability is an essential connect of software quality, composed with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software Reliability is hard to achieve because the complexity of software turn to be high. While any system with a high degree of complexity, containing software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the speedy growth of system size and ease of doing so by upgrading the software.

Software Reliability Metrics:

1. **Rate of Occurrence of Failure (ROCOF)**

   It measures the frequency of occurrence of unexpected behavior (i.e. failures).

2. **Mean Time to Failure (MTTF)**

   Average time between two successive failures observed over a large no. of failures.

3. **Mean-Time to Repair (MTTR)**

   Measures the average time it takes to track the errors and to fix them.

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

   MTBF=MTTF+MTTR

   E.g. MTBF of 300 hrs indicates that the next failure is expected after 300 hrs.

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

   It measures the likelihood of the system failing when a service request is made.

6. **Availability**

   It is a measure of how likely shall the system be available for use over a given period of time.

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}} \text{ x } 100\%$$

4.2.1 Availability

Measure of Reliability:

In hardware, failures due to physical wear (e.g., the effects of temperature, corrosion, shock) are more likely than a design-related failure. Unfortunately, the opposite is true for software. Thus, for computer system, a simple measure of reliability is *meantime-between-failure* (MTBF)

$$MTBF = MTTF + MTTR$$

Where the acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to repair* respectively

Availability:

Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:

$$Availability = \frac{MTTF}{MTTF + MTTR} \times 100\%$$

4.2.2 Software Safety

Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

A modeling and analysis process is conducted as part of software safety. Once hazards are identified and analyzed, safety-related requirements can be specified for the software. Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them.

Software Reliability Vs Software Safety:

Software Reliability	Software Safety
1. The occurrence of a failure does not necessarily result in hazard.	1. It examines the failure result that leads to hazard.
2. Use statistical analysis to determine software failure.	2. Uses the fault tree analysis, real time logic and Petri-net models to predict hazards.

4.3 ISO Standards

International Standard Organization (ISO) is a consortium of 63 countries to formulate and foster standardization. It specifies guidelines for maintaining a quality system. The main consideration of ISO is that "*If proper process is followed for production then good quality products are bound to follow automatically.*"

ISO 9000 is a generic name given to a family of standards developed to provide a framework around which a quality management system can effectively be implemented. The ISO 9000 is a series of 3 standards ISO 9001, 9002, and 9003.

- **ISO 9001:** Applies to organizations engaged in design, development, production, and servicing of goods, so applicable to most software development organization.
- **ISO 9002:** Applies to organizations which do not design products but are only involved in production. E.g. steel and car manufacturing industries (they buy plant and product design from others).
- **ISO 9003:** Applies to organizations involved only in installation and testing of products.

Shortcomings of ISO Certification

- Requires good software production process to adhere to high quality but no guidelines for defining an appropriate process.
- Variations in the norms of awarding certificates among different ISO agencies in different country and places.
- In manufacturing industry there exists a link between process quality and product quality. Good process quality product but software is not manufactured i.e. it is developed according as the end user requirements.

ISO vs CMM

ISO	CMM
1. Awarded by the International standard body.	1. Assessment for the internal use only
2. No software industry specific.	2. Developed specifically for software industry.
3. Confined to quality assurance but does not define the ways of improvement.	3. Quality assurance and also provides the way of gradual quality achievement.
4. Generally, the validity time duration is of 6 months,	4. Generally, the validity time duration of one year.

4.4 The CMM Model

The CMM is a benchmark for measuring the maturity of an organization's system process. It is a methodology used to develop and refine an organization's software development process. It describes the maturity of the company based upon the project the company is dealing with and the clients.

This model describes a five-level evolutionary part of increasingly organized and systematically more mature process. The higher the level, the better is the software development process.

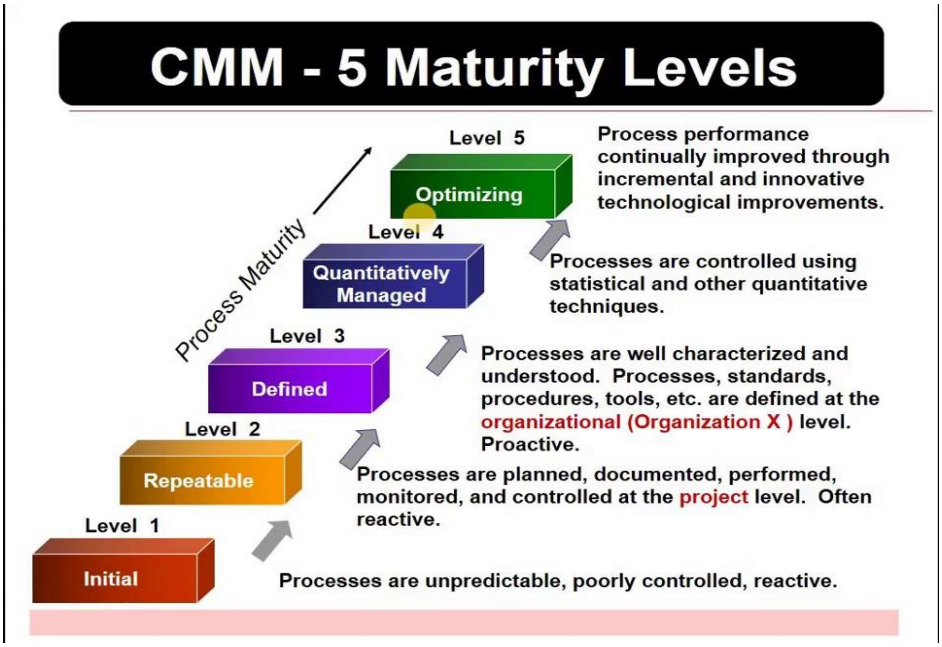


Figure: The Levels of CMM

1. **Maturity Level 1: Initial**
   - Processes are disorganized, ad-hoc and chaotic.
   - Success of organization depends on the effort and competence of the people within.
   - Company has no standard process for software development.
   - Organization may not be able to repeat their past success.

2. **Maturity Level 2: Repeatable**
   - Basic project management techniques are established and success could be repeatable.
   - Program management is a key characteristic of a level 2 organization.

3. **Maturity Level 3: Defined**
   - At this level, organization has developed its own standards.
   - The software process for both management and engineering activities are documented, standardized and integrated.

4. **Maturity Level 4: Managed**
   - The performance and process is controlled and monitored through statistical and quantitative techniques.

- Organization set a quantitative quality goal.

5. **Maturity Level 5: Optimizing**

- Processes are constantly being input through monitoring feedback from current processes and introducing innovative process.

4.5 Verification and Validation

Verification:

Verification is the process of evaluating the intermediary work products of a SDLC to check if we are in the right track of creating the final product. Verification is the step by step process which includes the documentation and all the reviews.

Verification in Software Testing is a process of checking documents, design, code, and program in order to check if the software has been built according to the requirements or not. The main goal of verification process is to ensure quality of software application, design, architecture etc. The verification process involves activities like reviews, walk-throughs and inspection.

It deals with: *"Are you building the software in right way?".*

Validation:

Validation is the process of evaluating the final product to check whether the software meets the business needs. Validation is done after the software is developed and to find the defects before it goes to customer's hand.

Validation in Software Testing is a dynamic mechanism of testing and validating if the software product actually meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. The validation process involves activities like unit testing, integration testing, system testing and user acceptance testing.

It deals with: *"Are you building the right product?"*

Example of verification and validation

In Software Engineering, consider the following specification: A clickable button with name *Submet*

- Verification would check the design doc and correcting the spelling mistake.
- Otherwise, the development team will create a button like

So new specification is: A clickable button with name *Submit*

Once the code is ready, Validation is done. A Validation test found –Button not clickable

Owing to Validation testing, the development team will make the submit button clickable

Verification Vs Validation

Verification	Validation
1. The verifying process includes checking documents, design, code, and program	1. It is a dynamic mechanism of testing and validating the actual product
2. It does not involve executing the code	2. It always involves executing the code
3. Verification uses methods like reviews, walkthroughs, inspections, and desk- checking etc.	3. It uses methods like Black Box Testing, White Box Testing, and non-functional testing
4. Whether the software conforms to specification is checked	4. It checks whether the software meets the requirements and expectations of a customer
5. It finds bugs early in the development cycle	5. It can find bugs that the verification process can not catch
6. Target is application and software architecture, specification, complete design, high level, and database design etc.	6. Target is an actual product
7. QA team does verification and make sure that the software is as per the requirement in the SRS document.	7. With the involvement of testing team validation is executed on software code.
8. It comes before validation	8. It comes after verification

4.6 Testing and Debugging

Testing:

Software testing is the process of executing a software system to determine whether if it matches its specification and executed in its intended environment or not.

Software testing is done to confirm that:

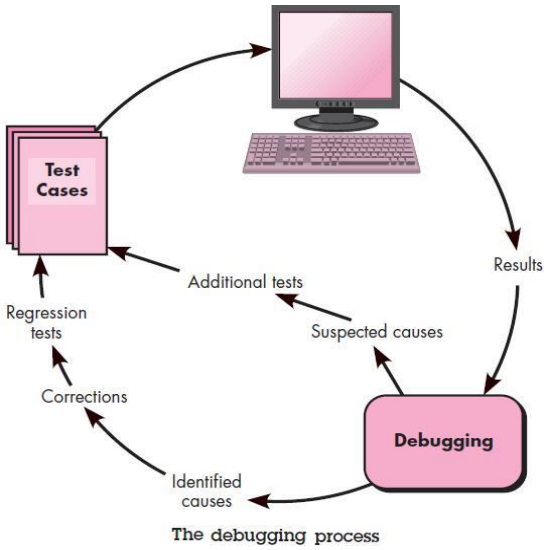
- It is error-free
- Working as to our expectation

During the software development process, we go through a series of events or process like analysis, design, coding, implementation and maintenance. Once the software has been designed, source code is generated. Even then there might be some errors that deflect the software product from its actual performance, and its implementation results in many unreliable and unrealistic consequences.

Therefore, testing the software so as to confirm that it is error-free and working as to our expectation, is very important. Thus, software testing is carried out for SQA, and represents the ultimate review od specification, design and code generation.

