

Content

1. Software Process and Requirements

- 1.1 Software Crisis
- 1.2 Software Characteristics
- 1.3 Software Quality Attributes
- 1.4 Software Process Model
 - 1.4.1 Waterfall Model
 - 1.4.2 Incremental / Prototyping Model
 - 1.4.3 Spiral Model
 - 1.4.4 Rapid Application Development (RAD) Model
 - 1.4.5 Agile Model
- 1.5 Process Iteration
- 1.6 Process Activities
- 1.7 Computer-aided software engineering
- 1.8 Functional and Non-functional requirements
- 1.9 User Requirements
- 1.10 System Requirements
- 1.11 Interface Specification
- 1.12 The Software Requirements Documents
- 1.13 Feasibility Study
- 1.14 Requirement Elicitation and Analysis
- 1.15 Requirements Validation and Management

2. System Modeling

- 2.1 Context Models
- 2.2 Behavioral Models
 - 2.2.1 Data Processing Model

- 2.2.2 State Machine Model 32
- 2.3 Data Models 33
- 2.4 Object Models 34
 - 2.4.1 Inheritance Model 34
 - 2.4.2 Aggregation Model 35
 - 2.4.3 Interaction Model 36
- 2.5 Structured Methods 36
- 2.6 CASE workbenches 37
- 2.7 Data Flow Diagrams (DFD) 37

3. Architectural Design

- 3.1 Architectural Design Decisions 42
- 3.2 System Organization 43
- 3.3 Modular Decomposition Styles 44
 - 3.3.1 Object Model 44
 - 3.3.2 Data-flow Model 45
- 3.4 Control Styles 45
 - 3.4.1 Centralized control 46
 - 3.4.2 Event-based control 46
- 3.5 Reference Architectures 46
- 3.6 Multiprocessor Architecture 47
- 3.7 Client Server Architecture 47
 - 3.7.1 Layers in a client-server system 48
 - 3.7.2 Two-tier Client Server Architectures 49
 - 3.7.3 Three-tier client server architecture 50
 - 3.7.4 Client-server characteristics 50
- 3.8 Distributed Object Architecture 50
- 3.9 Inter-organizational distributed computing 51
- 3.10 CORBA-Common Object Request Broker Architecture 52

4. Real-time Software Design

- 4.1 Real-time systems 53
- 4.2 System Design 54
 - 4.2.1 Real Time System Design Process 55
 - 4.2.2 Timing Constraints 55
 - 4.2.3 Real Time system Modeling 55
- 4.3 Real-time Operating Systems 56
- 4.4 Monitoring and Control Systems 58

- 4.4.1 Monitoring Systems
- 4.4.2 Control Systems
- 4.5 Data Acquisition Systems

5. Software Reuse

- 5.1 Reuse of software
- 5.2 The Reuse Landscape
- 5.3 Design Patterns
- 5.4 Generator Based Reuse
- 5.5 Application Frameworks
- 5.6 Model-view-controller (MVC)
- 5.7 Application System Reuse
 - 5.7.1 COTS production integration
 - 5.7.2 Product Line Development

6. Component-based Software Engineering

- 6.1 Components
- 6.2 Component Models
- 6.3 The CBSE Process
- 6.4 Component Composition

7. Verification and Validation

- 7.1 Verification
- 7.2 Validation
- 7.3 Planning Verification and Validation
- 7.4 Software Inspections
- 7.5 Verification and Formal Methods
- 7.6 Critical System Verification and Validation

8. Software Testing and Cost Estimation

- 8.1 Software Testing
- 8.2 System Testing
- 8.3 Component Testing
- 8.4 Integration Testing
 - 8.4.1. Big Bang Integration Testing
 - 8.4.2 Incremental Approach

- 8.5 Regression Testing 91
- 8.6 Alpha and Beta Testing 94
- 8.7 Black and White Box Testing 95
- 8.8 Test Case Design 97
- 8.9 Test Automation 98
- 8.10 Metrics for Testing 99
- 8.11 Cyclomatic Complexity 100
- 8.12 Symbolic Execution 102
- 8.13 Software Productivity 103
- 8.14 Estimation Techniques 104
- 8.15 Algorithmic Cost Modeling 105
- 8.16 Estimation Accuracy 105
- 8.17 The COCOMO Model 106
- 8.18 Project Duration and Staffing 108

9. Quality Management

- 9.1 Quality Concept 109
- 9.2 Software Quality Assurance 111
- 9.3 Software Reviews 113
- 9.4 Formal Technical Reviews 116
- 9.5 Formal Approaches to SQA 118
- 9.6 Statistical Software Quality Assurance 118
- 9.7 Software Reliability 120
- 9.8 A Framework for Software Metrics 121
- 9.9 Metrics for Analysis and Design Model 121
- 9.10 ISO Standards 122
- 9.11 CMMI 126
- 9.12 SQA Plan 127
- 9.13 Software Certification

10. Configuration Management

- 10.1 Configuration Management Planning 128
- 10.2 Change Management 130
- 10.3 Version Management 131
- 10.4 Release Management 134
- 10.5 System Building 135
- 10.6 CASE Tools for Configuration Management 135

11. Object Oriented Software Engineering

- 11.1 Object Oriented Analysis
- 11.2 Object Oriented Design
- 11.3 Unified Modeling Diagrams
 - 11.3.1 Use case Diagram
 - 11.3.2 Activity Diagram
 - 11.3.3 Class Diagram
 - 11.3.4 Sequence Diagram
 - 11.3.5 Component Diagram
 - 11.3.6 Deployment Diagram
 - 11.3.7 State Machine Diagram

12. Introduction to Software Engineering Trends and Technology

- 12.1 Agile Development
- 12.2 Extreme Programming
- 12.3 Cloud Computing
- 12.4 Grid Computing
- 12.5 Enterprise Mobility

Software Process and Requirements

Software engineering is an engineering discipline that is concerned with all the aspects of software production from the early stages of system specification to maintaining the system after it has gone into use.

1.1 Software Crisis

Software crisis is the situation resulted due to the catastrophic failure of software development which leads to incomplete and degrading performance of software products.

- > Term was coined in the year 1968.
- > Software crisis was due to the rapid increases in computer power and the complexity of the problems that could not be tackled.

Causes of software crisis:

- 1) Projects running over-budget
- 2) Projects running over-time
- 3) Software was very inefficient
- 4) Software was of low quality

- 5) Software often did not meet the requirements
 - 6) Projects were unmanageable and code was difficult to maintain.
- OS 360 as an example of software crisis:**

- Officially known as IBM System/360 Operating System.
- A discontinued batch processing operating system developed by IBM
- Announced in 1964.
- Entered the market in 1967, chock full of errors.
- The system was huge, involving more than a million lines of codes, written by hundreds of programmers.
- Was the disastrous result of an untried methodology.
- Unacceptable due to the tedious performance and the complexity.

1.2 Software Characteristics

The characteristic of the software can be written as:

- Software is developed or engineered; it is not manufactured in classical paradigm.
- Time and effort for software development are hard to estimate.
- Software does not tear out.
- Software is malleable.
- Software construction is human intensive.
- It has discontinuous operational nature.
- Software components should be manufactured in such a way that is can be reused for various application.

1.3 Software Quality Attributes

The software quality attributes has been given the acronym **FURPS**—functionality, usability, reliability, performance, and supportability.

The **FURPS** quality attributes represent a target for all software design:

- ❖ **Functionality:** Refers to the degree of performance of the software against its intended purpose.
- ❖ **Usability:** Refers to the extent to which the software can be used with ease.
- ❖ **Reliability:** Refers to the ability of the software to provide desired functionality under the given conditions.

- ❖ **Performance:** Is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
- ❖ **Supportability:** Refers to the ease with which software developers can transfer software from one platform to another, without (or with minimum) changes.

1.4 Software Process Model

A software process is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or, modifying an existing system. It can be known as the simplified representation of a software process. Each model represents a process from a specific perspective.

We study the following software development process models:

1. Waterfall Model
2. Incremental / Prototyping model
3. Spiral Model
4. Rapid application development (RAD) model
5. Agile Model

1.4.1 Waterfall Model

The waterfall model is a software development model in which the results of one activity is flowed sequentially into the next and seen as flowing steadily downwards (like a water) through different phases.

- It is the first formally defined software development life cycle model.
- In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.

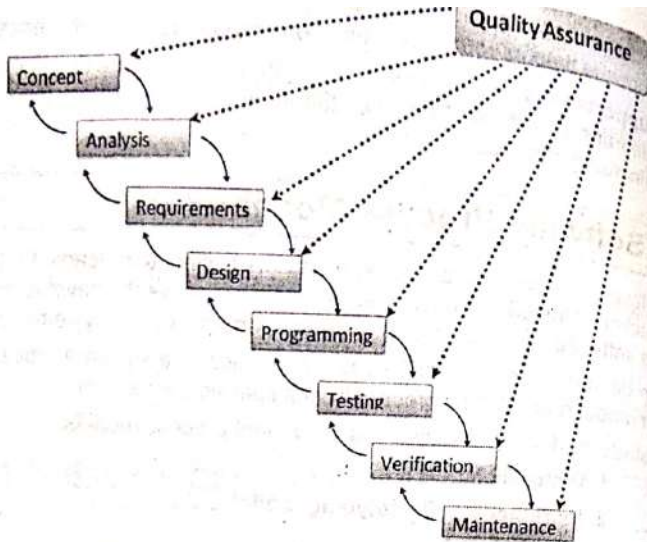


Fig: The waterfall model of Software development lifecycle

1. Quality assurance

- Oversees each steps of model towards producing useful, reliable software rather than connection of modules.
- All the phases are governed by quality assurance.

2. Concept and analysis

- Problem described in human language.
- Model the concept with mathematical formulas and algorithms.
- Analyze the software within the framework of the system description and concept.
- Analyze on the interactions and interfaces between the software, the hardware and data inputs.

3. Requirements

- All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.

4. Design

- The requirement specifications from first phase are studied in this phase and the system design is prepared.
- This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

5. Programming

- The methods of programming – assembly language, high level languages.
- CASE tools are on the horizon.
- Source file (Assembly language mnemonics) → assembler → object file → linker → Binary machine code → Burn → PROM
- A fully programmed software is then implemented.

6. Testing and verification

- All the units developed in the implementation phase are integrated into a system after testing of each unit.
- The entire system is tested for any faults and failures.
- Testing are done through alfa test, beta test, black box testing and white box testing.

7. Maintenance

- There are some issues which come up in the client environment. To fix those issues, patches are released.
- Also to enhance the product some better versions are released.
- Maintenance is done to deliver these changes in the customer environment.

Advantages of waterfall model:

- 1) This model is simple and easy to understand and use.
- 2) It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- 3) In this model phases are processed and completed one at a time. Phases do not overlap.
- 4) Waterfall model works well for smaller projects where requirements are very well understood.

Disadvantages of waterfall model:

- 1) Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- 2) No working software is produced until late during the life cycle.
- 3) High amounts of risk and uncertainty.
- 4) Not a good model for complex and object-oriented projects.
- 5) Poor model for long and ongoing projects.
- 6) Not suitable for the projects where requirements are at a moderate to high risk of changing.

When to use the waterfall model:

- ❖ This model is used only when the requirements are very known, clear and fixed.
- ❖ Product definition is stable.
- ❖ Technology is understood.
- ❖ There are no ambiguous requirements
- ❖ Ample resources with required expertise are available freely
- ❖ The project is short.

1.4.2 Incremental / Prototyping Model

Incremental model is the model of software development life cycle where the iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented.

- In this model the developer and client interact to establish the requirements of the software.
- Essence of prototyping is a quickly designed model that can undergo immediate evaluation.
- This is followed up by the quick design, in which the visible elements of the software, the input and the output are designed.
- The client then evaluates the prototype and provides its recommendations and suggestion to the analyst.
- The process continues in an iterative manner until all the user requirements are met.
- Accommodates problem of changing requirements

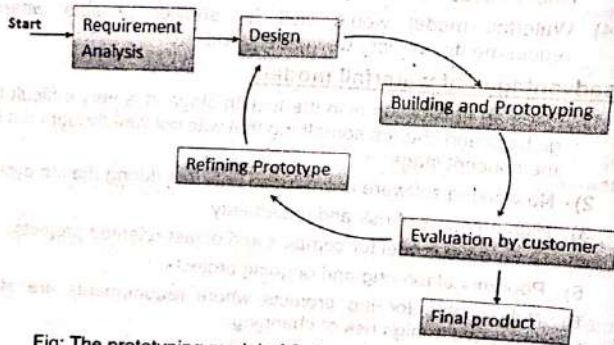


Fig: The prototyping model of Software development lifecycle

Advantages of Prototype model:

- 1) Clients are actively involved in the development.
- 2) Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- 3) Errors can be detected much earlier.
- 4) Quicker user feedback is available leading to better solutions.
- 5) Missing functionality can be identified easily.
- 6) Confusing or difficult functions can be identified.

Disadvantages of Prototype model:

- 1) Leads to implementing and then repairing way of building systems.
- 2) Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- 3) Incomplete application may cause application not to be used as the full system was designed.
- 4) Incomplete or inadequate problem analysis.

When to use Prototype model:

- ❖ Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- ❖ Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- ❖ Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

1.4.3 Spiral Model

It is the model which uses incremental approach to development that provides a combination of waterfall and prototyping model.

- Each cycle around the development spiral provides a successively more complete version of the software.
- Model allows flexibility to manage requirements control changes
- Risk driven rather than document driven.

➤ More emphasis placed on risk analysis.

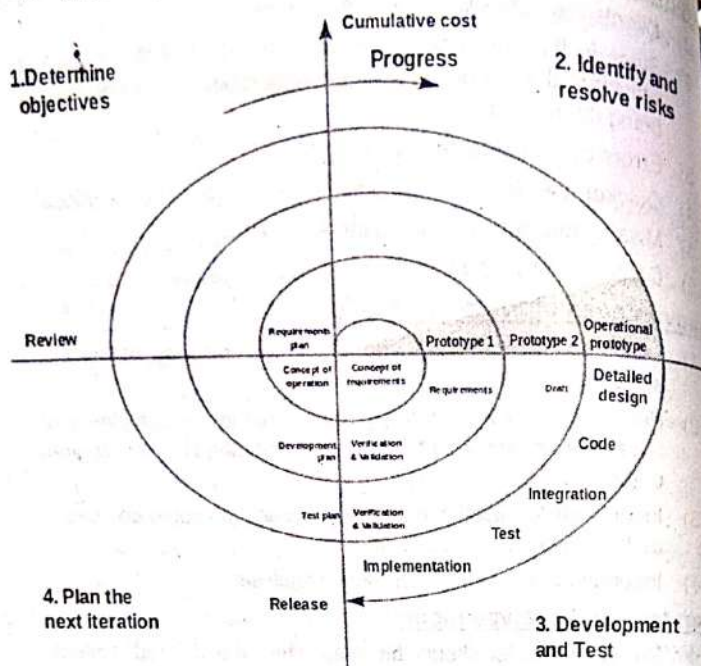


Fig: The spiral model of Software development lifecycle

Advantages of Spiral model:

- 1) High amount of risk analysis hence, avoidance of Risk is enhanced.
- 2) Good for large and mission-critical projects.
- 3) Strong approval and documentation control.
- 4) Additional Functionality can be added at a later date.
- 5) Software is produced early in the software life cycle.

Disadvantages of Spiral model:

- 1) Can be a costly model to use.
- 2) Risk analysis requires highly specific expertise.
- 3) Project's success is highly dependent on the risk analysis phase.
- 4) Doesn't work well for smaller projects.

When to use Spiral model:

- ❖ 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
- ❖ New product line

Differences between spiral and waterfall model:

Spiral model	Waterfall model
1. Spiral model is not suitable for small projects.	1. Waterfall model is suitable for small projects.
2. Better risk management.	2. High amount of risk and uncertainty.
3. Process is complex.	3. Easy to understand.
4. The process may go indefinitely.	4. Stages are clearly defined.
5. This model is suitable for long and ongoing projects.	5. This model is not suitable for long and ongoing projects.
6. Iterations are followed.	6. Sequence is followed.
7. Flexible with user requirements.	7. Requirements once fixed cannot be modified.
8. Refinements are easily possible.	8. Refinements are not so easy.

1.4.4 Rapid Application Development (RAD) Model

Rapid application development model is a software development methodology that uses minimal planning in favor of rapid prototyping.

- The functional modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery.
- Since there is no detailed preplanning, it makes it easier to incorporate the changes within the development process.
- It follows iterative and incremental model
- Have small teams comprising of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype.

- 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.

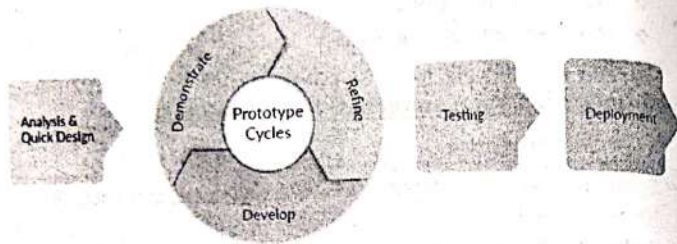


Fig: Brief view of RAD Model

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

Following are the various phases of the RAD Model:

Business Modeling

- Business model is designed in terms of flow of information and the distribution of information between various business channels.
- The information flow is identified between various business functions.
- Complete business analysis is performed to find the vital information for business
- Takes information gathered through many business related sources.

Data Modeling

- Information gathered in the Business Modeling phase is reviewed and analyzed to form sets of data objects vital for the business.
- The quality of every group of information is carefully examined and given an accurate description.
- Attributes of all data sets is identified and defined.
- Relation between these data objects are established and defined in detail in relevance to the business model.

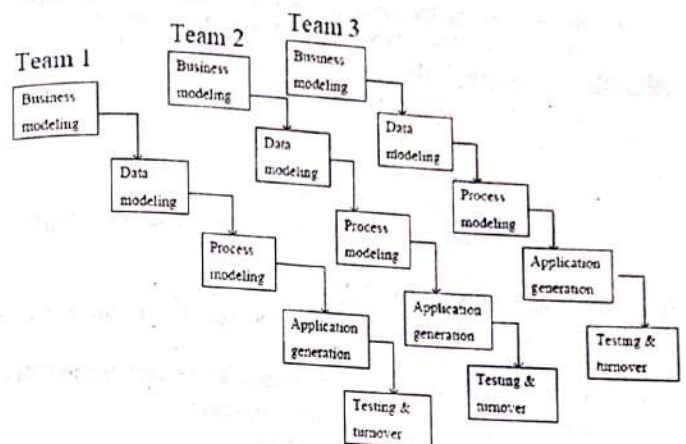


Fig: The RAD model of Software development lifecycle

Process Modeling

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

Application Generation

- The actual system is built and coding is done by using automation tools to convert process and data models into actual prototypes.
- Data models created are turned into actual prototypes that can be tested in the next step.

Testing and Turnover

- The overall testing time is reduced in the RAD model as the prototypes are independently tested during each iteration.
- However, the data flow and the interfaces between all the components need to be thoroughly tested with complete test coverage.

- Since most of the programming components have already been tested, it reduces the risk of any major issues.

Advantages of the RAD model:

- 1) Reduced development time.
- 2) Increases reusability of components.
- 3) Quick initial reviews occur.
- 4) Encourages customer feedback.
- 5) Integration from very beginning solves a lot of integration issues.

Disadvantages of RAD model:

- 1) Depends on strong team and individual performances for identifying business requirements.
- 2) Only system that can be modularized can be built using RAD.
- 3) Requires highly skilled developers/designers.
- 4) High dependency on modeling skills.
- 5) Inapplicable to cheaper projects as cost of modeling and automated code generation is very high.

When to use RAD model:

- ❖ RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
- ❖ It should be used if there's high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
- ❖ RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

1.4.5 Agile Model

Agile development model is a type of Incremental model which results in small incremental releases with each release building on previous functionality.

- Iterative approach is taken and working software build is delivered after each iterations.
- The tasks are divided to time boxes (small time frames) to deliver specific features for a release.
- Each release is thoroughly tested to ensure software quality is maintained.
- Used for time critical applications.

- Each build is incremental in terms of features; the final build holds all the features required by the customer.
- Extreme Programming (XP) is currently the most well known agile development life cycle model.

Here is a graphical illustration of the Agile Model:

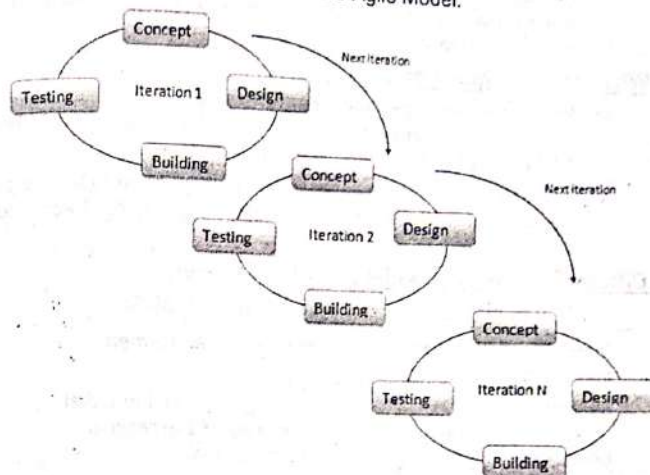


Fig: The Agile model of Software development lifecycle

Advantages of Agile model:

- 1) Customer satisfaction by rapid, continuous delivery of useful software.
- 2) People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- 3) Working software is delivered frequently (weeks rather than months).
- 4) Face-to-face conversation is the best form of communication.
- 5) Close cooperation between business people and developers.
- 6) Continuous attention to technical excellence and good design.
- 7) Regular adaptation to changing circumstances.
- 8) Even late changes in requirements are welcomed

Disadvantages of Agile model:

- 1) In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.

- 2) There is lack of emphasis on necessary designing and documentation.
- 3) The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- 4) Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers, unless combined with experienced resources.

When to use Agile model:

- ❖ When new changes are needed to be implemented. New changes can be implemented at very little cost because of the frequency of new increments that are produced.
- ❖ To implement a new feature the developers need to lose only the work of a few days, or even only hours, to roll back and implement it.

Differences between Agile and Spiral model:

AGILE MODEL	SPIRAL MODEL
1. More risk of sustainability and maintenance.	1. Better risk management.
2. Minimum rules, documentation easily employed.	2. Large number of intermediate stages requires excessive documentation.
3. Little or no planning required.	3. Planning is required.
4. Easy to manage.	4. Management is more complex.
5. Early delivery of partial working solutions.	5. End of project may not be known.
6. Suitable for small projects.	6. Not suitable for small or low risk projects.
7. Depends heavily on customer interaction.	7. Does not depend heavily on customer interaction.
8. Every iteration is a separate model.	8. Every iteration is not a separate model.

1.5 Process Iteration

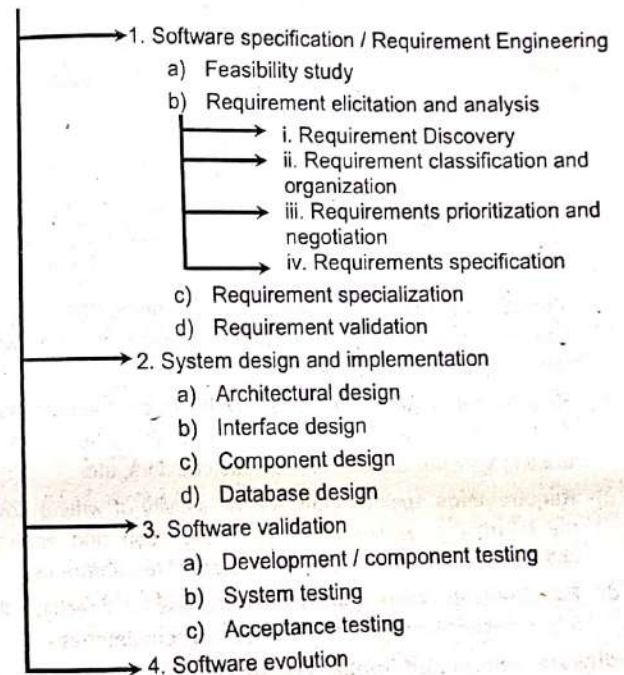
The Water Fall model was widely used in software development for many years but these days, the process iteration is being adopted more and more.

