

Chapter 1

Software Process and Requirements

[Prepared by Amir GC and shishir Rai]

1.1 Software crisis

Software crisis is the production of failed and challenged software as a result of

- Introduction of powerful computer hardware
- When larger and complex software were ordered
- Software built is over budget, late delivery unreliable, difficult to maintain properly.

It became clear that individual approaches to program development did not scale up to large and complex system. This term was purposed in 1968.

1.2 Software characteristics

To gain an understanding of software, it is important to examine the characteristics of software. Software is a logical rather than a system element.

Main characteristics of software are:

- 2 Software is developed and engineered; it is not manufactures in the classical sense.
- 3 Software doesn't wear out: i.e. it is maintainable with the introduction of new hardware.
- 4 A software component should be designed and implemented so that it can be reused in many different programs.
- 5 Software should have all required functionality and performance for user.

1.3 Software quality attributes

A. Runtime System Qualities

Runtime System Qualities can be measured as the system executes.

Functionality: the ability of the system to do the work for which it was intended.

Performance: the response time, utilization, and throughput behavior of the system. Not to be confused with human performance or system delivery time.

Security: a measure of system's ability to resist unauthorized attempts at usage or behavior modification, while still providing service to legitimate users.

Availability (Reliability quality attributes falls under this category): the measure of time that the system is up and running correctly; the length of time between failures and the length of time needed to resume operation after a failure.

Usability: the ease of use and of training the end users of the system. Sub qualities: learnability, efficiency, affect, helpfulness, control.

Interoperability: the ability of two or more systems to cooperate at runtime

B. Non-Runtime System Qualities

Non-Runtime System Qualities cannot be measured as the system executes.

Modifiability: the ease with which a software system can accommodate changes to its software

Portability: the ability of a system to run under different computing environments. The environment types can be either hardware or software, but is usually a combination of the two.

Reusability: the degree to which existing applications can be reused in new applications.

Integrability: the ability to make the separately developed components of

the system work correctly together.

Testability: the ease with which software can be made to demonstrate its faults

1.4 Software process models

Software process model is a simplified representation of a software process. Each process model represent a process from a particular perspective, so only provide only partial information about that process.

Main process model of software are:

1. The water fall model
2. Incremental development
3. Reuse-oriented software engineering
4. Spiral model

1.4.1 The water fall model

These takes the fundamental process activities of specification, development, validation, development, ad evolution and represent them as separate process phases such as requirements specification, software design, implementation, testing, and so on.

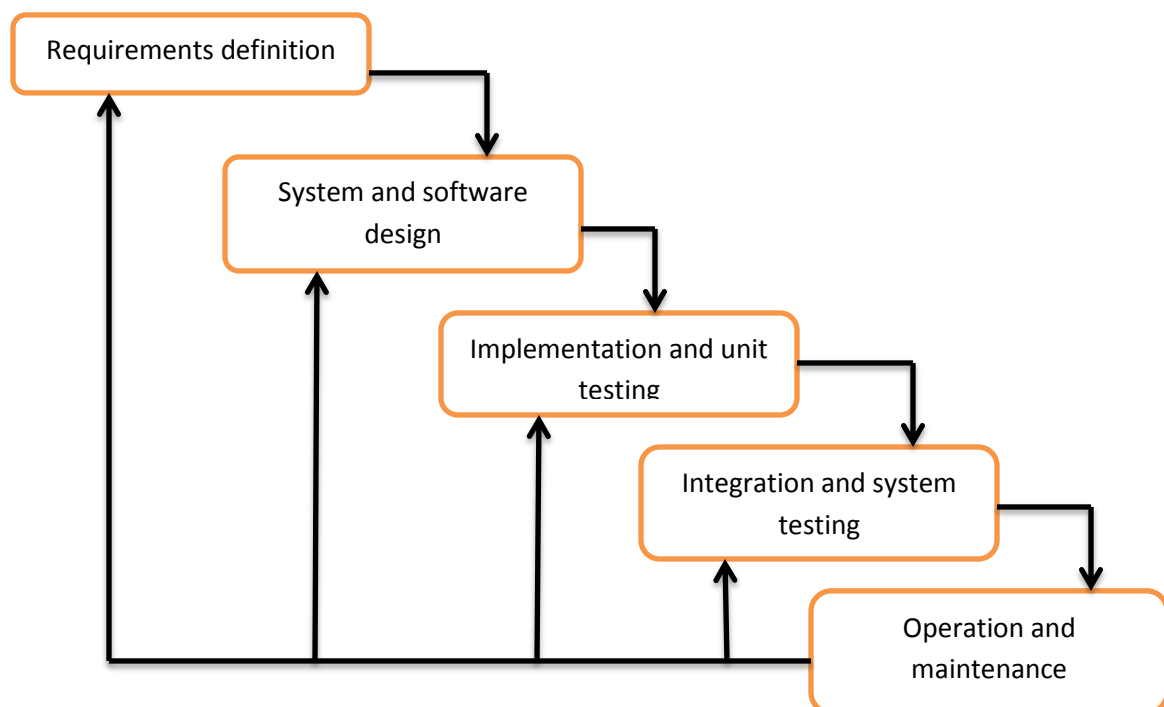


Fig 1.1 the water fall model

Because of the cascade from one phase to another is known as the water fall model. We should plan and schedule all of the process activities before starting works on them. Principle stages of the water fall model are listed below:-

- i. Requirements analysis and definition:
 - The system's services constraints and goals are established by consulting with the user.
 - They are defined in detail and serve as a system specification.
- ii. System and software design:
 - This process gives the requirements to either hardware or software systems by establishing an overall system architecture.
 - Involves identifying and describing the fundamental software system abstractions and their relationships.
- iii. Implementing and unit testing:
 - At this stage software design is realized as a set of programs or program units.
 - Unit testing involves verifying that each unit meets its specification.
- iv. Integration and system testing:
 - Individual program units or programs are integrated and tested as a complete system to ensure that software requirements have been made.
- v. Operation and maintenance:
 - Longest phase
 - System is put into practical use.
 - Maintenance involves correcting errors which were not discovered earlier.

Advantages

- ✓ Reflect systematic way of software process
- ✓ Useful for larger system engineering project

Disadvantages

- ✓ Inflexible partitioning of the project into distinct stages.
- ✓ Difficult to respond to changing customs requirements.

1.4.2 Incremental development model

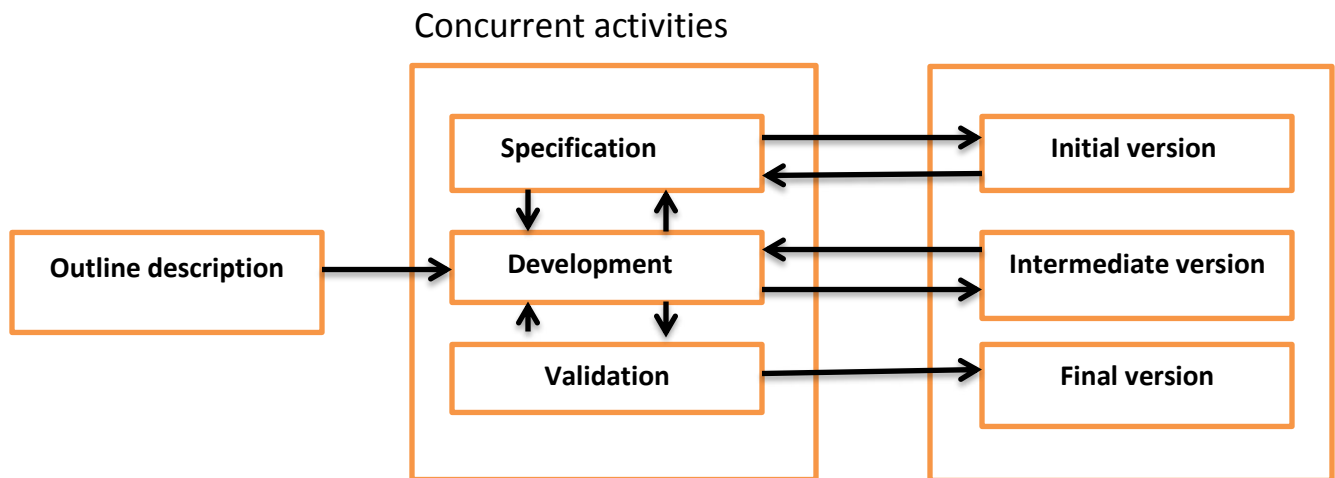


Fig 1.2 Incremental development

- ❖ Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving through several versions until a complete system has been developed.
- ❖ Interleaves the activity of specification, development & validation
- ❖ Developed as a series of version (increments) with each version adding functionalities to the previous one.

Advantages

- ✓ The cost of accommodating changing customer requirements is reduced.
- ✓ It is easier to get customer feedback on the development work that has been done.

- ✓ More rapid delivery and development of useful software to the customer is possible, even if all of the functionality has been included.

Disadvantages

- ✓ Hard to identify common facilities that are needed by all increments as requirement are not defined in detail at early stage.
- ✓ Difficult when replacement system is being developed as increments do not have full functionalities
- ✓ Conflicts arises with the pro current model of organization where complete system specification is part of the system development

1.4.3 Reuse oriented software engineering model

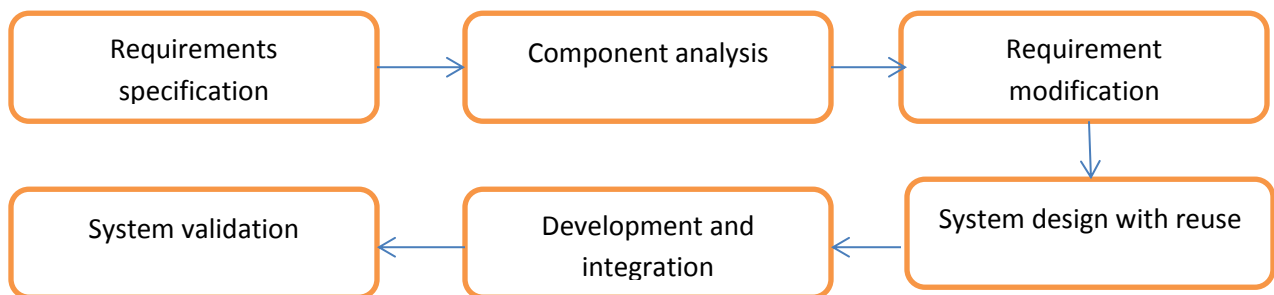


Fig 1.3 Reuse-oriented software engineering

The principle stages are

- I. Requirement specification:
 - same as of water fall model
- II. component analysis:
 - A search is made for components to implement that requirement specification.
 - Usually there is no exact match
 - Components which are discovered may only provide some of functionalities required.
- III. Requirement modification:

- The requirements are analyzed using the information about the components that have been discovered.
 - They are the modified to reflect the available components
- IV. System design with reuse:
- Framework of the system is designed or existing frame work is reused.
 - Some new software are designed if reusable components are not available.
- V. Development and integration
- Software that cannot be externally procured is developed
 - And the components and COTS (commercial-of-the-shelf) are integrated to create the new system.

There are three types of software component that may be reused:

- a. Web services that are developed according to services and which are available for remote invocation
- b. Collection of object that are developed as a package to be integrated with a component frame work such as .NET or J2EE.
- c. Stand-alone software system that are configured for use in a particular environment.

Advantages

- ✓ Reduce the amount of software to be developed
- ✓ Reduces cost and risk.
- ✓ Fast delivery of the software.

Disadvantages

- ✓ May lead to a system that does not meet the real necessary of the user requirement.
- ✓ Some control over the system evolution is lost as new as new versions of the reusable components are not under the control of the organization using them.

1.4.4 Spiral model

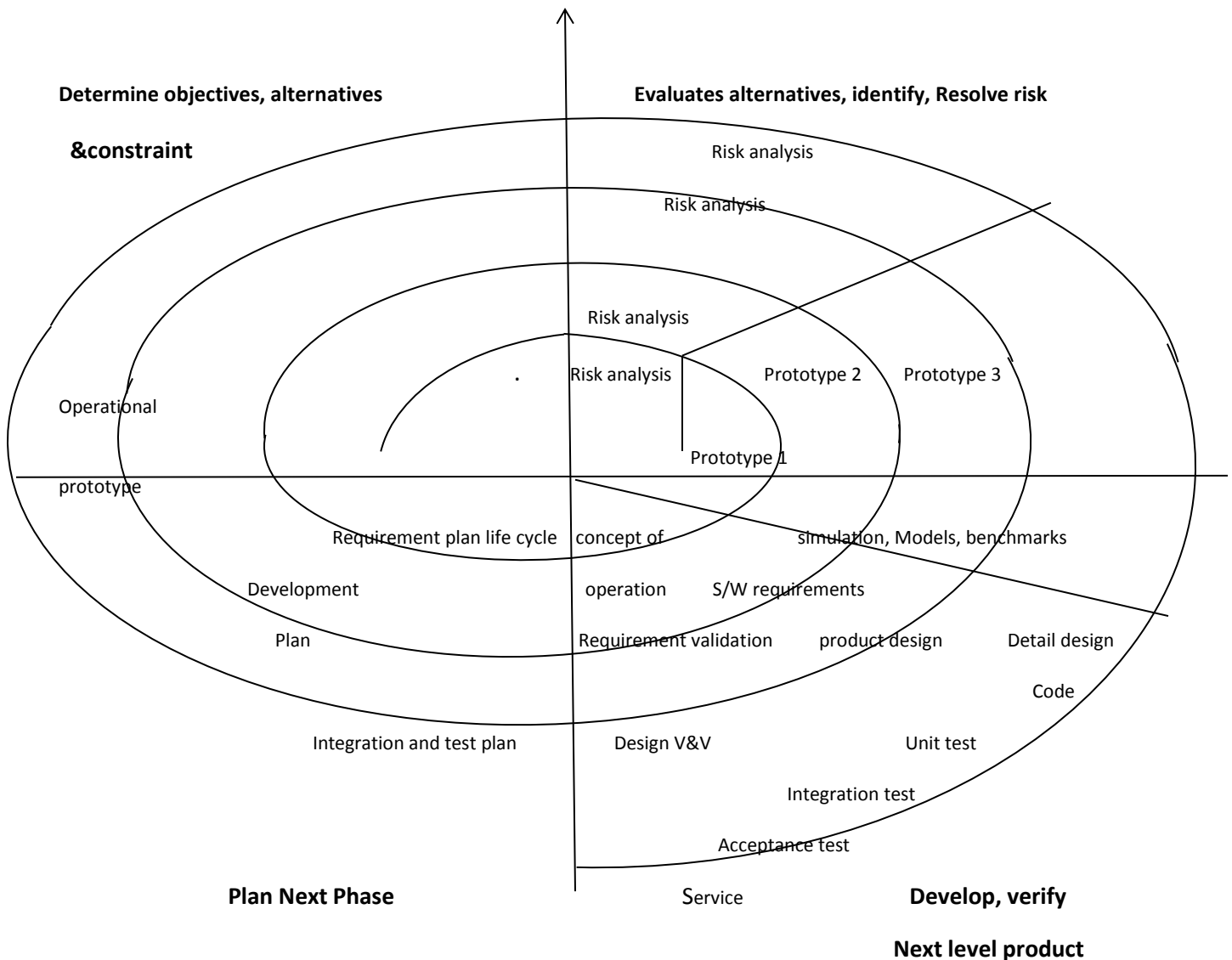


Fig 1.4 Spiral Model of Software process

In this model process is represented as spiral rather than a sequence of activities with some backtracking from one activity to another. Main principal stage of spiral model are as follow:

1. Objective setting:
 - Specific object for that phase of the project are defined

- Constraints and product are identified and detail management plot is drawn up.
 - Project risks are identified.
 - Strategies are planned to minimize risk
2. Risk assessment and reduction:
- For each of identified risks, a detailed analysis is carried out and step are taken to reduce them.
3. Development and validation
- After a risk evaluation, a development model is chosen.
 - For eg . if user interface risk are dominant then prototyping may be throw away
 - If safety risk are main dominant then development based on formal transformation is chosen
 - If risk is about sub system integration ,then waterfall model is best to use
4. Planning:
- Project is reviewed and decision is made whether to continue if it is decided then further plan for next phase are drawn up.

Advantages

- ✓ Explicit recognition of the risk.
- ✓ Flexibility to manage requirement and control changes
- ✓ Features for large business and complicated project
- ✓ Compromises both water fall model and prototype model

Disadvantages

- ✓ Not suitable for smaller project
- ✓ Not suitable for changes that happen frequently



1.5 Process iteration

The Waterfall model has dominated software development for many years, but iteration of processes is catching in. There are now a number of well-established iterative development process models that can be classified according to the levels where iteration is applied. Iteration can improve validation and verification

by allowing earlier quality feedback. Moreover, there seems to be a secret marriage between teamwork and iteration. Altogether, from a SPI (software process Iteration) point of view, changing to an iterative development process model could very well raise your professional standards in software development.

- ❖ Parts of the process are repeated as system requirement evolve.
- ❖ System design & implementation work must be reworked to implement the change requirement.
- ❖ It is alternative approach to S/w development.
- ❖ Makes the system that can do all to do little more.
- ❖ Minimize the risk of building wrong product .e.g. building a table instead of chair.
- ❖ Several development process use iteration in high level or level or both.

Development process that support process iteration:

-  Incremental development process
-  Spiral development
- During iteration process turns of iteration should be marked strictly.
- Effective iteration means optimizing the number of turns which requires the right stop criteria.

1.6 Process Activities

A software process is a set of related activities that leads to the production of a software product. This may involve the development of software from a scratch in a standard programming language like java or C. There are many different S/W processes but all must include four activities they are:

- a. Software specification
 - b. Software design and implementation
 - c. Software validation
 - d. Software evolution
- A. software specification:
- Software specification or requirements engineering process of the understanding & defining what services are required from the system and identifying the constraints on the system development.

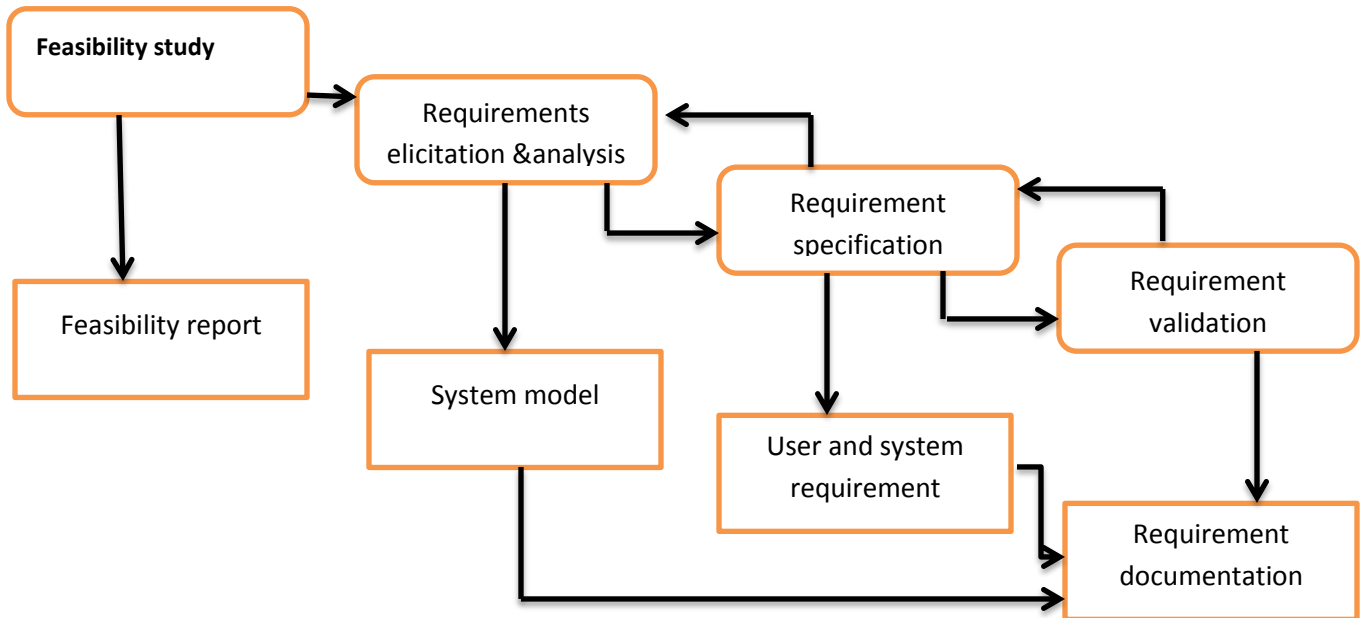


Fig 1.5 The requirement engineering process

The four main activities in engineering requirement process:

❖ Feasibility study:

- An estimate is made whether the user need may be satisfied using current software and hardware technologies.
- The study also considers whether the purposed system is cost-effective from business point of view.
- It should be quick and cheap.
- Should provide information to decide whether or not to go ahead with more detail analysis.

❖ Requirement elicitation and analysis:

- Derivation of the system requirements by observing the existing system, discussion with potential users & procurers, task analysis.
- Involve development of one or more system models & prototype.
- Helps us to understand the system to be specified.

❖ Requirement specification

- Activity of translating the information gathered during the analysis activity into document that defines the set of requirement.
- Two types of requirement are: i) user requirement b) system requirement

❖ Requirement validation

- Checks the realism, consistency, completeness of requirements.
- Errors in the requirements are discovered & modified to correct these problems.

