

VTU E-LEARNING



Software Engineering

[15CS42]

A Course By:

Prof. Padmashree T
Prof. Arun Kumar Khannur
Prof. Ramasesha
Prof. G.N. Srinivasan

Contents

Software Engineering.....	4
Module-1:Introduction	4
What is software?.....	4
What is Software Engineering?.....	4
Why is software engineering required?	4
Software process activities.....	6
Case studies.....	7
Software Processes.....	8
Requirements engineering process.....	9
Software design and implementation.....	10
Software validation	11
Software Evolution	13
Software Process Model	13
Software specification.....	14
Models.....	14
Requirements Engineering.....	18
Module 2 :System Models	35
Introduction.....	35
What is a Model	35
System Modeling	35
Why Modeling	36
Types of Models	37
How to represent a Model.....	37
UML Diagram Types	37
Universal Modeling Language.....	38
System Model Representation	39
Context Models.....	39
Process Models	40
Interaction Models	41
Sequence Models	43

Structural Models	44
Class Diagrams	45
Behavioral Models	47
Event-driven Modeling	48
State Machine Models.....	49
Model-driven Engineering	51
Agile Methods and MDA.....	52
Design and Implementation	54
Introduction.....	54
Design and Implementation	56
Design Patterns	63
Types of Design Patterns	Error! Bookmark not defined.
Creational Design Patterns.....	65
Software Implementation.....	68
Implementation Process	73
Module 3: Agile Software Development.....	75
Agile Model.....	77
Plan-Driven and Agile Development.....	79
Agile Methods: SCRUM Approach.....	85
Scaling Agile Methods.....	95
Summary.....	97
Module 4: Software Testing.....	98
Software Quality Control- Inspection and Testing.....	98
Verification & Validation techniques.....	102
Inspection Process.....	105
Testing Process.....	111

Software Project Management.....	115
Project Scheduling.....	119
Project Costing and estimation.....	123
Software metrics.....	128

Software Engineering

(15CS42)

Module-1:Introduction

What is software?

A program is a set of instructions that performs a specific task. Software is a set of programs that accomplish a collective functionality.

Properties of software differ from the properties of physical constructs. The abstract and intangible nature of the software makes it different. Software could be complex, difficult to understand and expensive to change depending on the type of software being developed. Depending on the context of its operation different types of software require different approaches for development. It is also possible that software can fail. The failure of software could be due to:

- 1) Increase in demand and
- 2) Low expectation

What is Software Engineering?

Software Engineering is an engineering discipline that involves the education of building software using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints. While doing so, all aspects of software production are considered and not just technical process of development. Also, project management and the development of tools, methods etc. to support software production should also be considered.

Why is software engineering required?

Software engineering principles involve techniques and practices that are used and tested. It is very much required to be able to produce reliable and trustworthy systems economically and quickly. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems.

Software products can be categorised into:

1. Generic products: These are stand-alone systems that are marketed and sold to any customer who wishes to buy them.

Examples – PC software such as graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.

2. Customized products: Software that is commissioned by a specific customer to meet their own needs. Such products are generally used to solve a problem in a specific domain.

Examples – embedded control systems, air traffic control software, traffic monitoring systems.

The attributes of good software are:

Maintainability –Every software developed should be able to meet the changing needs of customers.

Dependability and security- Software developed for any situation should not cause physical or economic damage in the event of system failure. Care should be taken that malicious users should not be able to access or damage the system.

Efficiency – Depending on the context, the developed software would be evaluated based on its responsiveness, processing time and memory utilisation

Acceptability- Any software developed should be understandable, usable and compatible with other systems in the context that work in unison with it.

Software crisis

It is obvious to note that it is very difficult to write efficient software within the right time specified. This could be due to various reasons such as

1. Heterogeneity –increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.
2. Business and social change: Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to rapidly develop new software.

Security and trust also plays a very important role in developing a software module. As software is intertwined with all aspects of our lives, it is essential that we can trust that

software. Hence appropriate security measures need to be taken to safeguard the software from external attack.

Software Engineering Ethics

Software development does not just involve technical skills, but also includes ethics. The team involved in the software development is expected to be honest and ethically responsible for the software at all times. The following factors need to be addressed during, before and after software development.

- a) Confidentiality: It is the employers' responsibility to maintain confidentiality of the employee information and it is the employee responsibility to maintain the secrecy and confidentiality involved in the development of software.
- b) Competence: It is the ethical responsibility of an employee to accept work that matches his technical skills. He should be competent enough to carry out the task assigned to him.
- c) IPR (Intellectual Property Rights) – Very tricky, but very essential. It has to be taken care that when reusing existing components for software development, the components is available for use as per IPR.
- d) Computer misuse- Using the computer provided by the employer for personal use is misconduct and misuse of the system. Watching videos, playing songs, browsing social networking sites all mark the misuse of computer.

Software process activities

Any software developed generally follows the following software process activities:

1. Software specification – Involves gathering requirements for the development of software
2. Software development- involves the implementation of the software.
3. Software validation – verifying if the software developed meets its requirements.
4. Software evolution- making sure that the software can be modified with respect to change in requirements over time.

Case studies

Insulin pump control system:

This is a system that collects data from a blood sugar sensor and calculates the amount of insulin required to be injected. Calculation is based on the rate of change of blood sugar levels. It is programmed to send signals to a micro-pump to deliver the correct dose of insulin.

This system is considered as a safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; and high-blood sugar levels have long-term consequences such as eye and kidney damage. Hence the software involved should function just as expected. The architecture of the system is shown in Fig 1.

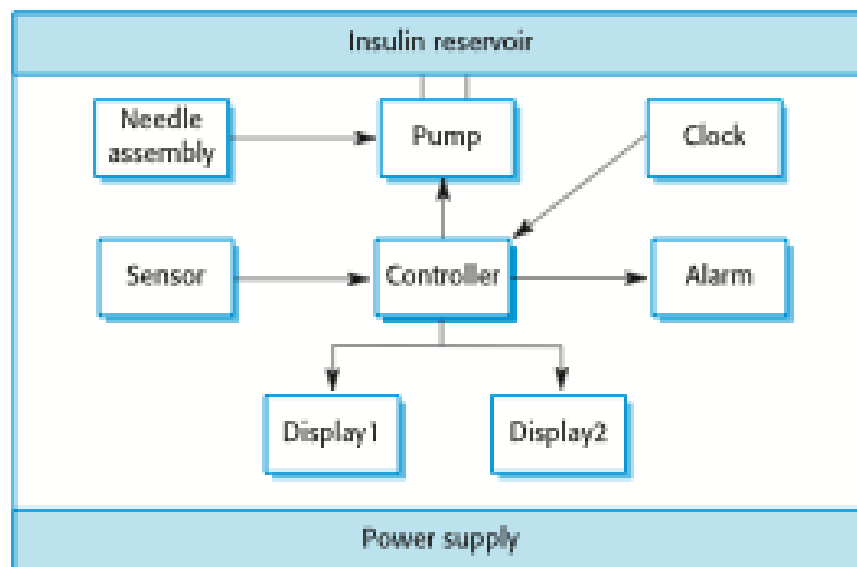


Fig 1: Architecture of insulin pump control system

High-level requirements specifications can be written as follows

1. The system shall be available to deliver insulin when required.
2. The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
3. The system must therefore be designed and implemented to ensure that the system always meets these requirements.

MHC-PMS (Mental Health Care-Patient Management System)

This is a system which makes use of a centralized database of patient information. When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected. Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics. Fig 2 shows the architecture of this system.