- There is now a number of well-established iterative development process model that can be classified according to the levels where iterations are applied.

- Iteration can improve validation and verification by allowing earlier quality feedback.
- Moreover, there is an important bond among teamwork and iteration.
- Altogether form a Software Process Iteration (SPI) points to view changing to an iterative development process model could very well raise our professional standards in software development.

1.6 Process Activities

The software process is the set of activities that eventually end up in the production of a software product. This may involve the development of software from a scratch. There are different software processes but all must include the major four activities. They are:



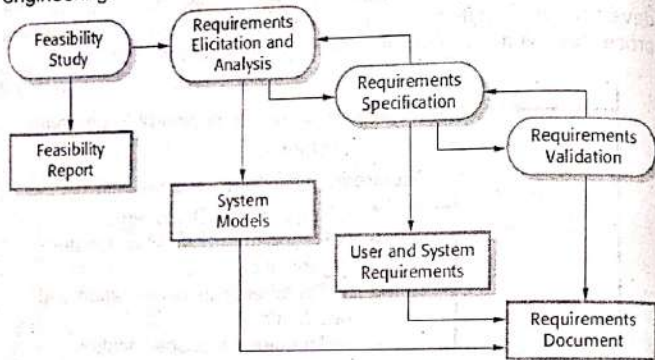
Now let's describe the software development processes in detail:

1. Software specification / Requirement Engineering:

Software specification or requirements engineering is the process of understanding and defining what services are required and identifying the constraints on these services.

- Ensures that the software will meet the user expectations and ending up with a high quality software.
- It's a critical stage of the software process as errors at this stage will reflect later on the next stages, which definitely will cause higher costs.

There are four main activities (or sub-activities) of requirements engineering:



- Feasibility study:** An estimate is made of whether the identified can be achieved using the current software and hardware technologies, under the current budget, etc.
- Requirements elicitation and analysis:** This is the process of deriving the system requirements through observation of existing systems, discussions with stakeholders, etc.
- Requirements specification:** It's the activity of writing down the information gathered during the elicitation and analysis activity into a document that defines a set of requirements.
- Requirements validation:** It's the process of checking the requirements for realism, consistency and completeness.

2. Software Design and Implementation:

- The implementation phase is the process of converting a system specification into an executable system.

- A software design is a description of the structure of the software to be implemented, data models, interfaces between system components, and maybe the algorithms used.
- The software designers develop the software design iteratively; they add formality and detail and correct the design as they develop their design.

Here's an abstract model of the design process showing the inputs, activities, and the documents to be produced as output.

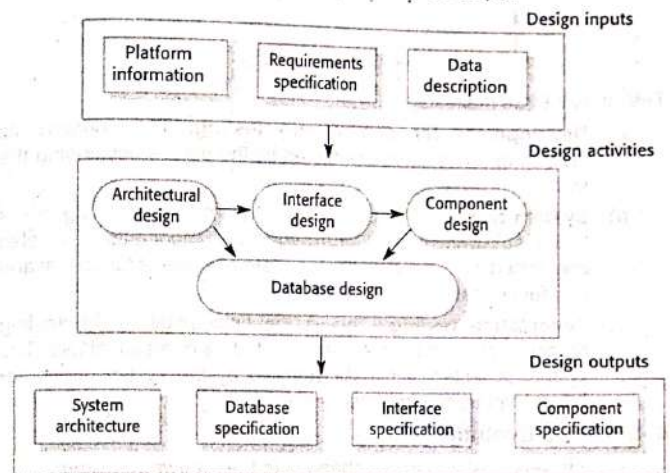


Fig: A general model of the design process

The four main activities that may be part of the design process are:

- Architectural design:** It defines the overall structure of the system, the main components and their relationships.
- Interface design:** It defines the interfaces between these components. The interface specification must be clear.
- Component design:** Take each component and design how it will operate, with the specific design left to the programmer, or a list of changes to be made to a reusable component.
- Database design:** The system data structures are designed and their representation in a database is defined. This depends on whether an existing database is to be reused or a new database to be created.

3. Software Verification and Validation:

Software validation or, more generally, verification and validation (V&V) is intended to show that a system both conforms to its specification and that it meets the expectations of the customer.

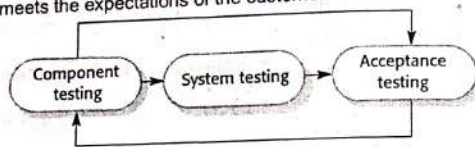


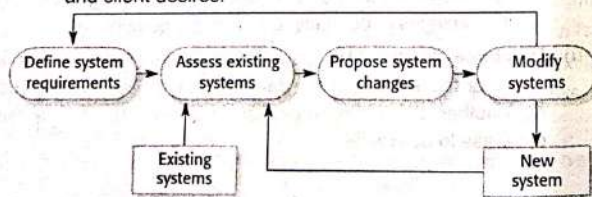
Fig: The stages of testing

Testing has three main stages:

- Development (or component) testing:** The components making up the system are tested by the people developing the system.
- System testing:** This process is concerned with finding errors that result from interactions between components. It is also concerned with showing that the system meets its functional and non-functional requirements.
- Acceptance testing:** This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than using simulated test data.

4. Software Evolution:

- It is very costly to make changes to the hardware system.
- However, changes are often created to software package at any time throughout or after the system development.
- There has continuously been a split between the process of software system development and also the process of software evolution (software maintenance).
- It's more realistic to think about software engineering as an evolutionary process wherever software is regularly modified over its period of time in response to ever-changing necessities and client desires.



1.7 Computer-aided Software Engineering

Computer-aided software engineering (CASE) is the application of computer-assisted tools and methods in software development to ensure a high-quality and defect-free software.

- CASE ensures a check-pointed and disciplined approach and helps designers, developers, testers, managers and others to see the project milestones during development.
- It can serve as a repository for project-related documents like business plans, requirements and design specifications.
- Delivery of the final product is more likely to meet real-world requirements as it ensures that customers remain part of the process.

Now we briefly go through various CASE tools

1. Diagram tools:

- These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form.
- For example, Flow Chart Maker tool for creating state-of-the-art flowcharts.

2. Process Modeling Tools:

- Process modeling is method to create software process model, which is used to develop the software.
- Process modeling tools help the managers to choose a process model or modify it as per the requirement of software product.
- For example, EPF Composer.

3. Project Management Tools:

- These tools are used for project planning, cost and effort estimation, project scheduling and resource planning.
- For example, Creative Pro Office, Trac Project, Basecamp.

4. Analysis Tools:

- These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions.
- For example, Accept 360, Accompa, CaseComplete for requirement analysis, Visible Analyst for total analysis.

5. Design Tools:

- These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques.
- For example, Animated Software Design.

6. Programming Tools:

- These tools consist of programming environments like IDE, in-built modules library and simulation tools.
- For example, Cscope to search code in C, Eclipse.

7. Web Development Tools:

- These tools assist in designing web pages with all allied elements like forms, text, script, graphic and so on.
- For example, Fontello, Adobe Edge Inspect, Foundation 3, Brackets.

8. Quality Assurance Tools:

- Quality assurance tools consist of configuration and change control tools and software testing tools.
- For example, SoapTest, AppSwatch, JMeter.

Importance of CASE tools:

- Reduce the cost as they automate many repetitive manual tasks.
- Reduce development time of the project as they support standardization and avoid repetition and reuse.
- Develop better quality complex projects as they provide greater consistency and coordination.
- Create good quality documentation.
- Create systems that are maintainable because of proper control of configuration item that support traceability requirements.

Disadvantages of using CASE Tools:

- Produce initial system that is more expensive to build and maintain.
- Require more extensive and accurate definitions of user needs and requirements.
- May be difficult to customize.
- Require training of maintenance staff.
- May be difficult to use with existing system.

1.8 Functional and Non-functional Requirements

Functional Requirements:

Functional requirements are those requirements which deal with what the system should do or provide for users.

- Describes the behavior of the system as it relates to the system's functionality.
- Includes the description of the required functions, outlines of associated reports or online queries, and details of data to be held in the system.
- Specified by users themselves.

Non Functional Requirements:

Non-functional requirements are those requirements which elaborate the performance characteristic of the system and define the constraints on how the system will do so.

- Defines the constraints, targets or control mechanisms for the new system.
- Describes how, how well or to what standard a function should be provided.
- Specified by technical peoples e.g. Architect, Technical leaders and software developers.
- They are sometimes defined in terms of metrics (something that can be measured about the system) to make them more tangible.
- Identify realistic, measurable target values for each service level.
- These include reliability, performance, service availability, responsiveness, throughput and security.

Example

If you are developing a Library system for your college, then the functional requirements can be listed as:

- Membership facility
- Issue of new books
- Return of books
- Visiting Books status
- Pre booking of books

And the non-functional requirements can be listed as:

- Throughput
- Service availability
- Security of the system and
- Reliability of the system

Difference between the functional and non-functional requirements:

Functional Requirements	Non-Functional Requirements
1. Functional requirements are those requirements that deal with what the system should do or provide for users.	1. Non-functional requirements are those requirements that define the constraints on how the system will do so.
2. It determines the product features.	2. It determines the product properties.
3. It is defined by the users.	3. It is determined by the software developers and architects.
4. It is easy to test the functional requirements.	4. It is comparatively difficult.

1.9 User Requirements

User requirements are the high level statements in a natural language with diagrams of what the system should do and the constraints under which it must operate.

- Should use as little technical terms as possible and must be understandable to the users.
- Often referred to as user needs, describe what the user does with the system, such as what activities that users must be able to perform.
- Generally documented in a User Requirements Document (URD) using narrative text.
- Signed off by the user and used as the primary input for creating system requirements.
- Are the functional requirements.
- Need to be described in the business domain and in any format the stakeholders want to allow such mutual understandings.

1.10 System Requirements

System requirements are the building blocks that the developers use to build the system which consist of the detailed description of the software system's functions and operational constraints.

- Defines what should be implemented so may be part of a contract between client and contractor.
- Are for the development team who need to understand how to practically meet user requirements using available technologies.
- Related to hardware and software.
- Usually describe the system that is required to host the solution.
- Deals with the characteristics that the system such as reliability, usability, maintainability and availability.
- Are the non-functional requirements.
- They are general characteristics of the system and may not be necessarily directly observable by the user.

1.11 Interface Specification

Large systems are decomposed into subsystems with well defined interfaces between these subsystems.

- Specification of subsystem interfaces allows independent development of the different subsystems.
- Interfaces may be defines as abstract data types or object classes.
- The algebraic approach to formal specification is particularly well suited to interface specification.
- Three types of interface may have to be defined:
 - Procedural interfaces
 - Data structures that are exchanged
 - Data representations

1.12 The Software Requirements Documents

A Software Requirements Specification (SRS) is a document or set of documentation that describes the features and behavior of a system or software application.

- It is a description of a software system to be developed.
- Traces out functional and non-functional requirements.
- Includes a set of use cases that describe user interactions that the software must provide.
- Establishes the basis for an agreement between customers and suppliers.

- Permits a rigorous assessment of requirements before design can begin and reduces later redesign.
- Provides a realistic basis for estimating product costs, risks, and schedules.
- Used appropriately, software requirements specifications can help prevent software project failure.
- Enlists enough and necessary requirements that are required for the project development.

Goals of SRS Document:

A well designed, well written SRS document accomplishes the following four major goals:

- ❖ Feedback to customers
- ❖ Problem decomposition
- ❖ Input to design specification
- ❖ Production validation check

Characteristics of SRS Document:

The main characteristics or features of SRS documents are written as:

1. **Accuracy:** This is the first and foremost requirement. The development team will get nowhere if the SRS which will be the basis of the process of software development, is not accurate.
2. **Clarity:** SRS should be clearly stating what the user wants in the software.
3. **Completeness:** The software requirement specification should not be missing any of the requirements stated in the business requirements documentation that the user specified.
4. **Consistency:** The document should be consistent from beginning till the end. It helps the readers understand the requirements well.
5. **Prioritization of Requirements:** Software Requirement Specification should not simply be a wish list. The requirements should follow the order of priority and preference.
6. **Verifiability:** At the end of the project, the user should be able to verify that all that all the agreed deliverables have in fact been produced and meet the project management requirements specified.
7. **Modifiability:** The SRS should be written in such a way that it can be modified when the development team and user feel the need.
8. **Traceability:** Each requirement stated in the SRS should be uniquely associated to a source such as a use case or interaction document.

1.13 Feasibility Study

The study which involves the analysis of the problem and collection of all relevant information relating to the product to select the best system that meets performance requirements is called as feasibility study.

- The main aim of the feasibility study activity is to determine whether it would be financially and technically feasible to develop the product.
- Analyzes whether the software will meet organizational requirements.
- Determines whether the software can be implemented using the current technology and within the specified budget and schedule.
- Determines whether the software can be integrated with other existing software.
- In this phase, the development team visits the user and studies their system.
- They investigate the need for development in the given system.
- The crucial purpose of this phase is to find the need and to define the problem that has to be solved.
- By the end of the feasibility study, the team furnishes a document that holds the different specific recommendations for the candidate system.

Topics under feasibility study:

The various topics included under feasibility study are:

1. Technical Feasibility:

- Concerned with specifying equipment and software that will successfully satisfy the user requirement.
- Technical needs of the system may vary considerably, but might include :
 - The facility to produce outputs in a given time.
 - Response time under certain conditions.
 - Ability to process a certain volume of transaction at a particular speed.
 - Facility to communicate data to distant locations.
- In examining technical feasibility, configuration of the system is given more importance than the actual make of hardware.
- The configuration should give the complete picture about the system's requirements.

2. Social Feasibility:

- Consideration of whether the proposed system would prove acceptable to the people who would be affected by its introduction.
- Describe the effect on users from the introduction of the new system considering whether there will be a need for retraining the workforce.
- Describe how you propose to ensure user co-operation before changes are introduced.

3. Economic Feasibility:

- Economic analysis is the most frequently used technique for evaluating the effectiveness of a proposed system.
- More commonly known as Cost / Benefit analysis, the procedure is to determine the benefits and savings that are expected from a proposed system and compare them with costs.
- If benefits outweigh costs, a decision is taken to design and implement the system.
- Otherwise, further justification or alternative in the proposed system will have to be made if it is to have a chance of being approved.
- An ongoing effort that improves in accuracy at each phase of the system life cycle.

4. Legal Feasibility:

- This is mainly related to human organizational and political aspects.
- Studies are carried out based on the legal perception.
- The political situation and the constitution of the country should be taken into consideration while performing the legal feasibility.

1.14 Requirement Elicitation and Analysis

It's a process of interacting with customers and end-users to find out about the domain requirements, what services the system should provide, and the other constraints.

It may also involve a different kinds of stockholders; end-users, managers, system engineers, test engineers, maintenance engineers, etc.

Here are the 4 main processes of requirements elicitation and analysis.

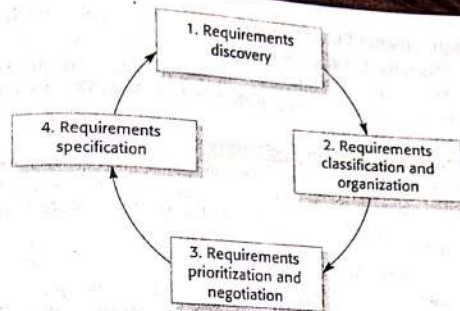


Fig: The process of requirements elicitation and analysis

1. Requirements Discovery

- It's the process of interacting and gathering the requirements from the stakeholders about the required system.
- It can be done using some techniques, like interviews, scenarios, prototypes, etc, which help the stockholders to understand what the system will be like.

2. Requirements Classification & Organization

- It's very important to organize the overall structure of the system.
- Putting related requirements together, and decomposing the system into sub components of related requirements.
- Then, we define the relationship between these components.
- What we do here will help us in the decision of identifying the most suitable architectural design patterns.

3. Requirements Prioritization & Negotiation

- It is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiations until you reach a situation where some of the stakeholders can compromise.
- We shouldn't reach a situation where a stakeholder is not satisfied because his requirements are not taken into consideration.
- Prioritizing your requirements will help us later to focus on the essentials and core features of the system.
- It can be achieved by giving every piece of function a priority level.

4. Requirements Specification

- This is the last stage of the cycle.
- All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

Requirement Elicitation Techniques:

Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others who have a stake in the software system development.

There are various ways to discover requirements:

a) Interviews:

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as structured, non-structured, oral and written interviews.

b) Surveys:

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

c) Questionnaires:

A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.

d) Task analysis:

Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

e) Domain Analysis:

Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

f) Brainstorming:

An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

g) Prototyping:

Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps giving better idea of requirements.

h) Observation:

A team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. The team itself draws some conclusions which aid to form requirements expected from the software.

1.15 Requirements Validation and Management

The process of checking the requirements for realism, consistency and completeness that determines whether the requirements are substantial management.

Requirements Validation Techniques

Test case generation:

- The requirements specified in the SRS document should be testable.
- The test in the validation process can reveal problems in the requirement.
- In some cases test becomes difficult to design, which implies that the requirement is difficult to implement and requires improvement.

Automated consistency analysis:

- If the requirements are expressed in the form of structured or formal notations, then CASE tools can be used to check the consistency of the system.
- A requirements database is created using a CASE tool that checks the entire requirements in the database using rules of method or notation.
- The report of all inconsistencies is identified and managed.

Prototyping:

- Prototyping is normally used for validating and eliciting new requirements of the system.
- This helps to interpret assumptions and provide an appropriate feedback about the requirements to the user.
- For example, if users have approved a prototype, which consists of graphical user interface, then the user interface can be considered validated.

2

System Modeling

System modelling is the graphical representation that helps the analyst to understand the functionality of the system and communicate with customers.

Different models present the system from different perspectives:

1. **External perspective:** It represents the system's context or environment.
2. **Behavioural perspective:** It represents the behaviour of the system.
3. **Structural perspective:** It represents the system or data architecture.

2.1 Context Models

Context Models are the models that show how the IT applications fit into the context of people and the organization they serve.

- Used to illustrate the operational context of a system.
- Show what lies outside the system boundaries.

- Social and organisational concerns may affect the decision on where to position system boundaries.

An example of the context model can be shown using the modelling diagram of the ATM system:

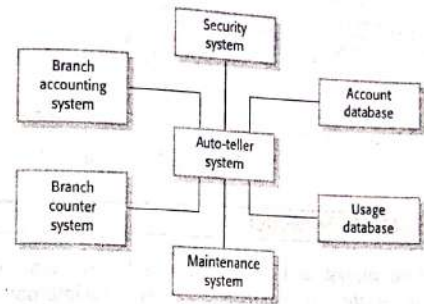
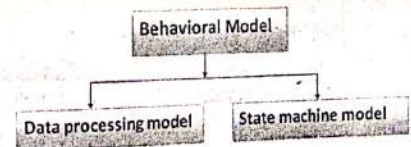


Fig: The context of an ATM system

2.2 Behavioral Models

Behavioral models are the models that are used to describe the overall behavior of a system.

- Two types of behavioral model are:
 - **Data processing models** that show how data is processed as it moves through the system.
 - **State machine models** that show the systems response to events.
- These models show different perspectives so both of them are required to describe the system's behavior.



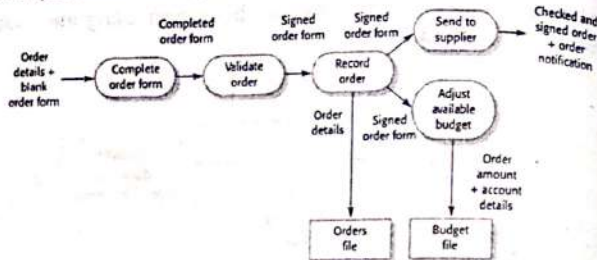
2.2.1 Data Processing Model

Data processing models is the modeling diagram that uses the data flow diagrams to model the system's data processing.

- It shows the processing steps as data flows through a system.
- Simple and intuitive notation that customers can understand.

- Show end-to-end processing of data.

An example of the order processing DPM can be shown as:



2.2.2 State Machine Model

State machine model is the modeling technique that models the behavior of the system in response to external and internal events.

- They show the system's responses to stimuli so are often used for modeling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- State charts are an integral part of the UML and are used to represent state machine models.
- State charts allow the decomposition of a model into sub-models.

An example of state machine model can be shown by drawing the state chart of Microwave oven model as shown below:

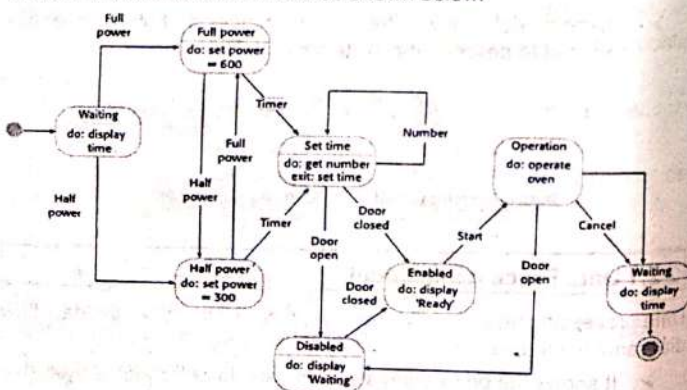


Fig: Microwave oven model

2.3 Data Models

The models that are used to describe the logical structure of data processed by the system are known as data models.

- An entity-relationship model sets out the entities in the system, the relationships between these entities and the entity attributes.
- Widely used in database design.
- Can readily be implemented using relational databases.
- No specific notation provided in the UML but objects and associations can be used.

An example of data model can be shown by drawing the Library semantic model as shown below:

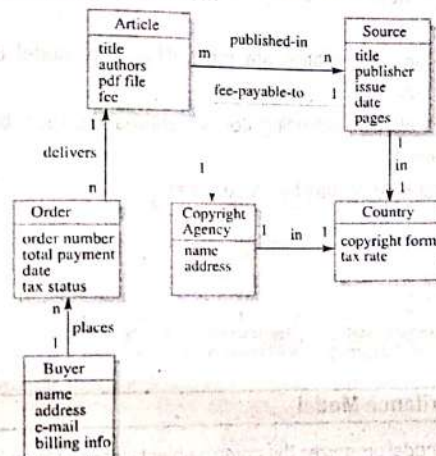


Fig: Library semantic model

Data dictionaries:

The set of information describing the contents, format, and structure of a database and the relationship between its elements, used to control access to and manipulation of the database are known as data dictionaries.

- Are the lists of all of the names used in the system models.
- Descriptions of the entities, relationships and attributes are also included.
- Many CASE workbenches support data dictionaries.

Advantages

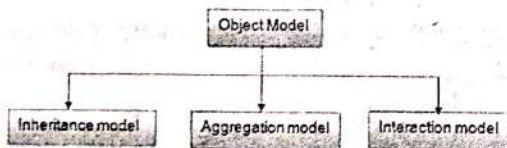
- ❖ Support name management and avoid duplication.
- ❖ Store of organizational knowledge linking analysis, design and implementation.

2.4 Object Models

Object models are the models that describe the system in terms of object classes and their associations.

- An object class is an abstraction over a set of objects with common attributes and the services provided by each object.
- Natural ways of reflecting the real-world entities manipulated by the system.
- More abstract entities are more difficult to model using this approach.
- Object classes reflecting domain entities are reusable across systems.

Various object models may be produced as:



2.4.1 Inheritance Model

Inheritance model organizes the domain object classes into a hierarchy.