B. Software design and implementation

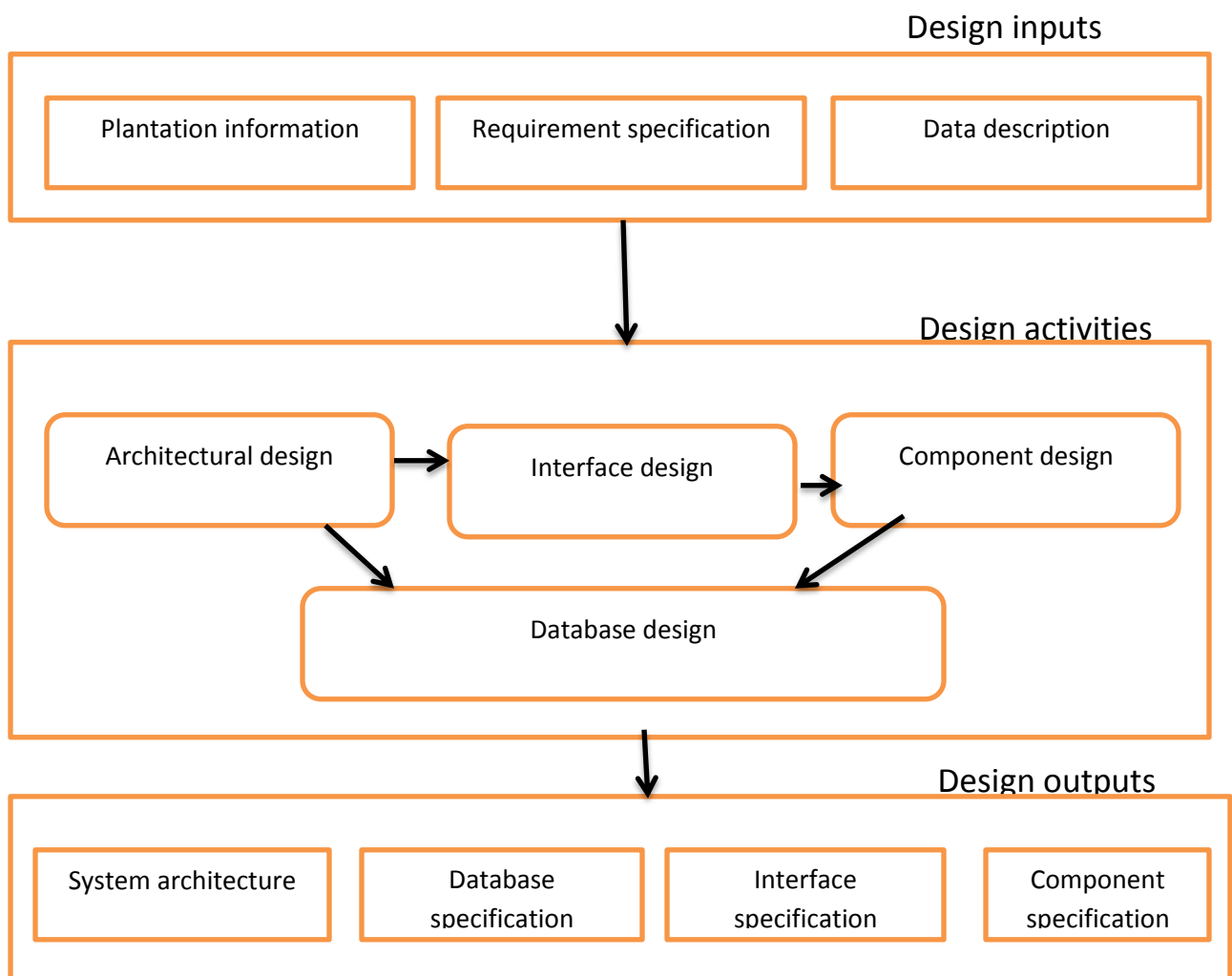


Fig 1.6 software design process

Four main activities that may be part of design process are

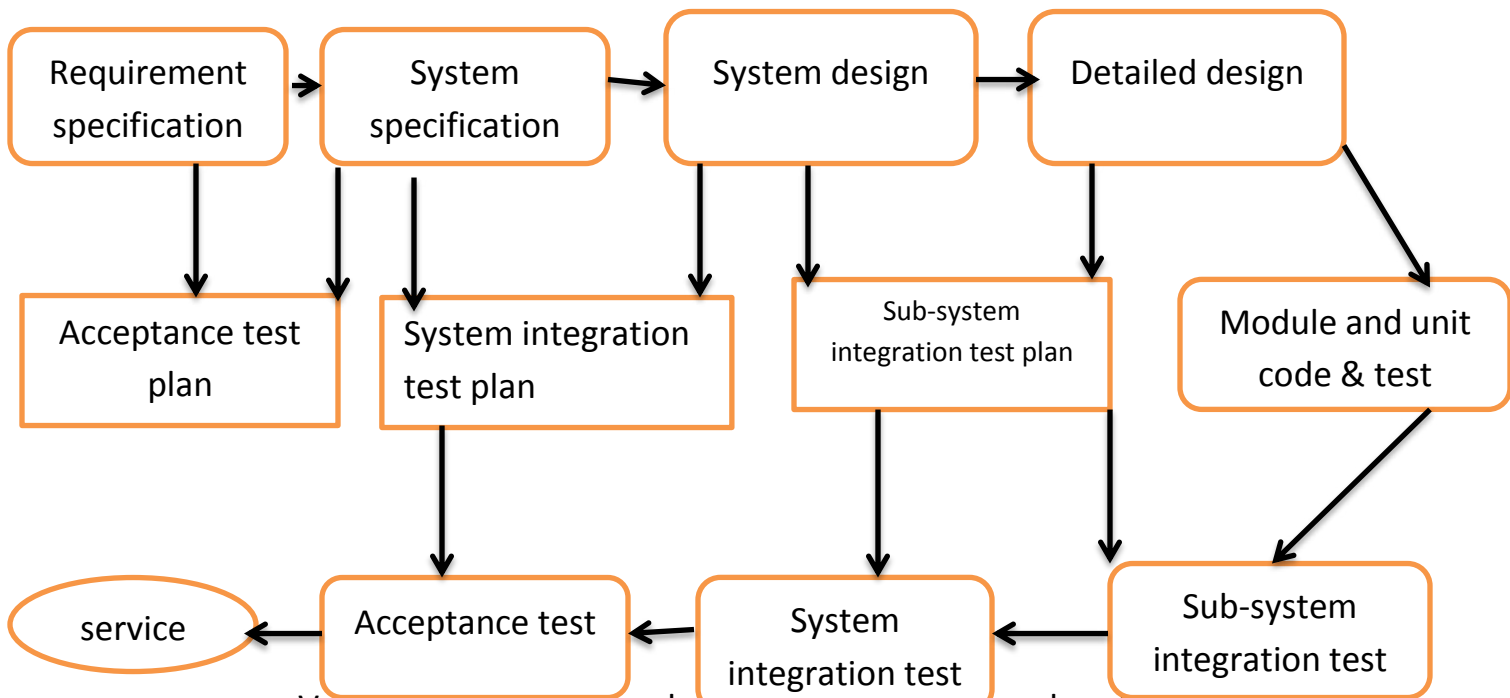
❖ Architectural design

- Identification of the overall system.
- Identify the relationship between principal component
- ❖ Interface design
 - Define the interfaces between system components.
- ❖ Component design
 - Here we take each system component and design how it will operate.
 - It may be the list of changes to be made to a reusable component or a detailed design model.
- ❖ Database design
 - Design the system data structure and how they are to be represented in a database.

C. System validation

Software validation & verification is intended to show that both that a System meets both specification and expectation of system customer.

Figure below is of testing phase of plan-driven software process



- Validation may also involves checking process such as inspection and review at each step of software process. Main process in software testing and validation are as follow

❖ Development testing

- Component making up the system are tested by the people developing the system.
- Each component is tested separately.
- Component may be simple entities such as function, object and class.

❖ System testing

- System component are integrated to create a complete system.
- Concerned with finding error that happens due to components and component interface problem.

❖ Acceptance testing

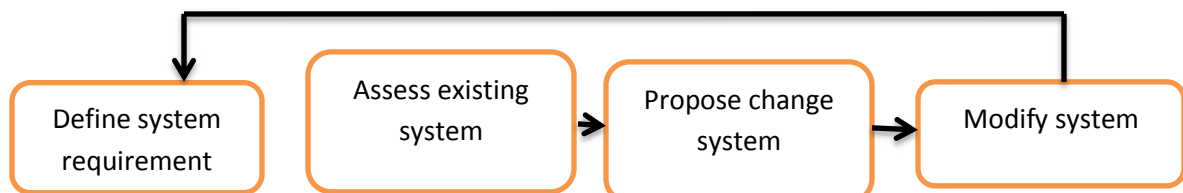
- Is the final stage in testing before the system is accepted for operational use?
- The system is tested with data supplied by a customer.
- May reveal errors and requirements problems.

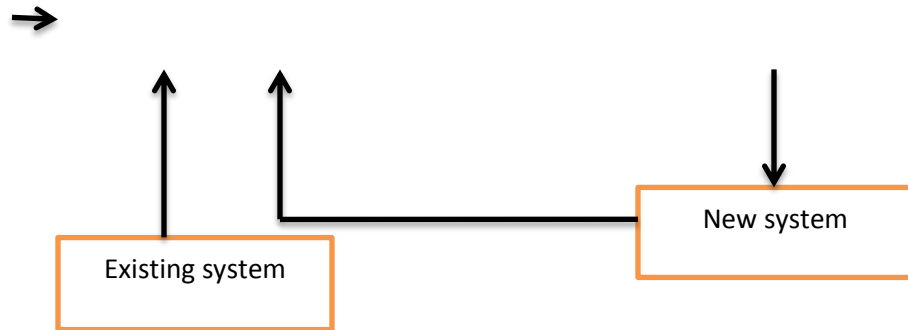
🚦 Alpha testing: some time acceptance testing is known as alpha testing. Custom system is developed for single client. It continues until the client and developer agreed that the system is acceptable.

🚦 Beta testing: involves delivering the system to multiple clients. They report the problem to the developer. After this developer modify it and release the system.

D. **Software evolution**

It is very expensive to make changes to hardware design but changes can be made to software at any time during or after the development in cheaper in correspondence to hardware change. Software engineering is a evolutionary process where software is continually changed over its life time with response to changing requirements and user needs.





1.7 Computer-aided software engineering(CASE)

- ❖ CASE tools are programs that are used to support software engineering process. These tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools, etc.
- ❖ CASE tools provide process support by automating some process activities and by providing information about the software that is being developed.
- ❖ Assist in development and maintenance of software
- ❖ Developed in 1970's to speed up the s/w build up process
- ❖ Allows rapid development of software to cope with the increasing speed of market demand.

Classification of CASE tools

- a. Business system planning
 - Information engineering tools
 - Process modeling and management tools
- b. Project management
 - Project planning tools
 - Risk analysis tools
 - Project management tools
 - Requirement tracing tools
- c. Programming tools
 - Integrating and testing tools
 - Client /server tools
- d. Maintenance tools
 - Requirement engineering tools

Specific examples:

- ✓ with class-object oriented design & code generation
- ✓ oracle designer/200-integrated CASE environment

1.8 Functional and non-functional requirements

Functional requirements

Functional requirements specify the product capabilities, or things that a product must do for its users. The functional requirements specify what the product must do. They relate to the actions that the product must carry out in order to satisfy the fundamental reasons for its existence. Functional requirement must fully describe the actions that the intended product can perform. They describe the relationship between the input and output of the system.

Non-functional requirements

Non-functional requirements define system properties such as reliability, performance, security, response time and storage requirements and constraints like Input output device capability, system representations.

Non-functional requirements are more critical than functional requirements. A system user can usually find ways to work around a system function that doesn't really meet their needs but if the non-functional requirements are not met, then the system will be useless.

They describe various quality factors, or attributes, which affect the functionality's effectiveness.

Functional	Nonfunctional
Product features	Product properties
Describe the work that is done	Describe the character of the work
Describe the actions with which the work is concerned	Describe the experience of the user while doing the work
Characterized by verbs	Characterized by adjectives

1.9 User requirements

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

User requirements should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.

User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

1.10 System requirements

They are more precise than user requirements.

They are more detailed descriptions of the software system's functions services and operational constraints.

The system requirements document should define exactly what is to be implemented.

They may be incorporated into the system contract between the system buyer and the software developers so as to define how the system should work.

They may be defined or illustrated using system models.

1.11 Interface specification

Interface specification describes the behavior of some software unit such as function or class.

Interface specification is an important part of any design process which describes the interfaces between the components in the design.

It is required so that objects and sub functions can be designed in parallel.

It is used to document the design of future software components and the correct usage of an existing component.

1.12 The software requirements documents

The software requirements document sometimes called the software requirements specification or SRS is an official statement of what the system developers should implement.

It should include both the user requirements for a system and a detailed specification of the system requirements.

The requirements document states 'what the software will do'. It does not state 'how the software will do it'.

The main purpose of a requirements document is to serve as an agreement between the developers and customers on what the application will do.

The characteristics of a good software requirements document are

1. **Complete:** A complete requirements specification must precisely define all the real world situations that will be encountered and the capability's responses to them. It must not include situations that will not be encountered or unnecessary capability features.
2. **Consistent:** System functions and performance level must be compatible and the required quality features (reliability, safety, security, etc.) must not contradict the utility of the system. For example, the only aircraft that is totally safe is one that cannot be started, contains no fuel or other liquids, and is securely tied down.
3. **Correct:** The specification must define the desired capability's real world operational environment, its interface to that environment and its interaction with that environment.
4. **Modifiable:** Related concerns must be grouped together and unrelated concerns must be separated. Requirements document must have a logical structure to be modifiable.
5. **Ranked:** Ranking specification statements according to stability and/or importance is established in the requirements document's organization and structure. The larger and more complex the problem addressed by the requirements specification, the more difficult the task is to design a document that aids rather than inhibits understanding.
6. **Testable:** A requirement specification must be stated in such a manner that one can test it against pass/fail or quantitative assessment criteria, all derived from the specification itself

and/or referenced information. Requiring that a system must be “easy” to use is subjective and therefore is not testable.

7. **Traceable:** Each requirement stated within the SRS document must be uniquely identified to achieve traceability. Uniqueness is facilitated by the use of a consistent and logical scheme for assigning identification to each specification statement within the requirements document.

8. **Unambiguous:** A statement of a requirement is unambiguous if it can only be interpreted one way. This perhaps, is the most difficult attribute to achieve using natural language. The use of weak phrases or poor sentence structure will open the specification statement to misunderstandings.

9. **Valid:** To validate a requirements specification all the project participants, managers, engineers and customer representatives, must be able to understand, analyze and accept or approve it. This is the primary reason that most specifications are expressed in natural language.

10. **Verifiable:** In order to be verifiable, requirement specifications at one level of abstraction must be consistent with those at another level of abstraction. Most, if not all, of these attributes are subjective and a conclusive assessment of the quality of a requirements specification requires review and analysis by technical and operational experts in the domain addressed by the requirements.

1.13 Feasibility study

A feasibility study is a short, focused study that is done earlier in requirement engineering process and is carried out to select the best system that meets performance requirements.

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

The feasibility study activity involves the analysis of the problem and collection of all relevant information relating to the product such as the different data items which would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system as well as various constraints on the behavior of the system.

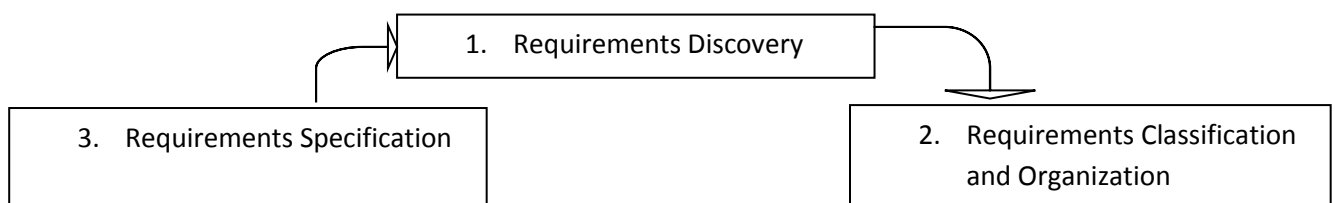
It should be relatively cheap and quick.

1.14 Requirements elicitation and analysis

After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis.

It is an iterative process that can be represented as a spiral of activities – requirements discovery, classification and organization, negotiation with prioritization and requirements specification.

In this process the software engineers work with the customers and system end users to find out about the application domain, what services the system should provide, the required performance of the system hardware constraints.



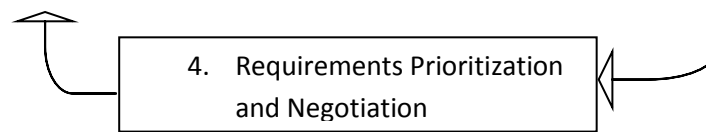


Fig. The requirements elicitation and analysis process

1. Requirements discovery

This is the process of interacting with stakeholders of the system to discover their requirements. A system stakeholder is anyone who should have some direct or indirect influence on the system requirements.

2. Requirements classification and organization

The discovered unstructured collection of requirements are then classified and structured properly. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system.

3. Requirements prioritization and negotiation

Inevitably, when multiple stakeholders are involved, requirements will conflict. So the prioritization of the requirements is necessary. Stakeholders have to meet and negotiate to resolve differences and agree on compromising requirements.

4. Requirements specification

Finally the requirements are documented and written in a requirements document.

Requirements elicitation is a **difficult process** for several reasons:

- a. Stakeholders often don't know what they want from a computer system in most general terms, they may make unrealistic demands because they don't know what is and isn't feasible.
- b. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain may not understand these requirements.
- c. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and find the commonalities and conflict.
- d. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
- e. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not previously consulted.

1.15 Requirements validation and management

Requirements validation is the process of checking that requirements actually define the system that the customer really wants.

It overlaps with analysis as it is concerned with finding problems with the requirements.

It is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The process of requirements validation includes different checks.

1. **Validity and verifiability:** Verification and validation is not the same thing.

Validation: Are we building the right product?

Verification: Are we building the product right?

Validation: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

Verification: The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. For verifiable software, we must be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Validation ensures that the product actually meets the user's needs, and that the specifications were correct in the first place, while verification is ensuring that the product has been built according to the requirements and design specifications.

Validation ensures that "you built the right thing". Verification ensures that "you built it right".

2. **Consistency:** Requirements in the document should not conflict. There should not be contradictory constraints or different descriptions of same system function.
3. **Completeness:** The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. **Realism:** The requirements should be checked to ensure that they can actually be implemented under constraints such as time and money.

The main problem with requirement validation is that the requirements change continuously during requirements elicitation.

Requirements validation techniques:

Requirement reviews: The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

Prototyping: An executable model of the system in question is used to check the validity.

Test-case generation: Requirements should be testable. If a test for a requirement is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered.

Requirement management is the process of managing changing requirements during the requirement engineering process by documenting, analyzing, tracing, and agreeing on requirements and then controlling change and communicating to relevant stakeholders.

It is the process of understanding and controlling changes to the system requirements.

Requirement management planning: Planning is important during requirements management.

- a. Requirements identification: Each requirement should be uniquely identified.
- b. A change management process: Process followed when analyzing a requirement change and the impact and cost of changes.

- c. Traceability policies: These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
- d. Tool support: Requirements management involves the processing of large amounts of information about the requirement. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

A good requirement engineering process consists of 4 main phases.

1. Feasibility study
2. Requirements elicitation and analysis
3. Requirements specification
4. Requirements validation

Past IOE question

- ❖ What is software crisis? Explain with the help of an example..
- ❖ Describe spiral model for software development. What are its advantages and disadvantages?
- ❖ Explain requirement management process with necessary illustration.
- ❖ What are the advantages and disadvantages of the water fall process? List out various model of the software development. Explain the limitation of waterfall model in detail.
- ❖ Explain software requirement specification (SRS).what are the good characteristics of good SRS document?

Chapter 2 System Modeling

-sushant chalise

System modeling is the process of developing abstract model of a system, with each model presenting a different view or perspective of the system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost based on notations in Unified Modeling Language (UML).

Models are used during the requirement engineering process to help to derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and structure and operation. The most important aspect of system model is that it leaves out detail. A model is an abstraction of system being studied rather than an alternative representation of that system. You may develop different models to represent the system from different perspective. For example:

1. An external perspective, where you model the context or environment of the system.
2. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
3. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

The UML has many diagrams types and so supports the creation of many different types of system model. However, a survey in 2007 showed that users of the UML thought that five diagram types could represent the essentials of a system:

1. Activity diagrams, which show the activities involved in process or in data processing.
2. Use case diagrams, which show the interactions between a system and its environment.
3. Sequence diagrams, which show interactions between actors and the system and between system components.
4. Class diagrams, which show the object classes in the system and the associations between these classes.
5. State diagrams, which show how the system reacts to internal and external events.

2.1 CONTEXT MODEL

- Context models are used to illustrate the boundaries of a system. Social and organizational concerns may affect the decision on where to position system boundaries.
- At the early stage in the requirement elicitation and analysis process one should decide on the boundaries of the system.
- It involves working with the stake holders to distinguish what the system.
- One makes this decision early in the process to limit the system cost and time needed for analysis.
- In some cases the boundary between a system and environment is relatively clear
eg: automated system repacking existing manual or computerized system, the error of the new system is usually the same as the existing system

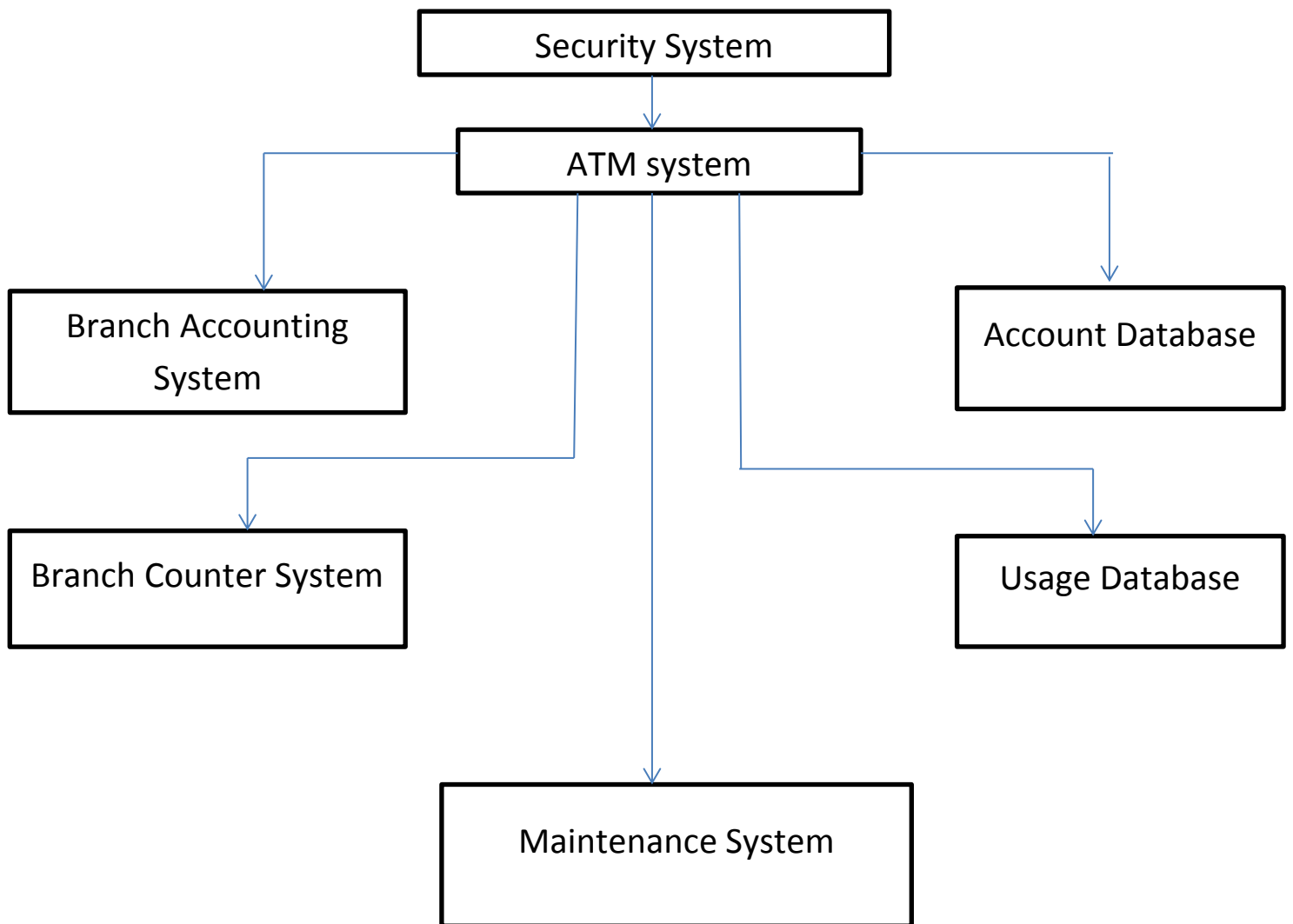


Fig: The Context of ATM system

2.2 Behavioral Model

- Behavioral models are used to describe the overall behavior system
- Here two types of behavioral models are discussed
 - data flow models— It models the data processing in the system
 - state machine system—It models how the system reacts to model
- These models may be used separately or together or together, depending on the type of system that is being developed.
- Most business system are primarily driven by data.
- They are controlled by the data inputs to the system with relatively little external event processing.