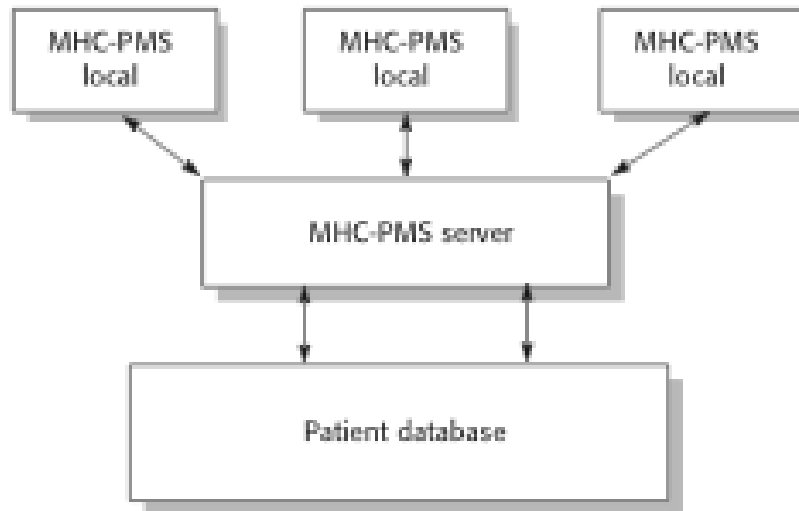


Fig 2: Architecture of MHC-PMS

The system is expected to support the following functionality:

1. Individual care management
2. Patient monitoring
3. Administrative reporting
4. Other Aspects that need to be considered other than functional are:

Privacy - patient information is confidential. At most care need to be taken to protect the privacy of the patient and his illness details.

Safety - prescribe the correct medication to patients.

Software Processes

Software processes are an integral part of software process models. Various process activities that are carried out during every phase of development form the software activities. To define technically, A software process is a structured set of activities required to develop a software system. The activities by and large include:

1. Specification – defining what the system should do;
2. Design and implementation – defining the organization of the system and implementing the system;
3. Validation – checking that it does what the customer wants;
4. Evolution – changing the system in response to changing customer needs.

Requirements engineering process

Requirements engineering process consists of the following activities:

- a) Feasibility study – Technical and Financial feasibility of the project needs to be considered.
- b) Requirements elicitation and analysis – Expectation from the system. The functionalities that the system is expected to provide.
- c) Requirements specification – Defining requirements for every activity performed by the user or set of users and the expected outcome for each of these activities.
- d) Requirements validation – Checking validity of the requirements. Verifying if the requirements specification contains the functionalities that the system is expected to perform.

The requirements engineering process is depicted in Fig 3.

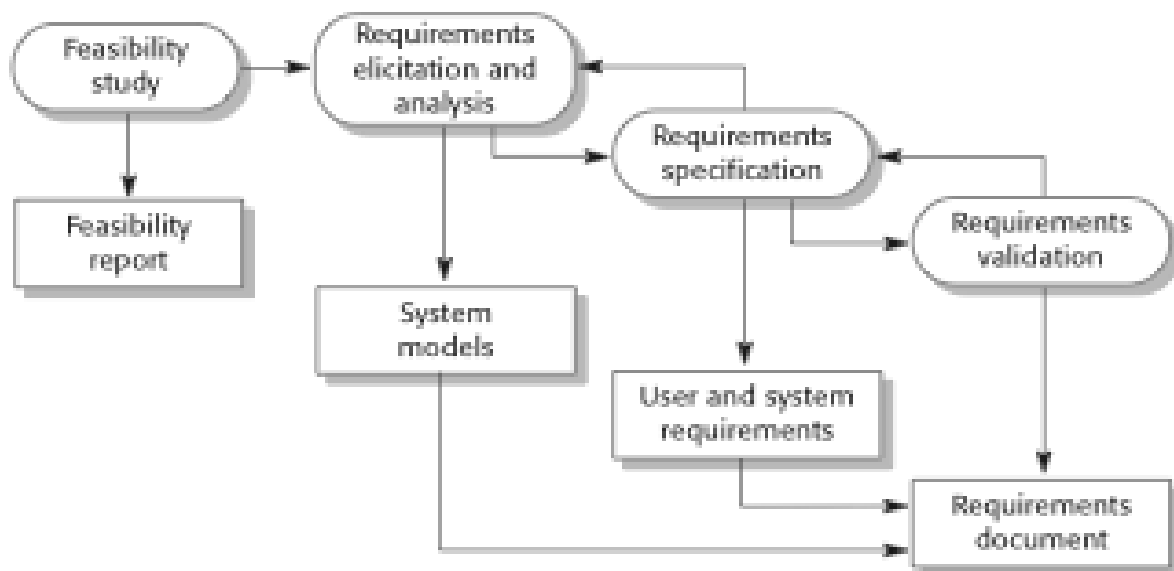


Fig 3: Requirements Engineering Process

Once the feasibility study has been completed a feasibility report is generated. It is observed that the requirements elicitation and analysis, requirements specification and requirements validation are iterative activities. Based on the analysis it is required to decide on a system model that needs to be incorporated as a part of requirements document. The requirements specification activity gives the complete user and system requirements. The requirements validation process checks if the system performs the required functions and the requirement specification covers all functionalities to be provided by the system. The outcome of each activity is an input to the requirements document.

Software design and implementation

Once the requirements for specific software have been gathered the next activity is to design and develop the software. These activities could be performed individually or in parallel depending on the project. Each of these activities have a set of process activities to be carried out. They are described in detail below:

Software design

The major aim of this phase of software development is to design a software structure that realises the specification as given by the requirements specification document.

Model of the design process

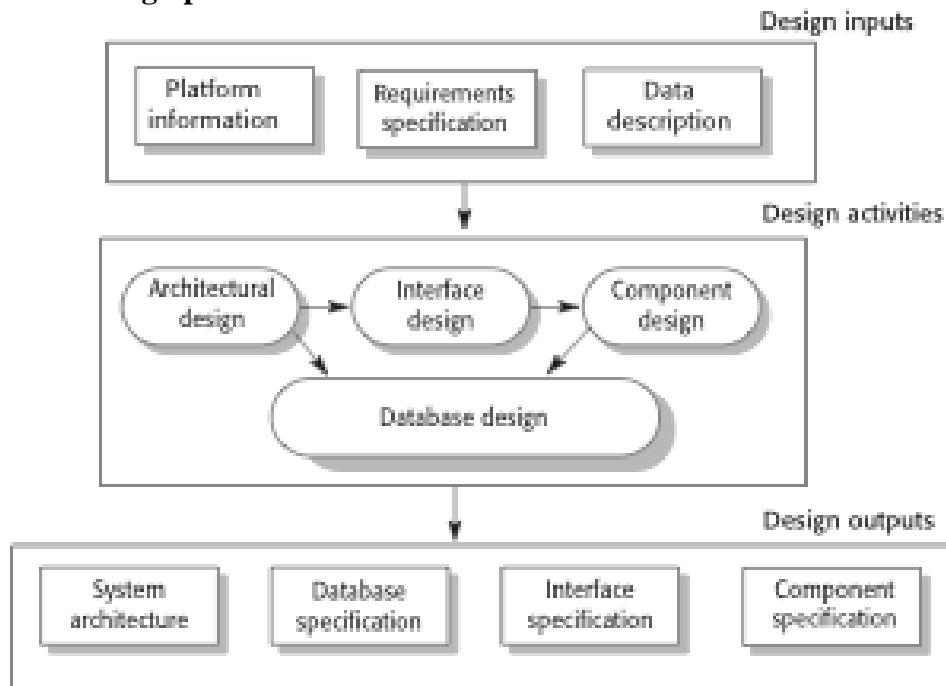


Fig 4: Design process and activities involved

Design activities involve Architectural design, interface design and component design. The system architecture and its components need to be defined and specified as a part of architectural design. The interface design involves the specification and working of interfaces for enables inter and intra component communication.

Implementation

Translate this structure into an executable program. This requires the use of a programming language. The choice of programming language is based on the complexity of the software, the requirements specification of the software, the reusability of components in the software etc.

Software validation

Validating the software involve checking for conformation of the system to its specification. This also involves checking and review processes and system testing. System testing involves executing the system with test cases. The system testing involves Component testing and acceptance testing. Component testing involves testing of each component that is used to build the software. The interfaces between the components and

the working of components with the existing system code needs to be tested. The process activities involved in testing and its iterations are shown in Fig 5. Every individual component is tested and then system is tested as a whole. Acceptance testing is done to determine if the system is functioning well in the domain of its application.