- Classes at the top of the hierarchy reflect the common features of all classes.
- Object classes inherit their attributes and services from one or more super-classes.
- Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

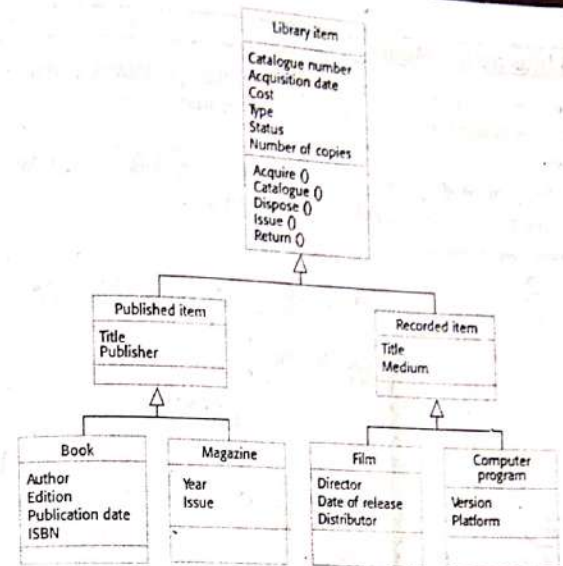


Fig: Library class hierarchy as Inheritance Model

2.4.2 Aggregation Model

An aggregation model shows how classes that are collections are composed of other classes.

- Aggregation models are similar to the part-of relationship in semantic data models.

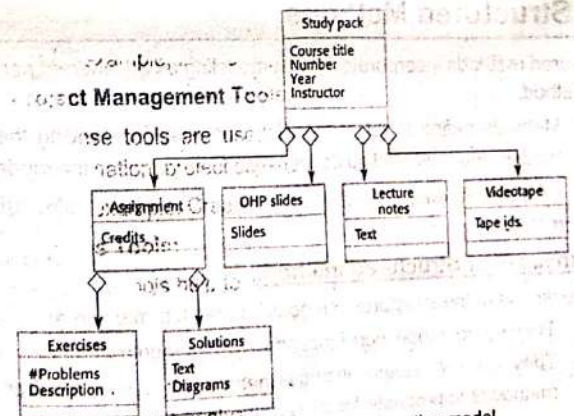


Fig: Demonstration of object aggregation model

2.4.3 Interaction Model

The interaction model shows the interactions between objects to produce some particular system behavior that is specified as a use case.

- Sequence diagrams or collaboration diagrams in the UML are used to model interaction between objects.

Medical Receptionist

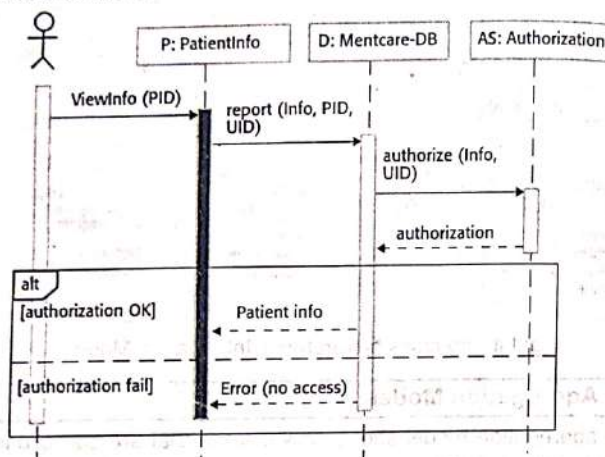


Fig: Sequence diagram as interaction model

2.5 Structured Methods

Structured methods incorporate system modeling as an inherent part of the method.

- Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models.
- CASE tools support system modeling as part of a structured method.

Weaknesses of Structured method:

The weakness of the structures methods can be summarized as:

- They do not model non-functional system requirements.
- They do not usually include information about whether a method is appropriate for a given problem.

- It may produce too much documentation.
- The system models are sometimes too detailed and difficult for users to understand.

2.6 CASE workbenches

A coherent set of tools that is designed to support related software process activities such as analysis, design or testing are known as CASE workbenches.

- Analysis and design workbenches support system modelling during both requirements engineering and system design.
- These workbenches may support a specific design method or may provide support for a creating several different types of system model.

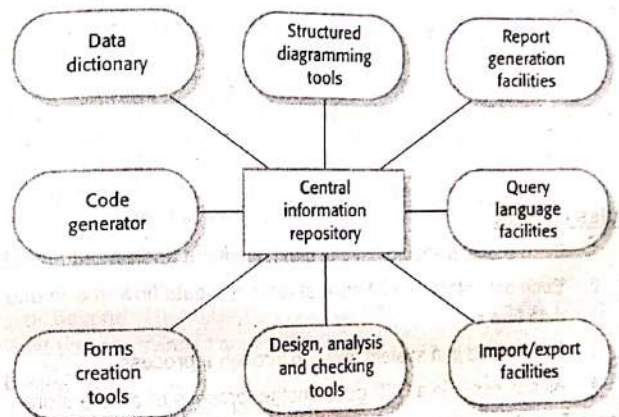


Fig: An analysis and design workbench

2.7 Data Flow Diagrams (DFD)

A data flow diagram (or DFD) is a graphical representation of the flow of data through an information system. They are also known as bubble charts.









- Shows how information is input to and output from the system, the sources and destinations of that information, and where that information is stored.

- > Maps out the flow of information for any process or system.
- > Uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination.

Symbols and Notations Used in DFDs:

Two common systems of symbols are named after their creators:

- ❖ Yourdon and Coad
- ❖ Gane and Sarson

Notation	Yourdon and Coad	Gane and Sarson
External Entity		
Process		
Data Store		
Data Flow		

Rules of DFD:

1. Each process should have at least one input and an output.
2. Each data store should have at least one data flow in and one data flow out.
3. Data stored in a system must go through a process.
4. All processes in a DFD go to another process or a data store.

Basic Terminologies:

Black hole: The situation where the processing step may have input flows but no output flows is called a black hole.

Miracle: The situation where the processing step may have output flows but no input flows is called a miracle.

Grey hole: The situation where the processing step may have outputs that are greater than the sum of its inputs - e.g., its inputs could not produce the output shown is referred to as a grey hole.

Major components of DFD:

DFDs are constructed using four major components:

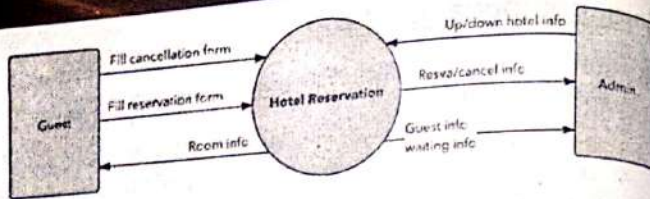
1. External entities
 2. Data stores
 3. Processes and
 4. Data flows
1. **External entities** represent the source of data as input to the system. They are also the destination of system data. External entities can be called data stores outside the system. These are represented by squares.
 2. **Data stores** represent stores of data within the system, for example, computer files or databases. An open-ended box represents a data, which implies store data at rest or a temporary repository of data.
 3. **Processes** represent activities in which data is manipulated by being stored or retrieved or transferred in some way. In other words, we can say that process transforms the input data into output data. Circles stand for a process that converts data into information.
 4. **Data flow** represents the movement of data from one component to the other. An arrow (→) identifies data flow, i.e. data in motion. It is a pipeline through which information flows. Data flows are generally shown as one-way only. Data flows between external entities are shown as dotted lines (-----→).

Different levels of DFD:

DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond. The necessary level of detail depends on the scope of what you are trying to accomplish.

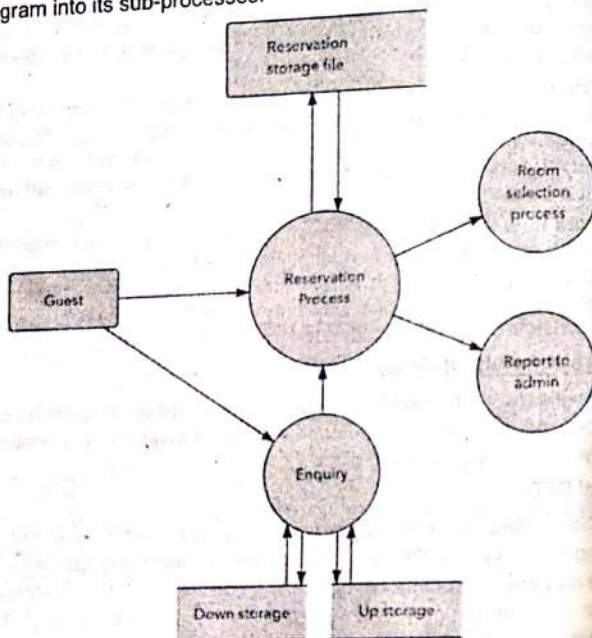
0 level DFD:

It is also called a Context Diagram. It shows a glance as if you are looking into a system through a helicopter. It's a basic overview of the whole system or process being analyzed or modeled. It's designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities. It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers.



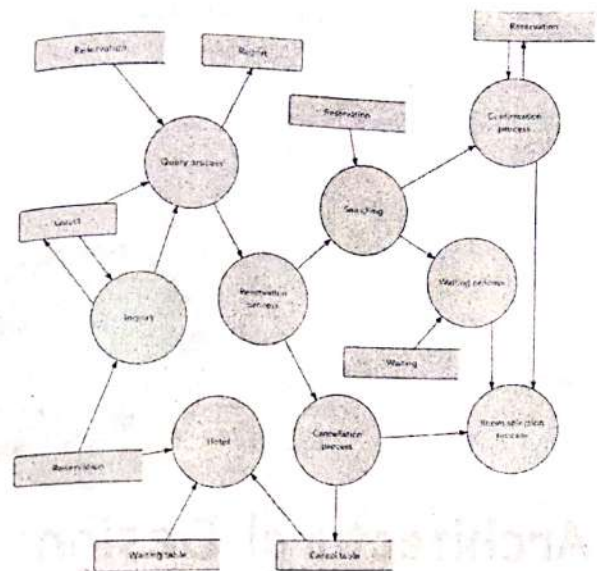
1 level DFD:

DFD Level 1 provides a more detailed breakout of pieces of the Context Level Diagram. Here the main functions carried out by the system are highlighted as we break down the high-level process of the Context Diagram into its sub-processes.



2 level DFD:

DFD Level 2 then goes one step deeper into parts of Level 1. It may require more text to reach the necessary level of detail about the system's functioning.



3 level DFD and so on:

Progression to Levels 3, 4 and beyond is possible, but going beyond Level 3 is uncommon. Doing so can create complexity that makes it difficult to communicate, compare or model effectively.

Difference between Logical and Physical DFD:

Logical DFD	Physical DFD
1. Logical DFD depicts how the business operates.	1. Physical DFD depicts how the system will be implemented.
2. The processes represent the business activities.	2. The processes represent the programs, program modules, and manual procedures.
3. The data stores represent the collection of data regardless of how the data are stored.	3. The data stores represent the physical files and databases, manual files.
4. It show business controls.	5. It show controls for validating input data, for obtaining a record, for ensuring successful completion of a process, and for system security.

3

Architectural Design

3.1 Architectural Design Decisions

Software architectural design is the design process that deals with the design and implementation of the high-level structure of software.

- Identifies the sub-systems that make up a system, and the framework for sub-system control and communication.
- Represents the link between specification and design processes.
- Provides well understood tools and techniques for constructing the system from its blueprint.
- Often carried out in parallel with some specification activities.
- Deals with abstraction, decomposition, composition, style, and aesthetics.
- Involves identifying major system components and their communications

Advantages of explicit architecture:

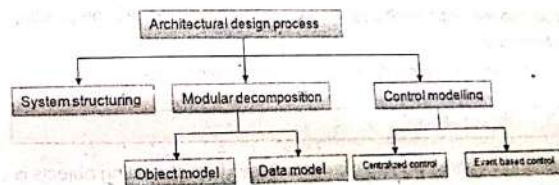
The advantages of the architectural design can be listed as:

- It is used for the stakeholders communication.
- It is used for the system analysis which means that analysis of whether the system can meet its non-functional requirements is possible.
- The architecture may be reusable across a range of systems.
- A framework for satisfying requirements.
- Managerial basis for cost estimation & process management.
- Effective basis for reuse.
- Basis for consistency, dependency, and tradeoff analysis.

Architecture attributes:

1. **Performance:** Localize operations to minimize sub-system communication
2. **Security:** Use a layered architecture with critical assets in inner layers
3. **Safety:** Isolate safety-critical components
4. **Availability:** Include redundant components in the architecture
5. **Maintainability:** Use fine-grain, self-contained components

The architectural design process can be organized into the following tree:



3.2 System Organization

System organization is the structuring method which is concerned with decomposing the system into interacting sub-systems.

- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.

- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

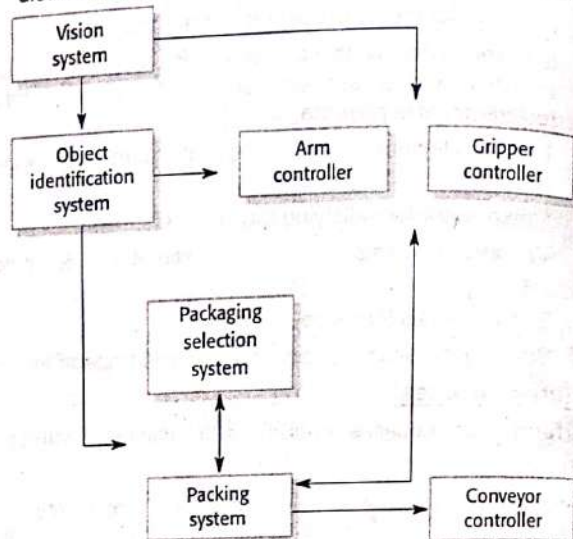


Fig: Packing robot control system

3.3 Modular Decomposition Styles

The architectural design process where the sub-systems are decomposed into modules is known as modular decomposition architecture.

Two modular decomposition models can be covered as:

3.3.1 Object Model

The model where the system is decomposed into interacting objects is known as object model.

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.

The example of object model can be shown as:

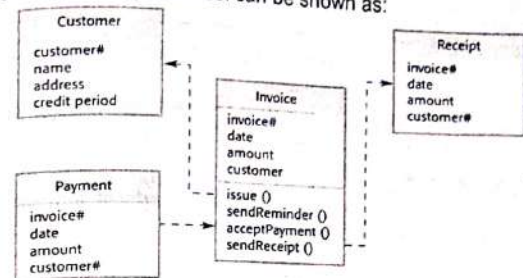


Fig: Invoice processing system as object model

3.3.2 Data-flow Model

A data-flow model is the model where the system is decomposed into functional modules which transform inputs to outputs.

- Also known as the pipeline model.
- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

The example of data-flow model can be shown as:

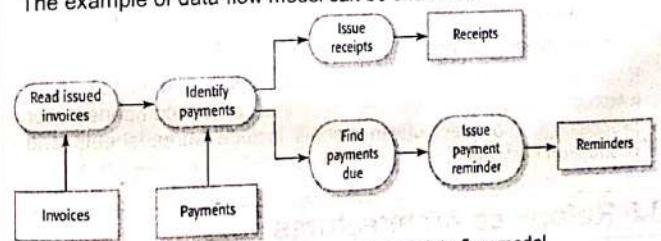


Fig: Invoice processing system as data-flow model

3.4 Control Styles

The modeling technique which is concerned with the control flow between sub-systems is called as control modeling. Control styles are classified as:

Control styles are classified as:

3.4.1 Centralized control

- One sub-system has overall responsibility for control and starts and stops other sub-systems.
- A control sub-system takes responsibility for managing the execution of other sub-systems.

a. Call-return model

Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. It is applicable to sequential systems.

b. Manager model

It is applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. It can be implemented in sequential systems as a case statement.

3.4.2 Event-Based Control

- Each sub-system can respond to externally generated events from other sub-systems or the system's environment.
- Driven by externally generated events where the timing of the event is out with the control of the sub-systems which process the event.

a. Broadcast models

An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.

b. Interrupt-driven models

It is used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing. The event driven models include spreadsheets and production systems.

3.5 Reference Architectures

The model used as a basis for system implementation or to compare different systems is known as reference architecture.

- Derived from a study of the application domain rather than from existing systems.
- It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems.

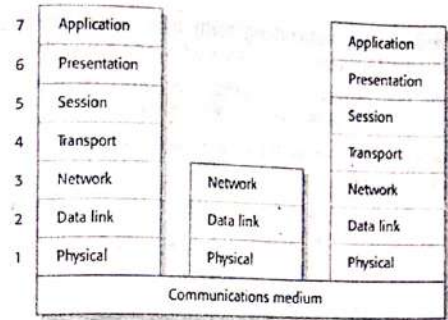


Fig: OSI reference model

3.6 Multiprocessor Architecture

The Simplest distributed system model in which the system is composed of multiple processes which may execute on different processors is called as multiprocessor architecture.

- Architectural model of many large real-time systems.
- Distribution of process to processor may be pre-ordered or may be under the control of a dispatcher.

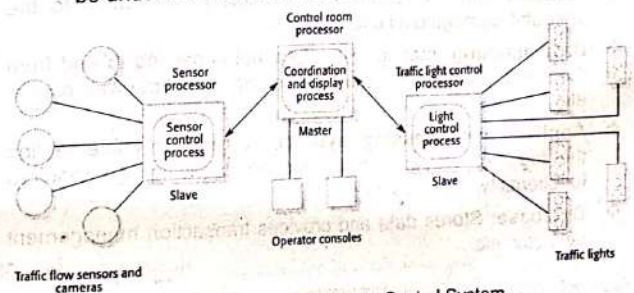


Fig: A Multiprocessor Traffic Control System

3.7 Client Server Architecture

Client-server architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the clients.

- Consists of one or more client computers connected to a central server over a network or Internet connection.
- System shares computing resources.

- Referred to as a networking computing model because all the requests and services are delivered over a network.
- Servers are powerful computers or processes dedicated to managing disk drives, printers, or network traffic.
- Clients are PCs or workstations on which users run applications.

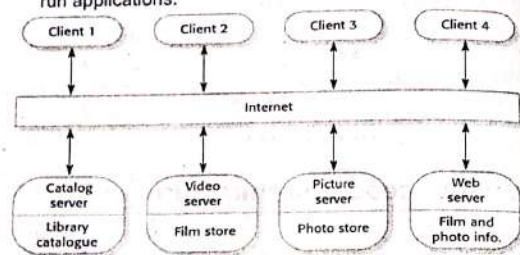


Fig: client-server architecture for a film library

3.7.1 Layers in a Client-Server System

- ❖ **Presentation:** concerned with presenting information to the user and managing all user interaction.
- ❖ **Data handling:** manages the data that is passed to and from the client. Implement checks on the data, generate web pages, etc.
- ❖ **Application processing layer:** concerned with implementing the logic of the application and so providing the required functionality to end users.
- ❖ **Database:** Stores data and provides transaction management services, etc.

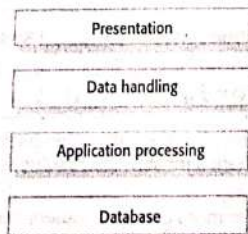
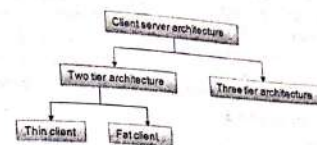


Fig: Layered architectural model for client-server applications

Moreover the client-server architecture can be classified as shown in the diagram:

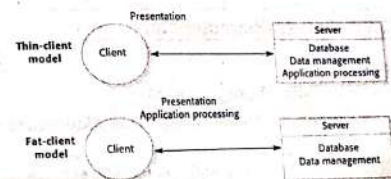


3.7.2 Two-tier Client Server Architectures

In two-tier client server architecture, the system is implemented as a single logical server plus an indefinite number of clients that use that server.

Thin-client model:

- In a thin-client model, all of the database, application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.
- A major disadvantage is that it places a heavy processing load on both the server and the network.



Fat-client model:

- In this model, the server is only responsible for data management and database functions.
- The software on the client implements the application logic and the interactions with the system user.
- More processing is delegated to the client as the application processing is locally executed.
- Most suitable for new client-server systems where the capabilities of the client system are known in advance.
- More complex than the thin client model, especially for management.

3.7.3 Three-tier client server architecture

- > In three-tier architecture, each of the application architecture layers may execute on a separate processor.
- > Allows for better performance than a thin-client approach and is simpler to manage than a fat-client approach.
- > A more scalable architecture - as demands increase, extra servers can be added.

Tier 1. Presentation

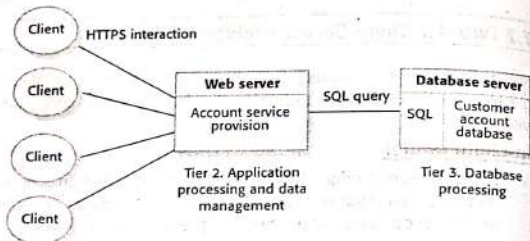


Fig: An internet banking system as three-tier C/S architecture

3.7.4 Client-Server Characteristics

Advantages

- ❖ Distribution of data is straightforward
- ❖ Makes effective use of networked systems. May require cheaper hardware
- ❖ Easy to add new servers or upgrade existing servers

Disadvantages

- ❖ No shared data model so sub-systems use different data organisation. data interchange may be inefficient
- ❖ Redundant management in each server
- ❖ No central register of names and services - it may be hard to find out what servers and services are available

3.8 Distributed Object Architecture

It is the system architecture where each distributable entity is an object that provides services to other objects and receives services from other objects.

- > There is no distinction in distributed object architectures between clients and servers.
- > Used as a logical model that allows you to structure and organise the system.
- > Object communication is through a middleware system called an object request broker.
- > However, distributed object architectures are more complex to design than C/S systems.
- > The ORB (object request broker) handles object communications. It knows of all objects in the system and their interfaces.

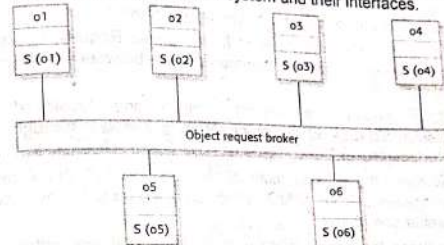


Fig: Distributed object architecture

Advantages of distributed object architecture:

- ❖ It allows the system designer to delay decisions on where and how services should be provided.
- ❖ It is a very open system architecture that allows new resources to be added to it as required.
- ❖ The system is flexible and scalable.
- ❖ It is possible to reconfigure the system dynamically with objects migrating across the network as required.

Disadvantages of distributed component architecture:

- ❖ Distributed component architectures are difficult for people to visualize and understand.
- ❖ They are more complex to design than client-server systems.
- ❖ Standardized middleware for distributed component systems has never been accepted by the community.

3.9 Inter-organizational Distributed Computing

- > Used for security and inter-operability reasons.

- Local standards, management and operational processes apply for such inter-organizational distribution computing.
- Newer models of distributed computing have been designed to support inter-organizational computing where different nodes are located in different organizations.

3.10 CORBA-Common Object Request Broker Architecture

CORBA is an architecture and specification for creating, distributing, and managing distributed program objects in a network.

- It is an international standard for an Object Request Broker - middleware to manage communications between distributed objects.
- Allows programs at different locations and developed by different vendors to communicate in a network through an "interface broker."
- Developed by a consortium of vendors through the Object Management Group (OMG), which currently includes over 500 member companies.
- Facilitates the communication of systems that are written in different languages and deployed on diverse platforms.
- Enables collaboration between systems on different operating systems, programming languages, and computing hardware.

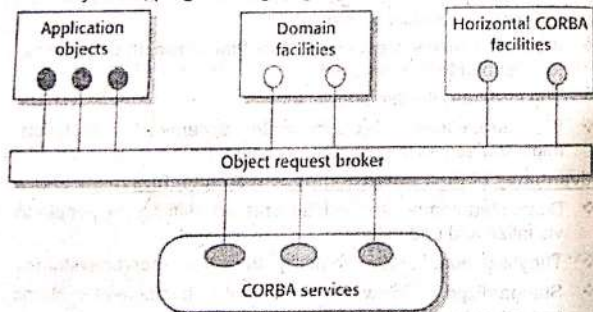


Fig: CORBA application structure

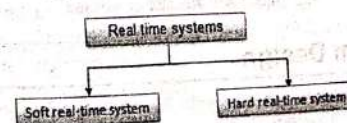
4

Real-time Software Design

4.1 Real-time Systems

A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced.