Data-flow model

- It is the system model that show a functional perspective where each transformation represents a single function or process.
- DFM are used to show how data flow through a sequence of processing steps.
Eg: processing could be to filter duplicate records in a customer database
- The data is transferred at each step before moving on to the next step.
- The processing steps or transformation represents software process or functions when data flow diagrams are used to document a software design.

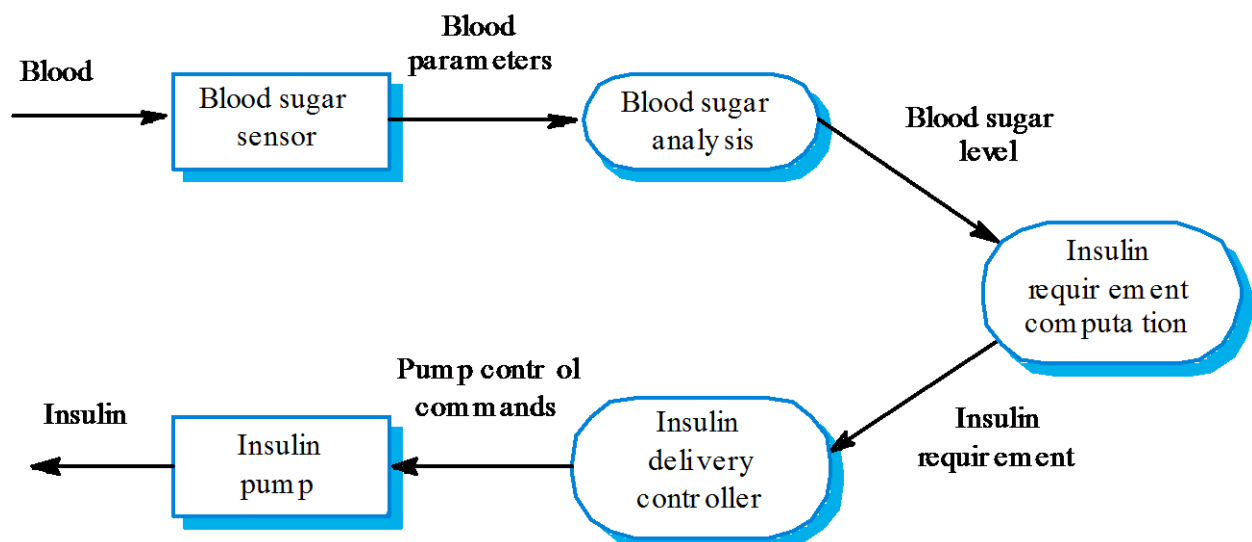


Fig: Insulin Pump DFD

State machine models

- These model 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

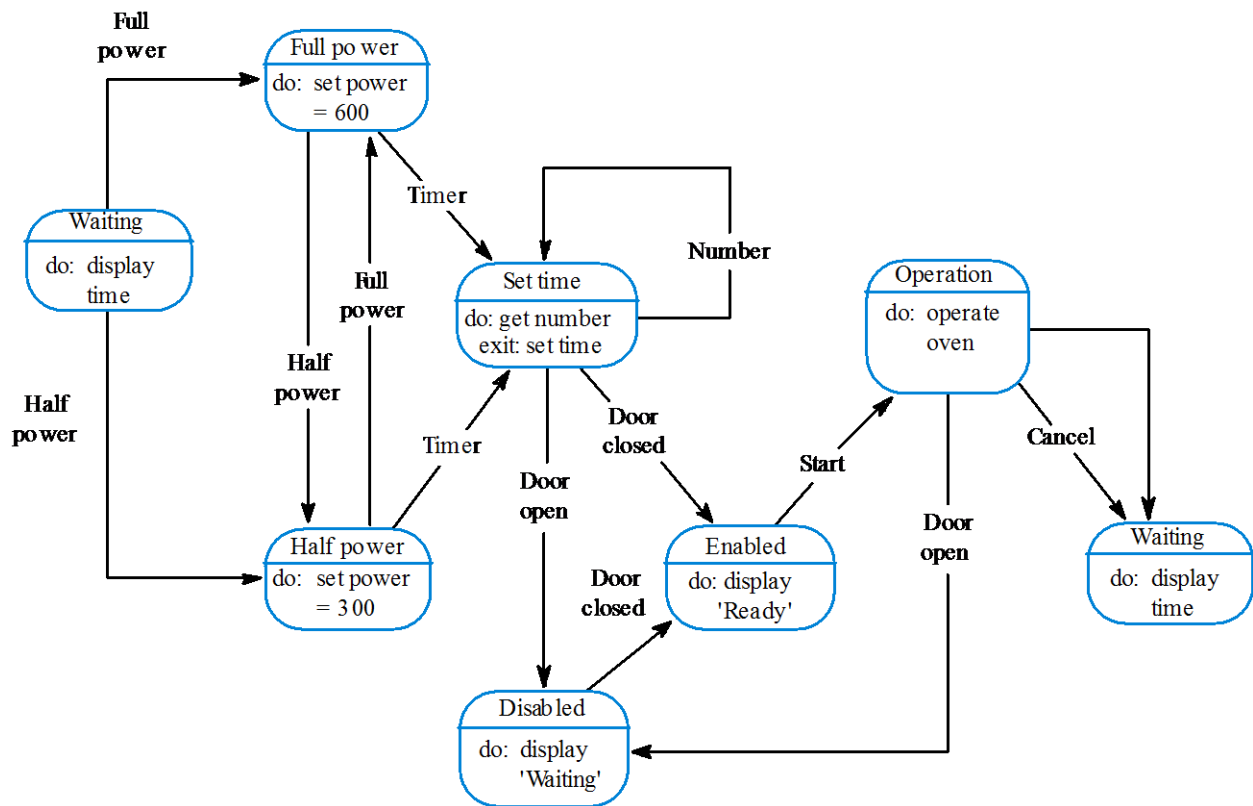


Fig: Microwave oven model

Microwave oven stimuli

Stimulus	description
Half Power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer button
Number	The user has pressed a numeric key
Door Open	The oven door switch is not closed
Door Closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

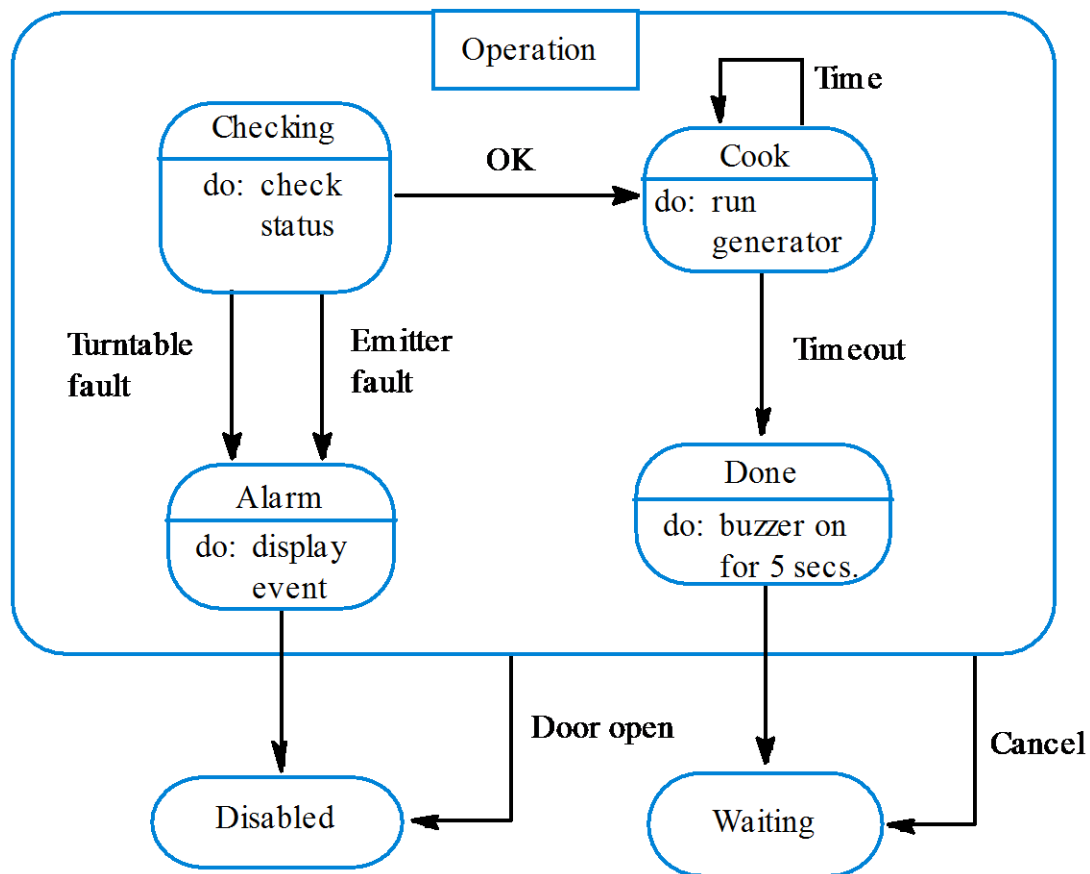


Fig: Microwave oven operation

2.3 Data Flow and Object Model

2.3.1 Data model

- It used to describe the logical structure of data processed by the system.
- It is an entity-relation-attribute model which sets out the entities in the system, the relationships between these entities and the entity attributes
- This model is widely used in database design. Can readily be implemented using relational databases.
- It has no specific notation provided in the UML but objects and associations can be used.

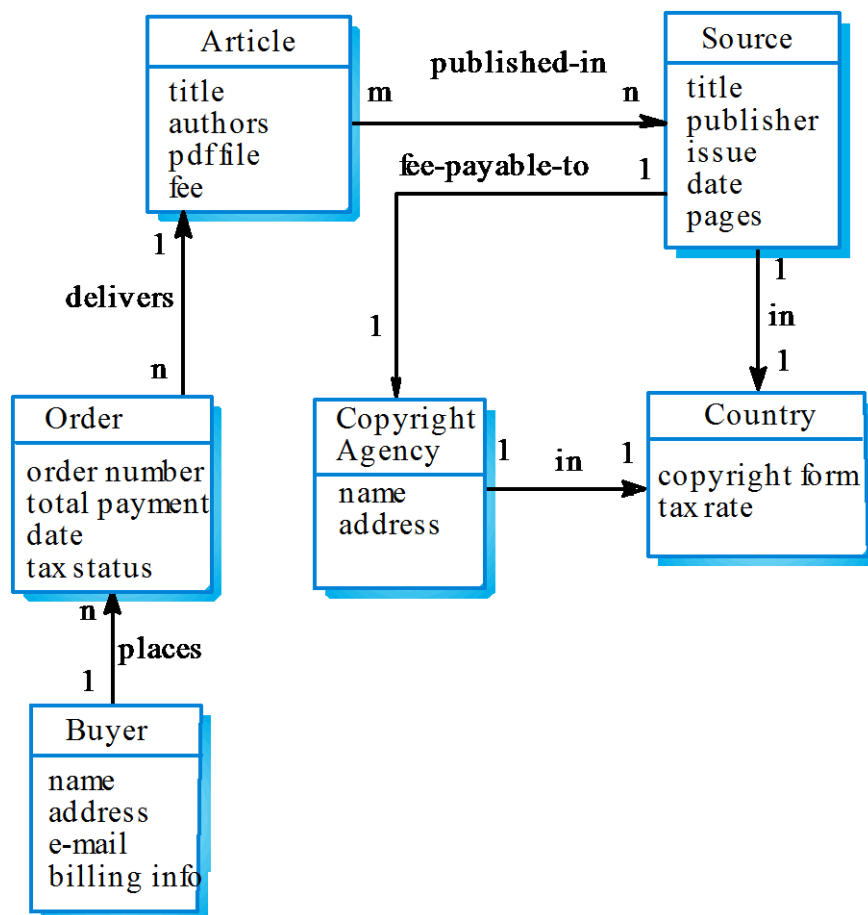


Fig: Library Symantec model

2.3.2 Object Model

- Object models 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 (operations) provided by each object.
- It is a natural way of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems
- Various object models may be produced
 - Inheritance models;
 - Aggregation models;
 - Interaction models.

Inheritance model

- Organise 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. These may then be specialised as necessary.
- Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

Object Model and the UML

- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
 - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
 - Relationships between object classes (known as associations) are shown as lines linking objects;
- Inheritance is referred to as generalisation and is shown 'upwards' rather than 'downwards' in a hierarchy

Object Aggregation

- 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.

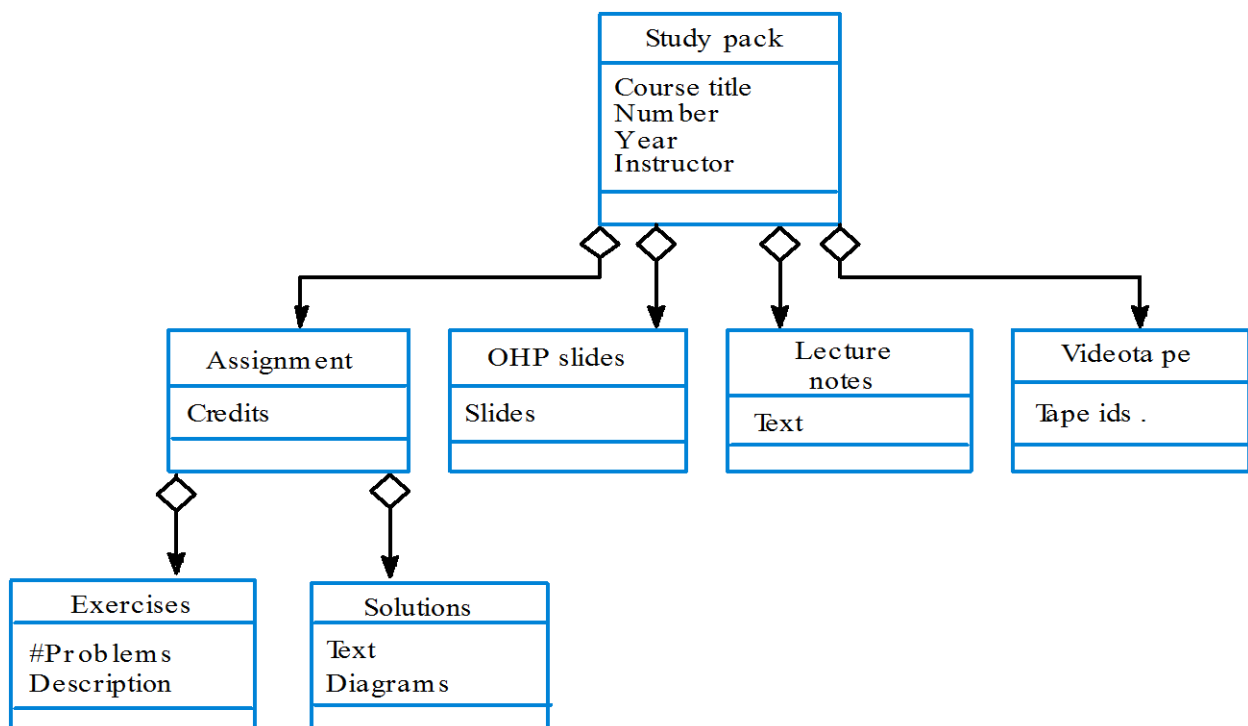


Fig:Object Aggregation

2.4 Structured Methods

- Structured methods incorporate system modelling 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 modelling as part of a structured method.
- This model of software display the organization of a system in terms of components that make up that system and their relationship.
- They may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing.

Weakness of the method:

- They do not model non-functional system requirements.
- They do not usually include information about whether a method is appropriate for a given problem.
- They may produce too much documentation
- The system models are sometimes too detailed and difficult for users to understand.

Chapter -3 Architectural design:

-Sunil Lama

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. In the model of the software development process, architectural design is the first stage in the software design process. It is the critical link between design and requirement engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

3.1 Architectural design decisions:

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there generic application architecture that can act as a template for the system that is being designed?
2. What strategy will be used to control the operation of the components in the system?
3. What architectural organization is best for delivering the non-functional requirement of the system?
4. How will the architectural design be evaluated?
5. How should the architecture of the system be documented?

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For embedded systems and systems designed for personal computers, there is usually only a single processor and you will not have to design a distributed architecture for the system. However, most large systems are now distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a key decision that affects the performance and reliability of the system.

Because of the close relationship between nonfunctional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the nonfunctional system requirements:

1. Performance: If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network.
2. Security: If security is critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.
3. Safety: If safety is the critical requirement, the architecture should be designed so that safety related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.
4. Availability: If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system.
5. Maintainability: If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

3.2 System organization:

- Reflects the basic strategy that is used to structure a system.
- Three organizational styles are widely used:
 1. A shared data repository style,
 2. A shared services and servers style,
 3. An abstract machine or layered style.

The repository model

- Sub-systems must exchange data. This may be done in two ways:
 1. Shared data is held in a central database or repository and may be accessed by all sub-systems,
 2. Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

Repository model characteristics:

- Advantages:
 1. Efficient way to share large amount of data.
 2. Sub-systems need not be concerned with how data is produced centralized management e.g. backup, security etc.
 3. Sharing model is published as the repository schema.
- Disadvantages
 1. Sub-systems must agree on a repository data model. Inevitably a compromise.
 2. Data evolution is difficult and expensive.
 3. No scope for specific management policies,
 4. Difficult to distribute efficiently.

3.3 Modular decomposition styles:

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organization and modular decomposition.

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.

Modular decomposition

- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered
 - I. An object model where the system is decomposed into interacting objects,
 - II. A pipeline or data-flow model where the systems is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

3.4 Control styles

- Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.
- Centralized control
 1. One sub-system has overall responsibility for control and starts and stops other sub-systems.
- Event-based control
 1. Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

Centralized control:

- A control sub-system takes responsibility for managing the execution of other sub-systems.
- Call-return model:

1. Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems.
- Manager model:
 1. Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

Event – driven systems:

- Driven by externally generated events where the timing of the event is out with the control of the sub-systems which process the event.
- Two principal event – driven models
 1. Broadcast models: An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
 2. Interrupt- driven models: Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.
- Other event driven models include spreadsheets and production systems.

3.5 Reference Architecture:

Reference architecture captures important features of system architectures in a domain. Essentially, they include everything that might be in application architecture although, in reality, it is very unlikely that any individual application would include all the features shown in a reference architecture. The main purpose of reference architectures is to evaluate and compare design proposal, and educate people about architectural characteristics in that domain.

- Architectural models may be applicable to some application domain:
 1. Generic model
 2. Reference model
 (Generic are bottom-top model whereas reference models follow top-down approach.)

- Reference models are derived from study of application domain rather than from existing system- maybe used as a basic system implementation or to compare different system.
- It acts as a standard against which system can be evaluated.