Debugging:

Once errors are identified, it is necessary to localize or identify the precise location of the errors and fix. Debugging occurs as a consequence of successful testing when a test case uncovers an error, debugging results in the removal of the error. Debugging is not testing but always occurs as a consequence of testing. The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered.



The debugging process

The debugging process will either find the cause or correct it, or the cause will not be found. If the cause if not found, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion. As the consequences of error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more. Debugging has one main objective

which is to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition and luck. The debugging must start from simple to complex.

4.5.1 Debugging Approaches:

Debugging is the process of fixing the errors in the program or software. There are several debugging techniques as discussed below:

1. **Brute Force Method:**

   This is the foremost common technique of debugging however is that the least economical method. during this approach, the program is loaded with print statements to print the intermediate values with the hope that a number of the written values can facilitate to spot the statement in error. This approach becomes a lot of systematic with the utilization of a symbolic program (also known as a source code debugger), as a result of values of various variables will be simply checked and breakpoints and watch-points can be easily set to check the values of variables effortlessly.

2. **Backtracking:**

   This is additionally a reasonably common approach. during this approach, starting from the statement at which an error symptom has been discovered, the source code is derived backward till the error is discovered. sadly, because the variety of supply lines to be derived back will increase, the quantity of potential backward methods will increase and should become unimaginably large so limiting the utilization of this approach.

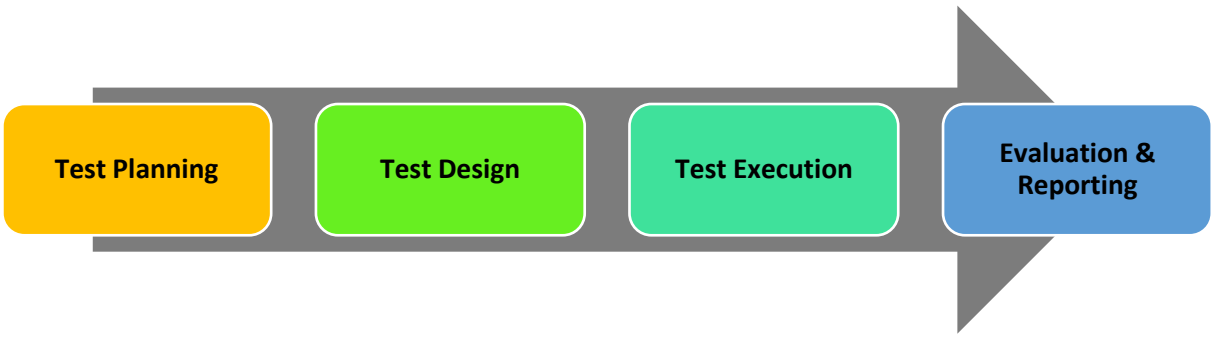
3. **Cause Elimination Method**:

   In this approach, a listing of causes that may presumably have contributed to the error symptom is developed and tests are conducted to eliminate every error. A connected technique of identification of the error from the error symptom is that the package fault tree analysis.

4. **Program Slicing:**

   This technique is analogous to backtracking. Here the search house is reduced by process slices. A slice of a program for a specific variable at a particular statement is that the set of supply lines preceding this statement which will influence the worth of that variable.

4.5.2 Testing Process



1. **Testing planning**

   Test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specification and other requirements.

2. **Test design**

   It is the act of creating and writing test suites for testing a software.

3. **Test execution**

   During execution, we take the test conditions into test cases and procedures and other test-ware such as scripts for automation, the test environment and any other test infrastructure.

4. **Evaluation and reporting**

   The test is measured against the exit criteria. Exit criteria refer to the benchmark or standard that the quality of software has to meet. Finally, a test report is documented.

4.5.3 Test cases ad Test suites

Test case

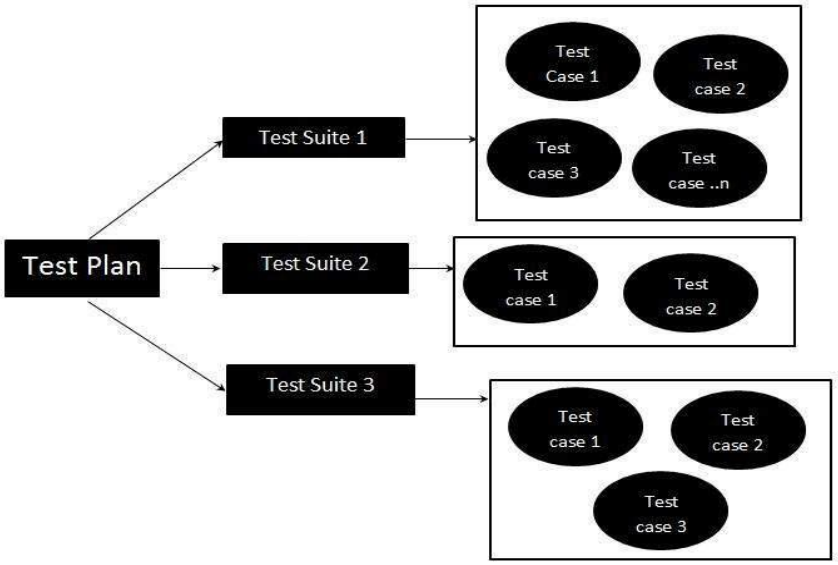
A test case is a document, which has a set of test data, preconditions, expected results and post-conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.

A test case sample:

1. Test suite ID:	TS001
2. Test case ID:	TC001
3. Test case summary:	To verify that clicking the “Generate Coin” button generates coins.
4. Prerequisites:	1. User is authorized. 2. Coin balance is available.
5. Test procedure:	1. Select the coin denomination in the denomination field. 2. Enter the number of coins in the Quantity field. 3. Click Generate coin.
6. Expected result:	……
Actual result:	……
7. Status:	Pass / Fail
8. Remarks:	This is a sample test case.
9. Created by:	Er. Shiva Ram Dam
10. Date of creation:	07 /08/ 2016
11. Executed by:	Er. Subash Raj Bhat
12. Date of execution:	07 /08/ 2016
13. Test environment:	OS : Windows 8 Browser : Google chrome

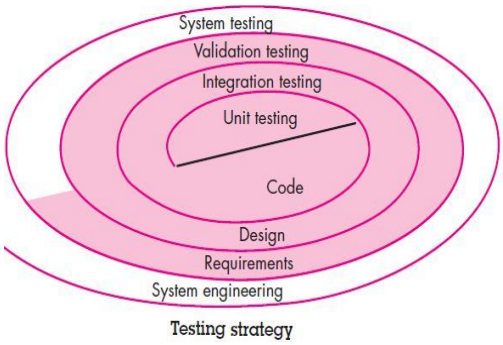
Test suite

Test suite is a collection of test cases that are intended to be used to test a software program.

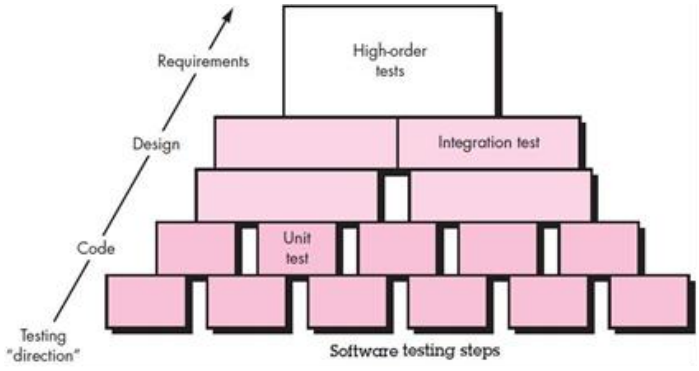


4.5.4 Testing Strategies

It explains the formal plan for testing small component to entire system; test the new changes on the existing modules, requirement of customer involvement time etc. A strategy provides guidance for the practitioner and a set of milestones for the manager. A strategy for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.



Testing strategy



Software testing steps

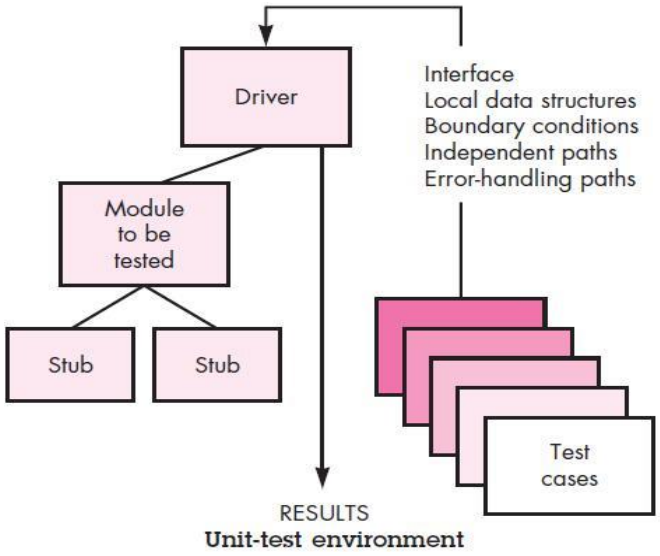
- Unit testing begins at the vortex of the spiral and concentrates on each unit or component of the software as implemented in source code. Unit test is white-box oriented.
- In integration testing, the focus is on design and the construction of the software architecture.
- In validation testing, requirements established as part of software requirements analysis are validated against the software that has been constructed. Black-box test case design techniques are the most prevalent during integration.

- At system testing, the software and other system elements (hardware, people, and databases) are tested as a whole.

1. Unit Testing

After source level code has been developed, reviewed, and verified, unit test case design begins. Each test case should be coupled with a set of expected results. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing Driver and stub are software, written but not delivered with the final software product. Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection in parallel for multiple components. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The main considerations towards the unit test are:



- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- Independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- All error handling paths are tested.

2. Integration Testing

After completion of the unit test, there will be problem by the collective result of individual modules on integration. The major problems on putting the individually tested components together will be:

- Data can be lost across an interface
- One module can have an adverse effect on another
- Sub-functions, when combined, may not produce the desired major function
- Global data structures can create problems

The individual components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction. Thus, Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The major general steps for the integration testing are listed as:

(a) The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

(b) Depending on the integration approach selected (depth or breadth first), subordinate stubs are replaced one at a time with actual components.