Fig 5: Testing activities

The various phases of testing are depicted below in Fig 6. It is a wrong notion that testing only begins after implementation is complete. As a fact, testing is carried out throughout the development process. The figure below depicts the outcome of each activity in the phase of testing.

Testing phases

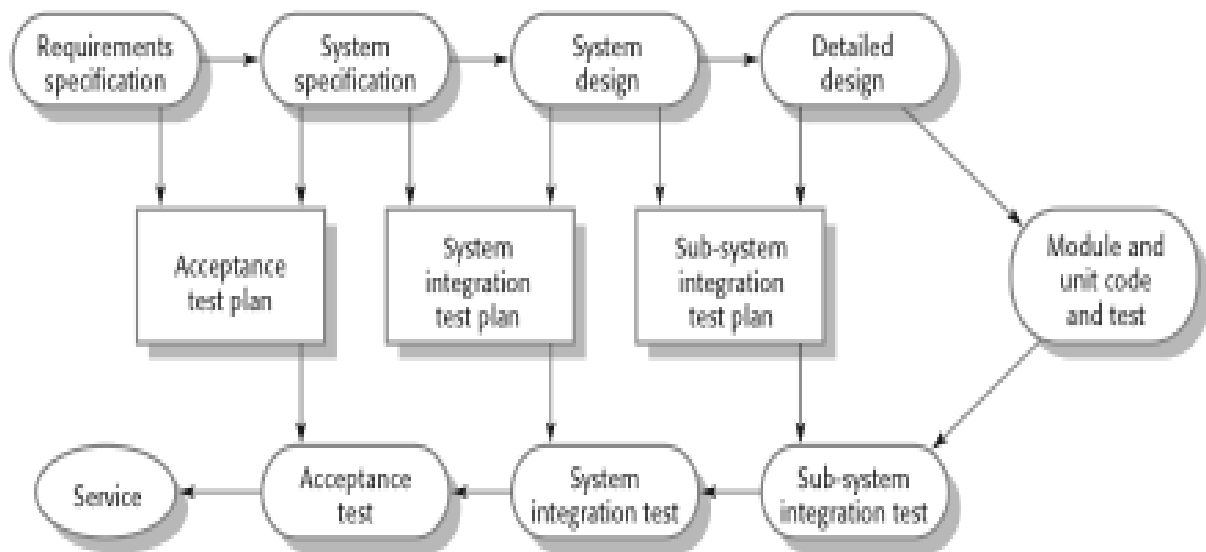


Fig 6: Phases of testing

As and when the requirements specification is complete the acceptance test plan is also parallel written because the acceptance testing involves testing of the software in its working domain. Acceptance test cases are written at this stage. The test plan gets its

input from both requirements specification as well as system specification activities. The system integration test plan is an output of system specification and design. Workings of individual components, testing for interfaces between them are all a part of system integration testing. Once the detailed design is available the sub system integration test plan is also drafted. Later on the tests are accomplished by making use of the test plan. This is also called as V-model for test plan driven software development process.

Software Evolution

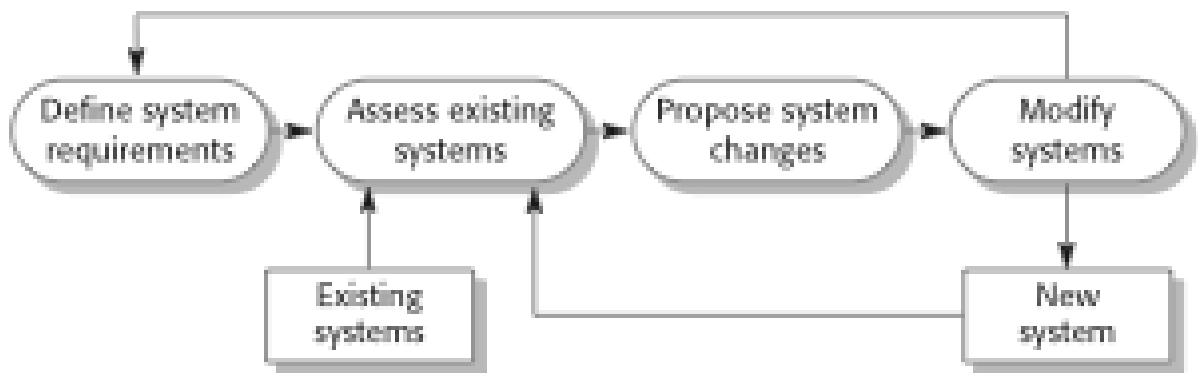


Fig 7: Activities in software evolution

It is obvious that software is inherently flexible and can change. As requirements change through changing business circumstances, the software that supports the business must also evolve and change. Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new. Fig 7 depicts the activities involved in software evolution. Existing system is continuously assessed based on the system requirements. If any change has been proposed the system needs to be modified and a new system is then released. This is an iterative process and continues until the required outcome has been obtained.

Software Process Model

A software process model is an abstract representation of a process. The process descriptions may also include:

1. Products, which are the outcomes of a process activity;

2. Roles, which reflect the responsibilities of the people involved in the process.
3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced.

Process activities

Processes are inter-leaved sequences of activities with the overall goal of specifying, designing, implementing and testing a software system. The four basic process activities of software development are organized differently in different development processes depending on the application being developed. The activities are: specification, development, validation and evolution. Each of these is explained in detail in the rest of the chapters.

Software specification

The specification of the software requires answering the following questions:

- a) What are the services required by the software? – To get the functionalities of the software to be developed.
- b) What are the constraints on system operation?- To get the domain information of the software to be developed.

Models

- a) The waterfall model: This is a plan-driven model. There are separate and distinct phases of specification and development.
- b) Incremental development: This is model where the specification, development and validation are interleaved. Incremental development may follow either plan-driven approach or agile approach.
- c) Reuse-oriented software engineering: The system is assembled from existing components. This approach also may be plan-driven or agile.

There is no hard rule on the model chosen for developing a particular application. In practice, most large systems are developed using a process that incorporates elements from all of these models. Each of these models are described in detail below:

Water fall model

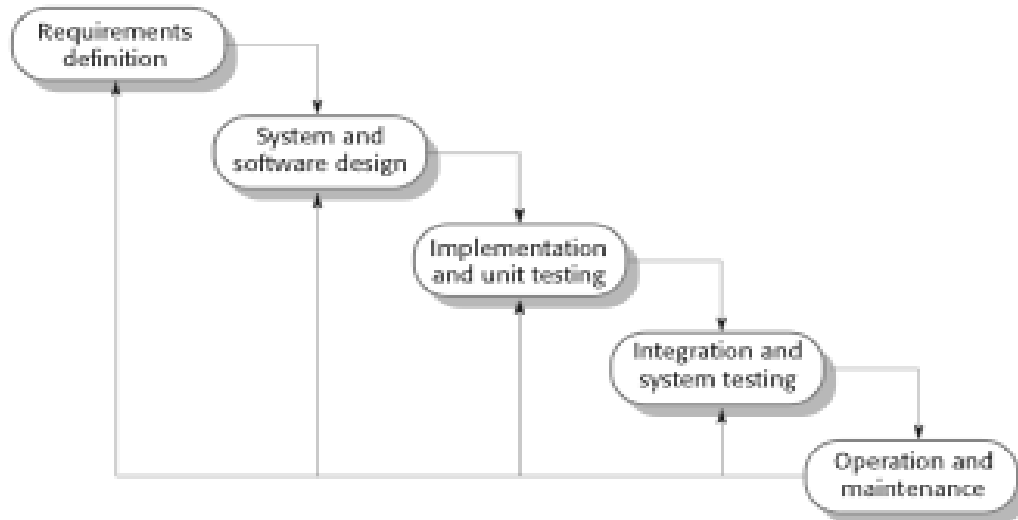


Fig 8: Waterfall model

The phases of Waterfall model depicted in Fig 8 are:

- a) Requirements analysis and definition – Gathering the requirements and defining the functionality of each requirement. This could be done through various techniques discussed in the next section.
- b) System and software design- Design and architecting the software as a whole. The components involved, the interface between them and other design considerations are depicted as diagrams. UML diagrams are most frequently used design representations.
- c) Implementation and unit testing – Developing the system and testing each module one by one as a single unit. Every component developed or reused are individually tested.
- d) Integration and system testing- The components are integrated together and the system is tested as a whole. The interfaces and communication modules between the systems are tested here.