OSI reference model is a layered model of communication system

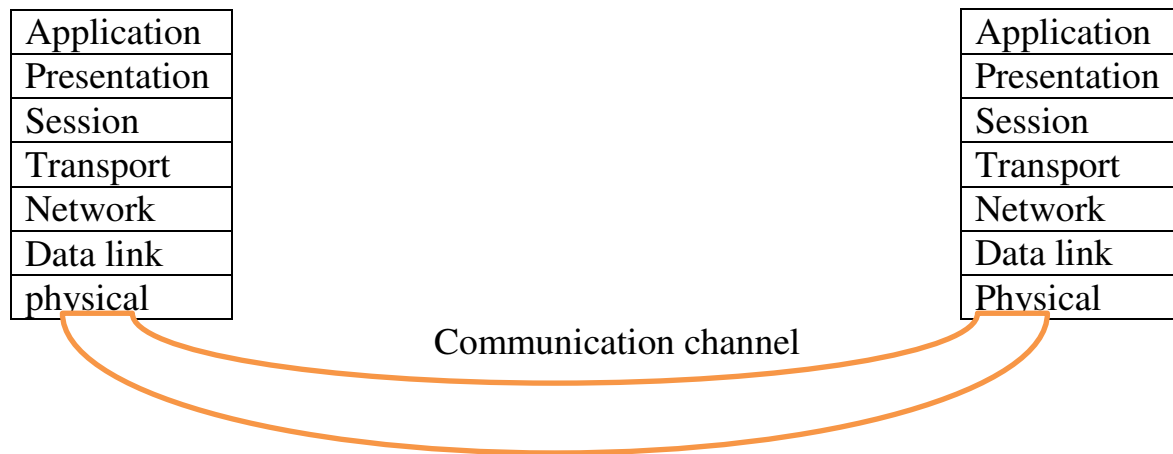
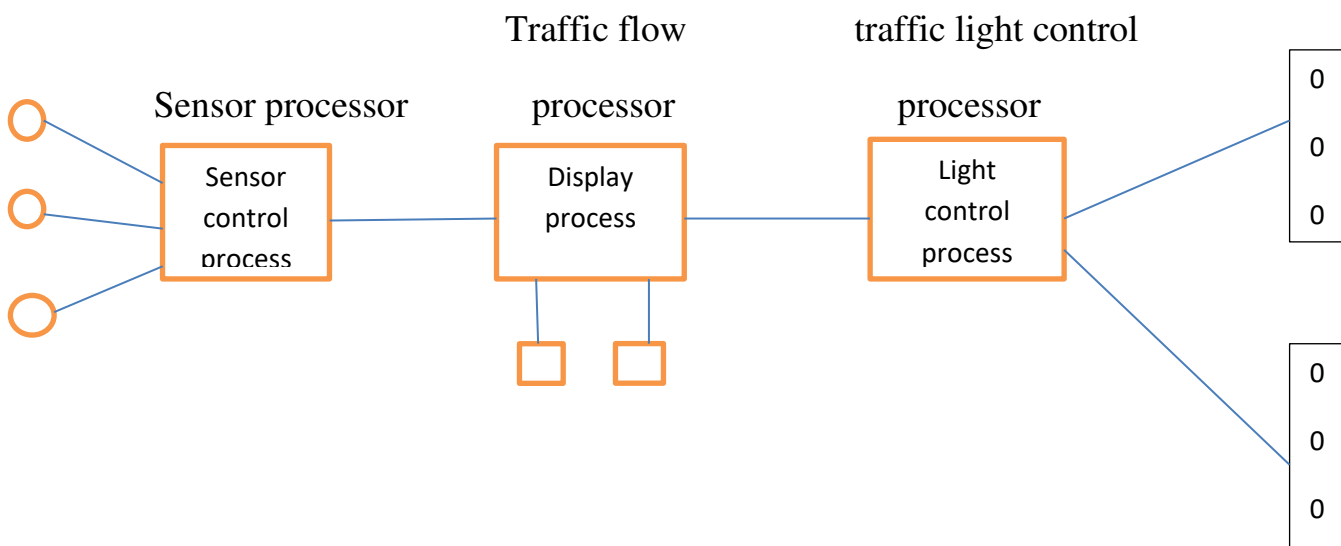


Fig: OSI reference model

3.6 Multiprocessor architecture:



Traffic flow sensor
and camera

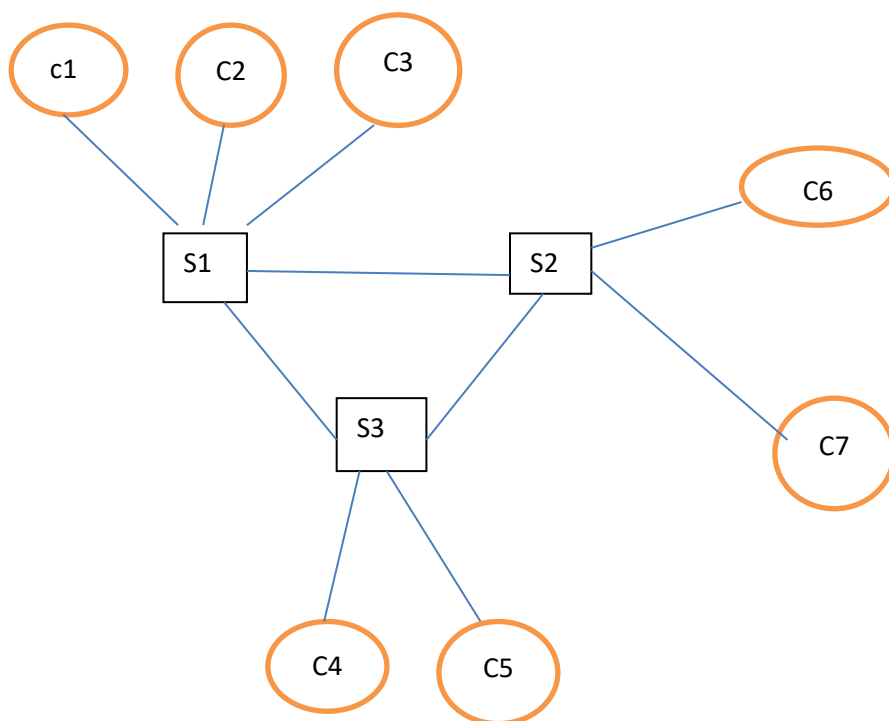
Operator console

Fig: Simple multiprocessor architecture for traffic light system

It is the simplest distributed system model. The system is composed of multiple processor processes which may execute on different processor. It is Architectural model of many large real time systems. In this architecture, distribution of process to processor may be preordered or may be under the control of the dispatcher.

3.7 Client – server architecture:

- This architectural model is used for the set of services that are provided by server and a set of clients that use this service.
- Client should know about the server but the server need not know about clients in client-server architecture.
- Mapping of processor to process is not necessarily 1:1 in case of client-server architecture.
- Client and server are logical processors in client-server architecture.



Note:

S1, s2, s3 are servers.

C1, c2, c3....c7 are the clients.

Fig: client-server architecture

There are different application layers in client-server architecture:

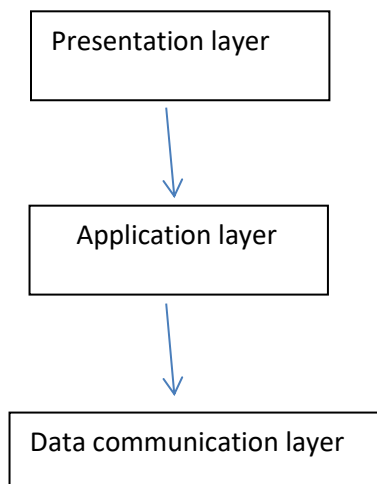


Fig: layer of application

- Presentation layer is concerned with presenting the result of a computation to system users and with collecting user inputs.
- Application layer is concerned with providing application specific functionality.
- Data management layer concerned with managing the system.

3.8 Distributed object architecture:

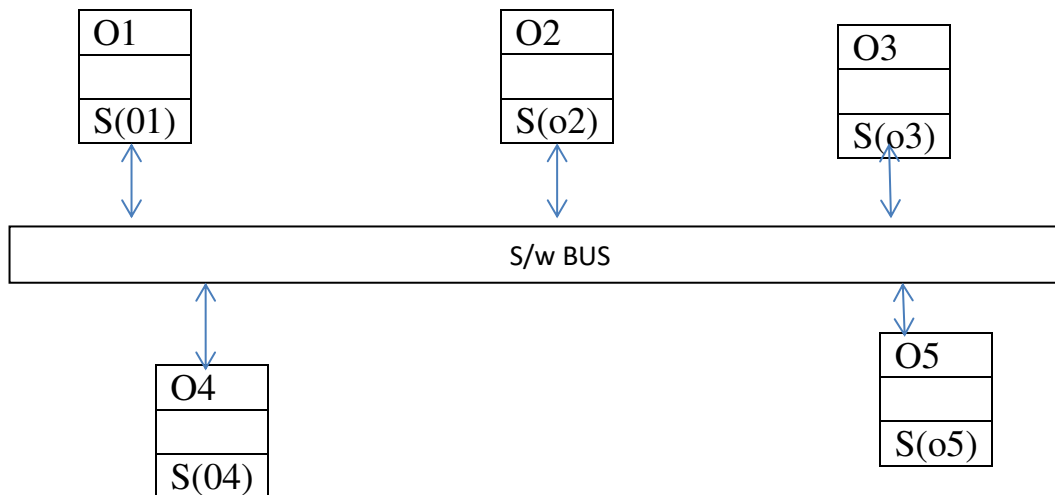


Fig: Distributed object architecture.

Features:

- No distinction
- Each distributed entity is objects that provide service other object and receive from other objects.

- Object communication is through middleware system called an object request booker or software bus.
- However, more complexes to design the client server architecture.

Advantages:

- It allows system designer to delay decision on where and how services should be provided.
- It is very open system architecture that allows to new resources to be as required.
- System is flexible and scalable.

3.9 Inter-organizational distributed computing

Features:

- Used for security and interoperability reason.
- Local standards management and operational processes apply for such inter-organizational distributed computing.
- Newer model of distributed computing have been designed to support inter-organizational computing where different node server are located at different organization.

Chapter 4

Real time software design

Definition

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 this result are produced.

- A 'soft' real-time system is a system groups whose operation is degraded if results are not produced according to the specified timing requirement.
- A 'hard' real-time system is a system whose operation is incorrect if results are not produced according to the timing specification.

Stimulus system

- Given a stimulus, system must produce a response within a specified time.
- Periodic stimulus system & stimuli which occurs at the prediction time intervals.
- Aperiodic stimuli which occur at unpredictable time interval.

General Model of Real Time System

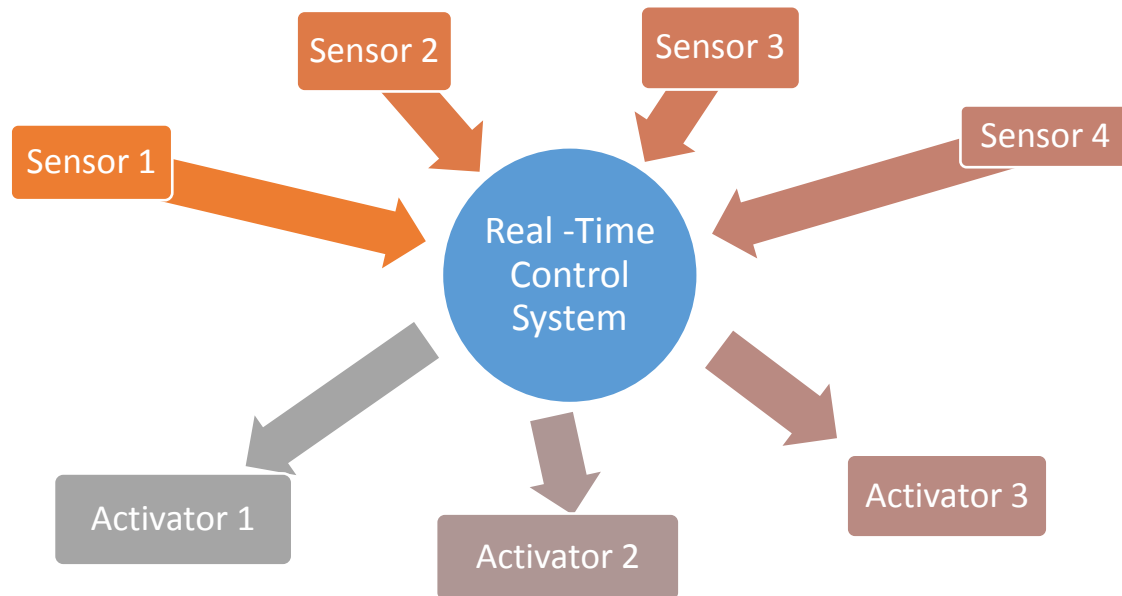


Fig : The sensor-system activator model of an embedded real-time system

Sensor/actuator process

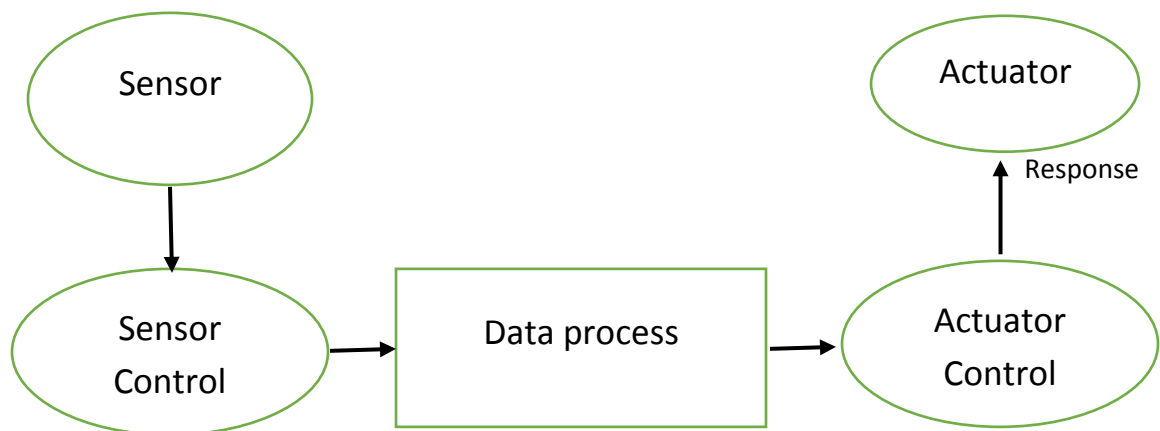


Fig : Sensor/actuator control process

- The generality of the stimulus response system of a real-time system leads to the generic architectural model where there are three types of process.

1. Sensor control process:

- ✓ collect information from sensor
- ✓ May buffer information collected in response to a sensor stimulus.

2. Data processor:

- Carries out processing of collected information and compute response.

3. Actuator

- Generates control signals for actuator.

4.1 System Design

- Design both the hardware & software associated with system. Partition function to either hardware or software.
- Design decision should be made on the basis of Non-functional system requirements.
- Hardware delivers better performance but potentially longer development & less scope for change.

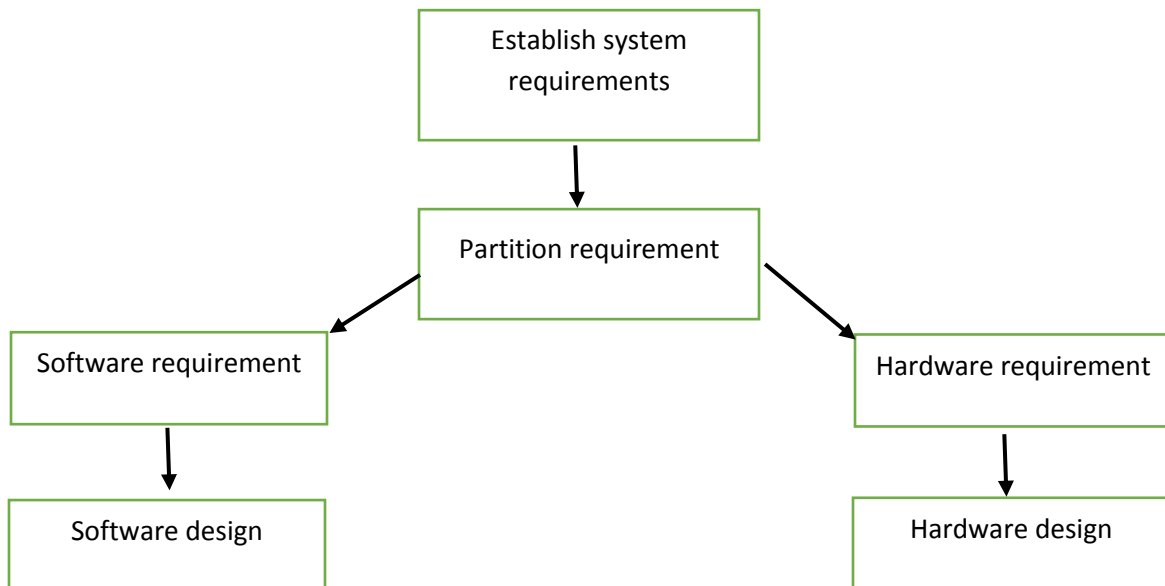


Fig: hardware and software design

- Identify the stimuli to be processed and the required responses to the stimuli.
- For each stimuli & response, identify the timing constraints
- Aggregate the stimulus & response processing into concurrent process

A process may be associated with each class of stimulus & response.

- Design algorithms to process each class of stimulus & 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 execution or operating system.

Timing constraints

- May require extensive simulation and experiment to ensure that they are made by the system.
- May mean that certain design strategies such as object oriented design cannot be used because of additional overhead involved.
- May mean that low level programming language features have to be used for performance reason.

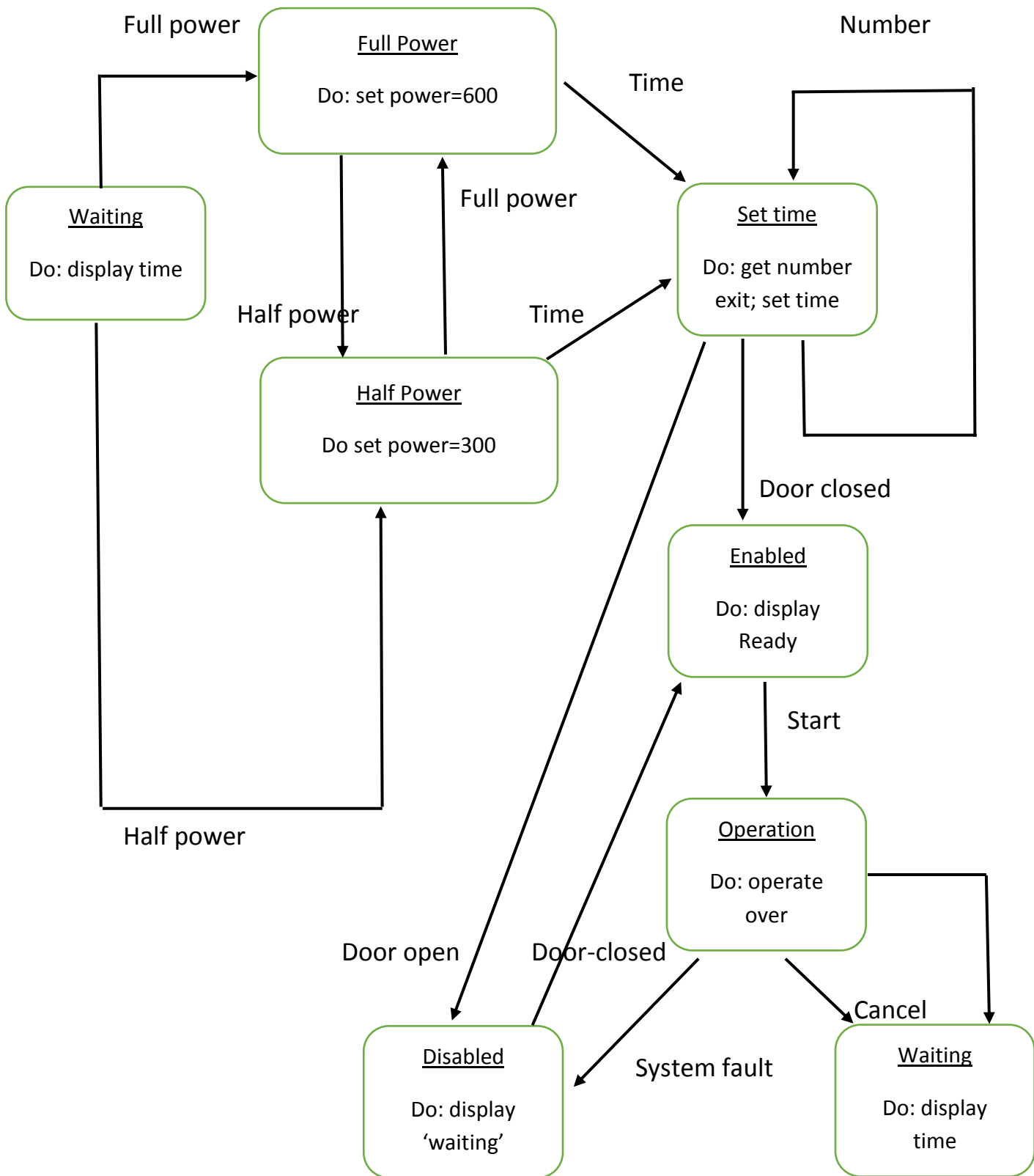


Fig: Microwave oven state machine

State machine modelling

- The effect of stimulus in real-time system may trigger a transition from one state to another.
- Finite state machine can be used for modelling real time system.
- UML includes notations for defining state machine model.

Real time Program

- I. Hard real time system may help to program in assembly language to ensure that deadline are made.
- II. Language such as C allows efficient program to be written but do not have constructs to support concurrency or shared resource management.
- III. Ada is a language design to support real time system design so includes a general purpose concurrency management.

4.2 Real –time operating system/executives

- Executive is analogous to an OS in a general purpose computer
- Are specialized operating system which manage the processes in RTS.
- Responsible for process management and resource allocation
- Doesn't include facilities such as file management.

Executive components

- ✚ Real time clock
 - ✓ Provides information for process scheduling
- ✚ Interrupt handle
 - ✓ Manage aperiodic request for service
- ✚ Scheduler
 - ✓ Chooses the next process to be run
- ✚ Resource manager
 - ✓ Allocate memory and processor resources
- ✚ Dispatcher
 - ✓ Start process execution

Non-stop system component

- **Responsible manager**
 - Responsible for dynamic re-configuration of the system software and hardware modules may be replaced and software upgrade stopping the system
- **Fault manager**
 - Responsible for detecting software and hardware faults and taking appropriate actions to ensure that the system continues in operation

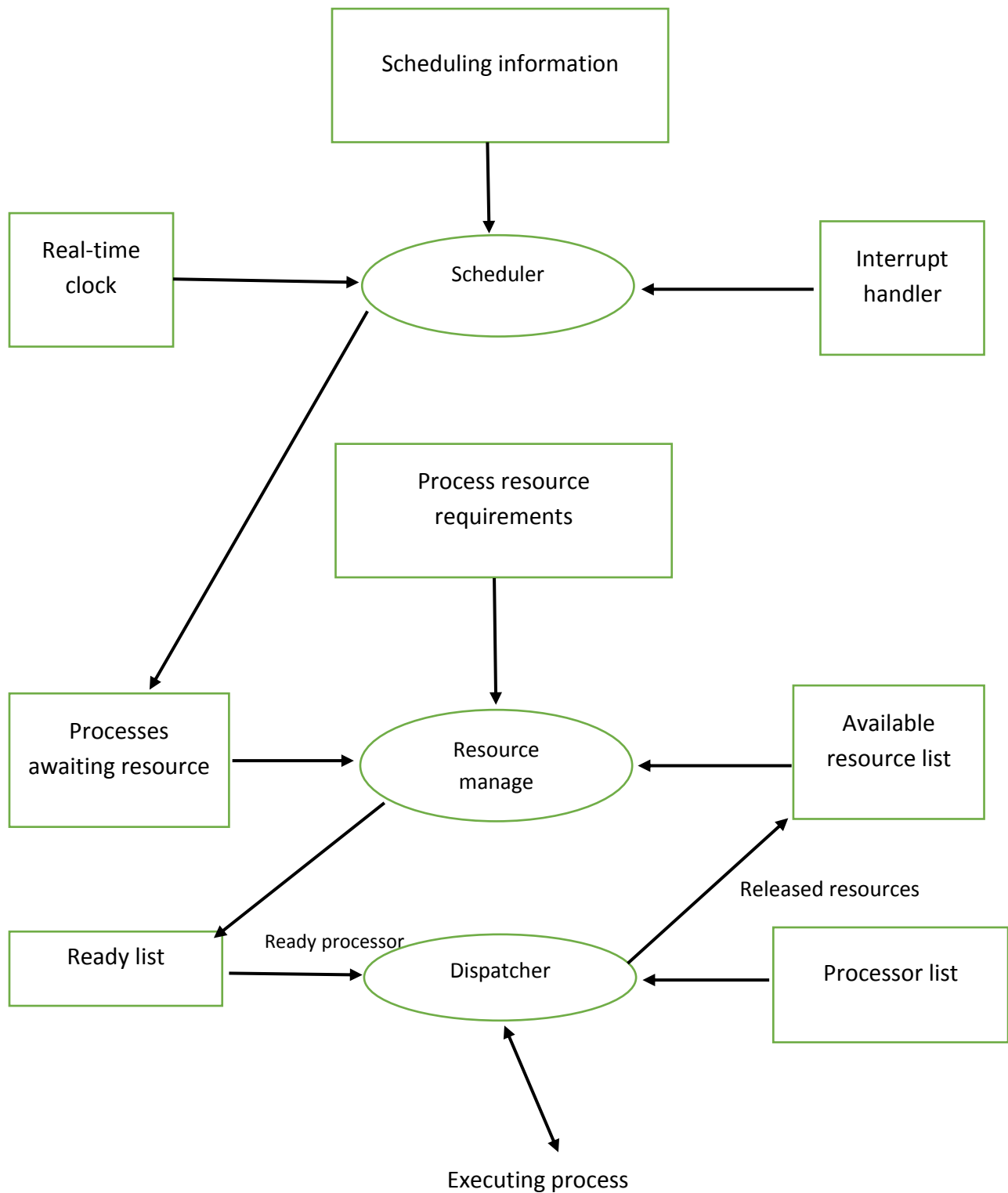


Fig: Real time executive components

4.3 Monitoring & control system

- Important class of real-time systems
- Continuously check sensory & take actions depending on sensor value
- Monitoring system examine sensor & report their results
- Control systems take values & control hardware actuator

Burglar alarm systems

- This system requires to monitor sensors on doors & windows to detect the presence of intruders in a building.
- When a sensor indicates a break-in, the system switches on light around the area & calls police automatically.
- The system should include provision for operation without a main power supply

Sensor

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

Action

- 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

Stimuli to be pressed

- Power failure -> generated aperiodically by a circuit monitor when received, the system must switch to backup power within 50 ms.
- Introduce alarm -> stimulus generated by system sensor, response is to call the police, switch on building lights & audible alarm.