(c) Tests are conducted as each component is integrated.

(d) On completion of each set of tests, another stub is replaced with the real component.

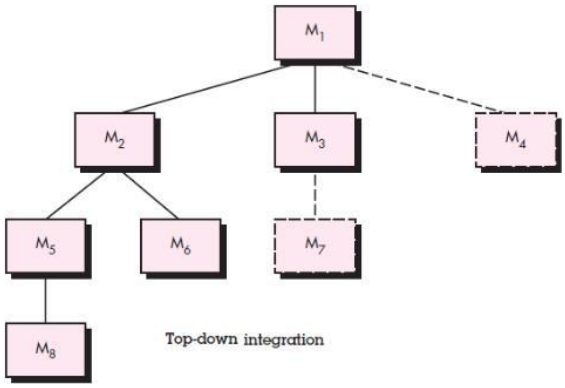
(e) Regression testing may be conducted to ensure that new errors have not been introduced.

The objective is to take unit-tested components and build a program structure that has been dictated by design. There are two approaches for integration testing: Top-down and Bottom-up.

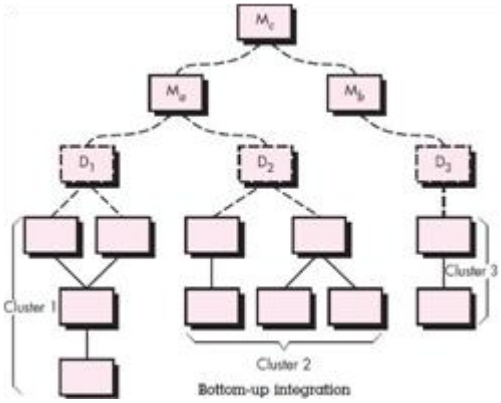
a) Top-down integration testing

It is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner and thus finally the whole programmed structure has been constructed.

- For depth first integration, selecting the left hand path, components M1, M2, M5 would be integrated first. Next,M8 or M6 would be integrated. Then, the central and right hand control paths are built.
- From breath first integration, components M2, M3, and M4 would be integrated first where M1 become the driver. The next control level, M5, M6 (M2 is driver component), and so on.



Top-down integration



Bottom-up integration

b) Bottom-up integration testing

This begins construction and testing with atomic modules (components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

- Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub-function.
- A driver (a control program for testing) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

c) Regression testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Successful tests result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

3. Validation Testing

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered

and corrected. At the validation or system level, the distinction between conventional software, object-oriented software, and Web Apps disappears. Testing focuses on user-visible actions and user-recognizable output from the system. Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end user. The validation testing mainly concerned with following tasks:

a) Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. After each validation test case has been conducted, one of two possible conditions exists:

- The function or performance characteristics conform to specification and are accepted
- A deviation from specification is uncovered and a deficiency list is created

b) Configuration Review

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an audit.

c) Alpha and Beta Testing

When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.

i. *Alpha test* is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

ii. *Beta test* is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. The Beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.

4. System Testing

Software is only one element of a larger computer-based system. Software is incorporated with other system elements (hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers.

A classic system testing problem is "finger-pointing", occurs when an error is uncovered, and each system element developer blames the other for the problem. Thus, system testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. The system testing can be classified in following diversified fields:

a) Recovery Testing

Many computer based systems must recover from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur. Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check-pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

b) Security Testing

Any system that manages sensitive information is a target for illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain. Security testing attempts to verify that protection mechanisms built into a system will protect it from improper penetration. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system.

c) Stress Testing

Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails? Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example:

- special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
- input data rates may be increased by an order of magnitude to determine how input functions will respond
- test cases that require maximum memory or other resources are executed
- test cases that may cause thrashing in a virtual operating system are designed
- Test cases that may cause excessive hunting for disk-resident data are created.

d) Performance Testing

For Real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. External instrumentation can monitor execution intervals, log events (interrupts) as they occur, and sample machine states on a regular basis.

4.5.5 White Box Testing

- The structural testing is the testing of the structure of the system or component.
- White-box testing is often referred to as 'Structural' or 'glass box' or 'clear-box testing' because in white-box testing, we are interested in what is happening 'inside the system/application'.
- In white-box testing, the testers are required to have the knowledge of the internal implementations of the code. Here the testers require knowledge of how the software is implemented, how it works.
- During white-box testing the tester is concentrating on how the software does it. For example, a structural technique wants to know how loops in the software are working. Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.
- White-box testing can be used at all levels of testing. Developers use structural testing in component testing and component integration testing, especially where there is good tool support for code coverage. Structural testing is also used in system and acceptance testing, but the structures are different. For example, the coverage of menu options or major business transactions could be the structural element in system or acceptance testing.

Types of White-Box Testing:

1. Basic Path Testing
2. Cyclomatic Complexity Testing
3. Graphics Matrix
4. Control structure Testing

1. Basic Path Testing

- Path testing is a structural testing method based on the source code or algorithm and NOT based on specification.
- Used flow graphs for representing the control flow or logical flow as show below:

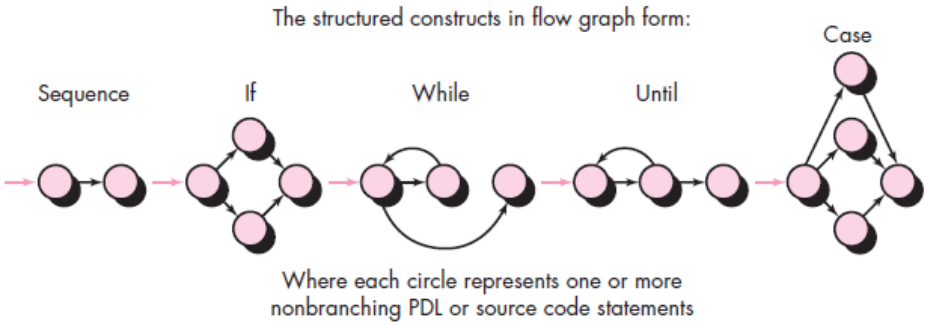


Fig: Flow graph notation

2. Cyclomatic Complexity

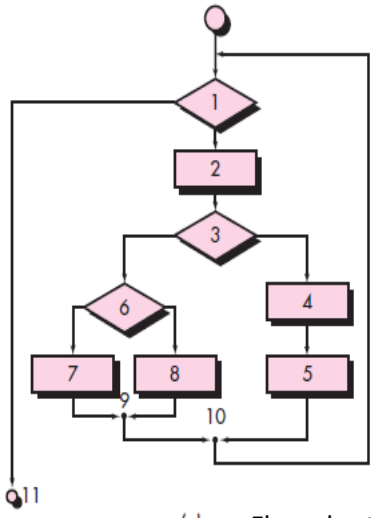
- Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors.
- It is calculated by developing a Control Flow Graph (CGF) of the code that measures the number of linearly- independent paths through a program module.
- Lower the cyclomatic complexity, lower the risk to modify and easier to understand.
- Cyclomatic complexity = E- N +2

  Where:

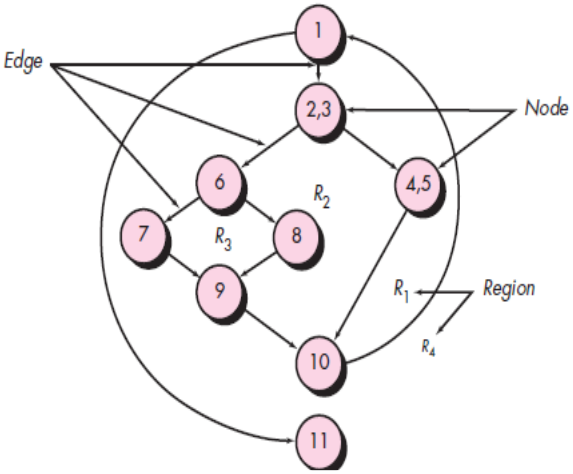
  E = no. of edges in CGF

  N= no. of nodes in CGF
- Also

  Cyclomatic complexity = no of regions in the CFG



(a) Flow chart



(b) Flow graph

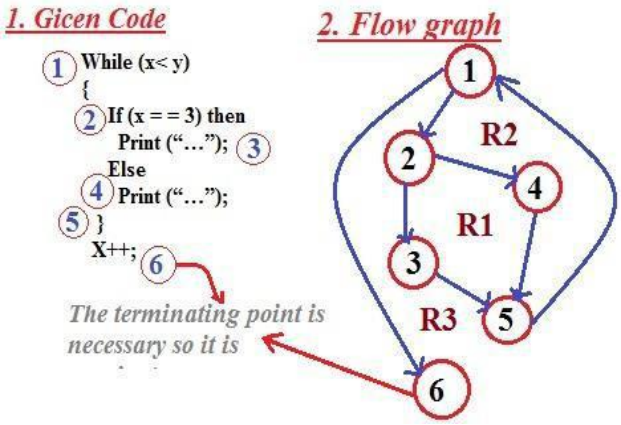
- Here , No of edges (E) =11

  No of nodes(N) = 9

  Cc = 11-9+2 = 4

Also, Cc = no of regions = 4

Another Example:



3) Cyclomatic Complexity:

a) Here,

Number of region (R) = 3

Thus, Cc = R = 3

b) Again,

Number of Edges (E) = 7

Number of Nodes (N) = 6

Thus, Cc = E- N + 2 = 7-6 +2 = 3

4. Basic paths

- Path 1: 1 – 6
- Path 2: 1 –2–3–5–1–6
- Path 3: 1 –2–4–5–1–6

5)Test Cases

- For Path: 1 – 6

  A test case is: X=4 & Y=3
- For Path: 1 – 2 – 3 – 5 – 1 – 6

  A test case is: X = 3
- For Path: 1 – 2 – 3 – 5 – 1 – 6

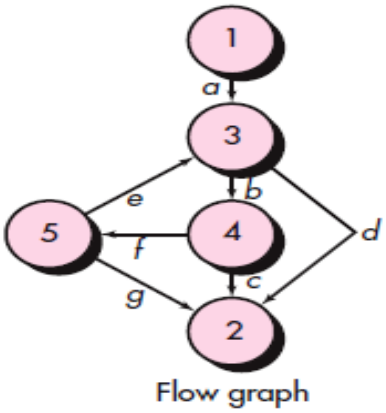
  A test case is: X = 2 or any other except 3

Note:

- Each & every nodes must be terminating on one node so we need the node – 6 on the given example.
- The Cyclomatic complexity must be equal in all these three cases.
- If there are other paths beside the Basic path then they are redundant path.
- The test case design has the highest probability of finding the most errors with a minimum amount of time and effort.