- e) Operation and maintenance- Once the system is put into use, the functionality of the system in the operational domain needs to be taken care of. The maintenance and operation of the system at the client end is important.

Problems with Waterfall model

The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements. The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.

Incremental development and delivery

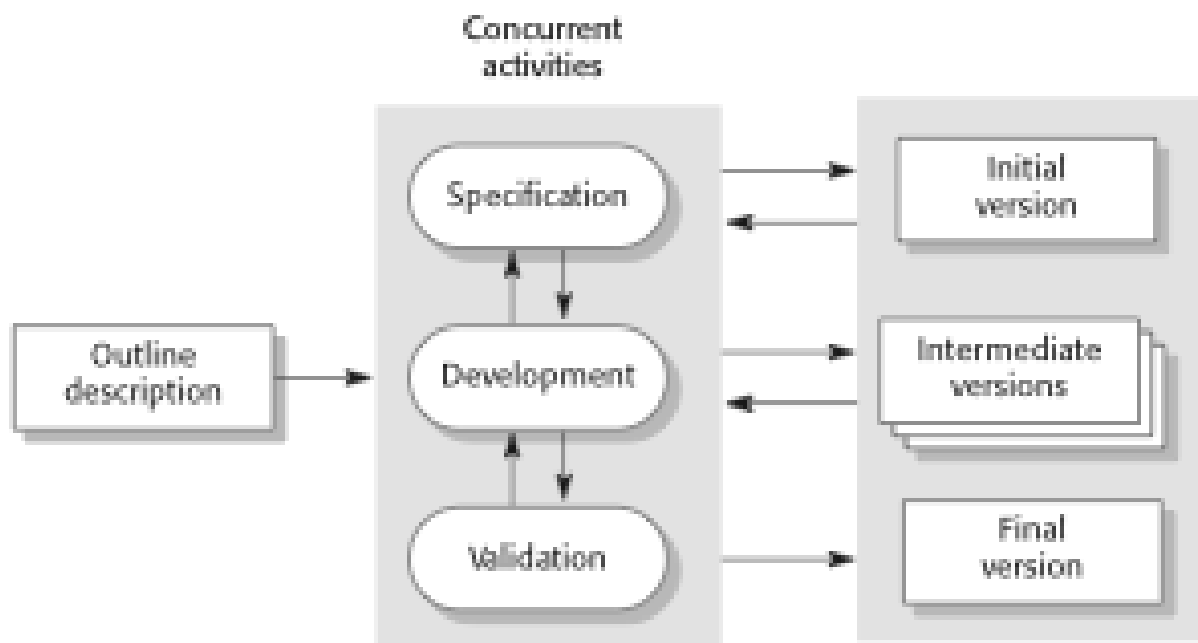


Fig 9: Process in incremental development

Incremental development involves developing the system in increments and evaluating each increment before proceeding to the development of the next increment. This is the normal approach used in agile methods. The evaluation of software developed using incremental approach is done by user/customer proxy.

Incremental delivery involves deploying an increment for use by end-users. This is a more realistic evaluation about practical use of software. It is difficult to implement for replacement systems as increments have less functionality than the system being replaced.

Advantages and disadvantages

Cost of accommodating changing customer requirements is reduced. Customer feedback on the development is obtained as and when an increment is ready to be released. Rapid delivery and deployment of useful software can be accomplished by making use of this approach.

The Problems with this approach is that the process is not visible. Also, it is inevitable that system structure tends to degrade as new increments are added.

Boehm's spiral model

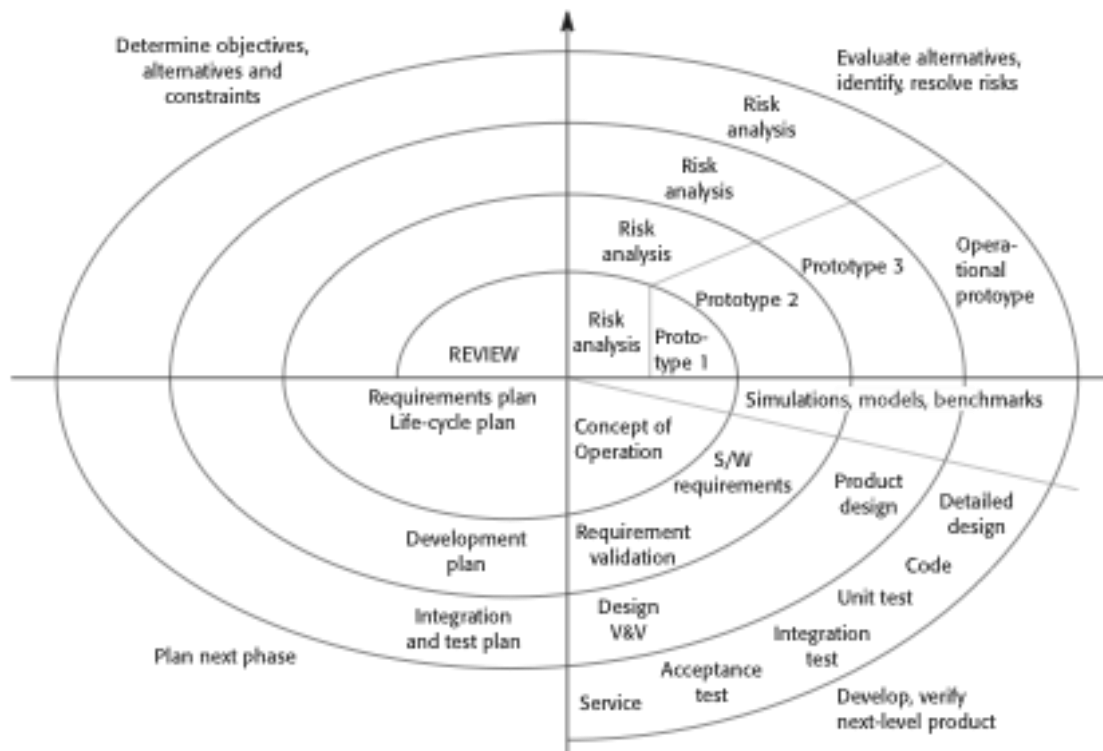


Fig 10: Boehm's spiral model

Fig 10 shows the Boehm's spiral model. Here the process is represented as a spiral rather than as a sequence of activities with backtracking. Each loop in the spiral represents a phase in the process. There are no fixed phases such as specification or design. The loops in the spiral are chosen depending on what is required. Risks are explicitly assessed and

resolved throughout the process. Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development. In practice, however, the model is rarely used as published for practical software development.

Sectors of spiral model

The following are the sectors of the spiral model:

1. Objective setting -Specific objectives for the phase are identified.
2. Risk assessment and reduction-Risks are assessed and activities put in place to reduce the key risks.
3. Development and validation-A development model for the system is chosen which can be any of the generic models.
4. Planning-The project is reviewed and the next phase of the spiral is planned.

Requirements Engineering

Requirements engineering definition: The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed. The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process. A requirement may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification. This is inevitable as requirements may serve a dual function as described in the following scenarios:

- 1) May be the basis for a bid for a contract - therefore must be open to interpretation;
- 2) May be the basis for the contract itself - therefore must be defined in detail;
- 3) Both these statements may be called requirements.