- Time is critical so real-time systems must respond within specified times.
- Systems which monitor and control their environment.
- Inevitably associated with hardware devices:
 - ❖ **Sensors:** Collect data from the system environment
 - ❖ **Actuators:** Change the system's environment in some way



The real time system can be classified as:

A. Soft real-time system:

A soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements.

B. Hard real-time system:

A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specification.

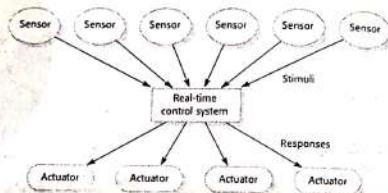


Fig: An embedded real-time system model

System elements:

- ❖ **Sensor control processes:** Collect information from sensors. May buffer information collected in response to a sensor stimulus.
- ❖ **Data processor:** Carries out processing of collected information and computes the system response.
- ❖ **Actuator control processes:** Generates control signals for the actuators.

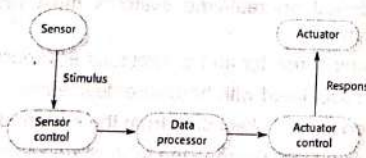


Fig: Sensor and actuator processes

4.2 System Design

System Design comprises of both the hardware and the software associated with system.

- Design decisions should be made on the basis on non-functional system requirements.
- Hardware delivers better performance but potentially longer development and less scope for change.

4.2.1 Real Time System Design Process

- Identify the stimuli to be processed and the required responses to these stimuli.
- For each stimulus and response, identify the timing constraints.
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response.
- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements.
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
- Integrate using a real-time operating system.

4.2.2 Timing Constraints

Timing constraints may require extensive simulation and experiment to ensure that these are met by the system.

- May mean that certain design strategies such as object-oriented design cannot be used because of the additional overhead involved.
- May mean that low-level programming language features have to be used for performance reasons.

4.2.3 Real Time system Modeling

The effect of a stimulus in a real-time system may trigger a transition from one state to another.

- Finite state machines can be used for modeling real-time systems.
- However, FSM models lack structure. Even simple systems can have a complex model.
- The UML includes notations for defining state machine models.

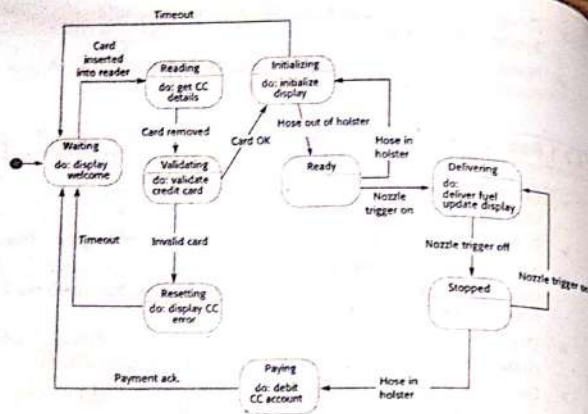


Fig: Petrol pump state model

4.3 Real-time Operating Systems

Real-time operating systems are specialised operating systems which manage the processes in the RTS.

- Responsible for process management and resource allocation.
- May be based on a standard kernel which is used unchanged or modified for a particular application.
- Do not normally include facilities such as file management.

Operating system components

1. **Real-time clock:** Provides information for process scheduling.
2. **Interrupt handler:** Manages aperiodic requests for service.
3. **Scheduler:** Chooses the next process to be run.
4. **Resource manager:** Allocates memory and processor resources.
5. **Dispatcher:** Starts process execution.

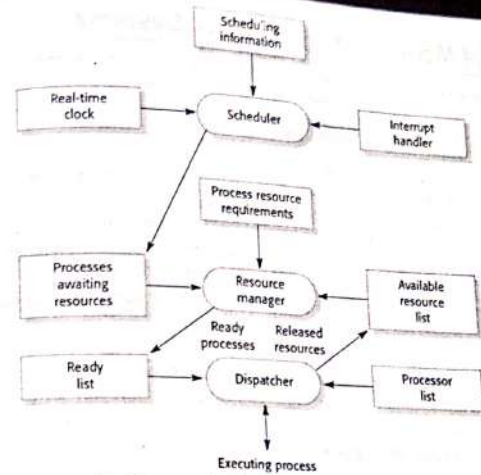


Fig: Components of a real-time operating system

Difference between real time OS and non-real time OS:

Real time OS	Non-real time OS
1. A real-time operating system is an operating system intended to serve real-time applications that process data as it comes in, typically without buffer delays.	1. A Non-real time OS or General purpose OS is the operating system made for high end, general purpose systems like a personal computer, a work station, a server system etc.
2. It is deterministic.	2. It is not deterministic.
3. It is time sensitive.	3. It is time insensitive.
4. It can't use virtual memory.	4. It can use virtual memory concept.
5. It is dedicated to single work.	5. It is used in multi-user environment.
6. It has flat memory model.	6. It has protected memory model.
7. It has low interrupt latency.	7. It has high interrupt latency

4.4 Monitoring and Control Systems

It is an important class of real-time systems which continuously check sensors and take actions depending on sensor values

4.4.1 Monitoring Systems

They examine sensors and report their results. Taking Burglar alarm system as an example of monitoring system,

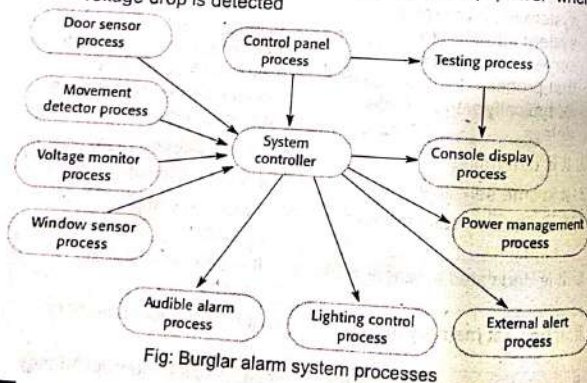
- The system is required to monitor sensors on doors and windows to detect the presence of intruders in a building.
- When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically.
- The system should include provision for operation without a mains power supply.

Sensors

- ❖ Movement detectors, window sensors, door sensors
- ❖ 50 window sensors, 30 door sensors and 200 movement detectors
- ❖ Voltage drop sensor

Actions

- ❖ When an intruder is detected, police are called automatically
- ❖ Lights are switched on in rooms with active sensors
- ❖ An audible alarm is switched on
- ❖ The system switches automatically to backup power when a voltage drop is detected



4.4.2 Control Systems

It takes the sensor values and controls hardware actuators.

- A burglar alarm system is primarily a monitoring system. It collects data from sensors but no real-time actuator control.
- Control systems are similar but, in response to sensor values, the system sends control signals to actuators.

An example of a monitoring and control system is a system that monitors temperature and switches heaters on and off.

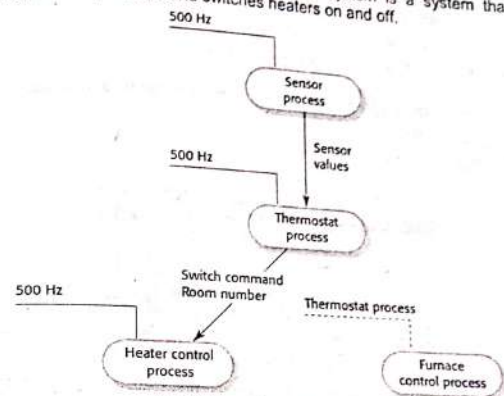


Fig: A temperature control system

4.5 Data Acquisition Systems

Data acquisition system (DAQ) is an information system that consists of DAQ software and hardware along with sensors and actuators for the collection, storage and distribution of information.

- Used in industrial and commercial electronics, and environmental and scientific equipment to capture electrical signals or environmental conditions on a computer device.
- The hardware typically consists of components in the form of external expansion cards which can be connected to the computer through a communication interface such as a PCI or USB.

- The hardware is connected with an input device such as a 3-D scanner or analog-to-digital converter.
 - The signal from the input device is sent to the hardware device, which processes and sends it to DAQ software, where it is recorded for further review and analysis.
 - A data acquisition system is also known as a data logger.
- A typical system consists of:

- Data acquisition (DAQ) hardware
- Sensors and actuators
- Signal conditioning hardware
- A computer running DAQ software

The block diagram of the data acquisition system can be shown as:

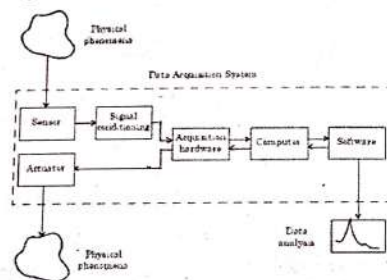


Fig: Block diagram of the data acquisition system



Software Reuse

5.1 Reuse of software

Reuse-based software engineering is one of the software engineering strategies where the development process is carried out reusing the existing software.

Reuse-based software engineering

Reuse may be in different sizes from program library to entire program. It is classified as per the degree of reuse:

- ❖ **System reuse:** Complete systems, which may include several application programs may be reused.
- ❖ **Application reuse:** An application may be reused either by incorporating it without change into other or by developing application families.
- ❖ **Component reuse:** Components of an application from sub-systems to single objects may be reused.
- ❖ **Object and function reuse:** Small-scale software components that implement a single well-defined object or function may be reused.

Advantages of software reuse

1. Increase software productivity.
2. Shorten software development time.
3. Improve software system interoperability.
4. Develop software with fewer people.
5. Produce more standardized software.
6. Produce better quality software and provide a powerful competitive advantage.

Disadvantages of software reuse

1. Needless complexity.
2. Inflexible design will cost too much to modify.
3. Domain irrelevance.
4. Inadequate documentation, training and awareness.
5. Increased development, testing, and maintenance costs.
6. Lack of tool support.

Reuse planning factors

Following are the factors that influence the reuse of the software components:

- ❖ The development schedule for the software.
- ❖ The expected software lifetime.
- ❖ The background, skills and experience of the development team.
- ❖ The criticality of the software and its non-functional requirements.
- ❖ The application domain.
- ❖ The execution platform for the software.

5.2 The Reuse Landscape

- Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used.
- Reuse is possible at a range of levels from simple functions to complete application systems.
- Many techniques have been developed to support software reuse.
- The reuse landscape covers the range of possible reuse techniques.

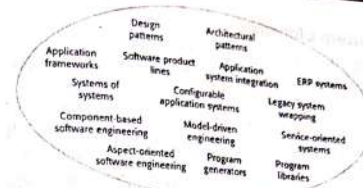


Fig: The reuse landscape

Concept reuse

When you reuse program or design components, you have to follow the design decisions made by the original developer of the component.

- This may limit the opportunities for reuse.
- However, a more abstract form of reuse is concept reuse when a particular approach is described in an implementation independent way and an implementation is then developed.

The two main approaches to concept reuse are:

- Design patterns
- Generator based reuse

5.3 Design Patterns

Design pattern is a general repeatable solution to a commonly occurring problem in software design.

- A design pattern is not a finished design that can be transformed directly into code. Rather, it is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Reusing design patterns helps to improve code readability for coders and architects familiar with the patterns.
- This should be sufficiently abstract so that it can be reused in different system settings.
- These patterns often rely on the characteristics of the object characterized by the inheritance and polymorphism.
- A template for design solution can be reused in different ways but not a concrete design.

Design Pattern Elements

- ❖ **Name:** A meaningful pattern identifier.
- ❖ **Problem description:** The complete description of the problem to be addressed.
- ❖ **Solution description:** A template for a design solution that can be instantiated in different operational context. It is often illustrated graphically.
- ❖ **Consequences:** The results and trade-offs of applying the pattern. It includes the analysis and experience.

Example: The Observer pattern

- ❖ **Name:** Observer
- ❖ **Description:** Separates the display of object state from the object itself allowing alternative displays.
- ❖ **Problem description:** Used when multiple display of states are needed.
- ❖ **Solution description:** It is based on the UML description.
- ❖ **Consequences:** Object optimizations to enhance the performance of a particular display are impractical.

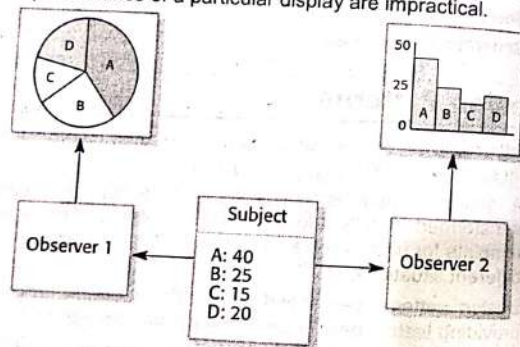


Fig: Multiple displays using observer pattern

5.4 Generator Based Reuse

The program generators involve the reuse of standard programs and algorithms.

These are embedded in the generator and parameterized by the user command and a program is then automatically generated.

- Generator-Based reuse is possible when domain abstraction and their mapping to executable code can be identified.
- A domain specific language is used to compose and control these abstractions.

Types of program generator:

1. Application generator for business data processing
2. Parse and lexical analyzer generator for language processing
3. Code generators in CASE tools

Generator based reuse is very cost effective but its applicability is limited to a relatively small number of application domains. It is easier for end-users to develop programs using generators compared to other component based approaches to reuse.

5.5 Application Frameworks

An application framework is a software library that provides a fundamental structure to support the development of applications for a specific environment.

- It consists of a software framework used by software developers to implement the standard structure of an application.
- Became popular with the rise of graphical user interfaces (GUIs) since these tended to promote a standard structure for applications.
- Programmers find it much simpler to create automatic GUI creation tools when using a standard framework since this defines the underlying code structure of the application in advance.
- Developers usually use object-oriented programming techniques to implement frameworks such that the unique parts of an application can simply inherit from pre-existing classes in the framework.

Framework classes

Frameworks are implemented as a collection of concrete and abstract object classes in an object-oriented programming language, therefore, are language-specific.

- There are frameworks available in all of the commonly used object-oriented programming languages like Java, C#, C++, as well as dynamic languages such as Ruby and Python.

- In fact, a framework can incorporate several other frameworks, where each of these is designed to support the development of a part of the application.
- We can use a framework to create a complete application or to implement part of an application, such as the graphical user interface.

The three classes of frameworks:

1. System infrastructure frameworks

These frameworks support the development of system infrastructures such as communications, user interfaces, and compilers.

2. Middleware integration frameworks

These consist of a set of standards and associated object classes that support component communication and information exchange. Examples of this type of framework include Microsoft's .NET and Enterprise Java Beans (EJB). These frameworks provide support for standardized component models.

3. Enterprise application frameworks

These are concerned with specific application domains such as telecommunications or financial systems. These embed application domain knowledge and support the development of end-user applications.

5.6 Model-view-controller (MVC)

The Model-View-Controller (MVC) is an architectural pattern that separates an application into three main logical components: the model, the view, and the controller.

- Each of these components is built to handle specific development aspects of an application.
- MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.
- It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.
- Traditionally used for desktop graphical user interfaces (GUIs), this architecture has become extremely popular for designing web applications.

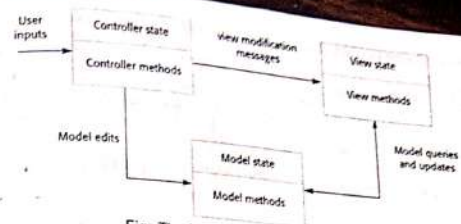


Fig: The Model-View-Controller pattern

Model

- The Model component corresponds to all the data-related logic that the user works with.
- This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data.
- For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

View

- The View component is used for all the UI logic of the application.
- For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

Controller

- Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.
- For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model.
- The same controller will be used to view the Customer data.

5.7 Application System Reuse

Application System Reuse involves the reuse of entire application either by configuring a system for an environment or by integrating two or more system to create a new application.

There may be two approaches for application system reuse. They are

5.7.1 COTS production integration

COTS (Commercial On The Shelf) are usually complete application system that often is an API which benefits in faster application development at lower cost.

E-Procurement System is one of the examples of COTS product reuse.

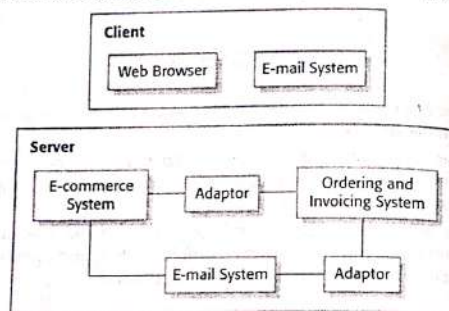


Fig: E-Procurement System

5.7.2 Product Line Development

A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements.

- The core system is designed to be configured and adapted to suit the needs of different system customers.
- This may involve the configuration of some components, implementing additional components, and modifying some of the components to reflect new requirements.
- Software product lines usually emerge from existing applications. That is, an organization develops an application then, when a similar system is required, informally reuses code from this in the new application.
- The same process is used as other similar applications are developed.

- However, change tends to corrupt application structure so, as more new instances are developed, it becomes increasingly difficult to create a new version. Consequently, a decision to design a generic product line may then be made.
- This involves identifying common functionality in product instances and including this in a base application, which is then used for future development.
- This base application is deliberately structured to simplify reuse and reconfiguration.

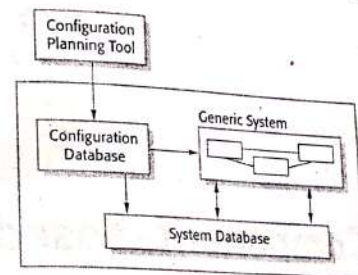


Fig: ERP system

Component-based Software Engineering

6.1 Components

A component is an independent executable entity that can be made up of one or more executable objects.

Component Characteristics

- ❖ **Standardized:** Component standardization means that a component that is used in a CBSE process has to conform to some standardized component model.
- ❖ **Independent:** A component should be independent – it should be possible to compose and deploy it without having to use other specific components.
- ❖ **Composable:** For a component to be composable, all external interactions must take place through publicly defined interfaces.
- ❖ **Deployable:** To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity

on some component platform that implements the component model.

- ❖ **Documented:** Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs.

Component interfaces

Component has two related interfaces. They are:

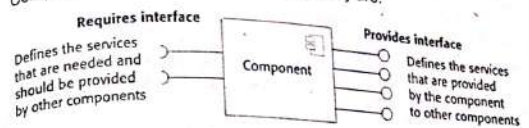


Fig: Component interfaces

- ❖ **Provides interface:** Defines the services that are provided by the component to other components.
- ❖ **Requires interface:** Defines the services that specifies what services must be made available for the component to execute as specified.

Example

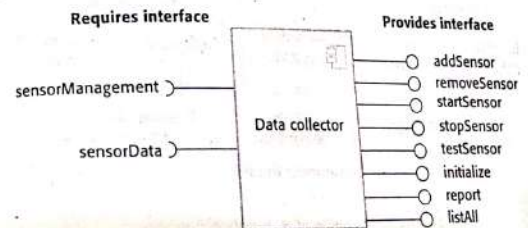


Fig: A model of a data collector component

6.2 Component Models

A component model is a definition of standards for component implementation, documentation and deployment.

- It specifies how interfaces should be defined and the elements that should be included in an interface definition.

Examples of component models

- ❖ EJB model (Enterprise Java Beans)

- ❖ COM+ model (.NET model)
- ❖ Corba Component Model

Elements of a component model

1. Interfaces

Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, which should be included in the interface definition.

2. Usage

In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique.

3. Deployment

The component model includes a specification of how components should be packaged for deployment as independent, executable entities.

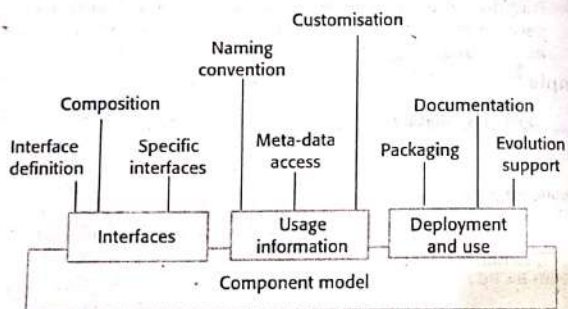


Fig: Basic elements of a component model

6.3 The CBSE Process

Component-based software engineering (CBSE) is an approach to software development emerged from the failure of object-oriented development that relies on the effective reuse of software.

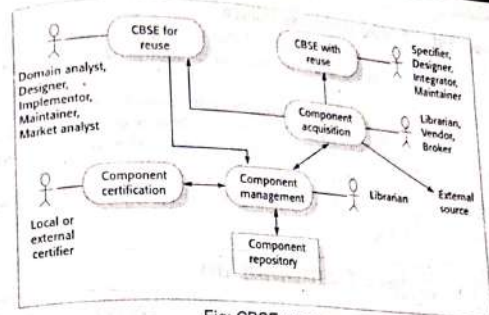


Fig: CBSE processes

Types of CBSE processes

1. CBSE for reuse

This process is concerned with developing components or services that will be reused in other applications.

2. CBSE with reuse

This process is the process of developing new applications using existing components and services. This involves:

- Developing outline requirements.
- Searching for components then modifying requirements according to available functionality.
- Searching again to find if there are better components that meet the revised requirements.
- Composing components to create the system.

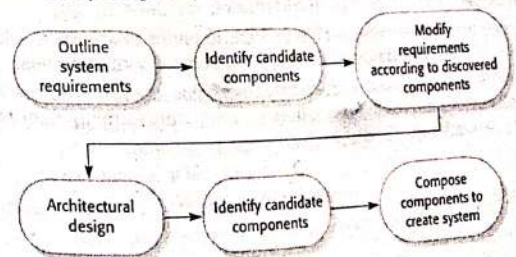


Fig: CBSE with reuse

Case study: Ariane launcher failure

- ✓ In 1996, the 1st test flight of the Ariane 5 rocket ended in disaster when the launcher went out of control 37 seconds after takeoff.
- ✓ The problem was due to a reused component from a previous version of the launcher (the Inertial Navigation System) that failed because assumptions made when that component was developed did not hold for Ariane 5.
- ✓ The functionality that failed in this component was not required in Ariane 5.

6.4 Component Composition

The process of assembling components to create a system is known as Component composition.

- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write 'glue code' to integrate components.

Types of composition

- ❖ **Sequential composition:** where the composed components are executed in sequence. This involves composing the provides interfaces of each component.
- ❖ **Hierarchical composition:** where one component calls on the services of another. The provides interface of one component is composed with the requires interface of another.
- ❖ **Additive composition:** where the interfaces of two components are put together to create a new component. Provides and requires interfaces of integrated component is a combination of interfaces of constituent components.

Glue code

The code that allows components to work together is known as glue code. Glue code may be used to resolve interface incompatibilities.

If A and B are composed sequentially, then glue code has to call A, collect its results then call B using these results, transforming them into the format required by B.



Verification and Validation

7.1 Verification

Verification is an act of testing checking and auditing that makes sure that the product is designed to deliver all functionality to the customer.

- Verification is done at the starting of the development process.
- It includes reviews and meetings, walk-throughs, inspection, etc. to evaluate documents, plans, code, requirements and specifications.
- It answers the questions like: Am I building the product right?
- It is a Low level activity.
- Demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle.

Advantages of Software Verification:

1. Verification helps in lowering down the count of the defect in the later stages of development.

2. Verifying the product at the starting phase of the development will help in understanding the product in a better way.
3. It reduces the chances of failures in the software application or product.
4. It helps in building the product as per the customer specifications and needs.