3. Graphics matrix

- A graph matrix is a square matrix whose size (i.e. no of rows and columns) is equal to the no of nodes on the flow graph.
- This graph matrix is useful for basic path testing.



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

4. Control Structure Testing

The basis path testing technique is one of a number of techniques for control structure testing but it is not sufficient on control structure testing. Some popular control structure testing techniques are described below:

a) Condition Testing

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. A relational expression takes the form:

```
E1 <relational-operator> E2
```

Where *E*1 and *E*2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠, >, or ≥. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&), and NOT (¬). A condition without relational expressions is referred to as a Boolean expression.

b) Data Flow Testing

The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program. To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number,

```
DEF(S) = {X | statement S contains a definition of X} USE(S) = {X |
statement S contains a use of X}
```

If statement *S* is an *if* or *loop statement*, its DEF set is empty and its USE set is based on the condition of statement *S*. The definition of variable *X* at statement *S* is said to be *live* at statement *S'* if there exists a path from statement *S* to statement *S'* that contains no other definition of *X*.

c) Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. *Loop testing* is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

- **Simple loops:** They are sequentially tested for the data of desired domain.
- **Nested loop:** They are started to test from innermost loop.
- **Concatenated loop:** Multiple loops are tested simultaneously if there is no dependency exists.
- **Unstructured loops:** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

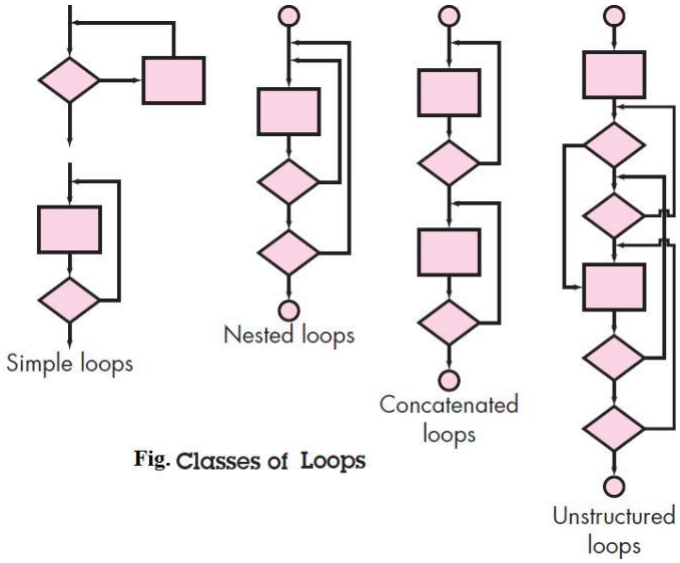


Fig. Classes of Loops

4.5.6 Black Box Testing

- Black-box Testing is a testing technique that is used to test the features/functionality of the system or Software.
- Black-box Testing is to test the functionality of the software application under test.
- In Black-box Testing we need to check if each components are functioning as expected or not, so it is also called as "**Component Testing**".
- Generates test cases based on the functionality of software.
- Also known as Functional Testing and Specification Testing
- The internal program structure is hidden from the testing process.

Types of Black Box Testing:

1. Boundary Value analysis (BVA)
2. Equivalence Class Testing
3. Decision Table based testing
4. Special Value Testing

1. Boundary Value analysis (BVA)

- BVA is the testing technique meant for testing the boundaries of the input domain for errors.
- Here, we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions).
- For eg. A form accepts product delivery days: a minimum of 2 days and maximum of 14 days are noted on the form.

  Boundary value test might put values of 1, 2, 14, 15 to determine how the function reacts to data.

2. Equivalence Class Testing

- It is a software testing technique that divides the input data of the application under test into each partition at least once of equivalent data from which test cases can be derived.
- It reduces the time required for performing testing of a software due to less number of test cases.
- For eg.

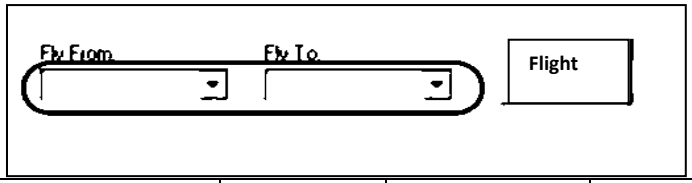
  Consider that a form accepts an integer in the range of 100 to 999.

  Valid equivalence class partition: 100 to 999 inclusive

  Non-valid equivalence class partition : less than 100,
  more than 999,
  decimal no,
  alphabets and non numeric characters.

3. Decision Table based testing

- A decision table based testing uses a decision table. A decision table is a good way to deal with combinations of inputs, which produce different results.
- Lets us consider the behavior of "FLIGHT" button for different combination of FLY FROM and FLY TO.



Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F	F	T	T
FLY TO	F	T	F	T
Outcome FLIGHT	F	F	F	T

Here, FLGHT button is enabled when both FLY FROM and FLY TO are set.

4. Special Value Testing

The operators used for comparison (>, <, >=, =<) have the 'special' property that programmers are likely to include or exclude the = sign.

There are other special values which it is always wise to include in test data sets because of their special properties:

- zeroes or nulls - these have the property that they don't generate a change in value for addition or subtraction *and*
- ones - these don't generate changes in values for multiplication and division.

4.7 Software Configuration Management

When you build computer software, change happens. And because it happens, you need to manage it effectively. Software configuration management (SCM), also called change management, is a set of activities designed to manage change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made.

Software configuration management (SCM) is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to

1. identify change,
2. control change,
3. ensure that change is being properly implemented, and
4. report changes to others who may have an interest.

The four fundamental sources of change on software product will be:

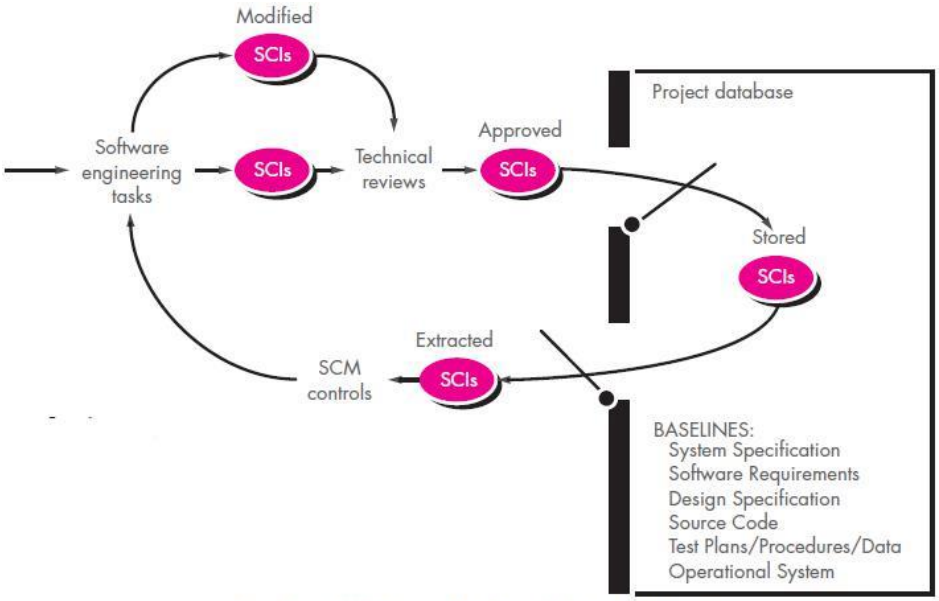
- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

Baselines

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a technical review. For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved.

Before a software configuration item (SCI) becomes a baseline (or placed in a project database, also called a project library or software repository), change can be made quickly or informally. However, once a baseline is established, we figuratively pass through a swinging one-way door. Now, the formal procedure must be applied to evaluate and verify each change. The procedure to make changes on baseline product will be:

- Send a private copy of module needing modification to change control board (CCB)
- Get permission from the CCB for restoring change module
- After review from CCB, the manager updates the old baseline by restore operation.

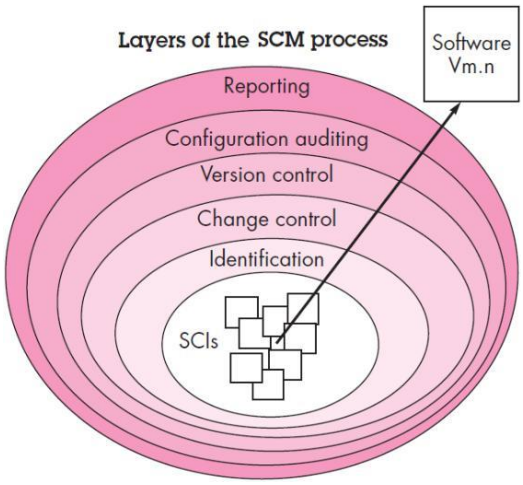


Baselined SCIs and the project database

4.6.1 SCM Process

The software configuration management process defines a series of tasks that have four primary objectives:

1. to identify all items that collectively define the software configuration,
2. to manage changes to one or more of these items,
3. to facilitate the construction of different versions of an application, and
4. to ensure that software quality is maintained as the configuration evolves over time.



1. Identification

To control and manage software configuration items, each should be separately named and then organized using an object-oriented approach. Two types of objects can be identified: basic objects and aggregate objects.

- A *basic object* is a unit of information that you create during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, part of a design model, source code for a component, or a suite of test cases that are used to exercise the code.
- An *aggregate object* is a collection of basic objects and other aggregate objects. For example, a *DesignSpecification* is an aggregate object.