Types of requirements:

Requirements can be basically categorised into:

1. User requirement which are statements in natural language plus diagrams of the services the system provides & its operational constraints. Basically specifies external system behavior. These are the requirements which are written for customers

2. Systems Requirements are a structured document setting out detailed descriptions of the system's functions, services & operational constraints. It is necessary that the systems requirements should reflect accurately what the customer wants and should also precisely define what should be implemented. This could be a part of a contract between client and contractor

Example of a user requirement:

In a Library management system, the following functionalities are expected to be present

1. The system will maintain records of all library items including books, serials, newspapers, magazines, video and audio tapes, reports, collections of transparencies, computer discs and CD-ROMs.
2. Paper-based library items are stored on open shelves in the library and the system records their reference position in the library.
3. No item will be removed from the library without the details of its borrowing being recorded in the system.
4. All items will have a bar code containing a unique reference number by which an item can be identified within the system.

For the same application the example of a System requirement could be:

1. The system will permit all users to search for an item by title, by author or by ISBN
2. Staff will be able to search for an item by bar code ref. number
3. Books can be borrowed for 15 days while CD-ROMs, Audio tapes & reports can be borrowed only for 3 days
4. Borrowed items that are one day overdue in their return will cause a reminder letter to be printed
5. Librarian should be able to find out details like Number of books & materials borrowed (on a given day, by a given client)
6. Selected items may be temporarily blocked by authorized staff

Readers of different types of requirements specification

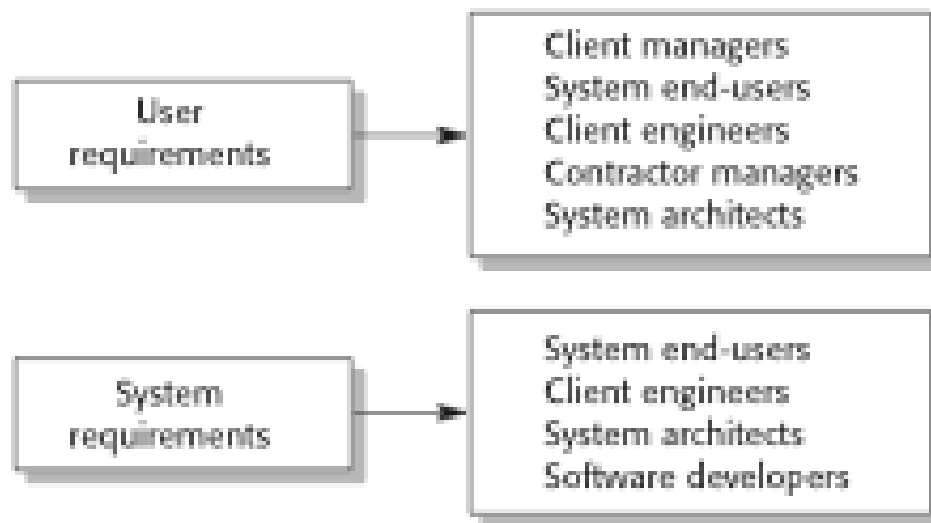


Fig 11: Readers of requirements

Fig 11 shows that the users of User requirements and system requirements are generally the System end-users, client engineers and system architects. Additionally, Contract managers and client managers will be using the user requirements and the system requirements are the guiding path for system developers.

Classification of requirements – Another way

1. Functional requirements: Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. These requirements may also state what the system should not do.
2. Non-functional requirements: Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
3. These requirements often apply to the system as a whole rather than individual features or services.

4. Domain requirements: Constraints on the system from the domain of operation of the final system.

Each of the above requirements is explained in detail.

Functional Requirements:

Describe functionality or system services and depend on the type of software, expected users and the organization where the software is used. This could be high-level statements of what the system should do. However, it should describe all the system services in detail which could include its inputs, its Outputs, exceptions and so on. Such requirements are generally described in fairly abstract but precise way.

Non-Functional Requirements:

Define system properties and constraints e.g. reliability, response time and storage occupancy, Security, etc.. Alternatively they may define platform constraints like, I/O devices capability, data representations. Process requirements may also be specified mandating a particular CASE system, programming language or development method. These requirements arise through organizational policies, budget limits, interoperability needs etc. They may be considered to be more critical than functional requirements because if these are not met, the system will be useless.

Non-functional requirements can be further classified as shown in Fig 12

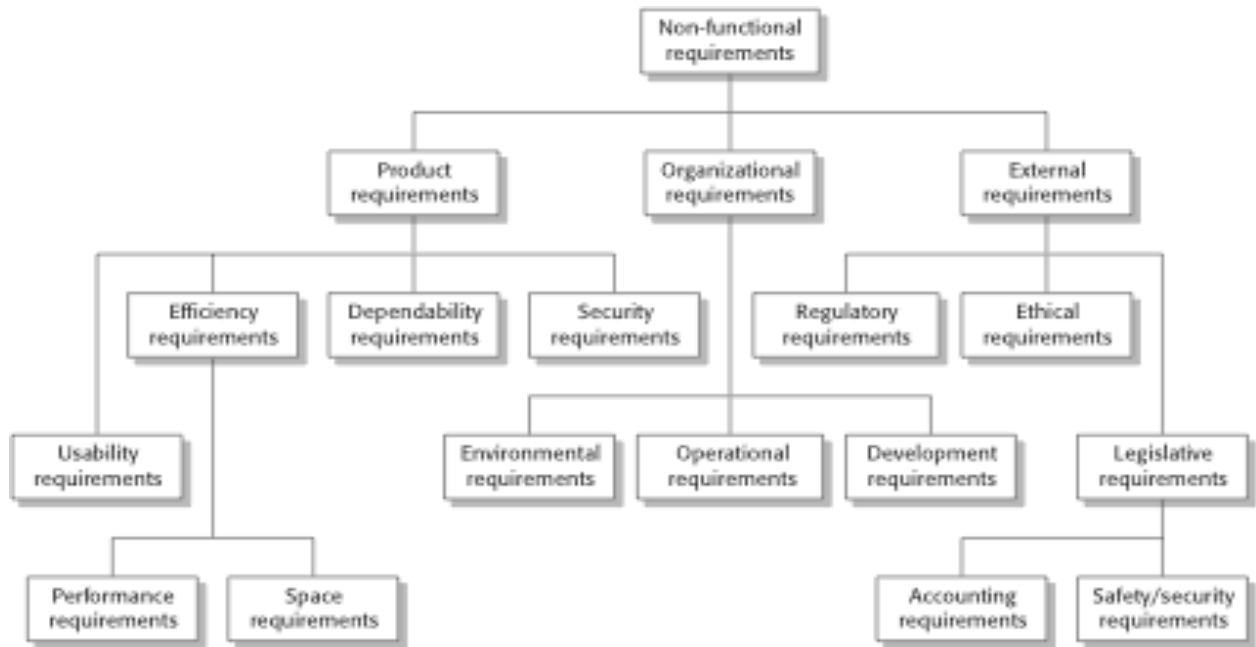


Fig 12: Classification of Non-functional requirements

The classification can be described as given below:

1. Product requirements: Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
2. Organisational requirements: Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
3. External requirements: Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required. It may also generate requirements that restrict existing requirements.

Metrics for specifying nonfunctional requirements

Since non-functional requirements are equally important for the operation of a software metrics have been defined to indicate non-functional requirements as a part of requirements specification document. This is indicated in Table 1.

Table 1 : Metrics for non-functional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

The software requirements document

The software requirements document is the official statement of what is required of the system developers. It should include both a definition of user requirements and a specification of the system requirements. It is not a design document. As far as possible, it should be a set of what the system should do rather than how it should do it.

Fig 13 shows the users of a requirements document. The illustration is self-explanatory. It should be noted that requirements specification is not just for end users or for developers. It is for all those entities who are involved through out the software development process.