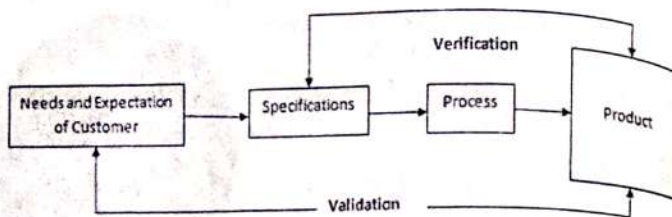


Fig: Software verification and validation

7.2 Validation

Validation is determining if the system complies with the requirements and performs functions for which it is intended and meets the organization's goals and user needs.

- Validation is done at the end of the development process and takes place after verifications are completed.
- It answers the question like: Am I building the right product?
- It is a High level activity.
- Performed after a work product is produced against established criteria ensuring that the product integrates correctly into the environment.
- Determination of correctness of the final software product by a development project with respect to the user needs and requirements.
- Validation is basically done by the testers during the testing.

Advantages of Validation:

1. During verification if some defects are missed then during validation process it can be caught as failures.
2. If during verification some specification is misunderstood and development had happened then during validation process while executing that functionality the difference between the actual result and expected result can be understood.

3. Validation is done during testing like feature testing, integration testing, system testing, load testing, compatibility testing, stress testing, etc.
4. Validation helps in building the right product as per the customer's requirement and helps in satisfying their needs.

Difference between Verification and Validation:

Verification	Validation
1. Verification is a static practice of verifying documents, design, code and program.	1. Validation is a dynamic mechanism of validating and testing the actual product.
2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	3. It is computer based execution of program.
4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. Validation uses methods like black box testing, gray box testing, and white box testing.
5. Verification is to check whether the software conforms to specifications.	5. Validation is to check whether software meets the customer expectations and requirements.
6. It can catch errors that validation cannot catch. It is low level exercise.	6. It can catch errors that verification cannot catch. It is High Level Exercise.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	8. Validation is carried out with the involvement of testing team.
9. It generally comes first-done before validation.	9. It generally follows after verification.

7.3 Planning Verification and Validation

The development of a V & V plan is essential to the success of a project. The plan must be developed early in the project. Careful planning is required to get the most out of testing and inspection process. Effective V & V plan requires many considerations that are:

1. Identification of V & V Goals:

V & V goals must be identified from the requirements and specifications. These goals must address those attributes of the product that correspond to its user expectations.

2. Selection of V & V Techniques:

Specific techniques must be selected for each of the project's evolving products.

3. Organizational Responsibilities:

The organizational structure of a project is a key planning consideration for project managers. An important aspect of this structure is a delegation of V & V activities to various organizations.

4. Integrating V & V Approaches:

Once a set of V & V objectives has been identified, an overall integrated V & V approach must be determined. This approach involves the integration of techniques applicable to the various life cycle phases as a delegation of these tasks among the project's organizations. Traditional integrated V & V approaches have followed the "Waterfall model".

5. Problem Tracking:

Software V & V plan to develop a mechanism for documenting problems

- When the problem occurred
- Where the problem occurred
- Evidence of the problem
- Priority for solving problem

7.4 Software Inspections

Inspection in software engineering, refers to peer review of any work product by trained individuals who look for defects using a well-defined process.

- It is a manual, static technique that can be applied early in the development cycle.
- It is the most formal review type.
- It is led by the trained moderators.

- During inspection the documents are prepared and checked thoroughly by the reviewers before the meeting.
- It involves peers to examine the product.
- A separate preparation is carried out during which the product is examined and the defects are found.
- The defects found are documented in a logging list or issue log.

The Inspection Process

The inspection process should have entry criteria that determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process.

The stages in the inspections process are:

- ❖ **Planning:** The inspection is planned by the moderator.
- ❖ **Overview meeting:** The author describes the background of the work product.
- ❖ **Preparation:** Each inspector examines the work product to identify possible defects.
- ❖ **Inspection meeting:** During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.
- ❖ **Rework:** The author makes changes to the work product according to the action plans from the inspection meeting.
- ❖ **Follow-up:** The changes by the author are checked to make sure everything is correct.

The process is ended by the moderator when it satisfies some predefined exit criteria. The term inspection refers to one of the most important elements of the entire process that surrounds the execution and successful completion of a software engineering project.

Inspection team and role

During an inspection the following roles are used.

- ❖ **Author:** The person who created the work product being inspected.
- ❖ **Moderator:** This is the leader of the inspection. The moderator plans the inspection and coordinates it.
- ❖ **Reader:** The person reading through the documents, one item at a time. The other inspectors then point out defects.
- ❖ **Recorder/Scribe:** The person that documents the defects that are found during the inspection.
- ❖ **Inspector:** The person that examines the work product to identify possible defects.

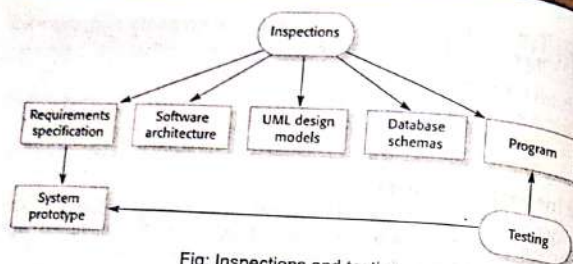


Fig: Inspections and testing

7.5 Verification and Formal Methods

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

- > Helpful in proving the correctness of systems such as: cryptographic protocols, and combinational circuits.
- > Verification of these systems is done by providing a formal proof on an abstract mathematical model of the system.
- > Can be used when a mathematical specification of the system is produced.
- > They are the ultimate static verification techniques.

Arguments for FM

- > Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- > They can detect implementation errors before testing when a program is analyzed alongside the specifications.

Arguments against FM

- > Requires specialized notations that are not understood by domain experts.
- > It is very expensive to develop a specification and even more expensive to show that a program meets that specification.
- > It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

7.6 Critical System Verification and Validation

Critical system verification and validation includes:

- ❖ **Reliability validation:**
Exercising the programs to access whether or not it has reached the required level of reliability.
- ❖ **Safety assurance:**
Concerned with establishing confidence labels in the system. However, quantitative measurements of safety are impossible.
- ❖ **Security assessment:**
Intended to demonstrate that the system can't enter some state rather than demonstrate the system can do something.
- ❖ **Safety and dependability cases:**
These are structured documents that set out detailed argument and evidence that a required level of safety or dependability has been achieved.

8

Software Testing and Cost Estimation

8.1 Software Testing

Software testing is a process of executing a program or application with the intent of finding the software bugs. It can also be stated as the process of validating and verifying that a software program or application or product:

- Meets the business and technical requirements that guided its design and development.
- Works as expected.

Software testing has different goals and objectives. The major objectives of software testing are as follows:

- ❖ To find defects that may be created by the programmer while developing the software.
- ❖ To gain confidence in and providing information about the level of quality.

- ❖ To prevent defects.
- ❖ To make sure that the end result meets the business and user requirements.
- ❖ To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- ❖ To gain the confidence of the customers by providing them a quality product.

Software Testing Hierarchy

As with almost any technical process, software testing has a prescribed order in which things should be done. Different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system. The following is lists of software testing categories arranged in sequentially organize.

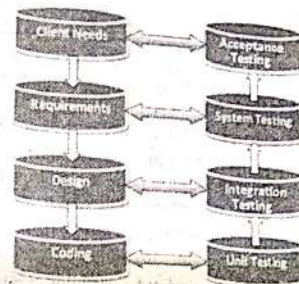


Fig: levels of testing hierarchy

1. **Unit testing:** Testing is done in the development process while developer completes the unit development. The object of this testing is to verify correctness of the module. The purpose of unit testing is to check that as individual parts are functioning as expected. Basically Unit testing is typically carried out by the developer.
2. **Integration testing:** System Integration Testing is started after the individual software modules are integrated as a group. A typical software project consists of multiple modules & these are developed by different developers. So in integration testing is focuses to check that after integrating modules is two modules are communicating with each other or not. It is critical to test every module's effect on the entire program model. Most of the issues are observed in this type of testing.

3. **System testing:** This is the first time end to end testing of application on the complete and fully integrated software product before it is launch to the market.
4. **Acceptance testing:** User acceptance is a type of testing performed by the Client to certify the system with respect to the requirements that was agreed upon. This is beta testing of the product & evaluated by the actual end users. The main purpose of this testing is to validate the end to end business flow.

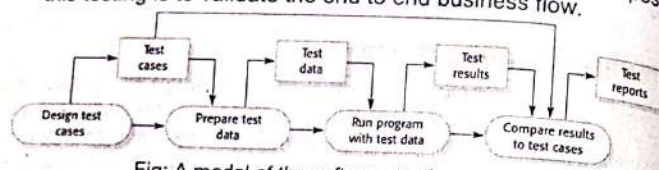


Fig: A model of the software testing process

8.2 System Testing

System testing is the type of testing to check the behavior of a complete and fully integrated software product based on the software requirements specification (SRS) document. The main focus of this testing is to evaluate Business / Functional / End-user requirements.

- This is black box type of testing where external working of the software is evaluated with the help of requirement documents & it is totally based on Users point of view.
- For this type of testing do not required knowledge of internal design or structure or code.
- This testing is to be carried out only after System Integration Testing is completed where both Functional & Non-Functional requirements are verified.
- In the integration testing testers are concentrated on finding bugs/defects on integrated modules.
- But in the Software System Testing testers are concentrated on finding bugs/defects based on software application behavior, software design and expectation of end user.
- System testing is the testing of a complete and fully integrated software product.

What do you verify in System Testing?

System testing involves testing the software code for following:

- ❖ Testing the fully integrated applications including external peripherals in order to check how components interact with one

another and with the system as a whole. This is also called End to End testing scenario.

- ❖ Verify thorough testing of every input in the application to check for desired outputs.
- ❖ Testing of the user's experience with the application.

That is a very basic description of what is involved in system testing. You need to build detailed test cases and test suites that test each aspect of the application as seen from the outside without looking at the actual source code.

Different Types of System Testing

1. **Usability Testing:** Usability Testing mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives.
2. **Load Testing:** Load Testing is necessary to know that a software solution will perform under real-life loads.
3. **Regression Testing:** Regression Testing involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.
4. **Recovery Testing:** Recovery testing is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.
5. **Migration Testing:** Migration testing is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.
6. **Functional Testing:** Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.
7. **Hardware/Software Testing:** This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

Differences between Unit and Component testing:

Unit Testing	Component Testing
1. Testing individual programs, modules to demonstrate that program executes as per the specification is called Unit Testing.	1. Testing each object or parts of the software separately with or without isolation of other objects is called Component Testing.
2. Its validated against design documents.	2. Its validated against test requirements, use cases.
3. Unit testing is done by Developers.	3. Component testing is done by Testers.
4. Unit testing is done first.	4. Component testing is done after unit testing is complete from the developers end.

8.3 Component Testing

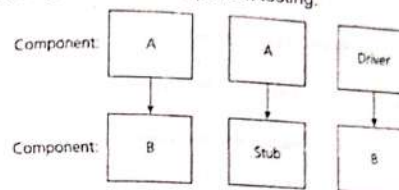
Component testing is a method where testing of each component in an application is done separately. Suppose, in an application there are 5 components. Testing of each 5 components separately and efficiently is called as component testing.

- Component testing is also known as module and program testing.
- It finds the defects in the module and verifies the functioning of software.
- Component testing is done by the tester.
- Component testing may be done in isolation from rest of the system depending on the development life cycle model chosen for that particular application.
- In such case the missing software is replaced by **Stubs** and **Drivers** and simulate the interface between the software components in a simple manner.
- Let's take an example to understand it in a better way. Suppose there is an application consisting of three modules say, module A, module B and module C. The developer has developed the module B and now wanted to test it. But in order to test the module B completely few of it's functionalities are dependent on module A and few on module C. But the module A and module C has not been developed yet. In that case to test the module B completely we can replace the module A and module C by stub and drivers as required.

➤ Integration testing is followed by the component testing.

- ❖ **Stub:** A stub is called from the software component to be tested. As shown in the diagram below 'Stub' is called by 'component A'.
- ❖ **Driver:** A driver calls the component to be tested. As shown in the diagram below 'component B' is called by the 'Driver'.

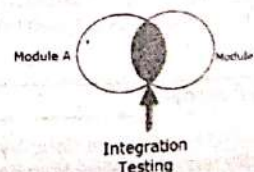
Below is the diagram of the component testing:



8.4 Integration Testing

Integration testing is a software testing methodology used to test individual software components or units of code to verify interaction between various software components and detect interface defects.

- It tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems.
- Components are tested as a single group or organized in an iterative manner. After the integration testing has been performed on the components, they are readily available for system testing.
- Also after integrating two different components together we do the integration testing.
- As displayed in the image below when two different modules 'Module A' and 'Module B' are integrated then the integration testing is done.
- Integration testing is done by a specific integration tester or test team.

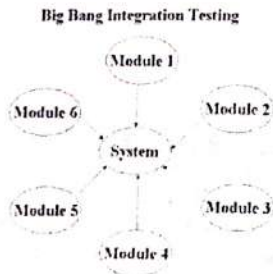


Below are the different strategies of Integration testing, the way they are executed and their limitations as well advantages.

8.4.1. Big Bang Integration Testing

Big Bang integration testing, all component are integrated together at once, and then tested.

- Here all components or modules are integrated simultaneously, after which everything is tested as a whole.
- As per the below image all the modules from 'Module 1' to 'Module 6' are integrated simultaneously then the testing is carried out.



Advantage:

- ❖ Big Bang testing has the advantage that everything is finished before integration testing starts.
- ❖ Convenient for small systems.

Disadvantage:

- ❖ Fault Localization is difficult.
- ❖ Given the sheer number of interfaces that need to be tested in this approach, some interfaces links to be tested could be missed easily.
- ❖ Since the integration testing can commence only after "all" the modules are designed, testing team will have less time for execution in the testing phase.
- ❖ Since all modules are tested at once, high risk critical modules are not isolated and tested on priority. Peripheral modules which deal with user interfaces are also not isolated and tested on priority.

8.4.2 Incremental Approach

The approach of testing where testing is done by joining two or more modules that are logically related is called as incremental approach.

- This process is carried out by using dummy programs called **Stubs and Drivers**.
- Stubs and Drivers do not implement the entire programming logic of the software module but just simulate data communication with the calling module.

The difference between stubs and driver can be shown as:

Stubs	Driver
1. Used in Top down approach	1. Used in Bottom up approach
2. Top most module is tested first	2. Lowest modules are tested first.
3. Stimulates the lower level of components	3. Stimulates the higher level of components
4. Dummy program of lower level components	4. Dummy program for Higher level component

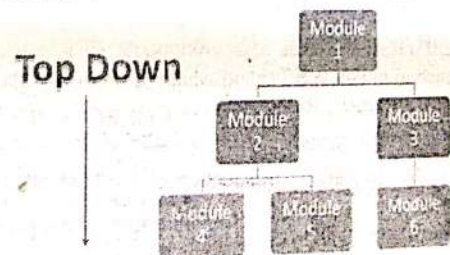
Incremental Approach is further divided into following:

- ❖ **Top Down Approach**
- ❖ **Bottom Up Approach**
- ❖ **Sandwich Approach** - Combination of Top Down and Bottom Up

a) Top-down integration testing:

In Top to down approach, testing takes place from top to down following the control flow of the software system.

- It takes help of stubs for testing.
- It Can be shown as below:



Advantages:

- ❖ Fault Localization is easier.
- ❖ Possibility to obtain an early prototype.
- ❖ Critical Modules are tested on priority; major design flaws could be found and fixed first.

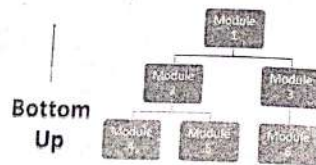
Disadvantages:

- ❖ Needs many Stubs.
- ❖ Modules at lower level are tested inadequately.

b) Bottom-up integration testing:

In bottom up testing takes place from the bottom of the control flow upwards.

- Components or systems are substituted by drivers.
- Below is the image of 'Bottom up approach':



Advantages:

- ❖ Fault localization is easier.
- ❖ No time is wasted waiting for all modules to be developed unlike Big-bang approach.

Disadvantages:

- ❖ Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
- ❖ Early prototype is not possible.

c) Sandwich/Hybrid testing:

It is an approach to Integration Testing which is a combination of Top Down and Bottom Up approaches.

Integration Testing Procedure

The integration test procedure irrespective of the test strategies are discussed here:

1. Prepare the Integration Tests Plan.
2. Design the Test Scenarios, Cases, and Scripts.
3. Executing the test Cases followed by reporting the defects.
4. Tracking & re-testing the defects.
5. Steps 3 and 4 are repeated until the completion of Integration is successfully.

Difference between System Testing and Integration Testing:

System Testing	Integration Testing
1. Testing the completed product to check if it meets the specification requirements.	1. Testing the collection and interface modules to check whether they give the expected result.
2. Both functional and non-functional testing are covered like sanity, usability, performance, stress and load.	2. Only the functional testing is performed to check whether the two modules when combined give correct outcome.
3. It is a high level testing performed after integration testing.	3. It is a low level testing performed after unit testing.
4. It is a black box testing technique so no knowledge of internal structure or code is required.	4. It is both black box and white box testing approach so it requires the knowledge of the two modules and the interface.
5. It is performed by test engineers only.	5. Integration testing is performed by developers as well test engineers.
6. Here the testing is performed on the system as a whole including all the external interfaces, so any defect found in it is regarded as defect of whole system.	6. Here the testing is performed on interface between individual module thus any defect found is only for individual modules and not the entire system.
7. In System Testing the test cases are developed to simulate real life scenarios.	7. Here the test cases are developed to simulate the interaction between the two module.
8. The System testing covers many different testing types like sanity, usability, maintenance, regression, retesting and performance.	8. Integration testing techniques includes big bang approach, top bottom, bottom to top and sandwich approach.

8.5 Regression Testing

Regression Testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features.

When any modification or changes are done to the application or even when any small change is done to the code then it can bring unexpected issues. Along with the new changes it becomes very important to test whether the existing functionality is intact or not. This can be achieved by doing the regression testing.

- The purpose of the regression testing is to find the bugs which may get introduced accidentally because of the new changes or modification.
- During confirmation testing the defect got fixed and that part of the application started working as intended. But there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these 'unexpected side-effects' of fixes is to do regression testing.
- This also ensures that the bugs found earlier are not creatable.
- Usually the regression testing is done by automation tools because in order to fix the defect the same test is carried out again and again and it will be very tedious and time consuming to do it manually.
- During regression testing the test cases are prioritized depending upon the changes done to the feature or module in the application.
- The feature or module where the changes or modification is done that entire feature is taken into priority for testing.
- This testing becomes very important when there are continuous modifications or enhancements done in the application or product.
- These changes or enhancements should not introduce new issues in the existing tested code.
- This helps in maintaining the quality of the product along with the new changes in the application.

Example

Let's assume that there is an application which maintains the details of all the students in school. This application has four buttons Add, Save, Delete and Refresh. All the buttons functionalities are working as expected. Recently a new button 'Update' is added in the application. This 'Update' button functionality is tested and confirmed that it's working as expected. But at the same time it becomes very important to know that the introduction of this new button should not impact the other existing buttons functionality. Along with the 'Update' button all the other buttons functionality are tested in order to find any new issues in the existing code. This process is known as regression testing.

Types of Regression testing techniques

There are four different types of regression testing techniques. They are as follows:

- 1) **Corrective Regression Testing:** Corrective regression testing can be used when there is no change in the specifications and test cases can be reused.
- 2) **Progressive Regression Testing:** Progressive regression testing is used when the modifications are done in the specifications and new test cases are designed.
- 3) **Retest-All Strategy:** The retest-all strategy is very tedious and time consuming because here we reuse all test which results in the execution of unnecessary test cases. When any small modification or change is done to the application then this strategy is not useful.
- 4) **Selective Strategy:** In selective strategy we use a subset of the existing test cases to cut down the retesting effort and cost. If any changes are done to the program entities, e.g. functions, variables etc. then a test unit must be rerun. Here the difficult part is to find out the dependencies between a test case and the program entities it covers.

When to use it

- ❖ Any new feature is added
- ❖ Any enhancement is done
- ❖ Any bug is fixed
- ❖ Any performance related issue is fixed

Advantages

- ❖ It helps us to make sure that any changes like bug fixes or any enhancements to the module or application have not impacted the existing tested code.
- ❖ It ensures that the bugs found earlier are not creatable.
- ❖ Regression testing can be done by using the automation tools
- ❖ It helps in improving the quality of the product.

Disadvantages

- ❖ If regression testing is done without using automated tools then it can be very tedious and time consuming because here we execute the same set of test cases again and again.
- ❖ Regression test is required even when a very small change is done in the code because this small modification can bring unexpected issues in the existing functionality.

8.6 Alpha and Beta Testing

Alpha Testing

Alpha testing is a type of acceptance testing performed to identify all possible issues/bugs before releasing the product to everyday users or public.

- The focus of this testing is to simulate real users by using blackbox and whitebox techniques.
- The aim is to carry out the tasks that a typical user might perform.
- Alpha testing is carried out in a lab environment and usually the testers are internal employees of the organization.
- To put it as simple as possible, this kind of testing is called alpha only because it is done early on, near the end of the development of the software, and before Beta Testing.

Beta Testing

Beta Testing of a product is performed by "real users" of the software application in a "real environment" and can be considered as a form of external User Acceptance Testing.

- Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality.
- Beta testing reduces product failure risks and provides increased quality of the product through customer validation.
- It is the final test before shipping a product to the customers.
- Direct feedback from customers is a major advantage of Beta Testing.
- This testing helps to test the product in real time environment.

Difference between Alpha testing and Beta testing:

Alpha Testing	Beta Testing (Field Testing)
1. It is always performed by the developers at the software development site.	1. It is always performed by the customers at their own site.
2. Alpha Testing is not open to the market and public	2. Beta Testing is always open to the market and public.
3. It is conducted for the software application and project.	3. It is usually conducted for software product.

4. It is always performed in Virtual Environment.

4. It is performed in Real Time Environment.

5. Alpha Testing is definitely performed and carried out at the developing organizations location with the involvement of developers.

5. Beta Testing (field testing) is performed and carried out by users or you can say people at their own locations and site using customer data.

6. Alpha Testing is always performed at the time of Acceptance Testing when developers test the product and project to check whether it meets the user requirements or not.

6. Beta Testing is always performed at the time when software product and project are marketed.

7. It is always performed at the developer's premises in the absence of the users.

7. It is always performed at the user's premises in the absence of the development team.

8.7 Black Box and White Box Testing

Black Box Testing

Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or Program

- Tester is aware of what the program should do but does not have the knowledge of how it does it.
- It provides external perspective of the software under test.

Advantages

- ❖ Efficient for large segments of code
- ❖ Code access is not required
- ❖ Separation between user's and developer's perspectives

Disadvantages

- ❖ Limited coverage since only a fraction of test scenarios is performed
- ❖ Inefficient testing due to tester's lack of knowledge about software internals
- ❖ Blind coverage since tester has limited knowledge about the application



Fig: Representation of black box testing

White Box Testing

White box testing is the software testing method in which internal structure is being known to tester who is going to test the software.

- White-box testing requires internal knowledge of the system and programming skills.
- It provides internal perspective of the software under test.

Advantages

- ❖ Efficient in finding errors and problems.
- ❖ Required knowledge of internals of the software under test is beneficial for thorough testing
- ❖ Allows finding hidden errors
- ❖ Programmers introspection
- ❖ Helps optimizing the code
- ❖ Due to required internal knowledge of the software, maximum coverage is obtained

Disadvantages

- ❖ Might not find unimplemented or missing features
- ❖ Requires high level knowledge of internals of the software under test
- ❖ Requires code access

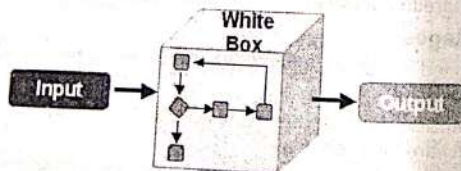


Fig: Representation of white box testing

Difference between Black Box and White Box Testing:

Black Box Testing	White Box Testing
1. Black box testing is the Software testing method which is used to test the software without knowing the internal structure of the program.	1. White box testing is the software testing method in which internal structure is being known to tester who is going to test the software.
2. This type of testing is carried out by testers.	2. Generally, this type of testing is carried out by software developers.
3. Implementation Knowledge is not required to carry out Black Box Testing.	3. Implementation Knowledge is required to carry out White Box Testing.
4. Programming Knowledge is not required to carry out Black Box Testing.	4. Programming Knowledge is required to carry out White Box Testing.
5. Testing is applicable on higher levels of testing like System Testing, Acceptance testing.	5. Testing is applicable on lower level of testing like Unit Testing, Integration testing.
6. Black box testing means functional test or external testing.	6. White box testing means structural test or interior testing.