2. Change Control

Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed.

3. Version control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities:

1. A project database (repository) that stores all relevant configuration objects,
2. A *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions),
3. A *make facility* that enables you to collect all relevant configuration objects and construct a specific version of the software.

In addition, version control and change control systems often implement an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

4. Configuration Audit

The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

5. Status Reporting

Configuration status Reporting (status accounting) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

The flow of information for configuration status reporting (CSR) is illustrated in figure:

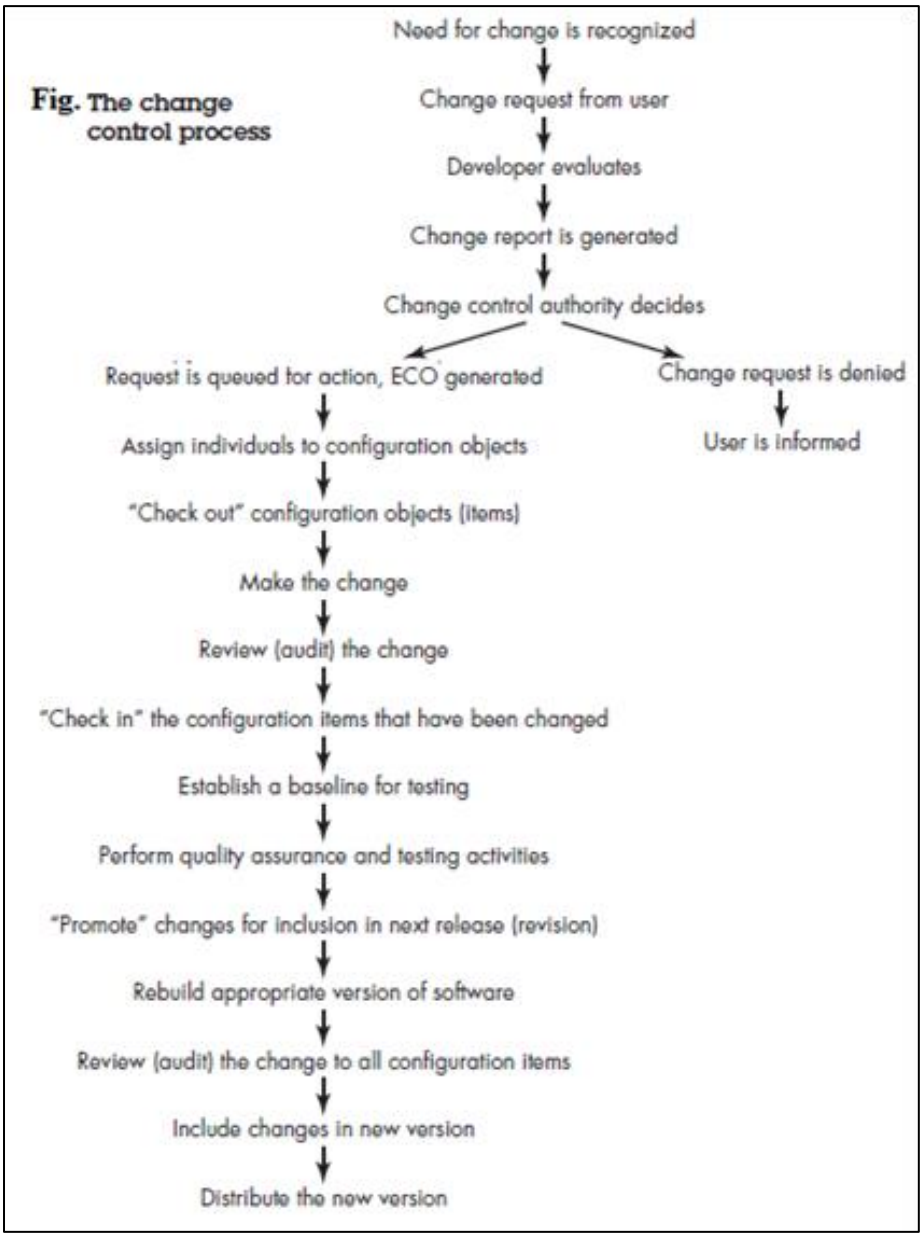


Fig. The change control process

Board Exam Questions:

1. Demonstrate the use of control structure testing with suitable example. [2019spring]
2. What is the difference between an SCM audit and a technical review? Can their function be folded into one review? What are the pros and cons? [2019 Spring]
3. Demonstrate the relationship between Mean-time-between-failure (MTBF). Mean-time-to-failure (MTTF), mean-time-to-repair (MTTR) with service availability with suitable example. [2019 Spring, 2019 fall]
4. What is FTR? Why is it important in SQA activities? Explain how FTR is conducted. [2019 fall]
5. How do you assure the quality of a software? Explain the software review and FTR. [201 fall]
6. What do you mean by Verification and Validation? Discuss the Basic path testing and Cyclomatic complexity of White box testing with an example. [2018 fall]
7. Why SCM is important during software development? Discuss the change control and version control in brief. [2018 fall]
8. What do you mean by software reliability? Explain the statistical SQA and six-sigma regarding in brief. [2018 fall]
9. Briefly explain the set of guidelines for FTR. [2017 spring]
10. Considering Online shopping software, explain test strategies. [2017 spring]
11. Define Cyclomatic Complexity (Cc). Prove that all methods that calculate Cc result the same value. [2017 spring]
12. Define software quality. Explain the activities of software quality assurance (SQA) group for achieving a high-quality end product. [2017 spring]
13. Define the meaning of software quality and detail the factors which affect the quality not productivity of a software product. [2017 fall]
14. What are test cases and what is the importance of testing and designing test cases. [2017 fall, 2015 spring]
15. What do you mean by SCM? List out the difference between change control and version control in SCM. [2017 fall]
16. Suppose a hotel is going to design an online hotel booking system for its guests. There are following requirements: [2016 spring, 2014 fall]
    - Cost-effectiveness
    - Reliability
    - User friendliness

    Suggest priority of the above-mentioned representative software qualities in most required to least required order. Also give appropriate reasons in favor of your answer.
17. What is FTR? Why is it important in SQA activities? Explain briefly about Defect amplification and removal. [2016 spring]
18. What do you mean by SCM? Explain with example. [2016 spring]
19. How FTR is taken as a measure to maintain the quality of a software project? Explain. [2016 fall]
20. Discuss the significance of unit testing and integration testing in object-oriented life cycle for system development. [2016 fall]

21. What are software quality controls and software quality assurance? How can we produce high-quality software products? [2016 fall]
22. What is integration testing? Differentiate between Top-down and Bottom-up integration testing. [2015 spring]
23. List out the differences between quality control and quality assurance. And explain the elements and goals of SQA. [2015 spring]
24. What is the purpose of Unit testing? Draw control flow of a program to find the largest number among three numbers and find the cyclomatic complexity of the program. [2015 fall]
25. What is software quality control and software quality assurance? Explain in brief about the representative qualities of software. [2015 fall]
26. What is software quality control and software quality assurance? Explain FTR as a measure to maintain the quality of a software project. [2014 spring]
27. How would you define Verification and Validation? Discuss briefly on White Box Test versus Black Box Test. [2014 spring, 2014 fall]
28. As a project manager, how can you ensure to your customer that your software product has quality? Discuss the procedures to control the quality of software during development. [2013 spring]
29. How testing control the quality of product? Discuss White box testing and Black box testing with suitable example. [2013 spring]
30. What is the difference between software quality and reliability? How can quality be assured in a software? [2013 fall]
31. Write short notes on:
    a) Baseline [2019 spring]
    b) People CMM [2019 fall]
    c) Alpha and Beta testing [2019 fall]
    d) ISO standards [2018 fall, 2017 fall]
    e) Verification and validation [2016 spring]
    f) Cyclomatic complexity testing method [2016 fall]
    g) Six-sigma [2015 spring]
    h) Debugging [2015 fall]
    i) SCM [2013 spring]

Object Oriented Software Engineering

Chapter 5

ADVANCED TOPICS

IN

SOFTWARE ENGINEERING

Er. Shiva Ram Dam

Shivaram.dam@pec.edu.np

5.1 Software Process Improvement

Some software organizations have little more than an ad hoc software process. As they work to improve their software engineering practices, they must address weaknesses in their existing process and try to improve their approach to software work. *Software process improvement* encompasses a set of activities that will lead to a better software process and, as a consequence, higher-quality software delivered in timely manner. The approach to SPI is iterative and continuous, but it can be viewed in five steps:

1. Assessment of the current software process,
2. Education and training of practitioners and managers,
3. Selection and justification of process elements, software engineering methods, and tools,
4. Implementation of the SPI plan, and
5. Evaluation and tuning based on the results of the plan.

5.1.1 Software Process Improvement Framework

Although an organization can choose a relatively informal approach to SPI, the vast majority choose one of a number of SPI frameworks. An *SPI framework* defines

1. A set of characteristics that must be present if an effective software process is to be achieved,
2. A method for assessing whether those characteristics are present,
3. A mechanism for summarizing the results of any assessment, and
4. A strategy for assisting a software organization in implementing those process characteristics that have been found to be weak or missing.

An SPI framework assesses the "maturity" of an organization's software process and provides a qualitative indication of a maturity level. Figure provides an overview of a typical SPI framework. The key elements of the framework and their relationship to one another are shown.

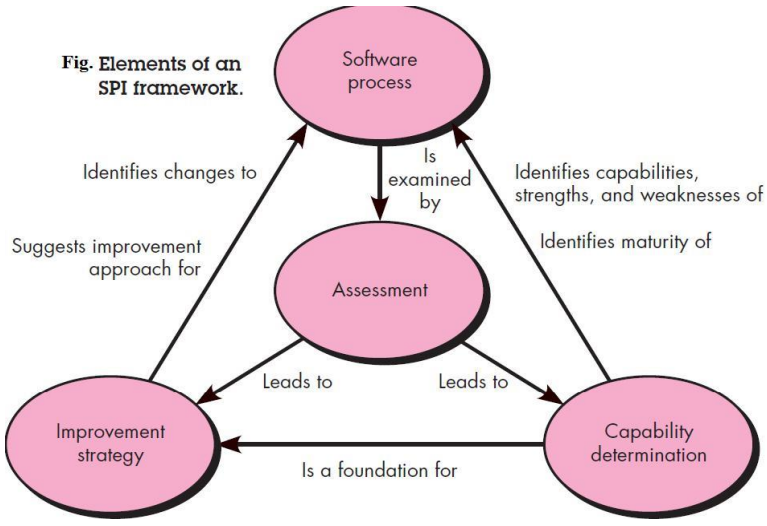


Fig. Elements of an SPI framework.