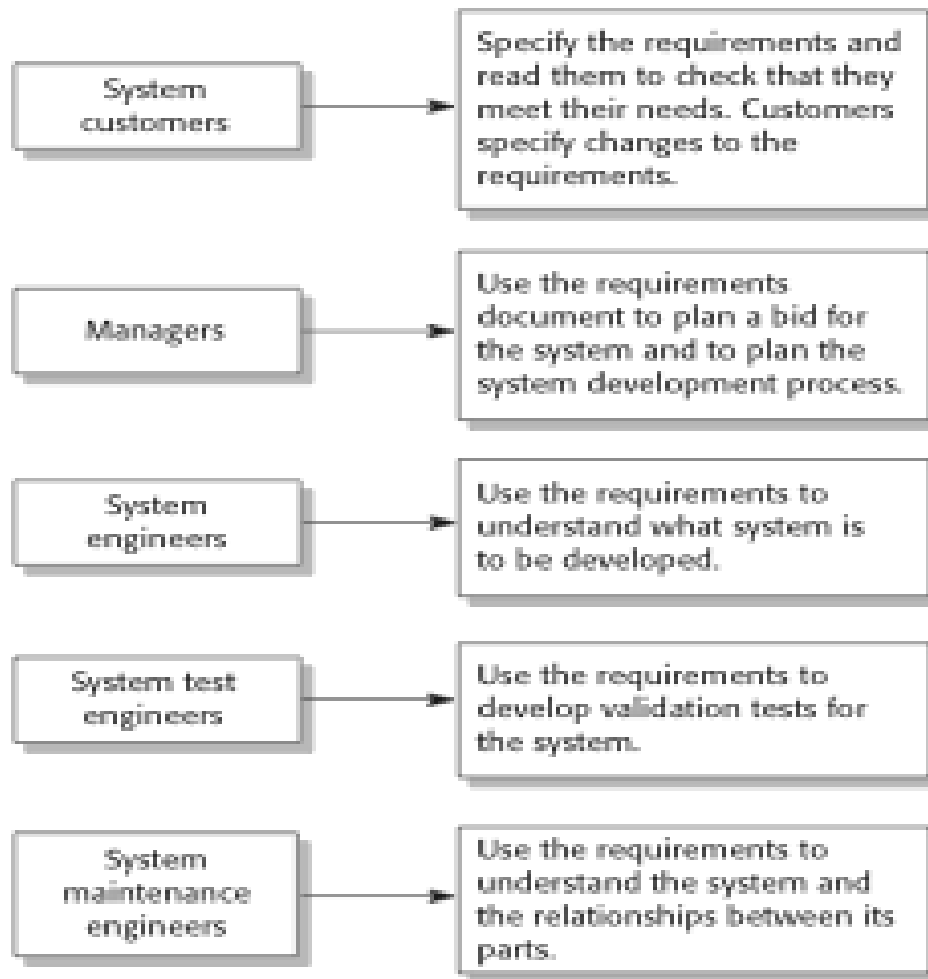


Fig 13: Users of a software requirements document

IEEE structure of a requirements document

The table below describes the various chapters and its description for a standard requirements document. This is given by the IEEE. All software do not necessarily follow this exact structure. But a standard structure tuned to the needs of the application is required to be followed.

Table 2: The IEEE structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.

Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.
-------	---

Ways of writing system requirements specification

There are different ways of writing requirements specification. Table 3 describes the notations generally used and its description:

Table 3: Notations for writing requirement specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Tools for Specifying System Requirements

This section deals with the ways to write requirements specification.

1. Structured language: A language with constructs similar to programming

Example: *System Requirements for Address book*

Case : 1

List Option selected

Show first 3 addresses

If down arrow pressed

scroll the addresses

else

If any “Alpha key pressed

Display the first address starting from that alpha

End if

Endif

2. Form Based approach: Creates a standard format for specifying requirements.

Typically can have entries like :

1. Definition of the function or entity.
2. Description of inputs and where they come from.
3. Description of outputs and where they go to.
4. Indication of other entities required.
5. Pre and post conditions (if appropriate)

This method eliminates problems of natural language. This also brings in uniformity & comprehensiveness. Not always useful Example specifying interactions)

3. Tabular Model

This is also used to supplement natural language. This is particularly useful when you have to define a number of possible alternative courses of action. Example:

Condition	Action
Sugar level falling ($r_2 < r_1$)	CompDose = 0
Sugar level stable ($r_2 = r_1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing ($(r_2 - r_1) < (r_1 - r_0)$)	CompDose = 0
Sugar level increasing and rate of increase stable or increasing. ($(r_2 - r_1) \geq (r_1 - r_0)$)	CompDose = round $((r_2 - r_1) / 4)$ If rounded result = 0 then CompDose = MinimumDose

4. Graphical Model (like sequence diagram of UML)

This is most useful when state changes need to be depicted OR Sequence of actions & interactions need to be described.

Sample Requirements Document

A typical SRS would look like the following example taken from a weblink free to download document.

To be strictly used for Educational purpose only:

A document of Global Digital Megacorp Student Information Management System available on:

web.uvic.ca/~cloke/Seng321Designer/SENG321-2008_Group4_RS1.0.doc

Requirements Engineering (RE) processes

The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements. However, there are a number of generic activities common to all processes which are:

1. Feasibility Study
2. Requirements elicitation;
3. Requirements analysis;
4. Requirements validation;
5. Requirements management.

In practice, RE is an iterative activity in which these processes are interleaved.

The spiral view of requirements engineering process is shown in Fig 14



Fig 14: Spiral view of Requirements engineering process

1. Feasibility Study : Decides whether or not the proposed system is worthwhile attempting. A short focused study that checks the following:

- a) If the system contributes to organisational objectives;
- b) If the system can be engineered using current technology and within budget;
- c) If the system can be integrated with other systems that are used
- d) If the system can fit into the cultural framework of the organizational culture and acceptable to all the stake-holders

Information collection is done by asking questions to the stake-holders of the system

Typical Questions could be:

- a) What if the system wasn't implemented?
- b) What are current process problems?
- c) How will the proposed system help?
- d) What will be the integration problems?

- e) Is new technology needed? What skills?
- f) What facilities must be supported by the proposed system?

2. Requirement Elicitation

Involves Interacting with technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints. This may involve stakeholders like end-users, managers, Engineers involved in maintenance, domain experts, trade unions, etc. Domain requirements are also discovered at this stage.

Domain Requirements are derived from the application domain rather than specific needs of a customer in that domain. They usually refer to specialized domain terminology / concepts. They describe system characteristics & features that reflect the domain.

Domain requirements could be

1. New functional requirements,
2. Constraints on existing requirements or
3. Define specific computations.

If domain requirements are not satisfied, the system may be unworkable.

The requirements elicitation and analysis process

Fig 15 shows the process of requirements elicitation and analysis.