8.8 Test Case Design

Test case design is the designing process that involves the designing of test cases i.e. inputs and outputs for the system test. The goal of test case design is to create a set of tests that are effective in validation and defect testing.

Test case design approaches:

1. Requirement-based testing

It is used in validation testing techniques where we consider each requirement and test for that requirement.

2. Partition testing

It is a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. This technique tries to define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed.

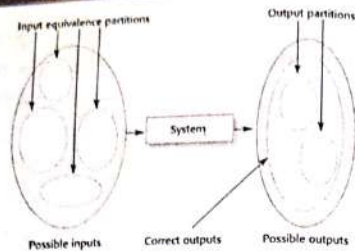


Fig: Equivalence partitioning

Input data and output results often fall into different classes where all members of a class are related. Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.

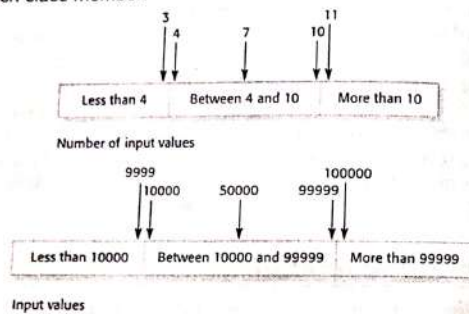


Fig: Example of equivalence partitions

3. Structural testing

It is a method of testing software that tests internal structures or working of an application, as opposed to its functionality (i.e. black-box testing). In white box testing an internal perspective of the system, as well as programming skills, are used to design test cases, the tester chooses inputs to exercise paths through the code and determine the appropriate outputs.

8.9 Test Automation

Test automation is a method in software testing that makes use of special software tools to control the execution of tests and then compares actual test results with predicted or expected results.

- Can automate previous repetitive but necessary testing in a formalized testing process already in place or add additional testing that would be difficult to perform manually.
- Reduces testing costs by supporting the test process with range of software tools.

There are two general approaches to test automation:

1. Code-driven testing:

The public interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.

2. Graphical user interface testing:

A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that results in the user interface, to validate that the observable behavior of the program is correct.

Automation Testing Tools

1. Selenium:

- It is a software testing tool used for regression testing.
- It is an open source testing tool that provides playback and recording facility for regression testing.
- The Selenium IDE only supports Mozilla Firefox WEB browser.

2. QTP (HP UFT):

- It is widely used for functional and regression testing, it addresses every major software application and environment.
- To simplify test creation and maintenance, it uses the concept of keyword driven testing.
- It allows the tester to build test cases directly from the application.

3. WATIR:

- It is an open source testing software for regression testing.
- It enables you to write tests that are easy to read and maintain.
- Watir supports only internet explorer on windows while Watir webdriver supports Chrome, Firefox, IE, Opera, etc.

8.10 Metrics for Testing

Majority of metrics for testing propose focus on the process of testing, not the technical characteristics of the test themselves.

Halstead metrics applied to testing:

Testing effort can be estimated using metrics derived from Halstead measures.

$$PL = \frac{1}{\frac{n1}{2} * \frac{N2}{n2}}$$
$$e = \frac{V}{PL}$$

Where,

PL is program Level

e is Halstead effort

V is program volume

n1 is no. of distinct operations that appears in program.

n2 is the no. of distinct operands that appears in the program.

N2 is the total no. of operand occurrence.

The percentage of overall testing effort to be allocated to a module 'K' can be estimated as:

$$K = \frac{e(K)}{\sum e(i)}$$

Where,

e(K) is computed for module K

$\sum e(i)$ is the sum of effort across all modules of the system.

8.11 Cyclomatic Complexity

Cyclomatic complexity is a software metric used to measure the complexity of a program. These metric measures independent paths through program source code.

- > Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths.
- > Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.
- > it is based on a control flow representation of the program.
- > Control flow depicts a program as a graph which consists of Nodes and Edges.
- > It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.

- > Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand.

It can be represented using the below formula:

$$\text{Cyclomatic complexity} = E - N + 2 \cdot P$$

where,

E = number of edges in the flow graph

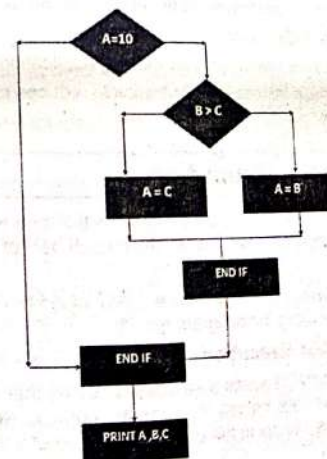
N = number of nodes in the flow graph

P = number of nodes that have exit points

Example

```
IF A = 10 THEN
  IF B > C THEN
    A = B
  ELSE
    A = C
  ENDIF
ENDIF
Print A
Print B
Print C
```

Flow Graph



The Cyclomatic complexity is calculated using the above control flow diagram that shows seven nodes (shapes) and eight edges (lines). Hence, the cyclomatic complexity is $8 - 7 + 2 = 3$

Tools for Cyclomatic Complexity calculation:

Many tools are available for determining the complexity of the application. Some complexity calculation tools are used for specific technologies. Complexity can be found by the number of decision points in a program.

Examples of tools are

- ❖ **OCLint** : Static code analyzer for C and Related Languages
- ❖ **devMetrics** : Analyzing metrics for C# projects
- ❖ **Reflector Add In** : Code metrics for .NET assemblies
- ❖ **GMetrics** : Find metrics in Java related applications
- ❖ **NDepends** : Metrics in Java applications

Uses of Cyclomatic Complexity:

The uses of the Cyclomatic Complexity can be summarized under the following points:

- ❖ Helps developers and testers to determine independent path executions.
- ❖ Developers can assure that all the paths have been tested at least once.
- ❖ Helps us to focus more on the uncovered paths.
- ❖ Improve code coverage.
- ❖ Evaluate the risk associated with the application or program.
- ❖ Using these metrics early in the cycle reduces more risk of the program.

8.12 Symbolic Execution

Symbolic execution or Symbolic evaluation is the means of analyzing a program to determine what inputs cause each part of a program to execute.

It is a software testing technique that is useful to aid the generation of test data and in proving the program quality.

Steps in Symbolic Execution

- ❖ The execution requires a selection of paths that are exercised by a set of data values. A program, which is executed using actual data, results in the output of a series of values.

- ❖ In symbolic execution, the data is replaced by symbolic values with set of expressions, one expression per output variable.
- ❖ The common approach for symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph.
- ❖ The flowgraph identifies the decision points and the assignments associated with each flow. By traversing the flow graph from an entry point, a list of assignment statements and branch predicates is produced.

Disadvantages of using symbolic execution

1. Symbolic execution cannot proceed if the number of iterations in the loop is known.
2. The second issue is the invocation of any out-of-line code or module calls.
3. Symbolic execution cannot be used with arrays.
4. The symbolic execution cannot identify of infeasible paths.

8.13 Software Productivity

Software productivity is the ratio between the amount of software produced to the labor and expense of producing it. There are two measures of software productivity:

1. Size related measures:

It measures the line of code delivered and measures no. of delivered object code instructions or no. of pages of system documentation.

For example: A system which might be coded in 5000 lines of assembly code. The development time for the various phases in 28 weeks:

$$\begin{aligned}\text{Then productivity} &= (5000 \times 28) / 4 \\ &= 714 \text{ lines/month}\end{aligned}$$

2. Function-related measures:

Productivity is expressed in terms of the amount of useful functionality produced in some given time. It is based on an estimate of the functionality of the delivered software. Function in a program is computed by measuring program features:

- External inputs and outputs
- User interactions
- External interfaces
- Files used by the system

A weight is associated with each of these and the function point cost is computed by multiplying each raw count by the weight and summing all values.

Factors affecting productivity:

- ❖ **Application domain experience:** Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
- ❖ **Process quality:** The development process used can have a significant effect on productivity.
- ❖ **Project size:** The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
- ❖ **Technology support:** Good support technology such as CASE tools, configuration management systems, etc. can improve productivity.
- ❖ **Working environment:** A quiet working environment with private work areas contributes to improved productivity.

8.14 Estimation Techniques

There is no simple way to make an accurate estimate of the effort required to develop a software system. Initial estimates are based on inadequate information in a user requirement definition. People in the project may be unknown. Project cost estimates may be self-fulfilling. The estimate defines the budget and the product is adjusted to meet the budget.

Some of the estimation techniques are:

1. **Algorithmic cost modeling:** A model based on historical cost information that relates some software metric (usually its size) to the project cost is used. An estimate is made of that metric and the model predicts the effort required.
2. **Expert judgment:** Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
3. **Estimation by analogy:** This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects.

4. **Parkinson's Law:** Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
5. **Pricing to win:** The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.

8.15 Algorithmic Cost Modeling

Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers:

$$\text{Effort} = A * \text{Size}^B * M$$

Where, A is an organization-dependent constant,
B reflects the disproportionate effort for large projects and
M is a multiplier reflecting product, process and people attributes.

The most commonly used product attribute for cost estimation is code size. Most of the models are similar but they use different values for A, B and M.

8.16 Estimation Accuracy

The size of a software system can only be known accurately when it is finished.

Several factors influence the final size

- Use of COTS and components
- Programming language
- Distribution of system

As the development process progresses, then the size estimate becomes more accurate.

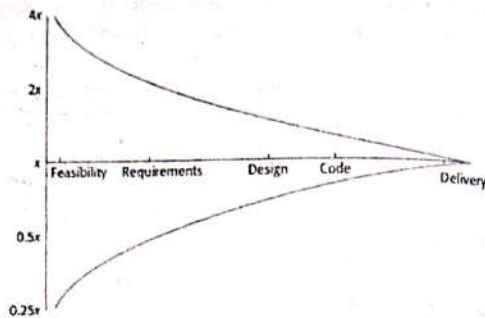


Fig: Estimate uncertainty graph

8.17 The COCOMO Model

The COCOMO model is an empirical model based on project experience.

- Well-documented, 'independent' model which is not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc.

COCOMO 81

Project complexity	Formula	Description
Simple	$PM = 2.4 (KDSI)^{1.05} * M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0 (KDSI)^{1.12} * M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6 (KDSI)^{1.20} * M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

COCOMO 2

COCOMO 81 was developed with the assumption that a waterfall process would be used and that all software would be developed from scratch.

Since its formulation, there have been many changes in software engineering practice and COCOMO 2 is designed to accommodate different approaches to software development.

COCOMO 2 models

COCOMO 2 incorporates a range of sub-models that produce increasingly detailed software estimates.

The sub-models in COCOMO 2 are:

- ❖ **Application composition model:** Used when software is composed from existing parts.
- ❖ **Early design model:** Used when requirements are available but design has not yet started.
- ❖ **Reuse model:** Used to compute the effort of integrating reusable components.
- ❖ **Post-architecture model:** Used once the system architecture has been designed and more information about the system is available.

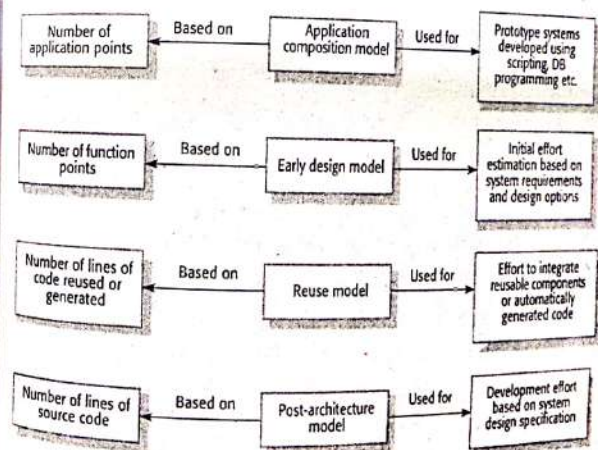


Fig: Use of COCOMO 2 models

8.18 Project Duration and Staffing

As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required.

- Calendar time can be estimated using a COCOMO 2 formula:
$$TDEV = 3 * (PM)^{(0.33 + 0.2 * (B - 1.01))}$$

Where PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model).

- This computation predicts the nominal schedule for the projects.
- The time required is independent of the number of people working on the project.
- Staff required can't be computed by dividing the development time by the required schedule.
- The number of people working on a project varies depending on the phase of the project.
- The more people who work on the project, the more total effort is usually required.
- A very rapid build-up of people often correlates with schedule slippage.



Quality Management

9.1 Quality Concept

Software quality is concerned with ensuring that the required level of quality is achieved in a software product.

- Involves defining appropriate quality standards and procedures and ensuring that these are followed.
- Quality management is particularly important for large, complex systems.
- The quality documentation is a record of progress and supports continuity of development as the development team changes.
- For smaller systems, quality management needs less documentation and should focus on establishing a quality culture.

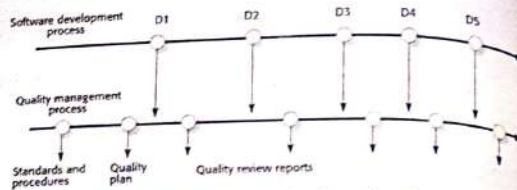


Fig: Quality management and software development

Quality management activities

Quality management should be separate from project management to ensure independence. The main three quality management activities are:

- ❖ **Quality assurance:** Establish organizational procedures and standards for quality.
- ❖ **Quality planning:** Select applicable procedures and standards for a particular project and modify these as required.
- ❖ **Quality control:** Ensure that procedures and standards are followed by the software development team.

Garvin's Quality Dimensions:

Garvin's eight dimensions can be summarized as follows:

1. **Performance:** Performance refers to a product's primary operating characteristics. This dimension of quality involves measurable attributes; brands can usually be ranked objectively on individual aspects of performance.
2. **Features:** Features are additional characteristics that enhance the appeal of the product or service to the user.
3. **Reliability:** Reliability is the likelihood that a product will not fail within a specific time period. This is a key element for users who need the product to work without fail.
4. **Conformance:** Conformance is the precision with which the product or service meets the specified standards.
5. **Durability:** Durability measures the length of a product's life. When the product can be repaired, estimating durability is more complicated. The item will be used until it is no longer economical to operate it. This happens when the repair rate and the associated costs increase significantly.

6. **Serviceability:** Serviceability is the speed with which the product can be put into service when it breaks down, as well as the competence and the behavior of the serviceperson.
7. **Aesthetics:** Aesthetics is the subjective dimension indicating the kind of response a user has to a product. It represents the individual's personal preference.
8. **Perceived Quality:** Perceived Quality is the quality attributed to a good or service based on indirect measures.

9.2 Software Quality Assurance

Software quality assurance (SQA) is a process that ensures that developed software meets and complies with defined or standardized quality specifications.

- SQA helps ensure the development of high-quality software.
- SQA practices are implemented in most types of software development, regardless of the underlying software development model being used.
- In a broader sense, SQA incorporates and implements software testing methodologies to test software. Rather than checking for quality after completion, SQA processes test for quality in each phase of development until the software is complete.
- With SQA, the software development process moves into the next phase only once the current/previous phase complies with the required quality standards.

It includes the following activities:

- Process definition and implementation
- Auditing
- Training

Processes could be:

- Software Development Methodology
- Project Management
- Configuration Management
- Requirements Development/Management
- Estimation
- Software Design
- Testing

Once the processes have been defined and implemented, Quality Assurance has the following responsibilities:

- identify weaknesses in the processes
- correct those weaknesses to continually improve the process

The quality management system under which the software system is created is normally based on one or more of the following models/standards:

- CMMI
- Six Sigma
- ISO 9000

Software Quality Assurance encompasses the entire software development life cycle and the goal is to ensure that the development and/or maintenance processes are continuously improved to produce products that meet specifications/requirements.

The process of Software Quality Control (SQC) is also governed by Software Quality Assurance (SQA). SQA is generally shortened to just QA.

Software Quality Control

Software Quality Control (SQC) is a set of activities for ensuring quality in software products.

- Software Quality Control is limited to the Review/Testing phases of the Software Development Life Cycle and the goal is to ensure that the products meet specifications/requirements.
- The process of Software Quality Control (SQC) is governed by Software Quality Assurance (SQA).
- While SQA is oriented towards prevention, SQC is oriented towards detection.

It includes the following activities:

Reviews

- Requirement Review
- Design Review
- Code Review
- Deployment Plan Review
- Test Plan Review
- Test Cases Review

Testing

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Difference between Software Quality Assurance and Software Quality Control:

Software Quality Assurance (SQA)	Software Quality Control (SQC)
1. SQA is a set of activities for ensuring quality in software engineering processes (that ultimately result in quality in software products). The activities establish and evaluate the processes that produce products.	1. SQC is a set of activities for ensuring quality in software products. The activities focus on identifying defects in the actual products produced.
2. It is process focused.	2. It is product focused.
3. Prevention oriented.	3. Detection oriented.
4. Organization wide.	4. Product/project specific.
5. It relates to all products that will ever be created by a process.	5. It relates to specific product.
6. The activities carried are: <ul style="list-style-type: none"> • Process Definition and Implementation • Audits • Training 	6. The activities carried are: <ul style="list-style-type: none"> • Reviews • Testing

9.3 Software Reviews

A software review is a process or meeting during which a software product is examined by a project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval.

- A group of people carefully examine part or all of a software system and its associated documentation.
- Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

Phases in the review process

- ❖ **Pre-review activities:** Pre-review activities are concerned with review planning and review preparation.
- ❖ **The review meeting:** During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team.

- ❖ **Post-review activities:** These address the problems and issues that have been raised during the review meeting.

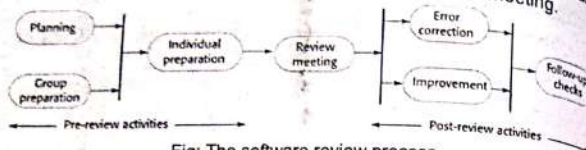


Fig: The software review process

The main review types are mentioned below:

1. Walkthrough

- It is not a formal process.
- It is led by the authors.
- Author guide the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.
- Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- Is especially useful for higher level documents like requirement specification, etc.

The goals of walkthrough:

- ❖ To present the documents both within and outside the software discipline in order to gather the information regarding the topic under documentation.
- ❖ To explain or do the knowledge transfer and evaluate the contents of the document.
- ❖ To achieve a common understanding and to gather feedback.
- ❖ To examine and discuss the validity of the proposed solutions.

2. Technical review

- It is less formal review.
- It is led by the trained moderator but can also be led by a technical expert.
- It is often performed as a peer review without management participation.
- Defects are found by the experts (such as architects, designers, key users) who focus on the content of the document.
- In practice, technical reviews vary from quite informal to very formal.

The goals of technical review:

- ❖ To ensure that an early stage the technical concepts are used correctly.
- ❖ To access the value of technical concepts and alternatives in the product.
- ❖ To have consistency in the use and representation of technical concepts.
- ❖ To inform participants about the technical content of the document.

3. Inspection

- It is the most formal review type.
- It is led by the trained moderators.
- During inspection the documents are prepared and checked thoroughly by the reviewers before the meeting.
- It involves peers to examine the product.
- A separate preparation is carried out during which the product is examined and the defects are found.
- The defects found are documented in a logging list or issue log.
- A formal follow-up is carried out by the moderator applying exit criteria.

The goals of inspection:

- ❖ It helps the author to improve the quality of the document under inspection.
- ❖ It removes defects efficiently and as early as possible.
- ❖ It improves product quality.
- ❖ It creates common understanding by exchanging information.
- ❖ It learn from defects found and prevent the occurrence of similar defects.

Differences between Inspections and Walkthrough

Inspection	Walkthrough
1. Inspection is a formalized method of improving a work product that deserves careful consideration by any organization concerned with the quality of the product they ship.	1. Walkthrough is an informal meeting for evaluation, usually no preparation is required for this.
2. It is Initiated by the project team	2. It is Initiated by the author

3. Planned meeting with fixed roles assigned to all the members involved	3. It is an unplanned approach.
4. Reader reads the product code. Everyone inspects it and comes up with defects.	4. Author reads the product code and his team mate comes up with defects or suggestions
5. Recorder records the defects	5. Author makes a note of defects and suggestions offered by team mate
6. Moderator has a role in making sure that the discussions proceed on the productive lines	6. Since it informal, there is no presence of moderator.

9.4 Formal Technical Reviews

Formal Technical review is a software quality assurance activity performed by software engineer.

Objectives of FTR

1. FTR is useful to uncover error in logic, function and implementation for any representation of the software.
 2. The purpose of FTR is to ensure that software meets specified requirements.
 3. It is also ensure that software is represented according to predefined standards.
 4. It helps to review the uniformity in software development process.
 5. It makes the project more manageable.
- Besides the above mentioned objectives, the purpose of FTR is to enable junior engineer to observe the analysis, design, coding and testing approach more closely.
- Each FTR is conducted as meeting and is considered successfully only if it is properly planned, controlled and attended.

Steps in FTR:

1. The review meeting

- Every review meeting should be conducted by considering the following constraints:
1. **Involvement of people:** Between 3 and 5 people should be involved in the review.

2. **Advance preparation:** Advance preparation should occur but it should be very short that is at the most 2 hours of work for each person can be spent in this preparation.
 3. **Short duration:** The short duration of the review meeting should be less than two hour.
- Rather than attempting to review the entire design walkthrough are conducted for modules or for small group of modules.
- The focus of the FTR is on work product (a software component to be reviewed). The review meeting is attended by the review leader, all reviewers and the producer.
- The review leader is responsible for evaluating for product for its deadlines.
- The copies of product material are then distributed to reviewers. The producer organizes "walkthrough" the product, explaining the material, while the reviewers raise the issues based on their advance preparation.
- One of the reviewers become recorder who records all the important issues raised during the review. When error are discovered, the recorder notes each.
- At the end of the review, the attendees decide whether to accept the product or not, with or without modification.
- #### 2. Review reporting and record keeping
- During the FTR, the reviewer actively records all the issues that have been raised.
- At the end of meeting these all raised issues are consolidated and review issue list is prepared.
- Finally, formal technical review summary report is produced.
- #### 3. Review guidelines
- Guidelines for the conducting of formal technical review must be established in advance.
- These guidelines must be distributed to all reviewers, agreed upon, and then followed.
- For example, Guideline for review may include following things
- ❖ Concentrate on work product only. That means review the product not the producers.
 - ❖ Set an agenda of a review and maintain it.
 - ❖ When certain issues are raised then debate or arguments should be limited. Reviews should not ultimately results in some hard feelings.

- ❖ Find out problem areas, but don't attempt to solve every problem noted.
- ❖ Take written notes (it is for record purpose)
- ❖ Limit the number of participants and insists upon advance preparation.
- ❖ Develop a checklist for each product that is likely to be reviewed.
- ❖ Allocate resources and time schedule for FTRs in order to maintain time schedule.
- ❖ Conduct meaningful trainings for all reviewers in order to make reviews effective.
- ❖ Reviews earlier reviews which serve as the base for the current review being conducted.

9.5 Formal Approaches to SQA

Formal approaches to SQA are a particular kind of mathematically based techniques for the assurance of the software.

- The use of formal methods for software design is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design.

9.6 Statistical Software Quality Assurance

Statistical SQA is a technique that measures the quality in a quantitative fashion.