5.1.2 Software Process Improvement Process

The Software Engineering Institute has developed IDEAL—"an organizational improvement model that serves as a roadmap for initiating, planning, and implementing improvement actions". IDEAL is representative of many process models for SPI, defining five distinct activities— initiating, diagnosing, establishing, acting, and learning—that guide an organization through SPI activities.

The five activities are applied during SPI process in an iterative (cyclical) manner in an effort to foster continuous process improvement.

1. **Assessment and Gap Analysis:**
   The intent of assessment is to uncover both strengths and weaknesses in the way our organization applies the existing software process and the software engineering practices that populate the process. Assessment examines a wide range of actions and tasks that will lead to a high quality process.

2. **Education and Training:**
   To follow new software engineering practices, practitioners and managers must provide the required education and training. Generally, three types of education and training should be conducted: *Generic concepts and methods*, *Specific technology and tools*, and *Business communication and quality-related topics*. In a modern context, education and training also can be delivered from podcasts, to Internet-based training, to DVDs, to classroom courses can be offered as part of an SPI strategy.

3. **Selection and Justification:**
   Selection and justification activity characteristics and specific software engineering methods and tools are chosen to populate the current software process. During this stage first the best process model is chosen for the organization, its stakeholders, and the software that build. Next, develop a work breakdown for each framework activity (e.g., modeling), defining the task set that would be applied for a typical project.

4. **Installation/Migration:**
   Installation and migration are actually *software process redesign* (SPR) activities. *Installation* is the first point at which a software organization feels the effects of changes implemented as a consequence of the SPI road map. In some cases, an entirely new process is recommended for an organization. Framework activities, software engineering actions, and individual work tasks must be defined and installed as part of a new software engineering culture. In other cases, changes associated with SPI are relatively minor, representing small, but meaningful modifications to an existing process model. Such changes are often referred to as *process migration*.

5. **Evaluation:**
   The evaluation activity assesses the degree to which changes have been instantiated and adopted, the degree to which such changes result in better software quality or other tangible process benefits, and the overall status of the process and the organizational culture as SPI activities proceed. Both qualitative factors and quantitative metrics are considered during the evaluation activity

5.1.3 CMMI

The original CMM was developed and upgraded by the Software Engineering Institute throughout the 1990s as a complete SPI framework. Today, it has evolved into the *Capability Maturity Model Integration* (CMMI), a comprehensive process meta-model that is predicated on a set of system and software engineering capabilities that should be present as organizations reach different levels of process capability and maturity.

- **Level 0:** *Incomplete*—the requirements management is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability for the process area.

- **Level 1:** *Performed*—all of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

- **Level 2:** *Managed*—all capability level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description".

- **Level 3:** *Defined*—all capability level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets".

- **Level 4:** *Quantitatively managed*—all capability level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process".

- **Level 5:** *Optimized*—all capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

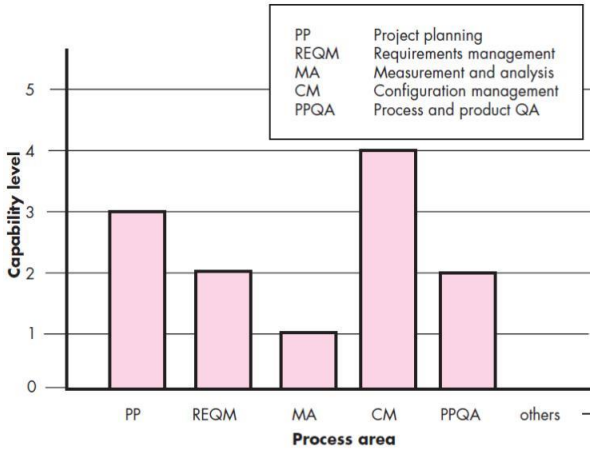


Fig. CMMI process area capability profile.

5.1.4 People CMM

A software process will not succeed without talented, motivated software people. The *People Capability Maturity Model* "is a roadmap for implementing workforce practices that continuously improve the capability of an organization's workforce". It was developed in the mid-1990s and refined over the intervening years. The goal of the People CMM is to encourage continuous improvement of generic workforce knowledge, specific software engineering and project management skills, and process-related abilities.

Level	Focus	Process Areas
Optimized	*Continuous improvement*	Continuous workforce innovation Organizational performance alignment Continuous capability improvement
Predictable	*Quantifies and manages knowledge, skills, and abilities*	Mentoring Organizational capability management Quantitative performance management Competency-based assets Empowered workgroups Competency integration
Defined	*Identifies and develops knowledge, skills, and abilities*	Participatory culture Workgroup development Competency-based practices Career development Competency development Workforce planning Competency analysis
Managed	*Repeatable, basic people management practices*	Compensation Training and development Performance management Work environment Communication and co-ordination Staffing
Initial	*Inconsistent practices*	

5.1.5 Other software Process Improvement Frameworks

Although the SEI's CMM and CMMI are the most widely applied SPI frameworks, a number of alternatives have been proposed and are in use. Among the most widely used of these alternatives are:

1. **SPICE:** The SPICE (*Software Process Improvement and Capability dEtermination*) model provides an SPI assessment framework that is compliant with ISO 15504:2003 and ISO 12207. The SPICE document suite presents a complete SPI framework including a model for process management, guidelines for conducting an assessment and rating the process under consideration, construction, selection, and use of assessment instruments and tools, and training for assessors.

2. **Bootstrap:** The *Bootstrap* SPI framework "has been developed to ensure conformance with the emerging ISO standard for software process assessment and improvement (SPICE) and to align the methodology with ISO 12207". The objective of Bootstrap is to evaluate a software process using a set of software engineering best practices as a basis for assessment. Like the CMMI, Bootstrap provides a process maturity level using the results of questionnaires that gather information about the "as is" software process and software projects. SPI guidelines are based on maturity level and organizational goals.

3. **PSP and TSP:** Although SPI is generally characterized as an organizational activity, there is no reason why process improvement cannot be conducted at an individual or team level. Both PSP and TSP emphasize the need to continuously collect data about the work that is being performed and to use that data to develop strategies for improvement.

4. **TickIT:** The Ticket auditing method ensures compliance with *ISO 9001:2000 for Software*—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

5.1.6 Software Process Improvement Return in Investment

SPI is hard work and requires substantial investment of dollars and people. Managers who approve the budget and resources for SPI will invariably ask the question: "How do I know that we'll achieve a reasonable return for the money we're spending?"

At a qualitative level, proponents of SPI argue that an improved software process will lead to improved software quality. They contend that improved process will result in the implementation of better quality filters, better control of change, and less technical rework. But can these qualitative benefits be translated into quantitative results? The classic return on investment (ROI) equation is:

$$\text{ROI} = \left[\frac{\Sigma(\textit{benefits}) - \Sigma(\textit{costs})}{\Sigma(\textit{costs})}\right] \times 100\%$$

Where

- *benefits* include the cost savings associated with higher product quality (fewer defects), less rework, reduced effort associated with changes, and the income that accrues from shorter time-to-market.
- *costs* include both direct SPI costs (e.g., training, measurement) and indirect costs associated with greater emphasis on quality control and change management activities and more rigorous application of software engineering methods (e.g., the creation of a design model).

5.2 Emerging Trends in Software Engineering

No one can predict the future with absolute certainty. But it is possible to assess trends in the software engineering area and from those trends to suggest possible directions for the technology. Throughout the relatively brief history of software engineering, practitioners and researchers have developed an array of process models, technical methods, and automated tools in an effort to foster fundamental change in the way we build computer software. Even though past experience indicates otherwise, there is a tacit desire to find the "silver bullet"—the magic process or transcendent technology that will allow us to build large, complex, software-based systems easily, without confusion, without mistakes, without delay— without the many problems that continue to plague software work.

But history indicates that our quest for the silver bullet appears doomed to failure. New technologies are introduced regularly, hyped as a "solution" to many of the problems, software engineers face, and incorporated into project: large and small. Industry pundits stress the importance of these "new" software technologies, the cognoscenti of the software community adopt them with enthusiasm, and ultimately, they do play a role in the software engineering world. But they tend not to meet their promise, and as a consequence, the quest continues.

5.2.1 Technology Evolution

Evolution occurs as a result of positive feedback—"the more capable methods resulting from one stage of evolutionary progress are used to create the next stage".

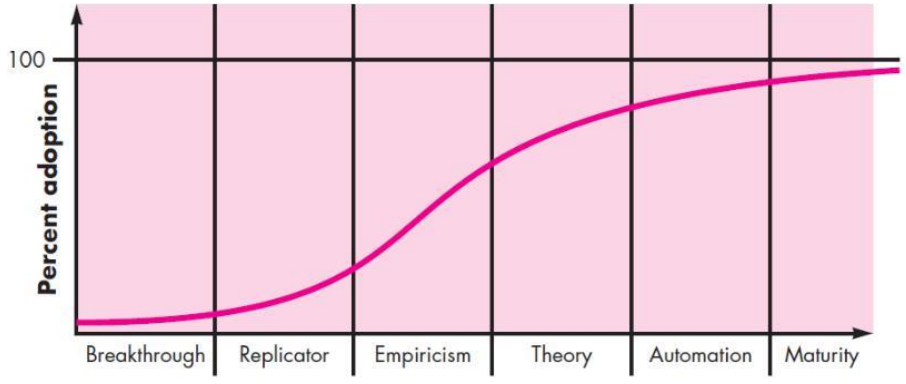


Fig. A technology innovation life cycle

When a successful new technology is introduced, the initial concept moves through a reasonably predictable "innovation life cycle" as shown in figure. In the *breakthrough* phase, a problem is recognized and repeated

attempts at a viable solution are attempted. At some point, a solution shows promise. The initial breakthrough work is reproduced in the *replicator* phase and gains wider usage. *Empiricism* leads to the creation of empirical rules that govern the use of the technology, and repeated success leads to a broader *theory* of usage that is followed by the creation of automated tools during the *automation* phase. Finally, the technology matures and is used widely.