Timing constraints

Stimulus/response	Timing requirement
1. Power failure	Switch to backup within 50 ms.
2. Door alarm	Polled twice per second
3. Window alarm	" " " "
4. Movement detector	" " " "
5. Audible	Switch on within ½ second
6. Lights switch	" " " " "
7. Communication	Call to police within 2 sec of alarm
8. Voice synthesizer	Message should be available within
.	4 sec of an alarm being raised by a
.	sensor

4.4 Data acquisition system

- Collect data from sensors for subsequent processing & analyzer
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing
- Circular or ring buffered, are mechanism for smoothing speed differences.

Reactor data collection

- A system collects data from a set of sensor monitoring the neutron flux from a nuclear reactor
- Flux data is placed in a ring buffer for processing. Ring buffer is itself implemented as concurrent processes so that the collection and processing processes may be synchronized.

Reactor flux monitoring

Sensor

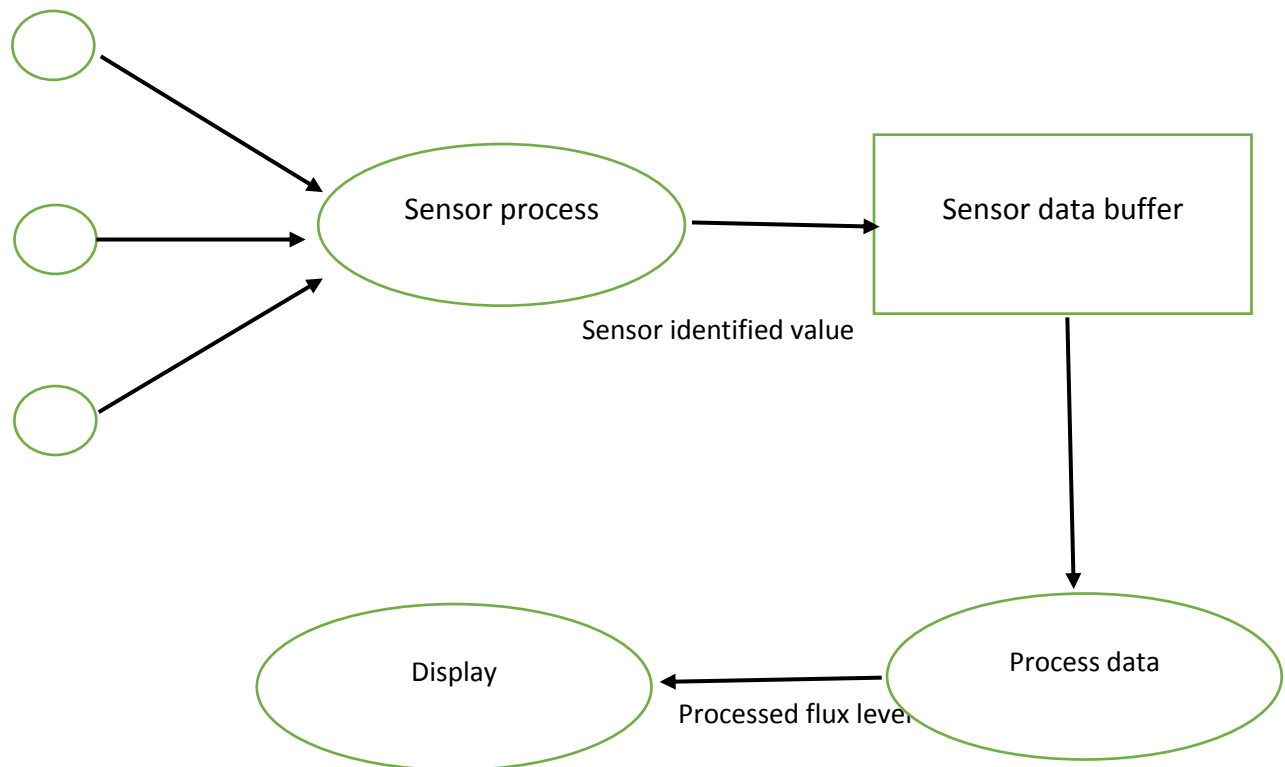


Fig: The architecture of a flux monitoring system

A ring buffer data acquisition

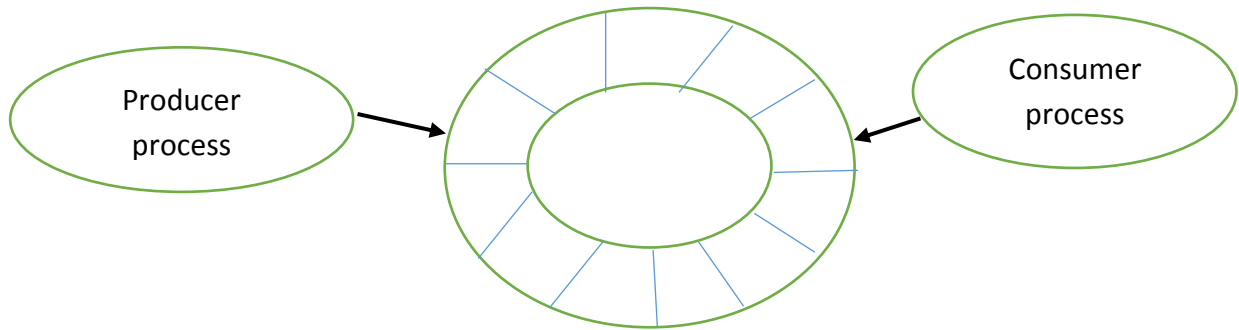


Fig: A ring buffer

- Producer doesn't add while the ring is full
- Consumer doesn't get data while ring buffer is empty

5 Software Reuse

─ Anuj Joshi

- Systems are designed by composing existing components.
- To achieve better s/w, more quickly at lower cost

Reuse- based Software Engineering

1. Application system reuse.
 - The whole of an application system may be reused either by incorporating it without change into other system(COTS reuse) or by developing application families.
2. Component reuse
 - Components of an application from sub- systems to single objects may be reused.
3. Object and function reuse
 - Software Components that implement a single well-defined object or function may be reused.

Advantages

- Increased reliability- components exercised in working system
- Reduced process risk- Reuse components instead of people.
- Standard compliance- Embed standards in reusable components.
- Accelerated development- Embed standards in reusable components.
- Effective use of specialists- Reuse components instead of people

5.1 Reuse landscape

- Although reuse is often simply thought of as the reuse of 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.
- Covers the range of possible reuse technologies

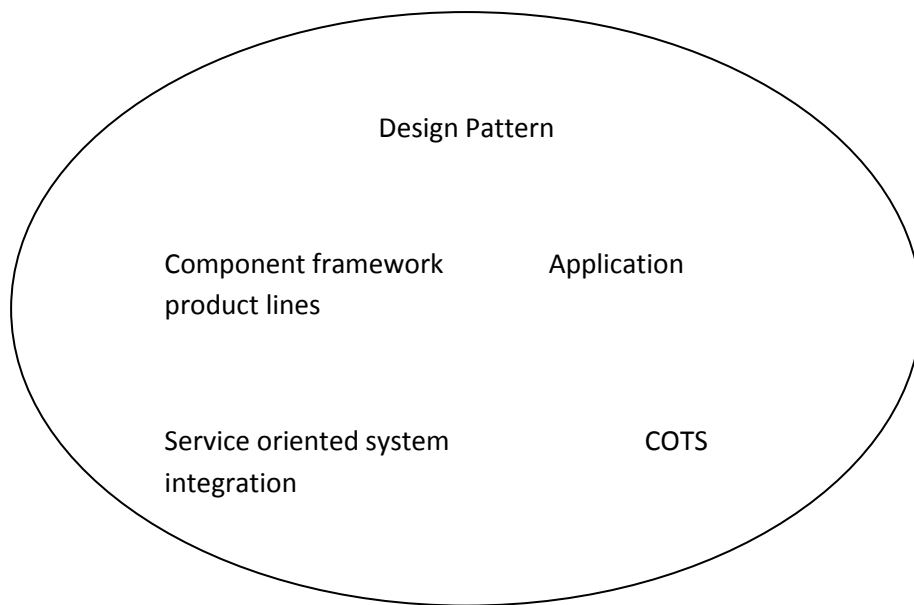


Fig: **Reuse Language**

Concept Reuse

- When we reuse program, we have to follow the design decision made by the original developer of the program
- This may limit the opportunities for reuse
- However, 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.
- Approach to concept reuse is
 1. Design Patterns
 2. Generative Programming

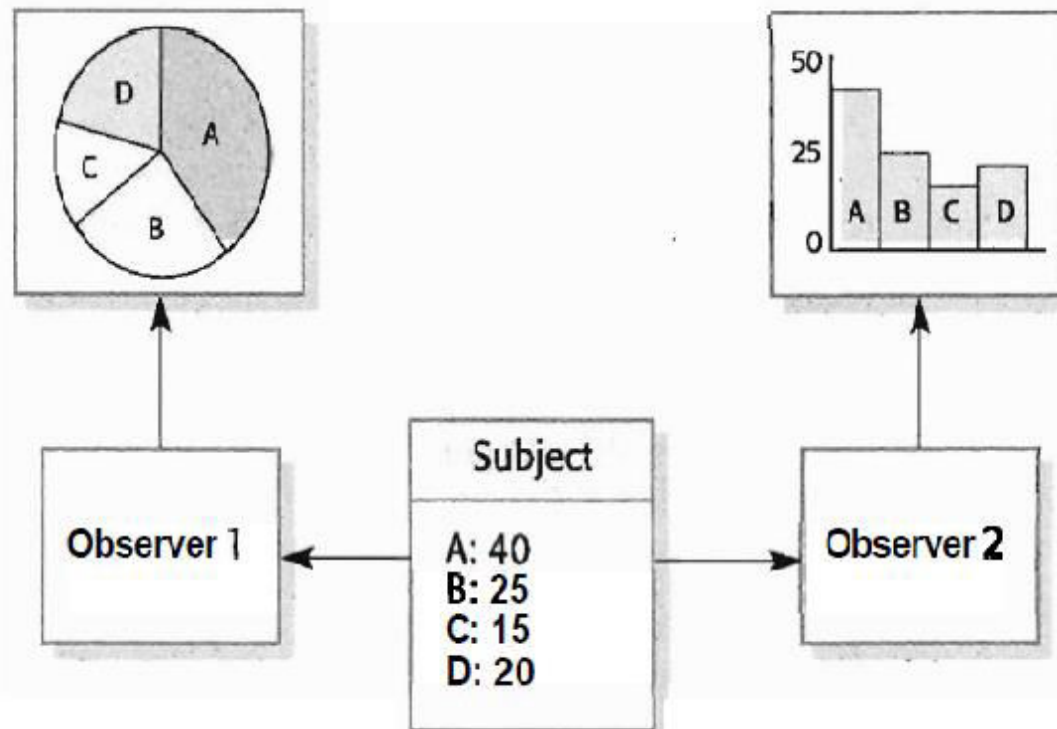
5.2 Design Pattern

- Way of reusing abstract knowledge about a problem and its solution.
- A pattern is a description of the problem and the essence of its solution.
- It should be sufficiently abstract to be reused in different settings.
- Pattern often rely on object characteristics such as inheritance and polymorphism.

Pattern element

- Name
 - A meaningful pattern identifier
- Problem Description
- Solution
- Not a concrete design but a template for a design solution that can be installed in different ways.

- Consequences
- The results and trade-offs of applying the pattern.



5.3 Generator- based reuse

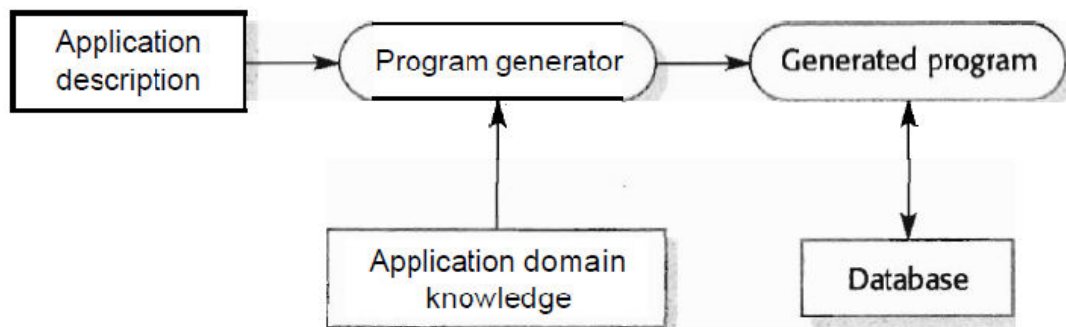
- Program generators involve the reuse of standard program and algorithms.
- These are embedded in the generator and parameterised by user command a program is then automatically generated.
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified.
- A domain specific language is used to compose and control these abstractions.

Types of program generator

- Types of program generator
 1. Application generator for business data processing.
 2. Parser and lexical analyser 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.

Reuse through program generation



5.4 Application Framework

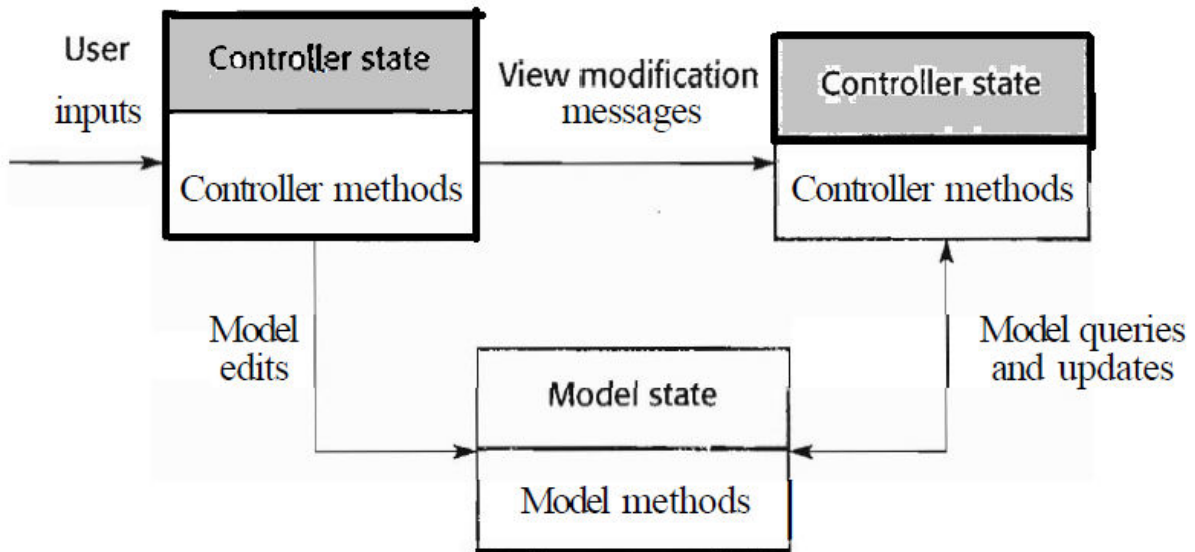
- Generic structure that can be extended to create a more specific sub- system or application
- Frameworks are sub-system design made up of collection of abstract and concrete classes and the interfaces between them.
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.
- Frameworks are moderately large entities than can be reuse.

Framework classes

1. System infrastructure framework
 - Support the development of system infrastructures such as communications, user interface and compilers.
2. Middleware integration framework
3. Enterprise application
 - Standards and classes that support component communications and information exchange.

MVC (Model- View Condition)

- One of the best known used frameworks for GUI design.
- MVC framework involves the instantiation of a number of patterns.
- Allows for multiple presentations of an object and separate interactions with these presentations.



5.5 Application system reuse

- Involves the reuse of entire application system either by configuring a system for an environment or by integrating two or more system to create a new application.
- **Two approaches**
 - i. COTS product integration
 - COTS(Commercial on the self system) are usually complete application systems that often is an API (Application Program Interface)
 - Benefits in faster application development lower cost.
 - ii. Product line development
e.g.: E- Procurement system

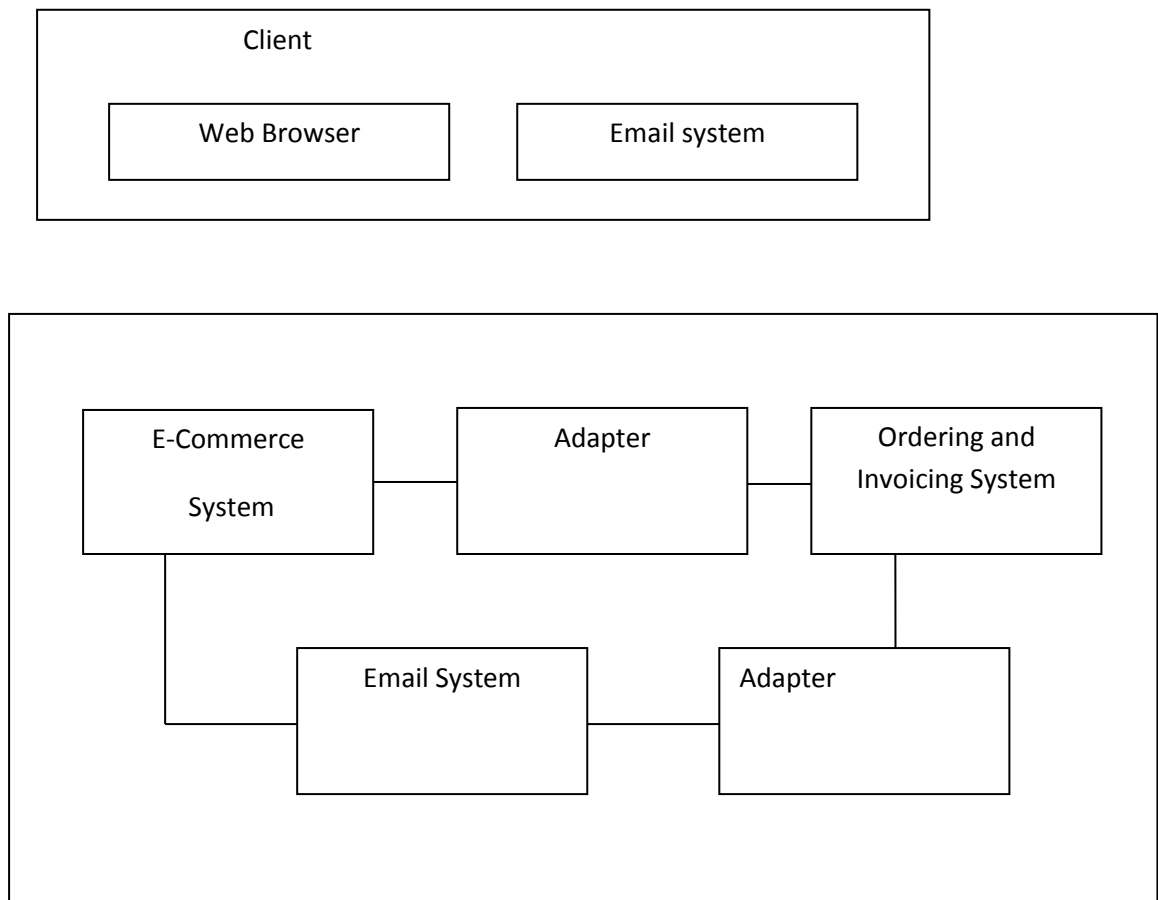


Fig: E-Procurement System

COTS product reused

- On client, standard e-mail and web browsing program are used.
- On server, an e-commerce platform has to be integrated with existing ordering system.
 - This involves writing an adaptor so that they can exchange data.

Software Product lines (Application Families)

- Application with generic functionality that can be adapted and configured for use in specific content.
- Adaptive involves
 1. Component and system configuration
 2. Adding new components to the system.
 3. Selecting from a library of existing components
 4. Modifying components to meet new requirements

E.g.; ERP system (Enterprise Resource Planning)

ERP system

- An Enterprise Resource Planning (ERP) system is a generic system that supports common business processes such as ordering and invoicing, manufacturing, etc.
- These are widely used in large companies- they represent probably the most common form of software reuse.
- The generic core is adapted by including modules and by incorporating knowledge of business process and rules.