- It implies that information about defects is collected and categorized and an attempt is made to trace each defect to underlying cause.
- It uses **Pareto Principle** to identify vital causes (80% of defects can be traced to 20% of causes) and moves to correct that problem that have caused the defects.

9.7 Software Reliability

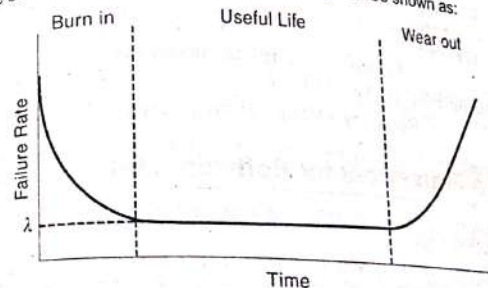
Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment.

- It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection.

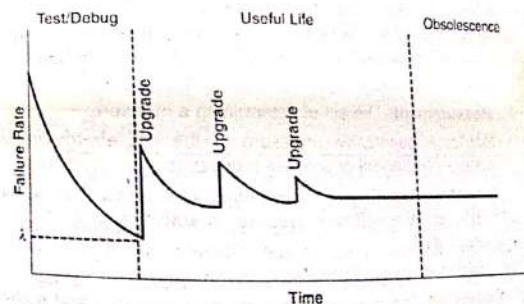
- The high complexity of software is the major contributing factor of Software Reliability problems.
- Software reliability can be measured, directed and estimated using historical and developmental data.

The change of failure rate over the product lifetime for a typical hardware and a software product are shown in the figure which is known as **Bathtub curve**.

The bathtub curve for the hardware reliability can be shown as:



Similarly, the Bathtub curve for the software reliability can be shown as in the figure:



Software reliability however does not show the same characteristics similar as hardware. For software, the failure rate is at its highest during integration and test. As the system is tested, more errors are identified and removed resulting in reduced failure rates.

Software Reliability Metrics

Reliability metrics are used to quantitatively express the reliability of a software product. Some reliability metrics are:

1. **MTTF (Mean time to failure):** It is the average time between observed system failures.
2. **MTTR (Mean time to repair):** It measures the average time it takes to track the errors causing the failure and then to fix them.
3. **Availability:** Software availability is the probability that a program is operating according to requirements at a given point time. It is given by:

$$\text{Availability} = \text{MTBF} / (\text{MTBF} + \text{MTTR})$$

Hence, Reliability can be calculated as:

$$\text{Reliability} = \text{MTBF} = \text{MTBF} / (1 + \text{MTBF})$$

9.8 A Framework for Software Metrics

Software metrics is a measure of some property of a piece of software or its specifications.

Good quality, reliability and maintainability are important attributes of enterprise applications and have a huge impact in the success on the economics of the business powered.

In framework for software Metrics, Measures, Measurement, Metrics and Indicators are often used interchangeably:

- ❖ **Measures:** Provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attributes of a product or process.
- ❖ **Measurement:** The act of determining a measure.
- ❖ **Metric:** A quantitative measure of the degree to which a system, component or process processes a given attribute.
- ❖ **Indicator:** A metric or combination of metrics that provide insight into the software process, a software project or the product itself.

Activities of a Measurement Process:

- ❖ **Formation:** The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
- ❖ **Collection:** The mechanism used to accumulate data required to derive the formulated metrics.
- ❖ **Analysis:** The computation of metrics and application of mathematical tools.

- ❖ **Interpretation:** The evaluation of metrics in an effort to gain insight into the quality of the representation.
- ❖ **Feedback:** Recommendations derived from the interpretation of product metrics and passed on to the software development team.

9.9 Metrics for Analysis and Design Model

The metrics for the analysis are:

- ❖ **Functionality delivered:** It provides an indirect measure of the functionality that is packaged within the software.
- ❖ **System Size:** It measures the overall size of the system defined in terms of information available as part of the analysis model.
- ❖ **Specification quality:** It provides an indication of the specific and completeness of a requirements specification.

The metrics for the design are:

- ❖ **Architectural metrics:** Provide an indication of the quality of the architectural design.
- ❖ **Component-level metrics:** It measures the complexity of software components and other characteristics that have a bearing on quality.
- ❖ **Interface design metrics:** It focuses primarily on usability.
- ❖ **Specialized object oriented design metrics:** It measures characteristics of classes and their communication and collaboration characteristics.

9.10 ISO Standards

The mission of the ISO is to promote the development of standardization and related activities to facilitate the international exchange of goods and services and to develop cooperation in the spheres of intellectuals, scientific, technological, and economic activity.

- ISO/IEC 9126 Software engineering — Product quality was an international standard for the evaluation of software quality.
- It has been replaced by ISO/IEC 25010:2011.
- The fundamental objective of the ISO/IEC 9126 standard is to address some of the well known human biases that can adversely affect the delivery and perception of a software development project.

The standard is divided into four parts:

1. **Quality model:** classifies software quality in a structured set of characteristics and sub-characteristics as follows:
 - functionality
 - Reliability
 - Usability
 - Efficiency
 - Maintainability
 - Portability
2. **External metrics:** External metrics are applicable to running software.
3. **Internal metrics:** Internal metrics are those which do not rely on software execution (static measure).
4. **Quality in use metrics:** Quality-in-use metrics are only available when the final product is used in real conditions

Why is ISO certification required by the software industry?

- ✓ It is the sign of customer confidence. This certification has become the standard for international bidding.
- ✓ It highlights weakness and suggests corrective measures for improvements.
- ✓ It makes processes more focused, efficient, and cost effective.
- ✓ It is a motivating factor for business organizations.
- ✓ It helps in designing high quality repeatable software products.

9.11 CMMI

The Capability Maturity Model Integration, or CMMI, is a process model that provides a clear definition of what an organization should do to promote behaviors that lead to improved performance.

Difference between CMM and CMMI

- CMM is a reference model of matured practices in a specified discipline like Systems Engineering CMM, Software CMM, People CMM, Software Acquisition CMM etc., but they were difficult to integrate as and when needed.
- CMMI is the successor of the CMM and evolved as a more matured set of guidelines and was built combining the best components of individual disciplines of CMM (Software CMM, People CMM, etc.). It can be applied to product manufacturing, people management, software development, etc.

CMM describes about the software engineering alone where as CMMI Integrated describes both software and system engineering. CMMI also incorporates the Integrated Process and Product Development and the supplier sourcing.

CMMI Objectives

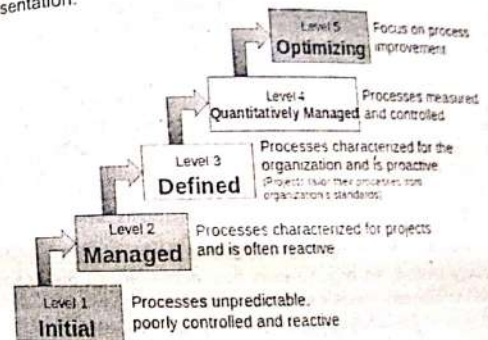
The objectives of CMMI are very obvious. They are as follows:

- ❖ Produce quality products or services
- ❖ Create value for the stockholders
- ❖ Enhance customer satisfaction
- ❖ Increase market share
- ❖ Gain an industry-wide recognition for excellence

CMMI maturity level

A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. Each maturity level provides a layer in the foundation for continuous process improvement.

The following image shows the maturity levels in a CMMI staged representation:



CMMI models with staged representation have five maturity levels designated by the numbers 1 through 5. They are:

Maturity Level 1 – Initial:

At maturity level 1, processes are usually ad hoc and chaotic. The organization usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization and not on the use of proven processes.

Maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects. They are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes.

Maturity Level 2 – Managed:

At maturity level 2, an organization has achieved all the **specific and generic goals** of the maturity level 2 process areas. In other words, the projects of the organization have ensured that requirements are managed and that processes are planned, performed, measured, and controlled.

At maturity level 2, requirements, processes, work products, and services are managed. The status of the work products and the delivery of services are visible to management at defined points. Commitments are established among relevant stakeholders and are revised as needed. Work products are reviewed with stakeholders and are controlled.

Maturity Level 3 – Defined:

At maturity level 3, an organization has achieved all the specific and generic goals of the process areas assigned to maturity levels 2 and 3.

At maturity level 3, processes are well characterized and understood, and are described in standards, procedures, tools, and methods. Here, processes are typically described in more detail and more rigorously than at maturity level 2. At maturity level 3, processes are managed more proactively using an understanding of the interrelationships of the process activities and detailed measures of the process, its work products, and its services.

Maturity Level 4 – Quantitatively Managed:

At maturity level 4, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, and 4 and the generic goals assigned to maturity levels 2 and 3.

At maturity level 4, sub-processes are selected that significantly contribute to the overall process performance. These selected sub-processes are controlled using statistical and other quantitative techniques.

Quantitative objectives for quality and process performance are established and used as criteria in managing the processes. Quantitative objectives are based on the needs of the customer, end users, organization, and process implementers. Quality and

process performances are understood in statistical terms and are managed throughout the life of the processes.

Maturity Level 5 – Optimizing:

At maturity level 5, an organization has achieved all the specific goals of the process areas assigned to maturity levels 2, 3, 4, and 5 and the generic goals assigned to maturity levels 2 and 3.

Processes are continually improved based on a quantitative understanding of the common causes of variation inherent in processes.

This level focuses on continually improving process performance through both incremental and innovative technological improvements.

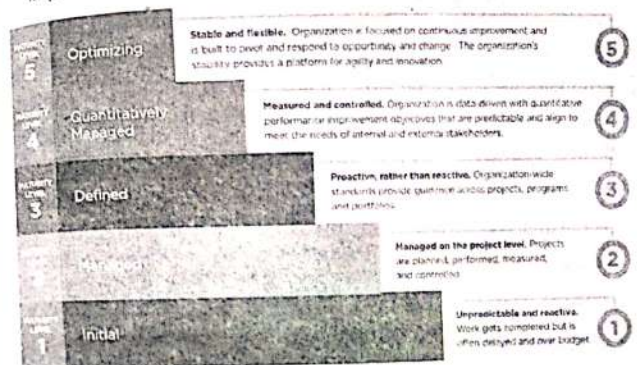


Fig: summarized view of CMMI levels

Difference between CMMI and ISO

CMMI	ISO
1. CMMI is a set of related "best practices" derived from industry leaders and relates to product engineering and software development.	1. ISO is a certification tool that certifies businesses whose processes conform to the laid down standards.
2. CMMI is a process model.	2. ISO is an audit standard.
3. CMMI is rigid and extends only to businesses developing software intensive systems.	3. ISO is flexible and applicable to all manufacturing industries.

4. CMMI compares the existing processes to industry best practices	4. ISO requires adjustment of existing processes to confirm to the specific ISO requirements.
5. CMMI is more focused, complex, and aligned with business objectives	5. ISO is flexible, wider in scope and not directly linked to business objectives.
6. It provides grade for process maturity.	6. ISO provides pass or fail criteria.
7. It reconnects the mechanism for step by step progress through its successive maturity levels.	7. ISO does not specify sequence of steps required to establish the quality system.
8. CMM is specially developed for software industry	8. It applies to any type of industry.
9. CMM has 5 levels: Initial Repeatable Defined Managed Optimization	9. ISO 9000 has no levels.

9.12 SQA Plan

The SQA plan is a document that specifies the process to be followed in each step of the software development and the procedures to be followed in each activity of such a process.

The objective of SQA plan is to ensure that the development of the software is based on a course of action and that from time to time the development can be measured controlled and monitored with respect to such a course of action-so that the end product is as per the specifications.

The plan is governed by several quality standards, policies and models such as ISO9000, SEI CMM and Baldrige.

9.13 Software Certification

Software certification is the certification that depicts the reliability and safety of software systems in such a way that it can be checked by an independent authority with minimal trust in the techniques and tools used in the certification process itself.

It builds on existing software assurance, validation and verification techniques but introduces the notion of explicit software certificates, which contain all information necessary for an independent assessment of the demonstrated properties.

Software certification comprises a wide range of formal, semi-formal and informal assurance techniques, including formal verification of compliance with explicit safety policies, system simulation, testing, code reviews and human "Sign Offs", and even references to supporting literature, consequently, the certificates can have different types and the certification process requires different mechanisms.

Configuration Management

10.1 Configuration Management Planning

Configuration management is a technique or discipline to systematically manage, organize and control the changes in the documents, codes, artifacts and other entities during the development life cycle.

- It is concerned with the policies, processes and tools for managing changing software systems.
- Software systems are constantly changing during development and use.
- You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- CM is essential for team projects to control changes made by different developers

CM activities

- ❖ **Version management:** Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
- ❖ **System building:** The process of assembling program components, data and libraries, then compiling these to create an executable system.
- ❖ **Change management:** Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented.
- ❖ **Release management:** Preparing software for external release and keeping track of the system versions that have been released for customer use.

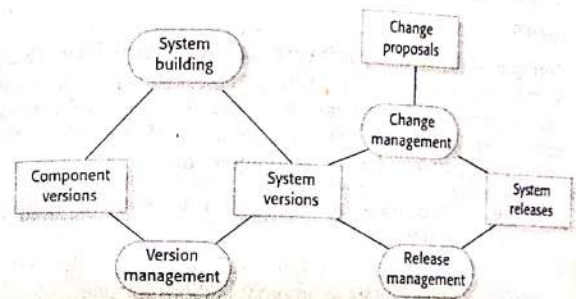


Fig: Configuration management activities

CM planning

- Starts during the early phases of the project
- Must define the documents or document classes which are to be managed (Formal documents).
- Documents which might be required for future system maintenance should be identified and specified as managed documents.
- Defines the types of documents to be managed and a document naming scheme.
- Defines who takes responsibility for the CM procedures and creation of baselines.
- Defines policies for change control and version management.
- Defines the CM records which must be maintained.

- Describes the tools which should be used to assist the CM process and any limitations on their use.

CM terminologies:

- ✓ **Baseline:** A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it is always possible to recreate a baseline from its constituent components.
- ✓ **Codeline:** A codeline is a set of versions of a software component and other configuration items on which that component depends.
- ✓ **Configuration (version) control:** The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
- ✓ **Configuration item or software configuration item (SCI):** Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
- ✓ **Mainline:** A sequence of baselines representing different versions of a system.
- ✓ **Release:** A version of a system that has been released to customers (or other users in an organization) for use.
- ✓ **Repository:** A shared database of versions of software components and meta-information about changes to these components.
- ✓ **Version:** An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
- ✓ **Workspace:** A private work area where software can be modified without affecting other developers who may be using or modifying that software.

10.2 Change Management

The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.

Factors in change analysis

- ❖ The consequences of not making the change
- ❖ The benefits of the change
- ❖ The number of users affected by the change
- ❖ The costs of making the change
- ❖ The product release cycle

10.3 Version Management

Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used.

- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- Therefore version management can be thought of as the process of managing codelines and baselines.

Codelines

- A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- Codelines normally apply to components of systems so that there are different versions of each component.

Baselines

- A baseline is a definition of a specific system.
- The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.
- Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system.
- Baselines are important because you often have to recreate a specific version of a complete system.

For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to

recreate the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

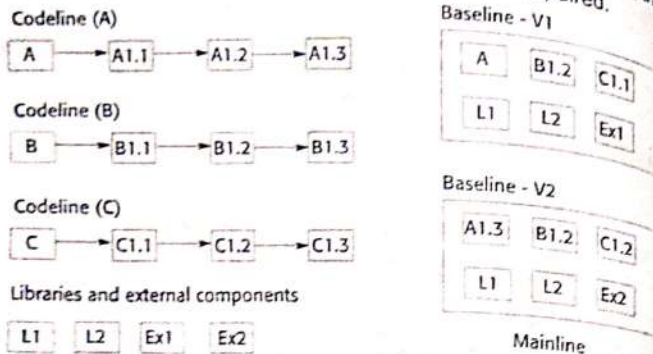


Fig. Codelines and baselines

Version control systems

Version control (VC) systems identify, store and control access to the different versions of components. There are two types of modern version control system:

- ❖ **Centralized systems:** Here, there is a single master repository that maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
- ❖ **Distributed systems:** Here, the multiple versions of the component repository exist at the same time. Git is a widely-used example of a distributed VC system.

Key features of version control systems

- Version and release identification
- Change history recording
- Support for independent development
- Project support
- Storage management

Branching and Merging

Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences.

Branching is the creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.

Merging is the creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.

- This is normal in system development, where different developers work independently on different versions of the source code and so change it in different ways.
- At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made.
- If the changes made involve different parts of the code, the component versions may be merged automatically by combining the deltas that apply to the code.

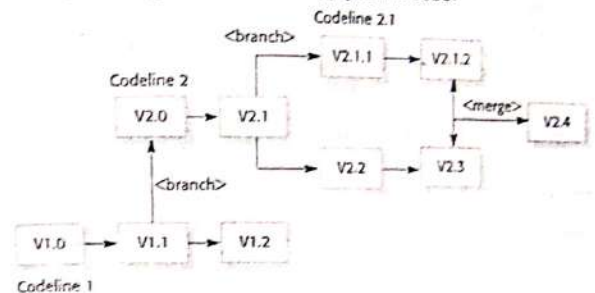


Fig. Branching and merging

Version identification

Procedures for version identification should define an unambiguous way of identifying component versions.

Three basic techniques for component identification are:

1. Version numbering

Simple naming scheme uses a linear derivation.

e.g. V1, V1.1, V1.2, V2.1, V2.2 etc.

- Actual derivation structure is a tree or a network rather than a sequence
- Names are not meaningful.
- Hierarchical naming scheme may be better.

2. Attribute-based identification

Attributes can be associated with a version with the combination of attributes identifying that version.

- Examples of attributes are Date, Creator, Programming Language, Customer, Status etc.
- More flexible than an explicit naming scheme for version retrieval; Can cause problems with uniqueness.
- Needs an associated name for easy reference.

3. Change-oriented Identification

Integrates versions and the changes made to create these versions.

- Used for systems rather than components.
- Each proposed change has a change set that describes changes made to implement that change.
- Change sets are applied in sequence so that, in principle, a version of the system that incorporates an arbitrary set of changes may be created.

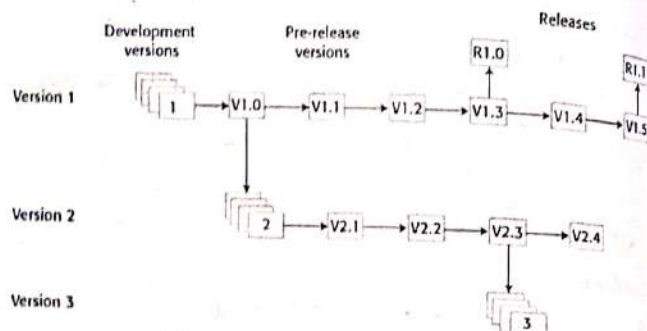


Fig: Multi-version system development

10.4 Release Management

A system release is a version of a software system that is distributed to customers.

- Releases must incorporate changes forced on the system by errors discovered by users and by hardware changes.
- They must also incorporate new system functionality.
- Release planning is concerned with when to issue a system version as a release.

For mass market software, it is usually possible to identify two types of release: major releases which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported.

For custom software or software product lines, releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time.

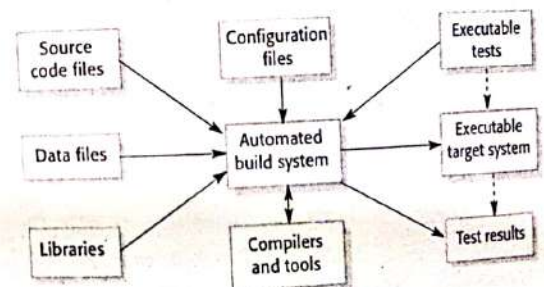
Factors influencing system release planning

- Competition
- Marketing requirements
- Platform changes
- Technical quality of the system

10.5 System Building

System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.

- System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system.
- The configuration description used to identify a baseline is also used by the system building tool.



10.6 CASE Tools for Configuration Management

CASE tools are set of software application programs, which are used to automate SDLC activities.

- CASE tools are available to support all CM activities.
- CM processes are standardized and involve applying pre-defined procedures.

- Mature CASE tools to support configuration management are available ranging from stand-alone tools to integrated CM workbenches.

CM workbenches

Open workbenches:

Tools for each stage in the CM process are integrated through organizational procedures and scripts. It gives flexibility in tool selection.

Integrated workbenches:

Provide whole-process, integrated support for configuration management. More tightly integrated tools so easier to use. However, the cost is less flexibility in the tools used.

Change management tools:

Change management is a procedural process so it can be modeled and integrated with a version management system.

Version management tools:

These tools mainly fulfill the following tasks:

- Version and release identification
- Storage management
- Change history recording
- Independent development
- Project support



Object Oriented Software Engineering

11.1 Object Oriented Analysis

Object-oriented analysis is the process that emphasizes on finding and describing the objects or concepts in the problem domain.

- It is the use of modeling to define and analyze the requirements necessary for success of a system.
- It groups items that interact with one another, typically by class, data or behavior, to create a model that accurately represents the intended purpose of the system as a whole.
- Emphasizes an investigation of the problem and requirements, rather than a solution.

Steps/Activities for Object oriented Analysis

1. Analyze the domain problem
2. Describe the process of systems
3. Identify the objects

4. Specify attributes
5. Defining operations
6. Define and establish Inter-object Communication mechanism

11.2 Object Oriented Design

Object oriented design is the process that emphasizes on defining software objects and how they collaborate to fulfill the requirements.

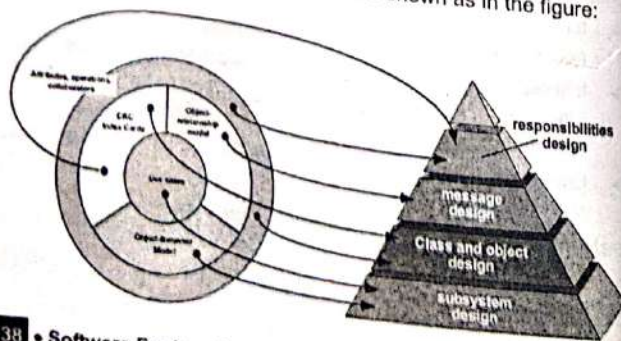
- It is the process of planning a system of interacting objects for the purpose of solving a software problem.
- Emphasizes a conceptual solution (in software and hardware) that fulfills the requirements, rather than its implementation.
- Transforms the analysis model created using OOA into a design model that serves as a blueprint for software construction.

Steps for Object oriented Design

1. First, build object model based on objects & relationship
2. Then iterate & refine model
 - a. Design & refine classes
 - b. Design & refine attributes
 - c. Design & refine methods
 - d. Design & refine structures
 - e. Design & refine associations

OOA to OOD:

The conversion from OOA to OOD can be shown as in the figure:



11.3 Unified Modeling Diagrams

Unified Modeling language (UML) is a standardized modeling language enabling developers to specify, visualize, construct and document artifacts of a software system.

- It is an important aspect involved in object-oriented software development.
- Makes these artifacts scalable, secure and robust in execution.
- Uses graphic notation to create visual models of software systems.
- It is a standard notation for the modeling of real-world objects as a first step in developing an object-oriented design methodology.

Importance of using UML

1. Provides extensibility and specialization mechanisms to extend the core concepts.
2. It is independent of particular programming languages and development processes.
3. Provides a formal basis for understanding the modeling language.
4. Encourages the growth of the OO tools market.
5. Supports higher-level development concepts such as collaborations, frameworks, patterns and components.

Relations in UML

1. **Association:** An association relation is established when two classes are connected to each other in any way. The symbol is:
2. **Aggregation:** When a class is formed as a collection of other classes, it is called an aggregation relationship between these classes. It is also called a "has a" relationship. There is existence of the derived class even if the base class is destroyed. For example, in the relationship between books and library, books can exist independently even if the library is destroyed.



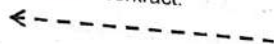
3. **Composition:** The composition is a variation of the aggregation relationship. Composition illustrates that a strong life cycle is present between the classes. For example, in the relationship between department and university, the departments cannot exist if the university is deleted.