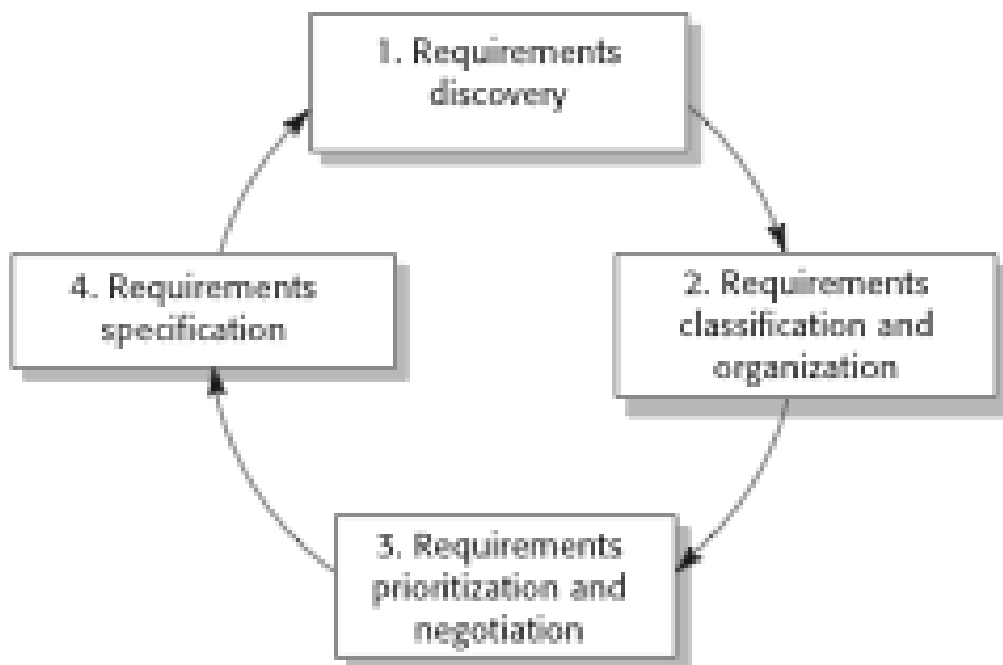


Fig 15: Requirements elicitation and analysis process

Requirements Elicitation is done through

1. Interviewing: The RE team puts questions to stakeholders about the system that they use and the system to be developed. There are two types of interviews:
 Closed interviews where a pre-defined set of questions are answered.
 Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.
 Normally a mix of closed and open-ended interviewing is good for getting an overall understanding of what stakeholders do & how they interact with the system. This method is not good for understanding domain requirements
2. Observation & study: Observation (in-situ): The RE team observes the manual process in action, in-situ and infers required information. This is good for process oriented systems. Minimal disturbance to user staff.
 Study- Additional information is gathered from study of documents, forms manuals , rulebooks and other artifacts used by the actors of the system . Here the analyst must be well experienced with the domain

Problems with Requirement Elicitation

1. Stakeholders don't know what they really want.
2. Stakeholders express requirements in their own terms.
3. Different stakeholders may have conflicting requirements.
4. Organisational and political factors may influence the system requirements.
5. The requirements change during the analysis process
6. New stakeholders may emerge and the business environment change

Activities of Requirement Analysis

1. Requirements classification and organisation : Grouping related requirements and organising them into coherent clusters
2. Prioritisation and negotiation: Prioritising requirements and resolving requirements conflicts requirements documentation
3. Requirements are documented and input into the next round of the spiral

Requirements validation

This is concerned with demonstrating that the requirements define the system that the customer really wants. Requirements error costs are high so validation is very important. Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking

To check the requirements for a software, the following parameters need to be considered:

1. Validity. Does the system provide the functions which best support the customer's needs?
2. Consistency. Are there any requirements conflicts?
3. Completeness. Are all functions required by the customer included?
4. Realism. Can the requirements be implemented given available budget and technology
5. Verifiability. Can the requirements be checked?

Requirements validation techniques

Various ways exist for validating the requirements:

1. Requirements reviews: This involves systematic manual analysis of the requirements.
2. Prototyping: This involves using an executable model of the system to check requirements.
3. Test-case generation: Developing tests for requirements to check testability.

Requirements reviews

It is essential that regular reviews should be held while the requirements definition is being formulated. Both client and contractor staff should be involved in reviews. Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks

Parameters for review checks consist of checking for the following:

1. Verifiability: Is the requirement realistically testable?
2. Comprehensibility: Is the requirement properly understood?
3. Traceability: Is the origin of the requirement clearly stated?

4. Adaptability: Can the requirement be changed without a large impact on other requirements?

Requirements management

Process of understanding and controlling the changing requirements during the requirements engineering process and system development forms the major activity of requirements management. Requirements are inevitably incomplete & inconsistent. New requirements emerge during the process as business needs change and a better understanding of the system is developed. Different viewpoints have different requirements and these are often contradictory. All this needs reconciliation & management.

Reasons for change in Requirements

1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery; new features may have to be added for user support if the system is to meet its goals.
3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements management planning

This activity establishes the level of requirements management detail that is required. It is essential that Requirements management decisions need to be taken for the following purposes:

1. Requirements identification- Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.

2. A change management process -This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. Traceability policies- These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
4. Tool support- Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Module 2 :System Models

Introduction

The aim of this Session is to introduce system modeling concepts that may be developed as part of the requirements engineering and system design processes.

At the end of the session, the students will:

1. Understand What is Modelling
2. Justify why Modeling is required before building a software system
3. Appreciate fundamental system modeling perspectives of context, interaction, structure, and behavior
4. Be aware of the ideas underlying model-driven engineering,

Difference between structural and behavioral models

- ✓ Software Systems are abstract and intangible and hence tend to be complex
- ✓ A Model represents essential characteristics of a complex system
- ✓ System modeling is the process of developing (abstract) models of a system, with each model presenting a view of that system
- ✓ A system model represents aspects of a system and its environment
- ✓ Modeling Helps Understand Information **systems**

What is a Model

A **Model** is a *simplified* representation of either reality or vision. Since “a picture is worth a thousand words,” most models use pictures to represent the reality or vision. Usually, the system model becomes the blueprint for designing and constructing an improved system.

A Model is a simplified representation of a complex system

System Modeling

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

Since “a picture is worth a thousand words,” most models use some kind of graphical notation representing a system, which is now almost always based on notations in the Unified Modeling Language (UML).

Model-driven analysis is a problem-solving approach that emphasizes the drawing of graphical or pictorial system models to document and validate both existing and/or proposed system.

Benefits of System Modeling

1. Ease project management tasks.
2. Can provide complete views of a system, as well as detailed views of subsystems.
3. Clarify structures and relationships.
4. Offer a communication framework for ideas within and between teams.
5. Can generate new ideas and possibilities.
6. Allow quality assurance and testing scenarios to be generated.
7. Are platform independent.

Why Modeling

Modeling is required to

1. Understand the existing software application to do any enhancement
2. Derive the requirements for a New software application
3. Discuss Different Design proposals to optimize the solution architecture
4. Document a Software Systems Structure and Operations to create Manuals
5. To Create Test Cases early in the Software Development Life Cycle
6. Modeling is used to
 - i. Conceptualize,
 - ii. Understand and
 - iii. Communicate

the functioning of a complex software system to stakeholders

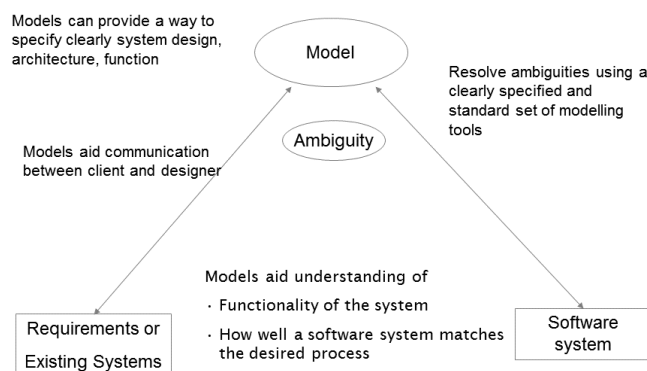


Fig 1.1 Models of a Software System

Types of Models

Different models may represent a system from different perspectives. For example:

1. An external perspective model representing the context or environment of the system
2. An interaction perspective Model where the interactions between a system and its environment or between the components of a system is represented.
3. A structural perspective model showing the organization of a system or the structure of the data that is processed by the system.
4. A behavioral perspective, where the model shows the dynamic behavior of the system and how it responds to events.