Computing technology is evolving at an exponential rate, and its growth may soon become explosive. Kurzweil suggests that within 20 years, technology evolution will accelerate at an increasingly rapid pace, ultimately leading to an era of non-biological intelligence that will merge with and extend human intelligence in ways that are fascinating to contemplate.

5.2.2 Observing Software Engineering Trends

Soft trends have a significant impact on the overall direction of software engineering. In the present context of software engineering, Barry Boehm suggests that "software engineers [will] face the often-formidable challenges of dealing with rapid change, uncertainty and emergence, dependability, diversity, and interdependence, but they also have opportunities to make significant contributions that will make a difference for the better."

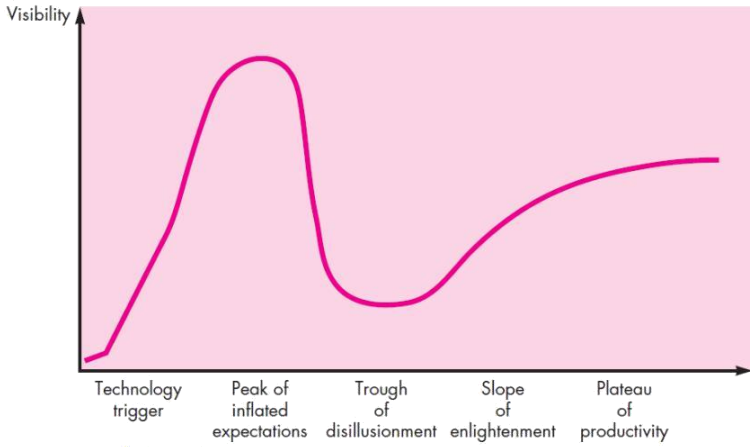


Fig. The Gartner Group's hype cycle for emerging technologies.

The Gartner Group—a consultancy that studies technology trends across many industries—has developed a *hype cycle for emerging technologies*. The "hype cycle" presents a realistic view of short-term technology integration. The Gartner Group cycle exhibits five phases:

1. *Technology trigger*—a research breakthrough or launch of an innovative new product that leads to media coverage and public enthusiasm.
2. *Peak of inflated expectations*—overenthusiasm and overly optimistic projections of impact based on limited, but well-publicized successes.
3. *Disillusionment*—overly optimistic projections of impact are not met and critics begin the drumbeat; the technology becomes unfashionable among the cognoscenti.
4. *Slope of enlightenment*—growing usage by a wide variety of companies leads to a better understanding of the technology's true potential; off-the-shelf methods and tools emerge to support the technology.
5. *Plateau of productivity*—real-world benefits are now obvious, and usage penetrates a significant percentage of the potential market.

5.2.3 Identifying Soft Trends

Each nation with a substantial IT industry has a set of unique characteristics that define the manner in which business is conducted, the organizational dynamics that arise within a company, the distinct marketing issues that apply to local customers, and the overriding culture that dictates all human interaction. *Globalization* leads to a diverse workforce. This, in turn, demands a flexible organizational structure.

It is estimated that over one billion new consumers will enter the worldwide marketplace over the next decade. Consumer spending in "emerging economies will double to well over $9 trillion". In some world regions, the population of software construction is aging and need to transfer the knowledge to new generation. Now the question is that, "Can new software engineering technologies be developed to meet this worldwide demand?"

1. **Managing Complexity:** Modern computer-based systems has increasing the LOC from million to billion and that challenge to manage the huge complexity.

2. **Open-World Software:** The "open-world software" is the software that is designed to adapt to a continually changing environment "by self-organizing its structure and self-adapting its behavior".

3. **Emergent Requirements:** Requirements will emerge as everyone involved in the engineering and construction of a complex system learns more about it, the environment in which it is to reside, and the users who will interact with it. This reality implies a number of software engineering trends. First, process models must be designed to embrace change and adopt the basic tenets of the agile philosophy. Next, methods that yield engineering models must be used judiciously because those models will change repeatedly as more knowledge about the system is acquired. Finally, tools that support both process and methods must make adaptation and change easy.

4. **The Talent Mix:** As software-based systems become more complex, as communication and collaboration among global teams becomes commonplace. Each software team must bring a variety of creative talent and technical skills to its part of a complex system, and the overall process must allow the output of these islands of talent to merge effectively.

5. **Software Building Blocks:** software engineering philosophy have emphasized the need for reuse—of source code, object-oriented classes, components, patterns, and COTS software. The important issues is that the components must be proven or well tested.

6. **Changing Perceptions of "Value":** Modern customers are interested on the software system having speed of delivery, richness of functionality, and overall product quality.

7. **Open Source:** The term *open source* when applied to computer software, implies that software engineering work products (models, source code, test suites) are open to the public and can be reviewed and extended (with controls) by anyone with interest and permission. The promise of open source is better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in."

5.2.4 Technology Directions

Software engineering will change more rapidly. a few trends in process, methods, and tools that are likely to have some influence on software engineering over the next decade.

1. **Process Trends:** A software procurer's goal is to select the best contractor objectively and rationally. A software company's goal is to survive and grow in a competitive market. An end user's goal is to acquire the software product that can solve the right problem, at the right time, at an acceptable price.

2. **The Grand Challenge:** software engineers can meet the daunting challenge of complex software systems development by creating new approaches to understanding system models and using those models as a basis for the construction of high-quality next generation software. The major challenges are: *Multi-functionality, Reactivity and timeliness, New modes of user interaction, Complex architectures, Heterogeneous, distributed systems, Criticality or Safety,* and *Maintenance variability*.

3. **Collaborative Development:** Collaboration involves the timely dissemination of information and an effective process for communication and decision making. Today, software engineers collaborate across time zones and international boundaries, and every one of them must share information. The same holds for open-source projects in which hundreds or thousands of software developers work to build an open-source app. Again, information must be disseminated so that open collaboration can occur.

4. **Requirements Engineering:** Basic requirements engineering actions are elicitation, elaboration, negotiation, specification, and validation. The success or failure of these actions has a very strong influence on the success or failure of the entire software engineering process. Today, most "informal" requirements engineering approaches begin with the creation of user scenarios (e.g., use cases).

5. **Model-Driven Software Development:** Model-driven approaches address a continuing challenge for all software developers—how to represent software at a higher level of abstraction than code.

6. **Postmodern Design:** Postmodern design will continue to emphasize the importance of software architecture. A designer must state an architectural issue, make a decision that addresses the issue, and then clearly define the assumptions, constraints, and implications that the decision places on the software as a whole.

7. **Test-Driven Development:** In *test-driven development* (TDD), requirements for a software component serve as the basis for the creation of a series of test cases that exercise the interface and attempt to find errors in the data structures and functionality delivered by the component. TDD is not really a new technology but rather a trend that emphasizes the design of test cases *before* the creation of source code.

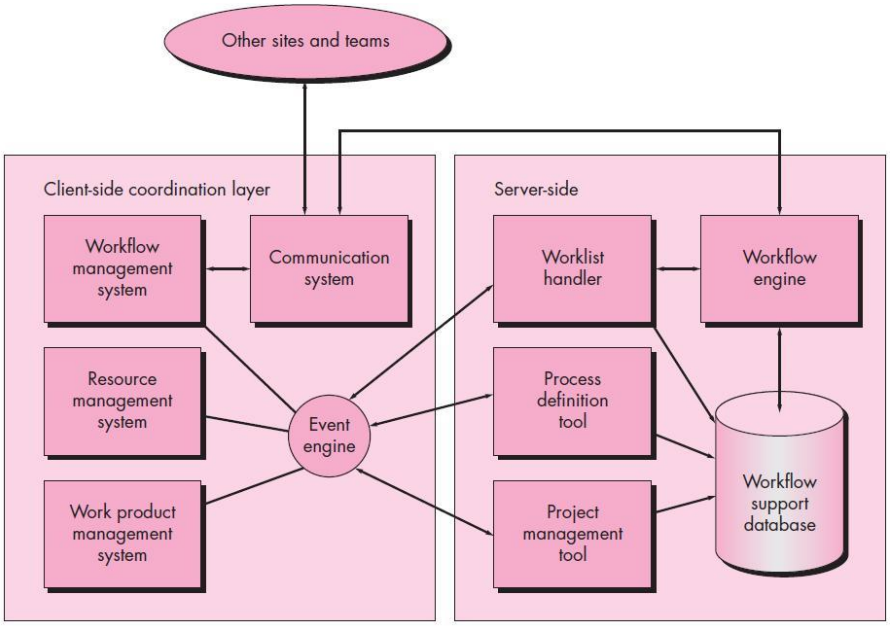
5.2.5 Tools Related Trends

Hundreds of software engineering tools are introduced each year. These claim to improve project management, or requirements analysis, or design modeling, or code generation, or testing, or change management, or any of the many software engineering activities, actions, and tasks. Other tools have been developed as open-source offerings. Future trends in software tools will follow two distinct paths—a *human-focused path* that responds to some of the "soft trends" and a technology-centered path that addresses new technologies.

a. Tools That Respond to Soft Trends:

The soft trends need to manage complexity, accommodate emergent requirements, establish process models that embrace change, coordinate global teams with a changing talent mix, among others—suggest a new era in which tools support for stakeholder collaboration will become as important as tools support for technology.

A collaborative software engineering environment (SEE) "supports co-operation and communication among software engineers belonging to distributed development teams involved in modeling, controlling, and measuring software development and maintenance processes. Moreover, it includes an artifact management function that stores and manages software artifacts (work products) produced by different teams in the course of their work". Figure illustrates an architecture for a collaborative SEE.



b. Tools That Address Technology Trends:

Tools environments will respond to a growing need for communication and collaboration and at the same time integrate domain-specific point solutions that may change the nature of current software engineering tasks. In some instances, modeling and testing tools targeted at a specific application domain will provide enhanced benefit when compared to their generic equivalents.