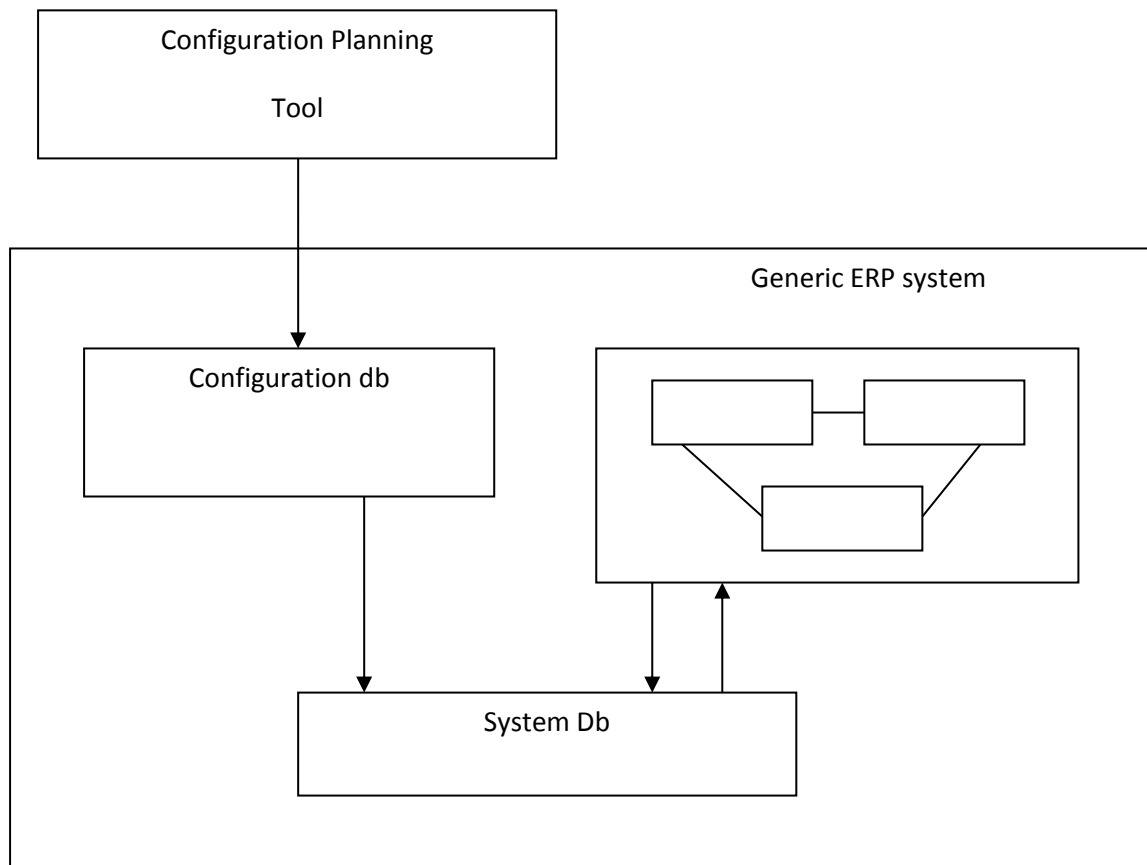


Fig: ERP System Organisation

Key points

- Advantages of reuse are lower costs, faster software development and lower risks.
- Design patterns are high-level abstractions that document successful design solutions.
- Program generators are also concerned with software reuse- the reusable concepts are embedded in a generator system.
- Application frameworks are collections of concrete and abstract objects that are designed for reuse through specialisation.
- COTS product reuse include lack of control over functionality, performance and evolution and problems with inter- operation.
- ERP systems are created by configuring a generic system with information about a customer's business.
- Software product lines are related applications developed around a common core of shared functionality.

6. Component Based Software Engineering

- Amir & Anuj

6.1 Component and Component Model

- A component is a s/w unit whose functionality and dependencies are completely defined by a set of public interface.
- Components can be composed with other components without knowledge of their implementation & can be deployed as an executable unit.

According to Szyperski:

- *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.*

According to Councill and Heinemann:

- *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.]*

Component characteristics

- *Standardised – Following a standard component model*
- *Independent – Usable without Adaptors*
- *Composable – External interactions use public interface*
- *Deployable – Stand-alone entity*
- *Documented – Full Documentation*

Components have two related interface

- Provides interface defines the services provided by the component.
- Requires interface specifies what services must be provided by other components in the system if a component is to operate correctly

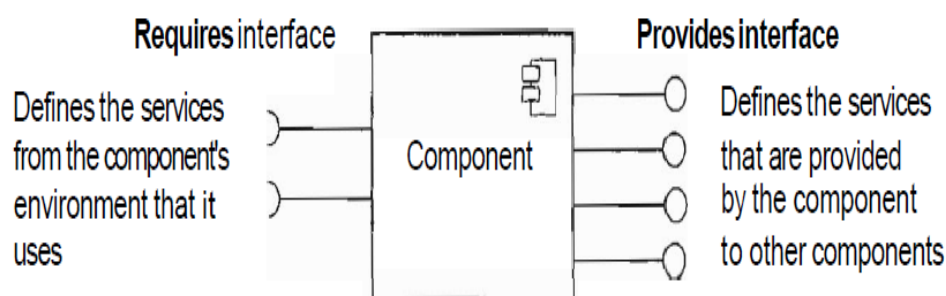


Fig: A component

Component Model

- Defines a set of standards for components, including interfaces standard, usage standards and deployment standards.

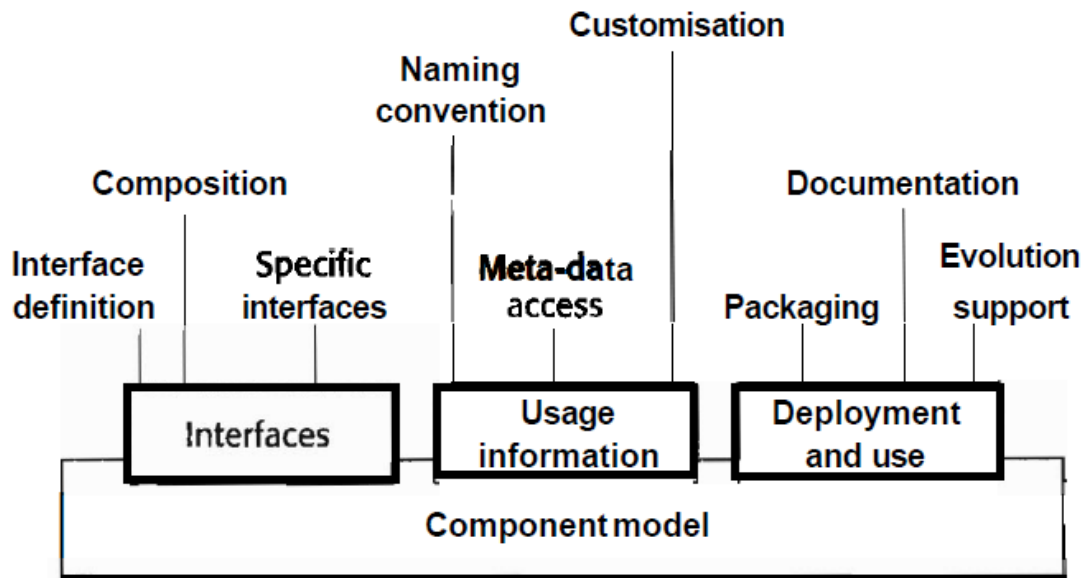


Fig: Basic Elements of Component Model

2 categories of service provided

- Platform service:
Which enables components to communicate & interoperate in a distributed environment?
- Support service:
Which are common services that are likely to be required by many different components

6.2 The CBSE process

- Component-based software engineering (CBSE) is an approach to software development that relies on software reuse.
- CBSE emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.

- Components are more abstract than objects and classes. Thus, they can be considered to be stand-alone service providers.
- CBSE requires knowledge, careful planning, and methodological support.
- CBSE is intended to provide an effective and efficient means for reuse, by building systems out of existing parts or components. Figuratively, the idea is comparable to that of lego bricks which can be used to develop a number of different „applications“ from a standard set of components. This will not only reduce the effort needed for system development but will also have a positive impact on the costs involved. Also this sounds easy the general idea is not new but hasn't been successfully realized yet. Effective reuse was/is also one of the major promises of OO. However, classes and objects are often too specific and fine-grained to be effectively reused (e.g., single operations instead of applications). The idea of CBSE is to raise the level of abstraction of reusable entities. Thus they can be seen as standalone entities or in other words: One man's component can be another man's system. Although this sounds easy, its practical application requires knowledge (provided services, side effects, providers, market, ..., etc.), careful planning (risks, impacts on the system, etc.) and methodological support applying sound engineering practices. Otherwise CBSE is likely to fail.

CBSE and design principles

- Apart from the reuse, CBSE is based on sound software engineering principles:
 - Component are independent
→do not interfere with each other;
 - Component implementations are hidden;
 - Communication is through well-defined interfaces;
 - Component platforms are shared and reduce development costs.
 - Component interfaces and application are „standardized“ to ease application

CBSE process

- When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components.
- 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.

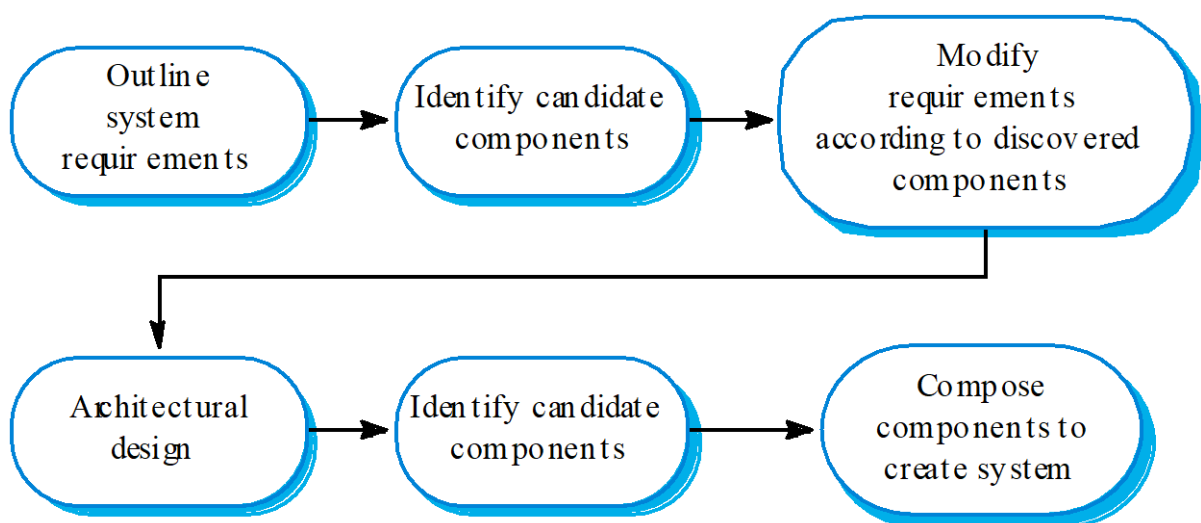
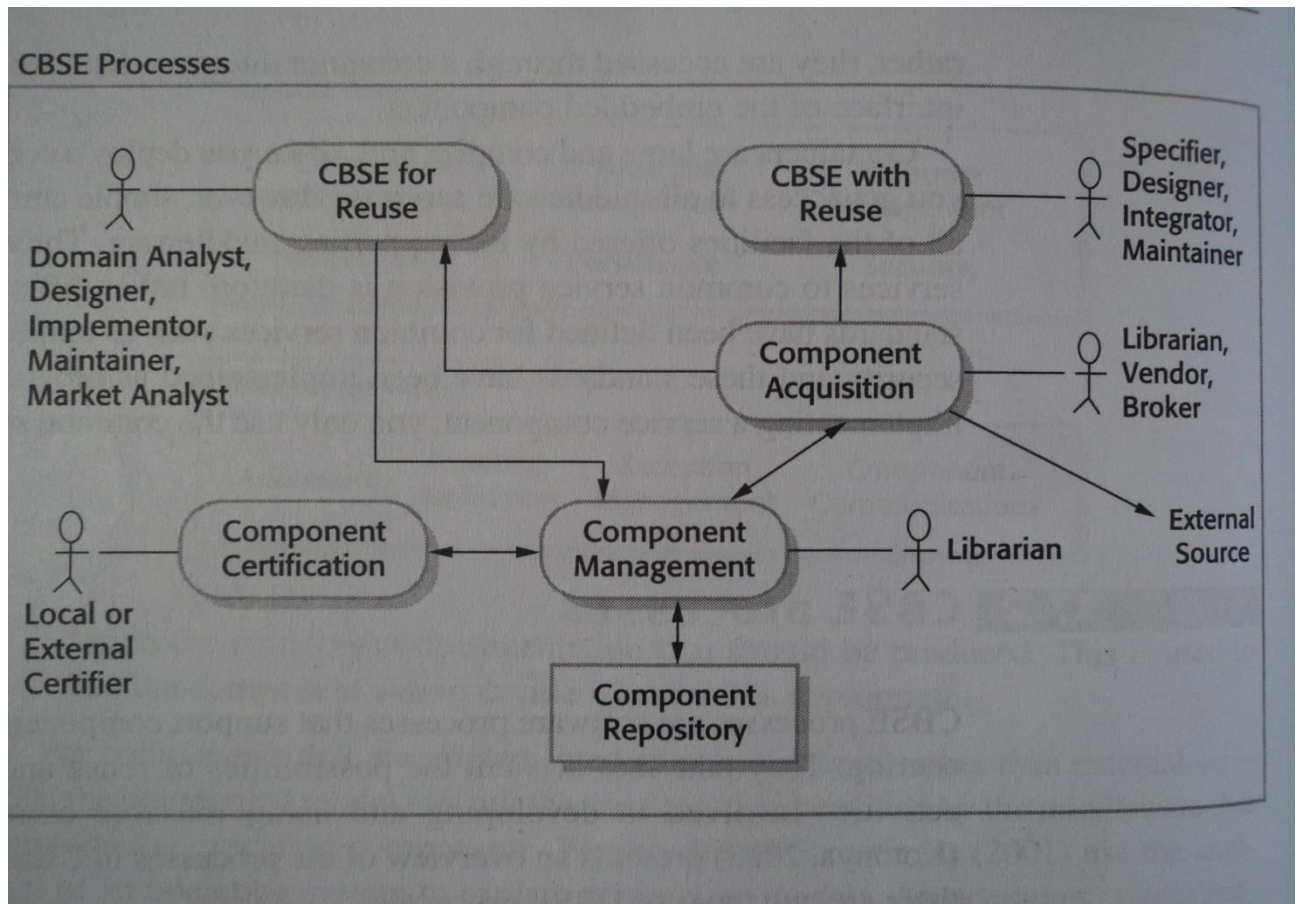


fig.1 CBSE process reuse

This figure shows the ideal/general process for CBSE without showing iterations and feedback loops. In practice there are many loops and feedback cycles (...), However, the principal sequence is the same.

There are s/w process that support component based s/w engineers

2 types of CBSE process are:

1. Development for reuse: Concerned with developing components that will be reused in other application.
2. Development with reuse: Process of developing new application using existing components and services.

Supporting Process:

1. Component acquisition
2. Component management
3. Component certification

6.3 Component Composition

Component composition is the process of integrating components with each other, and specially written 'glue code' to create a system or another component.

Types

1. Sequential composition
We can create new component from two existing components by calling the existing component in sequence.
2. Hierarchical composition
This type of composition occurs when one component calls directly on the services provided by another component.
3. Additive composition
This occurs when two or more components are put together to create a new component, which combine their functionality.

When composing reusable components that have not been written for specific application, one may need to write adaptors or 'glue code' to reconcile the different component interfaces.

Three types of incompatibility can occur

- i. Parameter incompatibility
- ii. Operator incompatibility
- iii. Operator incompleteness

CHAPTER 7: VERIFICATION AND VALIDATION

(3hrs,12 marks)

-Anuja Ghising

Syllabus

7.1 Planning Verification and validation

7.2 software inspections

7.3 Verification and formal methods

7.4 Critical system verification and validation

Verification

It is an act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services or documents conform to specified requirements.

It can also be defined as the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase.

Validation

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Validation is, therefore, 'end-to-end' verification. Validation occurs through the utilization of various testing approaches.

Verification and Validation goals

a. Correctness

The extent to which the product is fault free.

b. Consistency

The extent to which the product is consistent within itself and with other products

c. Necessity

The extent to which everything in the product is necessary.

d. Sufficiency

The extent to which the product is complete.

e. Performance

The extent to which the product satisfies its performance requirements.

7.1 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 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 integration of techniques applicable to the various life cycle phases

as delegation of these tasks among the project's organizations. The planning of this integrated V&V approach is very dependent upon the nature of the product and the process used to develop it. Traditional integrated V&V approaches have followed the "waterfall model".

5. Problem Tracking

Software V&V plandeveloping a mechanism for documenting problems

- when the problem occurred
- where the problem occurred
- evidence of the problem
- priority for solving problem

- The V model of development

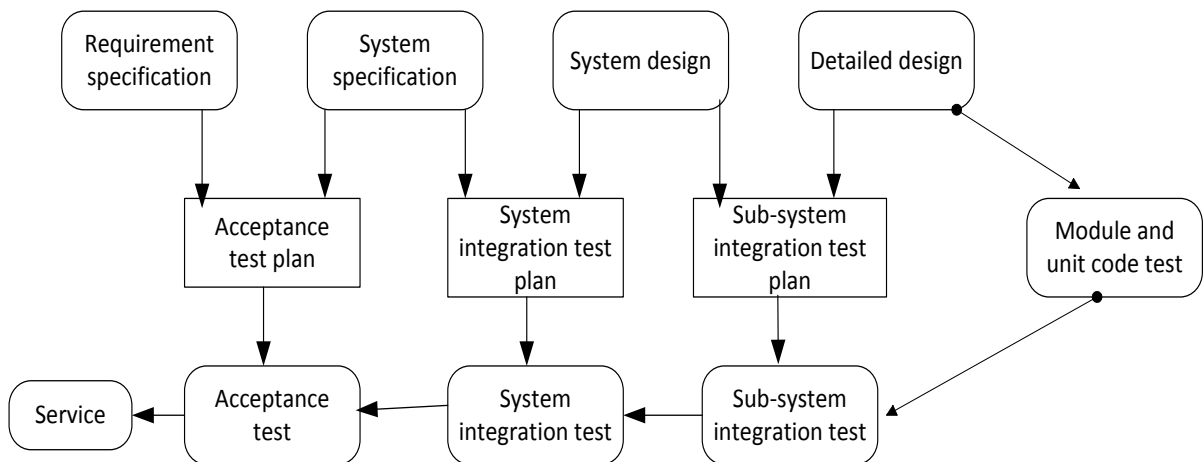


Fig: V model of development

7.2 Software Inspections (static verifications)

Software inspections can be used for the detection of defects in detailed designs before coding, and in code before testing. They may also be used to verify system requirements, design models and even the proposed system tests. Software inspections are concerned with the analysis of the static system representation to discover problems (static verification).

- Inspection is a manual, static technique that can be applied early in the development cycle.

- Inspection is based on reading techniques.

- Advantages of inspection over testing

1. During testing one error can hide other errors. As inspection is static process there is no interaction between errors, consequently an inspection can discover many errors.
2. Incomplete versions of a system can be inspected without additional costs.
3. Inspection considers inefficiencies, inappropriate algorithms and poor programming styles that could make system difficult.

- Inspection per conditions

1. A precise specification must be available.
2. Team members must be familiar with the organization standards.
3. Syntactically correct code must be available.
4. An error checklist must be present.
5. Management must accept that inspection will increase costs early in the software process.

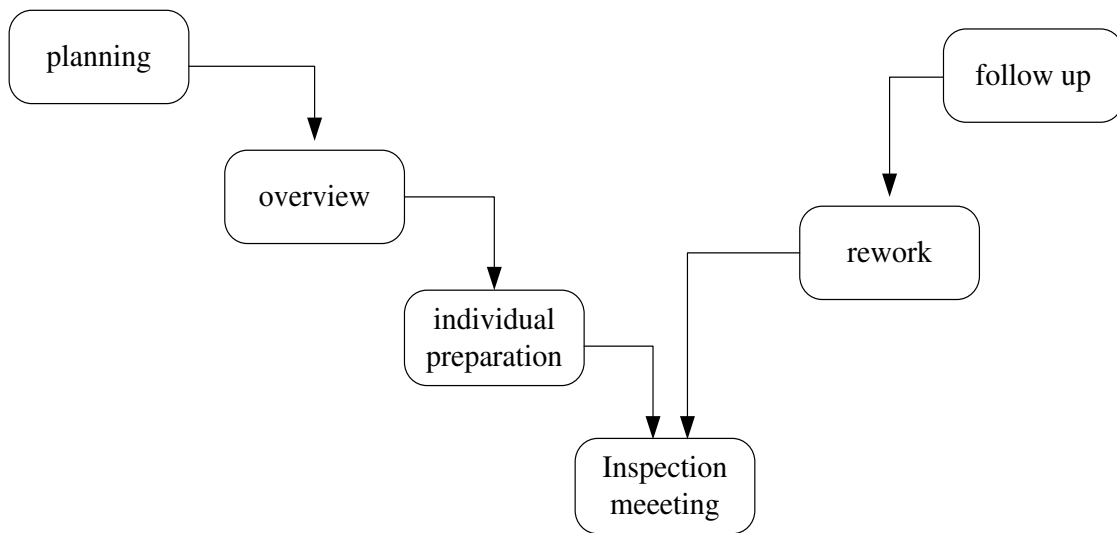


Fig: The inspection process

- Inspection procedure

1. First of all, system overview is presented to the inspection team.
2. Secondly, the required codes and documents are distributed to the inspection team in advance.
3. Then, inspection takes place and errors are discovered. Pre inspection however may or may not be required.

- Inspection team and role

1. Moderator:

The moderator leads the inspection and chairs the inspection meeting. The person should have implementation skills, but not necessarily be knowledgeable about the item under inspection. He or she must be partial and objective. For this reason moderators are often drawn from staff outside the project. Ideally they should receive some training in inspection procedures.

2. Inspector:

Inspectors identify and describe defects in the review items under inspection. They should be selected to represent a variety of viewpoints (e.g. designer, coder and tester).

3. Reader:

The reader guides the inspection team through the review items during the inspection meetings.

4. Author:

The author is the person who has produced the items under inspection. The author is present to answer questions about the items under inspection, and is responsible for all rework. A person may have one or more of the roles above. In the interests of objectivity, no person may share the author role with another role.