4. **Generalization:** It is the inheritance relation where the child classes "inherit" the common functionality defined in the parent class.



5. **Realization:** In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) "realizes" the contract by implementing the functionality defined in the contract.



UML diagrams can be divided into two categories.

1. Structural diagram

Structure diagrams are used in documenting the architecture of software systems and are involved in the system being modeled. Different structure diagrams are:

- ❖ **Class Diagram:** represents system class, attributes and relationships among the classes.
- ❖ **Component Diagram:** represents how components are split in a software system and dependencies among the components.
- ❖ **Deployment Diagram:** describes the hardware used in system implementations.
- ❖ **Object Diagram:** represents a complete or partial view of the structure of a modeled system.
- ❖ **Package Diagram:** represents splitting of a system into logical groupings and dependency among the grouping.

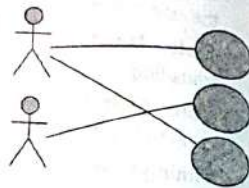
2. Behavioral diagram

Behavior diagrams represent functionality of software system and emphasize on what must happen in the system being modeled. The different behavior diagrams are:

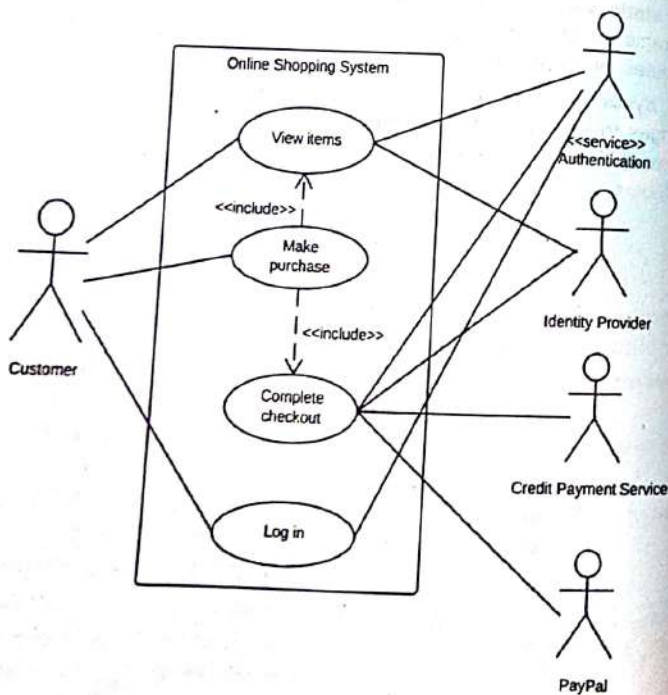
- ❖ **Activity Diagram:** represents step by step workflow of business and operational components.
 - ❖ **Use Case Diagram:** describes functionality of a system in terms of actors, goals as use cases and dependencies among the use cases.
 - ❖ **State Machine Diagram:** represents states and state transition.
 - ❖ **Communication Diagram:** represents interaction between objects in terms of sequenced messages.
 - ❖ **Timing Diagrams:** focuses on timing constraints.
 - ❖ **Sequence Diagram:** represents communication between objects in terms of a sequence of messages.
- UML diagrams represent static and dynamic views of a system model. The **static view** includes class diagrams and composite structure diagrams, which emphasize static structure of systems using objects, attributes, operations and relations. The **dynamic view** represents collaboration among objects and changes to internal states of objects through sequence, activity and state machine diagrams.

11.3.1 Use case Diagram

A use case diagram expresses how a user might use an object or system. In the diagram, the potential user is represented by a stick figure, called an "actor" symbol, and the various use cases are symbolized by oval shapes. Use case diagrams are ideal for communicating the principal functions of software systems, but they also have other valuable applications.

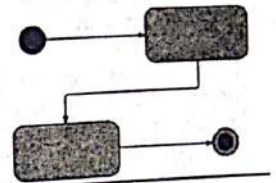


Example:

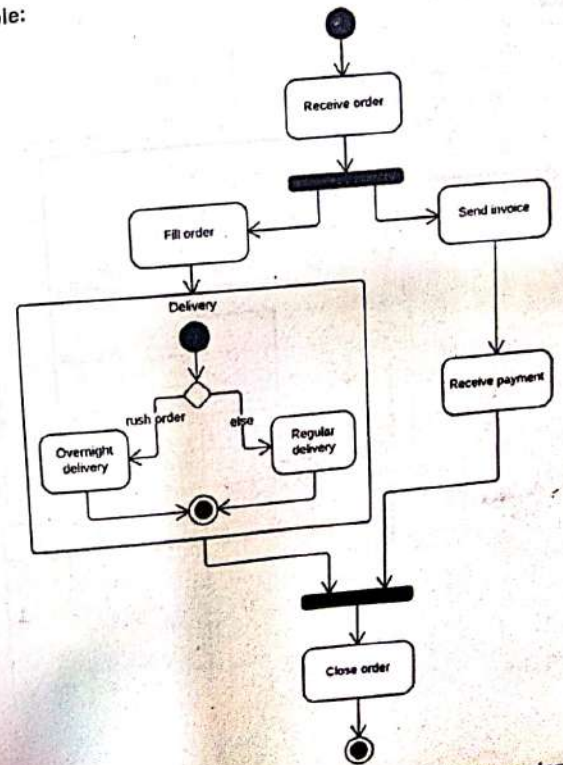


11.3.2 Activity Diagram

Activity diagrams are behavior-based flowcharts that capture processes from start to finish. Many UML diagrams generally describe software, but activity diagrams are often used to model other flows, such as showing how a patient checks into a hospital.

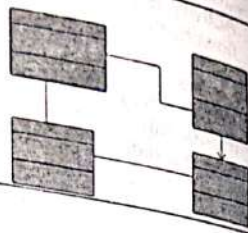


Example:

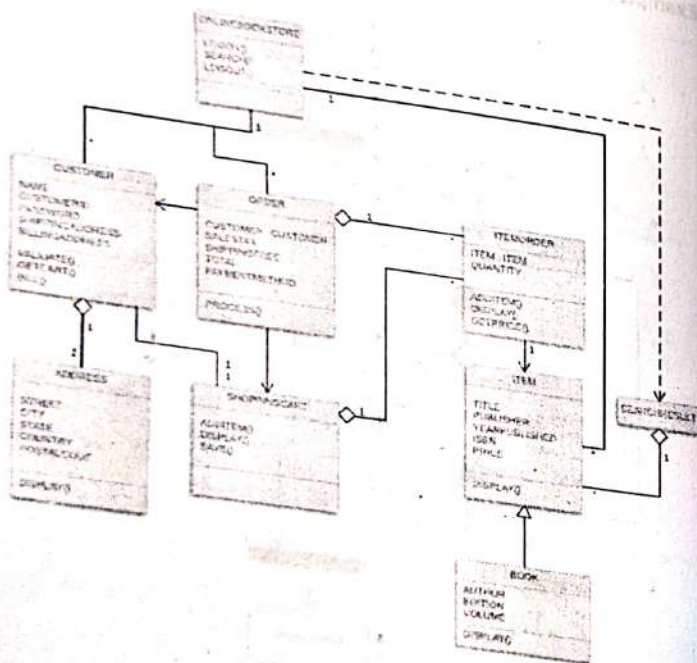


11.3.3 Class Diagram

Class diagrams organize elements into objects and classes. They are structural diagrams that are useful for identifying relationships between objects and for categorizing objects in an accessible and coherent manner.

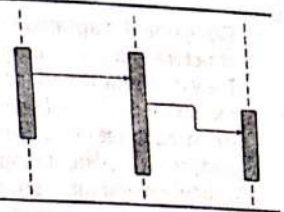


Example:

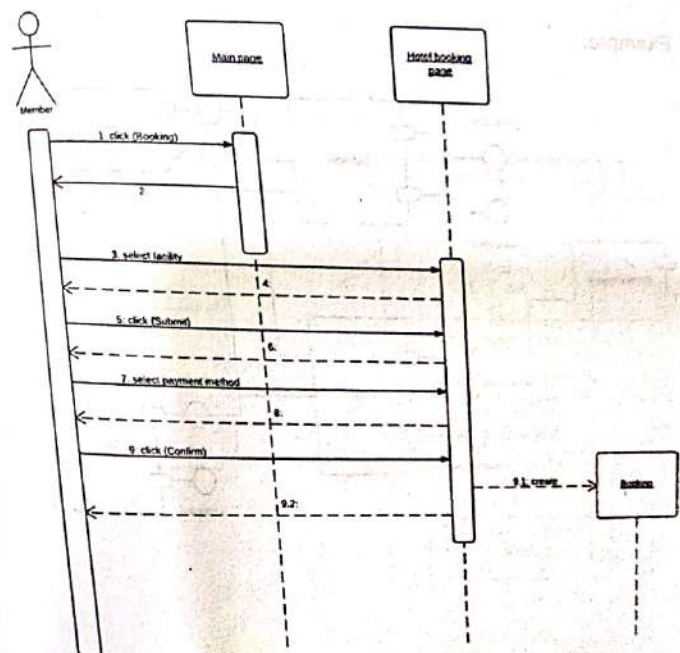


11.3.4 Sequence Diagram

A sequence diagram shows how events occur in sequential order over the flow of time. Objects and actors in the sequence are depicted with several different symbols. As time passes, the objects send messages to each other via arrows.

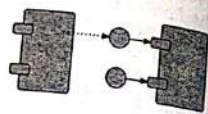


Example:

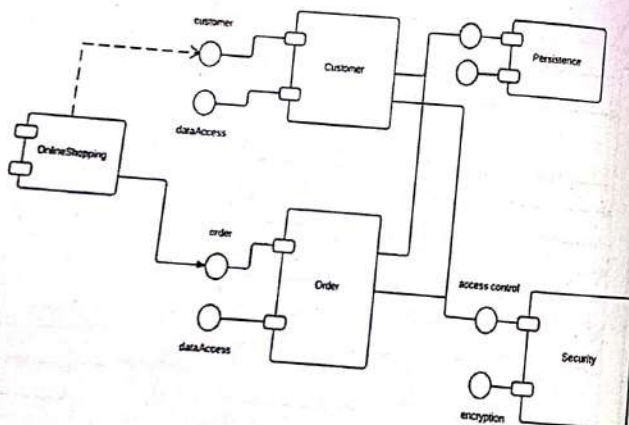


11.3.5 Component Diagram

Component diagrams fall under the structural diagram category in UML. They depict how various components in a software system are wired together to form a total product. Component diagrams include several main shapes, including the rectangular component shape and the circular lollipop shape.

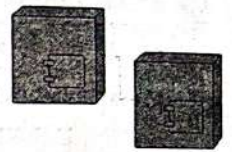


Example:

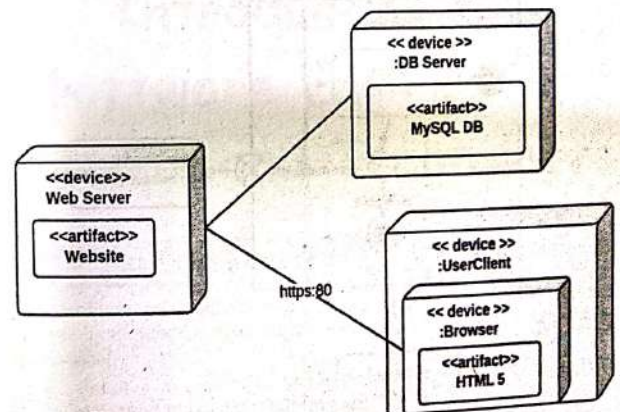


11.3.6 Deployment Diagram

Deployment diagrams, another group of structural diagrams, describe not only the software of a system but the hardware as well. They get their name from showing which software elements are deployed by the hardware. Their main shape is a large three-dimensional box known as a node, which represents the object doing the deploying.

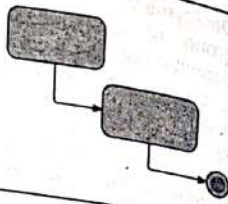


Example:

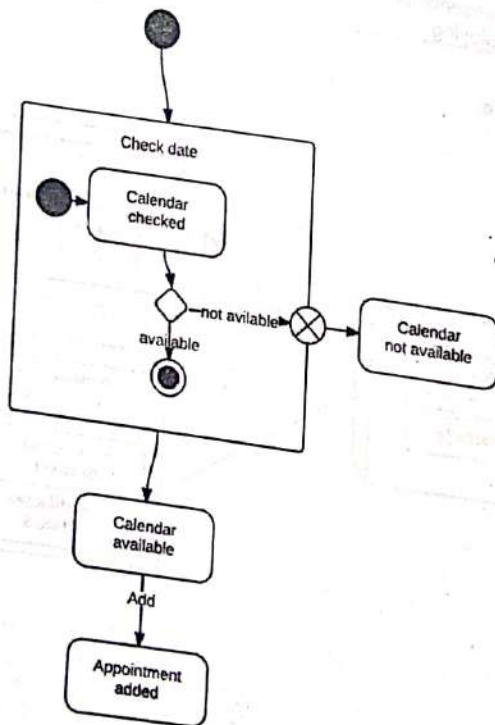


11.3.7 State Machine Diagram

State diagrams, also known as state machine diagrams, are a type of behavior diagram in UML. They illustrate how objects move from state to state via transition messages. The main shape for these is a rectangle with rounded corners



Example:



12

Introduction to Software Engineering Trends and Technology

12.1 Agile Development

Agile development model is a type of Incremental model which results in small incremental releases with each release building on previous functionality.

- Iterative approach is taken and working software build is delivered after each of iterations.
- The tasks are divided to time boxes (small time frames) to deliver specific features for a release.
- Each release is thoroughly tested to ensure software quality is maintained.
- Used for time critical applications.

- Each build is incremental in terms of features; the final build holds all the features required by the customer.
- Extreme Programming (XP) is currently the most well-known agile development life cycle model.

Here is a graphical illustration of the Agile Model:

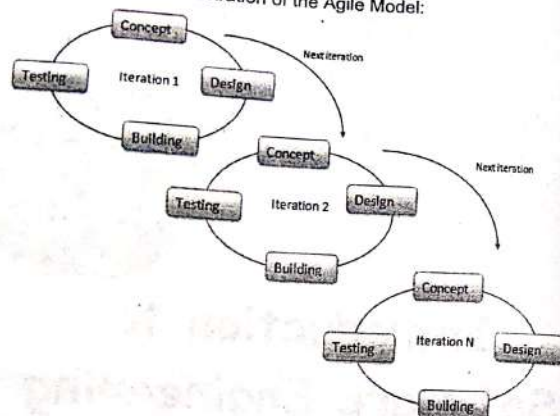


Fig: The Agile model of Software development lifecycle

Advantages of Agile model:

- 1) Customer satisfaction by rapid, continuous delivery of useful software.
- 2) People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- 3) Working software is delivered frequently (weeks rather than months).
- 4) Face-to-face conversation is the best form of communication.
- 5) Close, daily cooperation between business people and developers.
- 6) Continuous attention to technical excellence and good design.
- 7) Regular adaptation to changing circumstances.
- 8) Even late changes in requirements are welcomed

Disadvantages of Agile model:

- 1) In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.
- 2) There is lack of emphasis on necessary designing and documentation.
- 3) The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- 4) Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for newbie programmers, unless combined with experienced resources.

When to use Agile model:

- ❖ When new changes are needed to be implemented. New changes can be implemented at very little cost because of the frequency of new increments that are produced.
- ❖ To implement a new feature the developers need to lose only the work of a few days, or even only hours, to roll back and implement it.

12.2 Extreme Programming

Extreme programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements.

As a type of agile software development, it advocates frequent releases in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

- The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to extreme levels.
- The most prominent of several agile software development methodologies.
- Like other agile methodologies, Extreme Programming differs from traditional methodologies primarily in placing a higher value on adaptability than on predictability.

- Proponents of XP regard ongoing changes to requirements as an often natural and often inescapable aspect of software development projects.
- Uses an object-oriented approach as its preferred development paradigm.

Why is it called "Extreme?"

Extreme Programming takes the effective principles and practices to extreme levels.

- ✓ Code reviews are effective as the code is reviewed all the time.
- ✓ Testing is effective as there is continuous regression and testing.
- ✓ Design is effective as everybody needs to do refactoring daily.
- ✓ Integration testing is important as integrate and test several times a day.
- ✓ Short iterations are effective as the planning game for release planning and iteration planning.

Extreme programming encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing.

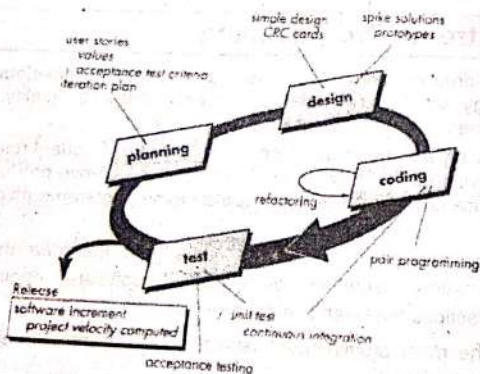


Fig: Extreme programming process

1. Planning

- The planning activity begins with the creation of a set of stories that describe required features and functionality for software to be built.

- Each story is written by the customer and is placed on an index card. The customer assigns a value to the story based on the overall business value of the feature of function.
- Members of the XP (Extreme Programming) team then assess each story and assign a cost – measured in development weeks – to it.
- If the story will require more than three development weeks, the customer is asked to split the story into smaller stories, and the assignment of value and cost occurs again.
- Customers and the XP team work together to decide how to group stories into the next release to be developed by the XP team.

2. Design

- XP design follows the KIS (Keep It Simple) principle. A simple design is always preferred over a more complex representation.
- XP encourages the use of CRC (Class Responsibility Collaborator) cards as an effective mechanism for thinking about the software in an object oriented context.
- CRC cards identify and organize the object oriented classes that are relevant to current software increment.
- The CRC cards are the only design work product produced as a part of XP process.
- If a difficult design is encountered as a part of the design of a story, XP recommends the immediate creation of that portion of the design called as 'spike solution'.
- XP encourages refactoring – a construction technique.

3. Coding

- Once the unit test has been created, the developer better able to focus on what must be implemented to pass the unit test.
- Once the code complete, it can be unit tested immediately, thereby providing instantaneous feedback to the developer.
- A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving and real time quality assurance.
- As pair programmers complete their work, the code they developed is integrated with the work of others.

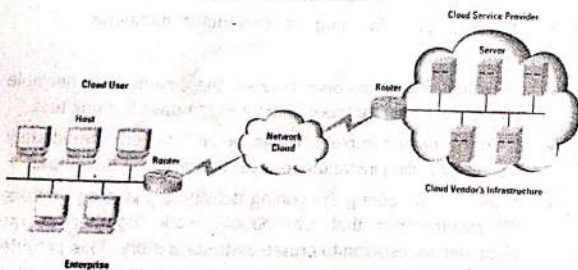
4. Testing

- The creation of unit test before coding is the key element of the XP approach.
- The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages regression testing strategy whenever code is modified.
- Individual unit tests are organized into a "Universal Testing Suit", integration and validation testing of the system can occur on daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things are going away.
- XP acceptance test, also called customer test, are specified by the customer and focus on the overall system feature and functionality that are visible and reviewable by the customer.

12.3 Cloud Computing

Cloud computing is a type of computing that relies on sharing computing resources rather than having local servers or personal devices to handle applications.

In cloud computing, the word cloud (also phrased as "the cloud") is used as a metaphor for "the Internet," so the phrase cloud computing means "a type of Internet-based computing," where different services - such as servers, storage and applications are delivered to an organization's computers and devices through the Internet.



Benefits of cloud computing

Cloud computing is a big shift from the traditional way businesses think about IT resources. There are 6 common reasons organizations are turning to cloud computing services:

154 • Software Engineering

1. **Cost:** Cloud computing eliminates the capital expense of buying hardware and software and setting up and running on-site datacenters. Hence it is comparatively cheaper.
2. **Speed:** Even vast amounts of computing resources can be provisioned in minutes, typically with just a few mouse clicks, giving businesses a lot of flexibility and taking the pressure off capacity planning.
3. **Global scale:** The benefits of cloud computing services include the ability to scale elastically. In cloud speak, that means delivering the right amount of IT resources. For example, more or less computing power, storage, bandwidth—right when its needed and from the right geographic location.
4. **Productivity:** Cloud computing removes the need for many of these tasks, so IT teams can spend time on achieving more important business goals. Hence is extremely productive.
5. **Performance:** The cloud computing offers several benefits over a single corporate datacenter, including reduced network latency for applications and greater economies of scale.
6. **Reliability:** Cloud computing makes data backup, disaster recovery and business continuity easier and less expensive, because data can be mirrored at multiple redundant sites on the cloud provider's network.

Types of cloud services

1. Infrastructure-as-a-service (IaaS)

- The most basic category of cloud computing services.
- With IaaS, we rent IT infrastructure, servers and virtual machines (VMs), storage, networks, operating systems from a cloud provider on a pay-as-you-go basis.

2. Platform as a service (PaaS)

- Platform-as-a-service (PaaS) refers to cloud computing services that supply an on-demand environment for developing, testing, delivering and managing software applications.
- PaaS is designed to make it easier for developers to quickly create web or mobile apps, without worrying about setting up or managing the underlying infrastructure of servers, storage, network and databases needed for development.

3. Software as a service (SaaS)

- Software-as-a-service (SaaS) is a method for delivering software applications over the Internet, on demand and typically on a subscription basis.

Software Engineering • 155

- With SaaS, cloud providers host and manage the software application and underlying infrastructure and handle any maintenance, like software upgrades and security patching.
- Users connect to the application over the Internet, usually with a web browser on their phone, tablet or PC.

Types of cloud deployments

1. Public cloud

- Public clouds are owned and operated by a third-party cloud service provider, which deliver their computing resources like servers and storage over the Internet.
- Microsoft Azure is an example of a public cloud.
- With a public cloud, all hardware, software and other supporting infrastructure is owned and managed by the cloud provider. We access these services and manage our account using a web browser.

2. Private cloud

- A private cloud refers to cloud computing resources used exclusively by a single business or organization.
- A private cloud can be physically located on the company's on-site datacenter.
- Some companies also pay third-party service providers to host their private cloud.
- A private cloud is one in which the services and infrastructure are maintained on a private network.

3. Hybrid cloud

- Hybrid cloud combines public and private clouds, bound together by technology that allows data and applications to be shared between them.
- By allowing data and applications to move between private and public clouds, hybrid cloud gives businesses greater flexibility and more deployment options.

12.4 Grid Computing

Grid computing is a processor architecture that combines computer resources from various domains to reach a main objective.

A grid is connected by parallel nodes that form a computer cluster, which runs on an operating system, Linux or free software. The cluster can vary in size from a small work station to several networks.

- In grid computing, the computers on the network can work on a task together, thus functioning as a supercomputer.

- Works on various tasks within a network, but it is also capable of working on specialized applications.
- Designed to solve problems that are too big for a supercomputer while maintaining the flexibility to process numerous smaller problems.
- Computing grids deliver a multiuser infrastructure that accommodates the discontinuous demands of large information processing.
- The technology is applied to a wide range of applications, such as mathematical, scientific or educational tasks through several computing resources.
- Often used in structural analysis, Web services such as ATM banking, back-office infrastructures, and scientific or marketing research.

12.5 Enterprise Mobility

Enterprise mobility is the trend toward a shift in work habits, with more employees working out of the office and using mobile devices and cloud services to perform business tasks.

The term refers not only to mobile workers and mobile devices, but also to the mobility of corporate data.

- An employee may upload a corporate presentation from his or her desktop PC to a cloud storage service then access it from a personal iPad to show at a client site, for example.
- Enterprise mobility can improve employee productivity, but it also creates security risks.
- Enterprise mobility management products, such as data loss prevention technologies, are available to help IT departments address these risks.
- A strong acceptable use policy for employees can also contribute to a successful enterprise mobility strategy.