5.2.6 CASE tools

CASE stands for Computer Aided Software Engineering. It means, development and maintenance of software projects with help of various automated software tools.

CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system. There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools are to name a few. Use of CASE tools accelerates the development of project to produce desired result and helps to uncover flaws before moving ahead with next stage in software development.

Advantages of the CASE approach:

- As special emphasis is placed on redesign as well as testing, the servicing cost of a product over its expected lifetime is considerably reduced.
- The overall quality of the product is improved as an organized approach is undertaken during the process of development.
- Chances to meet real-world requirements are more likely and easier with a computer-aided software engineering approach.
- CASE indirectly provides an organization with a competitive advantage by helping ensure the development of high-quality products.

Types of CASE Tools:

1. **Diagramming Tools:**
   It helps in diagrammatic and graphical representations of the data and system processes. It represents system elements, control flow and data flow among different software components and system structure in a pictorial form. For example, Flow Chart Maker tool for making state-of-the-art flowcharts.

2. **Computer Display and Report Generators:**
   It helps in understanding the data requirements and the relationships involved.

3. **Analysis Tools:**
   It focuses on inconsistent, incorrect specifications involved in the diagram and data flow. It helps in collecting requirements, automatically check for any irregularity, imprecision in the diagrams, data redundancies or erroneous omissions.
   For example,
   (i) Accept 360, Accompa, CaseComplete for requirement analysis.
   (ii) Visible Analyst for total analysis.

4. **Central Repository:**
   It provides the single point of storage for data diagrams, reports and documents related to project management.

5. **Documentation Generators:**
   It helps in generating user and technical documentation as per standards. It creates documents for technical users and end users.
   For example, Doxygen, DrExplain, Adobe RoboHelp for documentation.

6. **Code Generators:**
   It aids in the auto generation of code, including definitions, with the help of the designs, documents and diagrams.

UML Based CASE tools:

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing and documenting the artifacts of a software-intensive system. The UML based CASE tools are used to draw UML diagrams such as: Class, usecase, collaboration, sequence and activity diagram.
Some of the widely used UML based CASE tools are:

1. StarUML
2. Umbrello
3. AndroMDA
4. BOUML
5. AgroUML
6. IBM Rational Rose
7. Microsoft Visio
8. SmartDraw
9. Enterprise Architect
10. Boralnd Together

Reference: https://www.slideshare.net/curiousEngine/uml-case-tools

5.2.7 Software Re-engineering

Software Re-Engineering is the examination and alteration of a system to reconstitute it in a new form. The principles of Re-Engineering when applied to the software development process is called software re-engineering. It affects positively at software cost, quality, service to the customer and speed of delivery. In Software Re-engineering, we are improving the software to make it more efficient and effective. For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

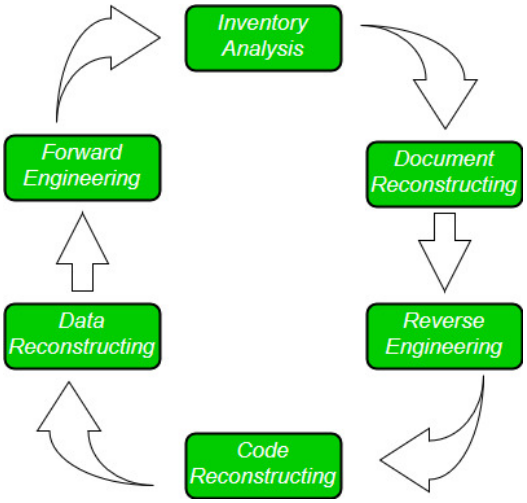


Fig: Software reengineering activities

1. **Inventory Analysis:**
   Every software organization should have an inventory of all the applications.
   - Inventory can be nothing more than a spreadsheet model containing information that provides a detailed description of every active application.
   - By sorting this information according to business criticality, longevity, current maintainability and other local important criteria, candidates for re-engineering appear.
   - Resource can then be allocated to candidate application for re-engineering work.

2. **Document reconstructing:**
   Documentation of a system either explains how it operate or how to use it.
   - Documentation must be updated.
   - It may not be necessary to fully document an application.
   - The system is business critical and must be fully re-documented.

3. **Reverse Engineering:**
   Reverse engineering is a process of design recovery. Reverse engineering tools extracts data, architectural and procedural design information from an existing program.

4. **Code Reconstructing:**
   - To accomplish code reconstructing, the source code is analysed using a reconstructing tool. Violations of structured programming construct are noted and code is then reconstruct.
   - The resultant restructured code is reviewed and tested to ensure that no anomalies have been introduced.

5. **Data Restructuring:**
   - Data restructuring begins with the reverse engineering activity.
   - Current data architecture is dissecred, and necessary data models are defined.
   - Data objects and attributes are identified, and existing data structure are reviewed for quality.

6. **Forward Engineering:**
   Forward Engineering also called as renovation or reclamation not only for recovers design information from existing software but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality.

5.2.8 Business Process Engineering

Business process engineering is a way in which organizations study their current business processes and develop new methods to improve productivity, efficiency, and operational costs. Business process engineering focuses on new business processes, how to diagnose problems with an organization's current methodology, and how to redesign, reconstruct, and monitor processes to ensure they are effective. Business Process Engineering (BPE) uses a proven systematic approach based on the latest experiences and research to achieve significant improvements. The BPE process helps clients fundamentally rethink and reinvent the business processes needed to achieve the firm's strategic objectives through the maximum use of enabling technologies and organizational strategies. A BPE effort can result in 15% to 50% improvement in performance of the targeted business processes, depending upon whether a reengineering or an improvement approach is used in the effort.

Business Process Engineering Approach

Business Process Engineering (BPE) Approach includes:

1. **Understanding the Present Mode of Operation (PMO).** We'll assemble an experienced team to analyze your current processes, technologies and systems. The result will be creation of a detailed PMO business process model showing interrelationships and dependencies between people, systems, and processes. This will serve as the baseline that proposed changes and actual implementations are evaluated against.
2. **Determining the Future Mode of Operation (FMO).** We'll work with you to build an advisory team to define an FMO business process model based on your business objectives and our combined knowledge of industry best practices.
3. **Gap Analysis and Transition Plan.** A gap analysis of needed business process improvements and transition to the FMO plan will be developed. You'll gain an understanding of the business strategy, timing, personnel, and system/process evolution that will take place.
4. **Implementation.** By trialing and deploying new system or operational process improvements, we'll help you determine whether the intended results will be achieved. If additional system and operational process improvements need to be made, we'll repeat the appropriate PMO, FMO, Gap Analysis and Transition Planning steps as necessary.

5.2.9 Reverse and Forward Engineering

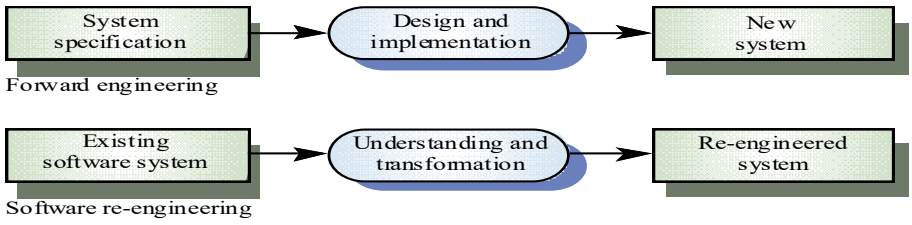


Fig. Forward Engineering and Reengineering

Forward Engineering is a method of creating or making an application with the help of the given requirements. Forward engineering is also known as **Renovation and Reclamation**. Forward engineering is required high proficiency skill. It takes more time to construct or develop an application.

Reverse Engineering is also known as backward engineering, is the process of forward engineering in reverse. In this, the information are collected from the given or exist application. It takes less time than forward engineering to develop an application. In reverse engineering the application are broken to extract knowledge or its architecture.

Difference between Forward Engineering and Reverse Engineering:

S.NO	FORWARD ENGINEERING	REVERSE ENGINEERING
1.	In forward engineering, the application are developed with the given requirements.	In reverse engineering or backward engineering, the information are collected from the given application.
2.	Forward Engineering is high proficiency skill.	Reverse Engineering or backward engineering is low proficiency skill.
3.	Forward Engineering takes more time to develop an application.	While Reverse Engineering or backward engineering takes less time to develop an application.
4.	The nature of forward engineering is Prescriptive.	The nature of reverse engineering or backward engineering is Adaptive.
5.	In forward engineering, production is started with given requirements.	In reverse engineering, production is started by taking existing product.
6.	The example of forward engineering are construction of electronic kit, construction of DC MOTOR etc.	The example of backward engineering are research on Instruments etc.

Board Exam Questions:

1. What is the capacity maturity model? Describe the five levels defined in CMMI. [2019 spring, 2015 fall, 2014 spring]]
2. Software engineering will change more rapidly. Explain few trends, methods and tools that are likely to have influence on software engineering. [2019 fall]
3. Explain the capability maturity model in brief. [2018 fall]
4. With diagram, explain software process improvement (SPI) framework. [2017 spring, 2017 fall]

5. What is SPI? Explain process improvement according to CMMI. [2016 spring]
6. Explain why software system which is used in a real-world environment must change or become progressively less useful. [2015 spring]
7. What is SPI and SPI framework? Explain briefly about CMMI. [2015 spring]
8. Differentiate between Reverse and Forward engineering with relevant example. [2014 spring]
9. Why do we need to improve software process? Discuss about the emerging trends in SE. [2014 spring]
10. Describe the UML based CASE tools. [2014 fall]
11. Define requirement traceability, and explain why it is relevant to the maintenance of software systems. [2014 fall]
12. What is the difference between software engineering and system engineering? Explain the different levels of business process engineering. [2013 fall]
13. Write short notes on:
    a) People CMM [2019 fall]
    b) CASE tools [2017 fall, 2013 fall]
    c) Emerging trends in SE [2015 fall]
    d) Business process engineering [2013 spring]
    e) Product engineering [2013 fall]

***