7.4 Verification and Formal Methods (FM)

FM can be used when a mathematical specification of the system is produced. They are the ultimate static verification techniques. They involve detailed mathematical analysis of the specifications and may develop formal arguments that a program conforms to its mathematical specification.

- 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 program is analyzed alongside the specifications.

- Arguments against FM
 1. Requires specialized notations that are not understood by domain experts.
 2. It is very expensive to develop a specification and even more expensive to show that a program meets that specification.

7.4 Critical system Verification and Validation

- Verification and Validation cost

V and V costs for critical system involves additional validation processes and analysis than for other system. The cost of failure is so high that it is cheaper to find and remove faults than to pay for system failure.

Normally, verification and validation costs take up more than 50% of the total system development costs which may be as follows:

1. 33% - life sustaining medical devices or nuclear weapons
2. 20-25% - telecommunications or financial systems
3. 10-18% systems desiring software quality but not high-integrity.

- Critical system verification and validation includes:

1. Reliability Validation:

Exercising the programs to access whether or not it has reached the required level of reliability.

2. Safety assurance:

Concerned with establishing confidence labels in the system. However, quantitative measurements of safety is impossible.

3. Security assessment:

Intended to demonstrate that the system can't enter some state rather than to demonstrate that the system can do something.

4. 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.

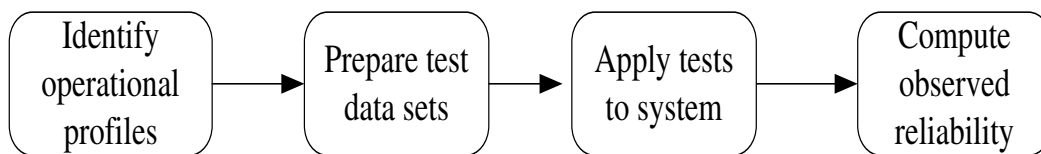


Fig :The reliability measurement process

- Process et al(1990) suggests five types of review for safety critical system:
 1. Review for correct intended function.
 2. Review for maintainable, understandable structure.
 3. Review to verify that the algorithm and data structure design are consistent with the specified behavior.
 4. Review the consistency of the code and the algorithm and the data structure design.
 5. Review the adequacy of the system test cases.

- Security assessment

There are four complementary approaches for security checking:

1. Experience-based validation
2. Tool-based validation
3. Tiger team
4. Formal verification

PAST QUESTIONS

Qn. Distinguish between verification and validation. [066 Bhadra, 5 marks]

Qn. Explain why program inspection are an effective technique for discovering errors in program? What types of errors are unlikely to be discovered through inspections? [068 chaitra, 10 marks]

CHAPTER-8

SOFTWARE TESTING AND COST ESTIMATION

-Ela Thakur

Software testing is the process of exercising with the specific intent of finding errors prior to delivery to the end user. Testing is part of a broader process of software verification and validation. Testing results in higher quality software, more satisfied user and lower maintenance cost, more accurate and reliable results. Testing costs 1/3 to 1/2 of the total cost of software development process.

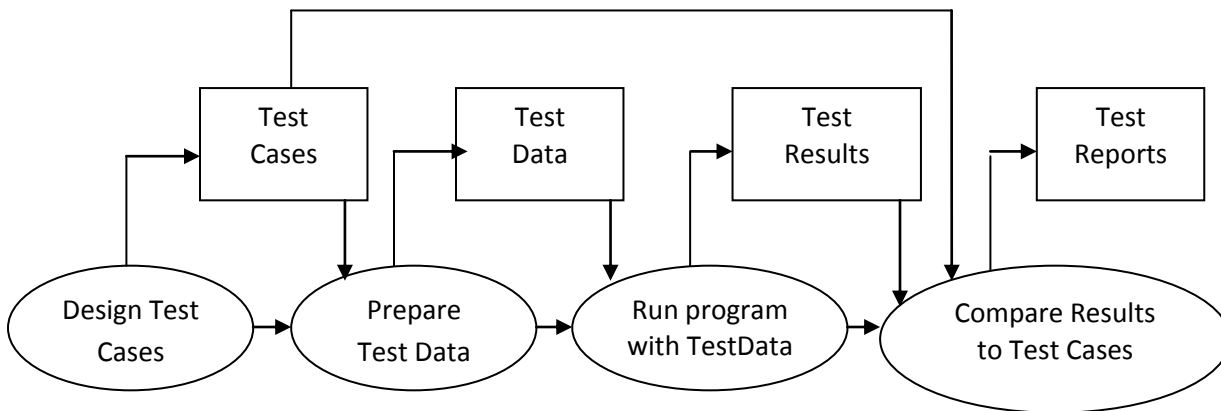


Fig: A model of the software testing process

8.1. SYSTEM TESTING:

System testing fully exercise the computer based system to verify the system elements have been properly integrated and perform allocated functions. An independent testing team is responsible for system testing. The tests are based on system specification.

There are two phases in system testing:

a) Integration testing: (IOE Q: Explain Integration testing 067 asadh)

In this type of testing the test team has access to the system code. The system is tested as components are integrated.

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using Black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing. The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.

Some different types of integration testing are big bang, top-down, and bottom-up.

Big Bang

In this approach, all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. The Big Bang method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing.

Top-down and Bottom-up

Bottom Up Testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for

integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.

Top Down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.

The main advantage of the Bottom-Up approach is that bugs are more easily found. With Top-Down, it is easier to find a missing branch link

b) Release testing:

In this type of testing a separate testing team tests a complete version of the system before it is released to users. System testing by the development team should focus on discovering bugs in the system. The aim of release testing is to check that the system meets the requirements of system stakeholders. System testing by the development team should focus on discovering bugs in the system. Release testing is usually a **black-box testing** process where tests are derived from the system specification. The system is treated as black-box whose behavior. Another name for this is 'functional testing', so called because the tester is only concerned with functionality and not the implementation of the software.

8.2. COMPONENT TESTING:

Several individual units are integrated to create composite components. Component testing should focus on testing component interfaces. It is a defect testing process. The components may be:

- Individual function or methods within an object
- Object classes with several attributes and methods
- Composite components with defined interfaces used to access their functionality

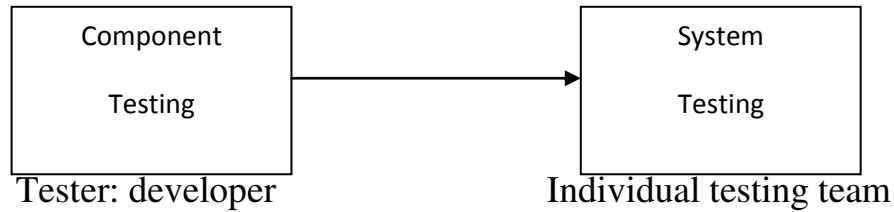


Fig: Testing Phases

8.3. TEST CASE DESIGN:

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

❖ Design approaches:

- Requirement-based testing:
Used in validation testing technique where we consider each requirement and tests for that requirement.
- 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.
- Structural Testing(White-Box Testing):
It is a method of testing software that tests internal structures or workings 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.

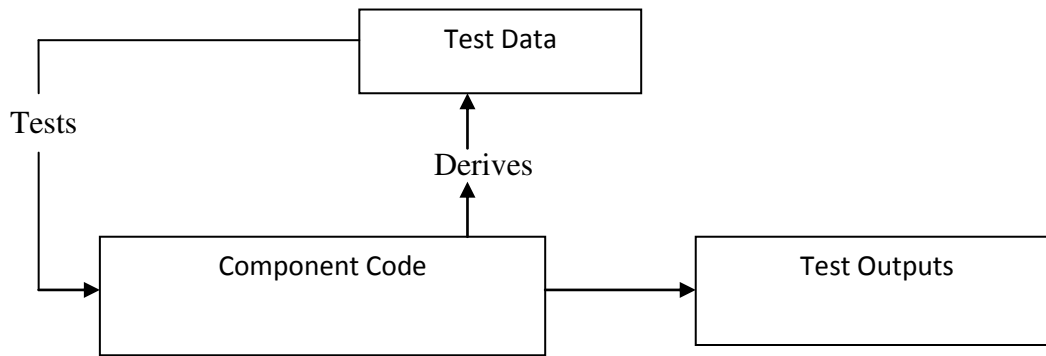


Fig: Structural Testing

8.4. TEST AUTOMATION:

In software testing, test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes to predicted outcomes. Test automation 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. It reduces testing costs by supporting the test process with range of software tools. System such as 'JUnit' supports the automatic execution of tests. There are two general approaches to test automation:

- **Code-driven testing.** The public (usually) interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.
- **Graphical user interface testing.** A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

8.5. METRICS FOR TESTING:

Majority of metric for testing proposed 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.

$$PL = 1 / [(n1/2) * (N2/n2)]$$

$$e = V / PL$$

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 a 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:

$$K = e(K) / \sum e(i) \quad \text{where; } e(K) \text{ is computed for module } K$$

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

8.6. 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:

I. Function-related measures:

Productivity is expressed in terms of the amount of useful functionality produced in some given time. Function point in a program is computed by measuring program features:-

- A. External inputs and outputs
- B. User interactions
- C. External interfaces
- D. Files used by the system

II. Size:

- Line of code delivered

- Also measure no. of delivered object code instruction or no. of pages of system documentation
- Useful for programming in FORTRAN, Assembly or COBOL
- More expressive the programming language, the lower apparent productivity

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

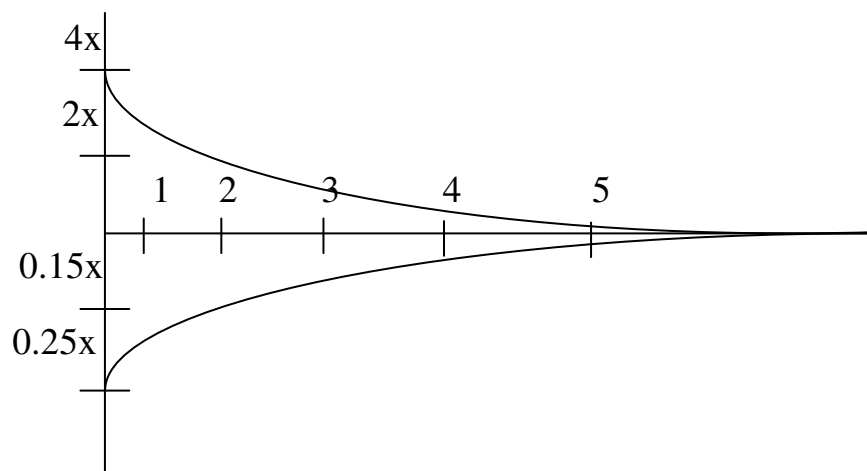
$$\begin{aligned}\text{Then; productivity} &= (5000/28) * 4 \\ &= 714 \text{ lines/ month}\end{aligned}$$

8.7. 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 estimation techniques are:

- Algorithm cost modeling
- Expert judgment
- Estimation by analogy
- Parkinson's law
- Pricing to win



- 1: Feasibility
- 2: Requirement Design
- 3: Code
- 4: Delivery

Fig: Estimation Uncertainty

8.8. ALGORITHM COST MODELLING

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

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

Where; A is an organization-dependent constant

B reflects the disproportionate effort for large project

M is a multiplier reflecting product, process and people attributes

❖ COCOMO MODEL(CONSTRUCTIVE COST MODEDL):

This is an empirical model that was derived by collecting data from a large number of software projects. This is a well-documented and non-proprietary estimation model.

Formula for effort computation for system prototype:

$$PM = [NAP * (1 - \%reuse/100)] / [PROD]$$

Where; PM is effort estimation in person-motion

NAP is the total no. of application points in the delivered system

%reuse is an estimation of the amount of reused code in the development

PROD is the application-point productivity

8.9. PROJECT DURATION AND STAFFING:

As well as effort estimation, managers must estimate the calendar time required in completing a project and the staff required.

Calendar time estimation (COCOMO)

$$TDEV = 3 * (PM) ^ [(0.33 + 0.2) * (B - 1.01)]$$

Where; PM is the effort computation

B is the exponent computed (B is 1 for the early prototyping mode)

Time required is independent of the number of people working in the project. Staff required cannot be computed by dividing the development time by the required schedule. The number of people working in a project varies depending on the phase of the project.

✓ IOE QUESTIONS:

- What is regression testing?
- A test suite is developed incrementally as a program is developed. We can always run regression tests to check the changes to the program have not introduced new bugs.

The intent of regression testing is to ensure that a change has not introduced new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software.

Common methods of regression testing include rerunning previously-completed tests and checking whether program behavior has changed and whether previously-fixed faults have re-emerged. Regression testing can be used to test a system efficiently by systematically selecting the appropriate minimum set of tests needed to adequately cover a particular change.

Regression testing can be used not only for testing the correctness of a program, but often also for tracking the quality of its output. For instance, in the design of a compiler, regression testing could track the code size, simulation time and compilation time of the test suite cases.

- What are the basic principles of software testing? List the characteristics of testability of software. List out possible errors of black-box testing.

➤ The basic principles of software testing are:

- 1) Testing an application exhaustively is impossible

Example

Assume that we have been given an application which produces bank statements that are sent to the customers. It is impossible for us to test each and every notice that is being generated to each customer. Only way to perform is to identify a suitable sample for it.

- 2) Testing is context based - Software testing is always based on the purpose to which the software built will be used.

Example:

An application built to be used inside an aircraft, requires rigorous testing and subject it to high quality standards. But an application built for storing the addresses in a Personal computer need not be tested rigorous similar to the previous application.

- 3) Testing a software is to find out the defects - not to prove that the software is error free.

The main objective of testing a software is to find out as many defects as possible. Testing is not done to prove that the software is error free.

Example

A software may not have any reported defects (all defects identified are fixed) but still it may fail in the production environment.

- 4) Testing starts from requirements gathering, In other words early testing reduces the amount of money, rework involved etc.

Testing must be started as early as the Testing life cycle begins. The earlier we identify and fix defects, the greater the money is saved.

- 5) The number of Defects in an application seems to come from one or few areas or modules of the application and not spread evenly.

Example:

The module was prepared by new programmer

The complexity of that particular module is very high etc..

- 6) Performing the similar kind of testing again and again does not identify the defects.

Executing same set of test cases will not identify the defects present in the software.

- 7) Absence of errors in an application does not mean that, the application is free from defects.

The characteristics of testability of software are:

- Operability
- Observability
- Controllability
- Decomposability
- Simplicity
- Stability
- Understandability

Possible errors of Black-box testing are:

- Only a small number of possible inputs can be tested and many program paths will be left untested
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design
- Tests can be redundant if the software designer/ developer has already run a test case

- Testing is one of the very important core parts of software development and implementation. Comment on this statement and explain various testing techniques.
- Software testing is the process of exercising with the specific intent of finding errors prior to delivery to the end user. Testing is part of a broader process of software verification and validation. Testing results in higher quality software, more satisfied user and lower maintenance cost, more accurate and reliable results. Testing costs 1/3 to 1/2 of the total cost of software development process. Hence testing is the very important core parts of software development and implementation.

Various testing techniques are:

A. SYSTEM TESTING:

System testing fully exercise the computer based system to verify the system elements have been properly integrated and perform allocated functions. An independent testing team is responsible for system testing. The tests are based on system specification.

There are two phases in system testing:

- Integration testing:

In this type of testing the test team has access to the system code. The system is tested as components are integrated.

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using Black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing. The overall idea is a "building block"

approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.

Some different types of integration testing are big bang, top-down, and bottom-up.

Big Bang

In this approach, all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. The Big Bang method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing.

Top-down and Bottom-up

Bottom Up Testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.

Top Down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.

The main advantage of the Bottom-Up approach is that bugs are more easily found. With Top-Down, it is easier to find a missing branch link

- Release testing:

In this type of testing a separate testing team tests a complete version of the system before it is released to users. System testing by the development team should focus on discovering bugs in the system. The aim of release testing is to check that the system meets the requirements of system stakeholders.

System testing by the development team should focus on discovering bugs in the system. Release testing is usually a **black-box testing** process where tests are derived from the system specification. The system is treated as black-box whose behavior. Another name for this is 'functional testing', so called because the tester is only concerned with functionality and not the implementation of the software.

B.COMPONENT TESTING:

Several individual units are integrated to create composite components. Component testing should focus on testing component interfaces. It is a defect testing process. The components may be:

- Individual function or methods within an object
- Object classes with several attributes and methods
- Composite components with defined interfaces used to access their functionality



Fig: Testing Phases

- What problems may be encountered when top down integration is chosen?
 - The problems encountered are:
 - The solution provides limited coverage in the first phases.
 - A minimal percentage of user accounts are managed in the first phases.
 - You might have to develop custom adapters at an early stage.
 - The support and overall business will not realize the benefit of the solution as rapidly.
 - The implementation cost is likely to be higher.

- Why does software project fail after it has passed through acceptance testing?
 - The reasons are:
 - Poor user input
 - Stakeholder conflicts
 - Vague requirements
 - Late failure warning signals
 - Communication breakdowns
 - Hidden costs of going “Lean And Mean”
 - Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a, b, c) and values may be from interval [0, 100]. The program output may have one of the following words. [Not a quadratic equation; Real root, Imaginary roots, Equal roots]. Design test cases to test this program.

Chapter 9

QUALITY MANAGEMENT

─Amir, Sudeep Rawal & Shristi Maharjan

9.1 Quality concept

With development of software it is necessary to develop a concept. How might a software development organization need to control variation? From one project to another, we want to minimize the difference between the predicted resources needed to complete a project and the actual resources used, including staff, equipment, and calendar time. In general, we would like to make sure our testing program covers a known percentage of the software, from one release to another. Not only do we want to minimize the number of defects that are released to the field, we'd like to ensure that the variance in the number of bugs is also minimized from one release to another. (Our customers will likely be upset if the third release of a product has ten times as many defects as the previous release.) We would like to minimize the differences in speed and accuracy of our hotline support responses to customer problems. The list goes on and on.

9.2 Quality Assurance

The terms quality assurance is widely used in manufacturing industry. Quality assurance (QA) is the definition of process and the standards that should lead to high quality products and introduction of quality process into manufacturing process

In software industry, different companies and industry sectors quality assurance is used in different ways; sometimes Q/A simply means the definition of procedures, processes and standards that are aimed at ensuring software quality is achieved. In other cases Q/A also includes all configuration, management, verification and validation activities that are applied after a product has been handed over by a development team.

The QA team in most companies is responsible for managing the release testing process. This means they manage the testing of the software before it is released to customers.

Software Quality Assurance encompasses

- A quality management approach,

- Effective software engineering technology (methods and tools),
- Formal technical reviews that are applied throughout the software process,
- A multitiered testing strategy,
- Control of software documentation and the changes made to it,
- A procedure to ensure compliance with software development standards (when applicable), and
- Measurement and reporting mechanisms.

9.3 SOFTWARE REVIEWS

Software reviews is a “filter” for the software process. This are applied at various points during software engineering and serve to uncover errors and defects that can be removed. Software reviews purify software engineering activities we have called analysis, design and coding,

Reviews can be done in different ways. An informal meeting around a coffee shop is a form of review, if technical problem are discussed. A formal presentation of software design to an audience of customers management and technical staffs is also form of review. A formal technical review is the most effective filter form a quality assurance point.

9.4 FORMAL TECHNICAL REVIEWS

A formal technical review is software quality activity performed by software engineers for following objectives;

- To uncover error in function, logic or implementation for any representation of the software.
- Verify software if it has met is requirement.
- Ensure that software has been represented according to predefined standards.
- To achieve software that is developed in uniform manner.
- To make projects more manageable.

In addition FTR serves as a training ground for junior engineers to observe different approach of software analysis, design and construction. The FTR also serves to promote backup & continuing because number of people becomes familiar with parts of the software. Each FTR is conducted as meeting and will be successful only if properly planned, controlled, and attended.

Review Meeting:

Regardless of the FTR format that is chosen every meeting should abide by following constraints:

- Between 3 to 5 people should be involved in review meeting
- Advance preparation should occur but should be less than 2 hours of work for each person
- The duration of the review meeting should not be more than 2 hours

Given these constraints it should be obvious that an FTR focuses on a specific part of the overall software for example rather than attempting to review an entire design, walkthroughs are conducted for each component the individuals who has developed the work product – the producers- informs the project leader that the work product is complete and a review is required. The project leader contacts a review leader who evaluates the product for readiness, generates copies of product materials and distributes it to two or three reviewers for advance preparation. The reviewer including review leader reviews the product and establishes an agenda for the review meeting. After the review meeting one of the reviewer records the valid problems or errors that are discovered.

At the end of the reviews all the attendees of FTR must decide whether to

1. Whether to accept the product without further modification.
2. Reject the product due to several errors.
3. Accept the product provisionally (with minor errors that could be correct but further additional review is not required).

Review Reporting and Record Keeping

During the FTR, a reviewer (the recorder) actively records all issues that have been raised. In addition a FTR summary report is completed which answers 3 questions:

- Who was reviewed?
- Who reviewed it?
- What were the findings and conclusions?

The review issue is a single page form. The review issue serves two purposes:

1. To identify the problem areas within the product.
2. To serve an action item check list that guides the producer as correctness are made.

Review Guidelines

Guidelines for conducting formal technical reviews must be established in advance all reviews, agreed upon and then followed:

1. Review the product not the producer:
 - FTR should leave warm feeling of accomplishment.
 - Errors should be pointed out gently.
 - Tone of meeting should be loose and constructive.
2. Set an agenda and maintain it.
 - FTR must be kept on track and on schedule
 - FTR must be responsive and shouldn't be afraid to nudge people.

3 Limit debate and rebuttal

- When an issue is raised there may be universal agreement. Rather than spending time debating the question, the issue shouldn't be recorded for further discussion.

4 Enunciate problems but don't try to solve every problem noted:

Review is time for discussing related problems rather than solving problems instantly.

5 Take written notes:

It is good to make notes on wall board so that wording and priorities can be assessed by other reviewers;.

6 Limit the number of participants and insist upon advance preparation:

Number of people must be minimum and review team member must be preparing in advance.

7 Develop a checklist for each product that is likely to be reviewed:

A checklist helps the review leader to structure the FTR meeting and helps each to focus on important issues.

8 Conduct meaningful training for all review:

For effective FTR all participants should receive training. The training should stress both issues and human Psychological side of review.

9 Review your early review:

Debriefing can be beneficial in uncovering problems with the review process itself.

9.5 FORMAL APPROACHES TO SQA

Over past 2 decades software engineering community has argued for more formal SQA is required. A strict and detailed syntax and semantics (meaning of words and sentences) can be defined for every programming language and a strict and detailed approach to the specification of the software requirement is required, if requirement and programming language can be represented in strict and detailed manner then we can apply mathematical proof of correctness to demonstrate that a program conforms exactly to its specifications.

To prove programs correctness we can use Dijkstra and Hoare, Floyd and Witt etc. proofs.

9.6 STATISTICAL SOFTWARE QUALITY ASSURANCE

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

- Information about software defects is collected and categorized.
- An attempt is made to trace each defect to its underlying cause.
- Using the Pareto principle, isolate 20percent.
- Once the vital few causes have been identified move to correct the problems that have caused the defects.

9.7 Software reliability

Ability: The probability that the system, at a point in time, will be operational and able to deliver the requested service.

Reliability: The probability of failure free operation over a specific time, in a given environment, for a specific purpose.

Software reliability is not independent. Hardware failure can generate spurious signals that are outside the range of inputs expected by the software. The software can then behave unpredictably and produce unexpected outputs. These may confuse and consequently stress the system operator. Failure is non-conformance to the software requirement.

For reliability testing, data is gathered from various stages of development, such as the design and operating stages. The tests are limited due to restrictions such as cost and time restrictions. Statistical samples are obtained from the software products to test for reliability of the software. Once sufficient data or information is gathered, statistical studies are done. Time constraints are handled by applying fixed dates or deadlines for the tests to be performed. After this phase, design of the software is stopped and the actual implementation phase starts. As there are restrictions on costs and time the data is gathered carefully so that each data has some purpose and gets its expected precision. To achieve the satisfactory results from reliability testing one must take care of some reliability characteristics. For example Mean Time to Failure (MTTF) is measured in terms of three factors

1. operating time,
2. number of on off cycles,
3. Calendar time.

If the restrictions are on operation time or if the focus is on first point for improvement, then one can apply compressed time accelerations to reduce the testing time. If the focus is on calendar time (i.e. if there are predefined deadlines), then intensified stress testing is used.

Measurement of reliability in **Mean Time between Failures (MTBF)**

$$MTBF = MTTF + MTTR$$

Where,

MTTF: Mean time to failure

MTTR: Mean time to repair

$$\text{Ability} = \frac{MTTF}{MTTF + MTTR} * 100$$

$$\text{Reliability} = \frac{MTTF}{1 + MTBF}$$

Over 225 models have been developed since early 1970s, but how to quantify software reliability still remains unsolved. There is no single model which can be used in every situation. There is no model which either complete or fully developed.

Many software models contain:

1. Assumptions
2. Factors
3. Mathematical function

Software reliability can be divided into categories:

- a. Prediction modeling
- b. Estimation modeling

These modeling techniques follow observation and analyzes with statistical inference.

Prediction Model This model uses historical data. They analyze previous data and some observations. They usually made prior development and regular test phases. The model follow the concept phase and the predication from the future time.

Estimation Model Estimation model uses the current data from the current software development effort and doesn't use the conceptual development phases and can estimate at any time.

9.8 A framework for software metrics

SOFTWARE METRIC 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 ON THE SUCCESS ON THE ECONOMICS OF THE BUSINESSES POWERED. IT REALLY, REALLY MATTERS.

A Framework for Software Metrics

Measures, Metrics, and Indicators. These three terms are often used interchangeably, but they can have subtle differences

- ❖ Measure :Provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- ❖ Measurement: The act of determining a measure
- ❖ Metric:(IEEE) A quantitative measure of the degree to which a system, component, or process possesses a given attribute
- ❖ Indicator: A metric or combination of metrics that provides insight into the software process, a software project, or the product itself

Activities of a Measurement Process

- Formulation: The derivation (i.e., identification) 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 the 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 Matrices for analysis and design model

Metrics for the analysis

- Functionality delivered
 - Provides an indirect measure of the functionality that is packaged within the software
- System size
 - Measures the overall size of the system defined in terms of information available as part of the analysis model
- Specification quality
 - Provides an indication of the specificity and completeness of a requirements specification

Metrics for the Design Model

- Architectural metrics
 - Provide an indication of the quality of the architectural design
- Component-level metrics
 - Measure the complexity of software components and other characteristics that have a bearing on quality
- Interface design metrics
 - Focus primarily on usability
- Specialized object-oriented design metrics
 - Measure characteristics of classes and their communication and collaboration characteristics

9.10 ISO Standard

ISO 9126 is an international standard for the evaluation of software quality.

The standard is divided into four parts, which address, respectively, the following subjects: quality model; external metrics; internal metrics; and quality in use metrics.

Quality Mode The quality model established in the first part of the standard, ISO 9126-1, classifies software quality in a structured set of characteristics and sub-characteristics. These are also considered as nonfunctional requirements metrics. These are:

1-Functionality - A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

- ❖ Suitability
- ❖ Accuracy
- ❖ Interoperability – the capability of different programs to exchange data via a common set of exchange formats, to read and write the same file formats, and to use the same protocols
- ❖ Compliance – the flexibility of the software to accept new features and enhancements.
- ❖ Security – preventing unauthorized access to the software.

2- Reliability - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- ❖ Maturity
- ❖ Recoverability - software product can be modified in order to correct defects, meet new requirements, make future maintenance easier, or cope with a changed environment
- ❖ Fault Tolerance - the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components.

3- Usability - A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.

- ❖ Learnability
- ❖ Understandability
- ❖ Operability - ability to keep a system in a functioning and operating condition.

4- Efficiency - A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

- ❖ Behavior
- ❖ Resource Behavior

5- Maintainability - A set of attributes that bear on the effort needed to make specified modifications.

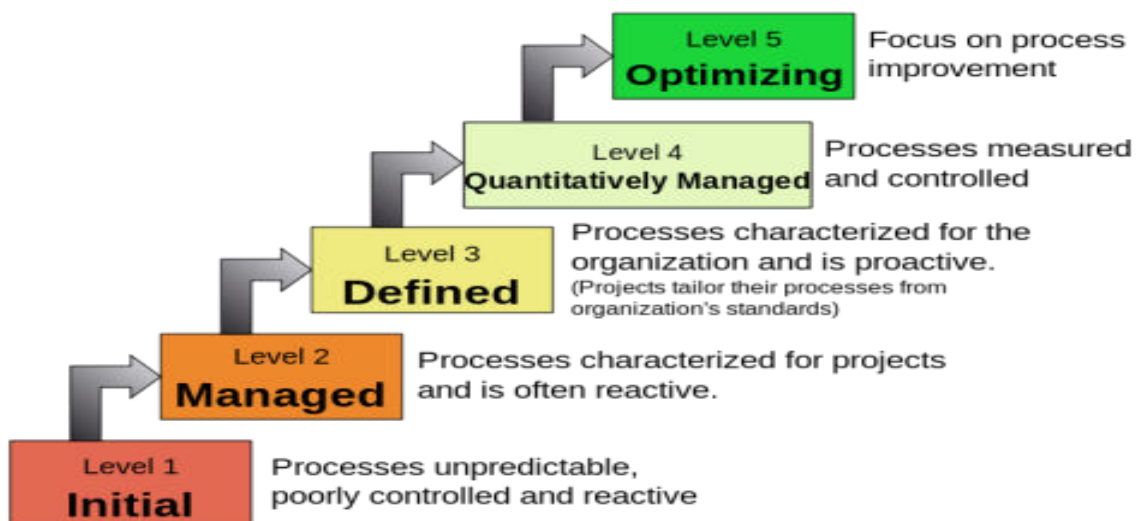
- 3. Stability
- 4. Changeability
- 5. Testability

6- Portability - A set of attributes that bear on the ability of software to be transferred from one environment to another.

- 6. Installability
- 7. Replaceability
- 8. Adaptability

9.11 CMM

Characteristics of the Maturity levels



Capability Maturity Model Integration (CMMI)

is a process improvement approach. CMMI can be used to guide process improvement across a project, a division, or an entire organization. Processes are

rated according to their maturity levels, which are defined as: Initial, Repeatable, Defined, Quantitatively Managed, and Optimizing. CMMI currently addresses three areas of interest:

1. Product and service development — CMMI for Development (CMMI-DEV),
2. Service establishment, management, — CMMI for Services (CMMI-SVC), and
3. Product and service acquisition — CMMI for Acquisition (CMMI-ACQ).

CMMI was developed by a group of experts from industry, government, and the Software Engineering Institute (SEI) at Carnegie Mellon University. CMMI models provide guidance for developing or improving processes that meet the business goals of an organization. A CMMI model may also be used as a framework for appraising the process maturity of the organization. By January of 2013, the entire CMMI product suite was transferred from the SEI to the CMMI Institute, a newly created organization at Carnegie-Mellon.

CMMI originated in software engineering but has been highly generalized over the years to embrace other areas of interest, such as the development of hardware products, the delivery of all kinds of services, and the acquisition of products and services. The word "software" does not appear in definitions of CMMI. This generalization of improvement concepts makes CMMI extremely abstract. It is not as specific to software engineering as its predecessor, the Software CMM.

9.12 SQA plan

The SQA Plan provides a road map for instituting software quality assurance. Developed by the SQA group, the plan serves as a template for SQA activities that are instituted for each software project. A standard for SQA plans has been recommended by the IEEE.

Initial sections describe the purpose and scope of the document and indicate those software process activities that are covered by quality assurance. All documents noted in the SQA Plan are listed and all applicable standards are noted. The management section of the plan describes SQA's place in the organizational structure, SQA tasks and activities and their placement throughout the software process, and the organizational roles and responsibilities relative to product quality.

The documentation section describes (by reference) each of the work products produced as part of the software process. These include:

- Project documents (e.g., project plan)
- Models (e.g., ERDs, class hierarchies)
- Technical documents (e.g., specifications, test plans)
- User documents (e.g., help files)

In addition, this section defines the minimum set of work products that are acceptable to achieve high quality. The standards, practices, and conventions section lists all applicable standards and practices that are applied during the software process (e.g., document standards, coding standards, and review guidelines). In addition, all project, process, and (in some instances) product metrics that are to be collected as part of software engineering⁴² work are listed.

The reviews and audits section of the plan identifies the reviews and audits to be conducted by the software engineering team, the SQA group, and the customer. It provides an overview of the approach for each review and audit.

The test section references the Software Test Plan and Procedure. It also defines test record-keeping requirements. Problem reporting and corrective action defines procedures for reporting, tracking, and resolving errors and defects, and identifies the organizational responsibilities for these activities. The remainder of the SQA Plan identifies the tools and methods that support SQA activities and tasks; references software configuration management procedures for controlling change; defines a contract management approach; establishes methods for assembling, safeguarding, and maintaining all records; identifies training required to meet the needs of the plan; and defines methods for identifying, assessing, monitoring, and controlling risk.

9.13 Software certification

Software certification demonstrates 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 the 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.

Certificates A certificate contains all information necessary for an independent assessment of the properties claimed for an artifact. At its most abstract, a certificate thus has to represent the three entities involved in the certification process,

- (i) the artifact being certified,
- (ii) the property being asserted, and
- (iii) The certification authority.

Certifiable Artifacts Certifiable artifacts include not only the conventional software artifacts (e.g., product families, completed systems, individual

components, or even code fragments) but also supporting non-software artifacts: requirements documents, system designs, component specifications, test plans, individual test cases, scientific and engineering data sets, and others.

Certificate Hierarchies The certificates for an artifact are not an unstructured collection but exhibit some hierarchical structure. This structure is determined by two independent dimensions,

- (i) The system structure, and
- (ii) The certificate types.

The internal structure of a *system* is reflected in the certificate hierarchy.

Certifiable Properties and Certification Authorities Traditional V&V has only addressed a restricted range of formal properties. Realistically, however, software development requires a wide range of notions of software reliability, safety, and validity, each with an appropriate certification authority. This must all be supported by a customizable SCMS.

Release Policies In the context of certification, a release refers to the transition of an artifact into a new defined state: for example, launch, system integration test, alpha and beta testing phases, spiral anchor-point milestones, or code inspection. A release policy formally describes under which conditions an artifact is deemed to be in an adequately certified state and can thus be released safely to another state.

CHAPTER 10

Configuration Management

-Mr. Sanjay Adhikari

10.1 Configuration management planning

The output of the software process is information that may be divided into three broad categories:

1. Computer programs
2. Work products
3. Data

The items that comprise all information produced as part of the software process are collectively called a *software configuration*.

Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems. CM planning defines:

1. the types of documents to be managed and a document scheme.
2. who takes responsibility for the CM procedures and creation baseline
3. policies for the change control and version management.
4. the tools which should be used to assist the CM process and only limitation on their use.
5. the CM database used to record configuration information

CM is useful for the individual projects as it is easy for one person to forget what changes have been made. It is essential for the team projects where several developers are working at the same time on a software system. The CM ensures that teams have access to information about a system that is underdevelopment and do not interfere with each other's work.

The CM of a software system product involves four closely related activities:

1. *Change Management:*
This involves keeping track of requests for the changes to the software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
2. *Version Management:*
This involves 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.
3. *System Building:*
This is the process of assembling program components, data, and libraries, and then compiling and linking these to create an executable system.

4. *Release management:*

This involves preparing software for external release and keeping track of the system versions that have been released for customers use.

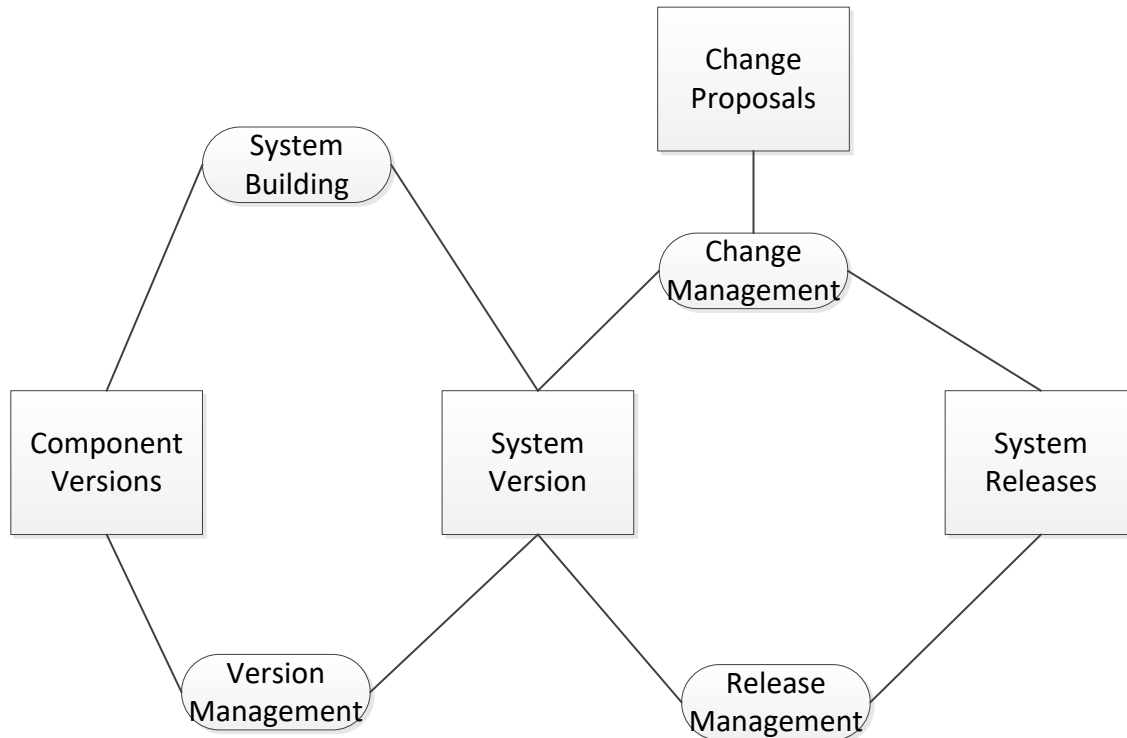


Fig10.1: Configuration activities

10.2 Change Management

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

Software systems are subject to continual change requests:

- ◆ From users, developers, market forces

□ Change management is concerned with keeping track of these changes and ensuring that they are implemented in the most cost-effective way.

Change Management Process

Request change by completing a change request for

Analyze change request

if change is valid **then**

Assess how change might be implemented

Assess change cost

Submit request to change control board

if change is accepted **then**

repeat

make changes to software

submit changed software for quality approval

until software quality is adequate create new system version

else

reject change

request

else

reject change request

Change Management System

- The definition of a change request form is part of the CM planning process.
- This form records the change proposed, requestor of change, the reason why change was suggested and the urgency of change (from requestor of the change).
- It also records change evaluation, impact analysis, change cost and recommendations (System maintenance staff).

10.3 Version and release management

- Invent an identification scheme for system versions.
- Plan when a new system version is to be produced.
- Ensure that version management procedures and tools are properly applied.
- Plan and distribute new system releases.

Versions/variants/releases

- **Version** An instance of a system which is functionally distinct in some way from other system instances.
- **Variant** An instance of a system which is functionally identical but non-functionally distinct from other instances of a system.

- **Release** An instance of a system which is distributed to users outside of the development team.

Version identification

- Procedures for version identification should define an unambiguous way of identifying component versions.
- There are three basic techniques for component identification
 - a) Version numbering
 - b) Attribute-based identification
 - c) Change-oriented identification

A) Version numbering

- Simple naming scheme uses a linear derivation

Eg. V1, V1.1, V1.2, V2.1, V2.2 etc.

- The actual derivation structure is a tree or a network rather than a sequence.
- A hierarchical naming scheme leads to fewer errors in version identification.

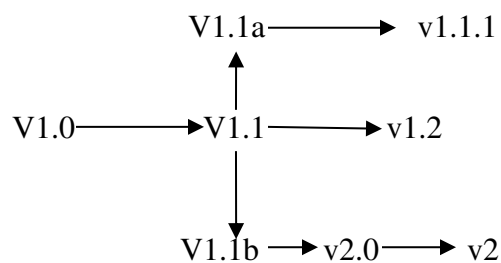


Fig: version numbering

b) 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.
- This is more flexible than an explicit naming scheme for version retrieval; However, it can cause problems with uniqueness - the set of attributes have to be chosen so that all versions can be uniquely identified.
- In practice, a version also needs an associated name for easy reference.

c) 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.

Release management

- Release management 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.

Release problems

- Customer may not want a new release of the System due to unwanted facilities.
- Release management should not assume that all previous releases have been accepted. All files required for a release should be re-created when a new release is installed.

10.4. System Building

System building is the process of creating a complete ,executable system by compiling and linking the system components, external libraries etc. system building tools and version management tools must communicate .Different system are build from different combinations of components, this process is nowadays supported by automated tools that are driven by ‘build scripts ‘and hundreds of files may be used.

System building tools may provide:

- A dependency specification language and interpreter
- Tools selection and investigation support
- Distributed compilation
- Desires object management

System building problems:

- Do the build instructions include all required components?
 - When there are many hundreds of components making up a system, it is easy to miss one out. This should normally be detected by the linker.
- Is the appropriate component version specified?
 - A more significant problem. A system built with the wrong version may work initially but fail after delivery.
- Are all data files available?
 - The build should not rely on 'standard' data files. Standards vary from place to place.
- Are data file references within components correct?
 - Embedding absolute names in code almost always causes problems as naming conventions differ from place to place.
- Is the system being built for the right platform
 - Sometimes you must build for a specific OS version or hardware configuration.
- Is the right version of the compiler and other software tools specified?
 - Different compiler versions may actually generate different code and the compiled component will exhibit different behavior.

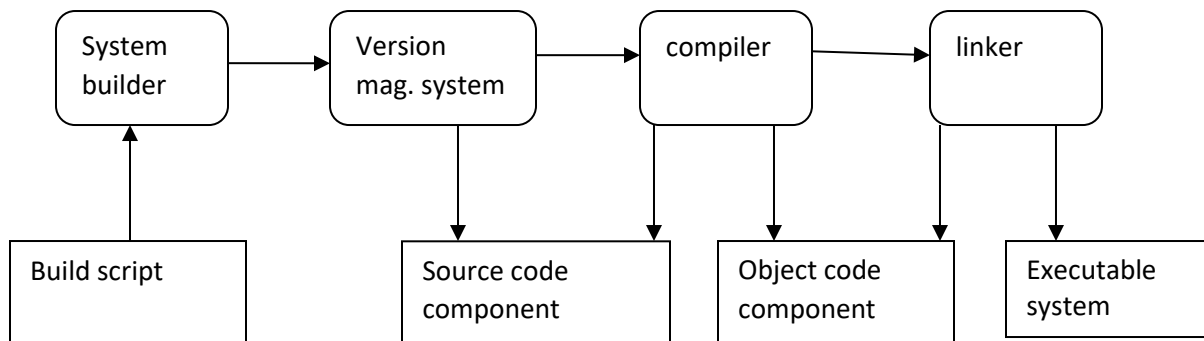


Fig: system building

10.5 CASE tools for configuration management

- CASE tools are available to support all CM activities.
- CM processes are standardized and involve applying pre-defined procedures.
- Large amounts of data must be managed.
- Mature CASE tools to support configuration management are available ranging from stand-alone tools to integrated CM workbenches.

CM workbenches

1. Open workbenches

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

2. 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.

3. Change management tools

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

Change management tools

- Form editor to support processing the change request forms
- Workflow system to define who does what and to automate information transfer
- Change database that manages change proposals and is linked to a VM system.
- Change reporting system that generates management reports on the status of change requests.

4. Version management tools

- **Version and release identification**
 - Systems assign identifiers automatically when a new version is submitted to the system.
- **Storage management**
 - System stores the differences between versions rather than all the version code.
- **Change history recording**
 - Record reasons for version creation.
- **Independent development**
 - Only one version at a time may be checked out for change. Parallel working on different versions.
- **Project support**
 - Can manage groups of files associated with a project rather than just single files.

Key points

- Configuration management is the management of system change to software products.
- A formal document naming scheme should be established and documents should be managed in a database.
- The configuration data base should record information about changes and change requests.
- A consistent scheme of version identification should be established using version numbers, attributes or change sets.
- System releases include executable code, data, configuration files and documentation.
- System building involves assembling components into a system..
- CASE tools are available to support all CM activities
- CASE tools may be stand-alone tools or may be integrated systems which integrate support for version management, system building and change management.

Questions

A restaurant uses an information system that takes customer orders, sends the order to the kitchen, the goods sold and inventory and generates reports for management. List functional and non-functional requirements for this Restaurant Information System. (068 Chaitra)

Explain requirement management process with necessary illustration. (068 Chaitra)

Explain software requirement specification (SRS). What are the characteristics of a good software requirement specification document? (068 Baisakh)

Explain the importance of requirement engineering. List out requirement elicitation techniques. What are the problems in formation of requirements? (067 Asadh)

Distinguish between verification and validation. (066 Bhadra)