How to represent a Model

1. System Models are Usually represented graphically and so are the software system Models.
2. Graphical models are very popular because they are easy to understand and construct.
3. The **Unified Modeling Language (UML)** provides a standard for the artifacts of development (semantic models, syntactic notation, and diagrams
4. UML is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system.
5. The creation of UML was originally developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software in 1996.
6. In 2005 UML was also published by the International Organization for Standardization (ISO) as an approved ISO standard.
7. The UML standard is being periodically revised

UML Diagram Types

UML diagrams represent two different views of a system model

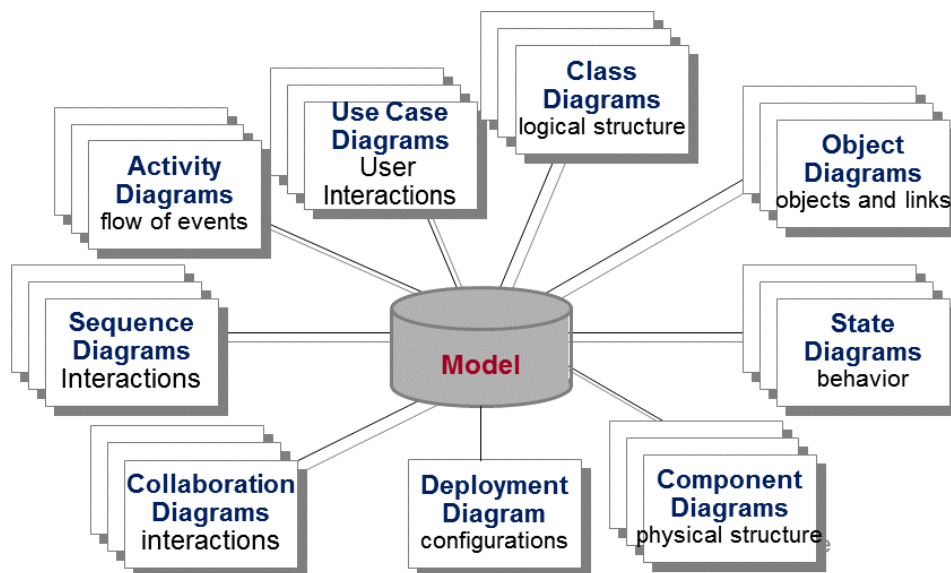
- i. Static (or *structural*) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. It includes class diagrams and composite structure diagrams.

- ii. Dynamic (or *behavioral*) view: emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

Universal Modeling Language

In building a visual model of a system, many different diagrams are needed to represent different views of the system. The UML provides a rich notation for visualizing our models. This includes the following key diagrams:

1. Use Case diagrams to illustrate user interactions with the system.
2. Class diagrams to illustrate logical structure.
3. Object diagrams to illustrate objects and links.
4. State diagrams to illustrate behavior.
5. Component diagrams to illustrate physical structure of the software.
6. Deployment diagrams to show the mapping of software to hardware configurations.
7. Interaction diagrams (i.e., collaboration and sequence diagrams) to illustrate behavior.
8. Activity diagrams to illustrate the flow of events in a use case.



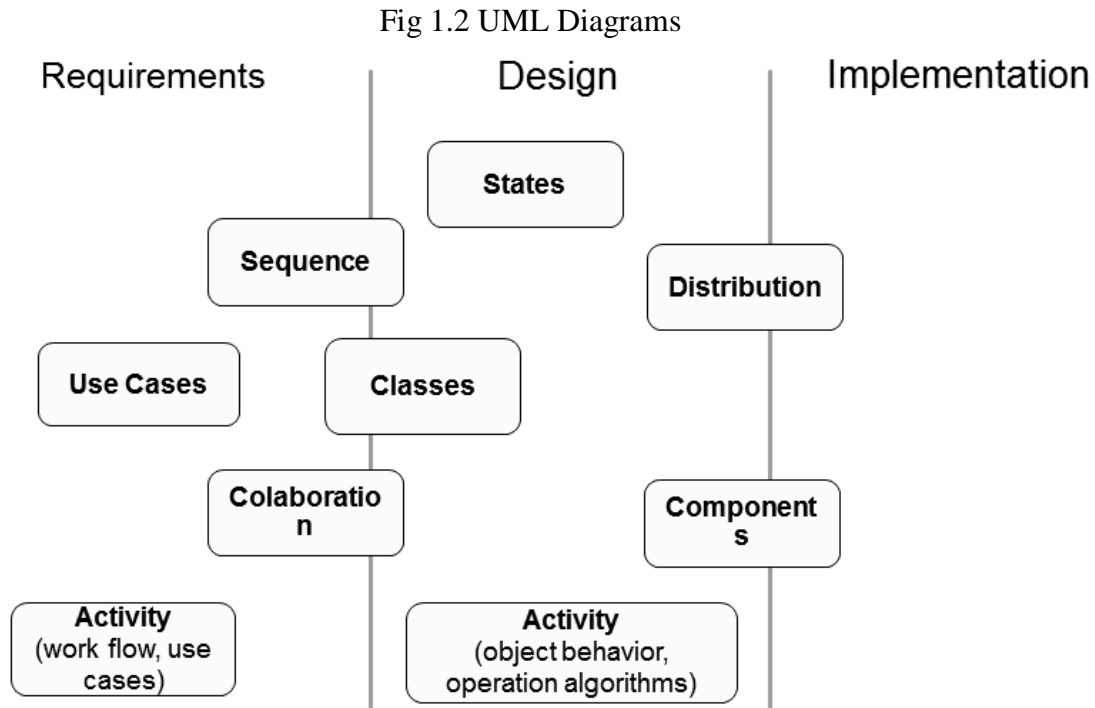


Fig 1.3 : UML Diagrams at different stages of SDLC

System Model Representation

1. Context models
2. Interaction models
3. Structural models
4. Behavioral models
5. Model-driven engineering

Context Models

- A context model is a model that shows how a system fit into the context of the environment.
- Shows the scope and boundaries of a system at a glance including the other systems that interface with it
- No technical knowledge is assumed or required to understand the diagram
- Easy to draw and amend due to its limited notation

- Easy to expand by adding related systems
- Can benefit a wide audience including stakeholders, business analyst, data analysts, developers
- Context models provide an overview (abstraction) of an entire system, and shows the most important aspects.
- Details are not included.
- Context models are most useful in the requirements analysis and design stages.
- A context model is a model that shows how a system fit into the context of the environment.
- Context models provide an overview (abstraction) of an entire system, and shows the most important aspects.
- Details are not included.
- Context models are most useful in the requirements analysis and design stages.

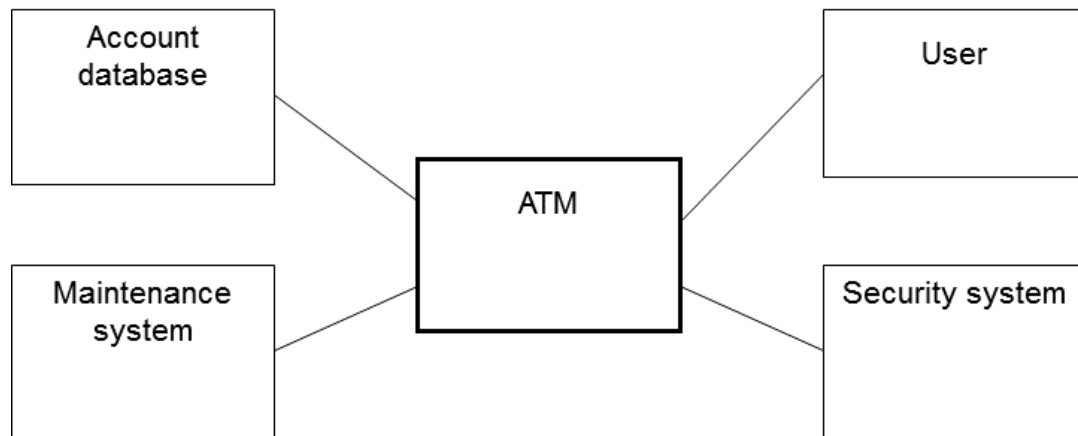


Fig 1.4 Context Model of an ATM showing the external perspective

Process Models

- Context models simply show the other systems in the environment, NOT how the system being developed is used in that environment.
- Process models reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.

Activity Diagrams

Activity diagrams are intended to show the *activities that make up a system process* and the flow of control from one activity to another.

- The start of a process is indicated by a filled circle; the end by a filled circle inside another circle.
- Rectangles with round corners represent activities, which are sub-processes that must be carried out.
- A solid bar is used to indicate activity coordination.

When the flow from more than one activity leads to a solid bar then all of these activities must be complete before progress is possible.

When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Arrows may be annotated with guards that indicate the condition when that flow is taken.

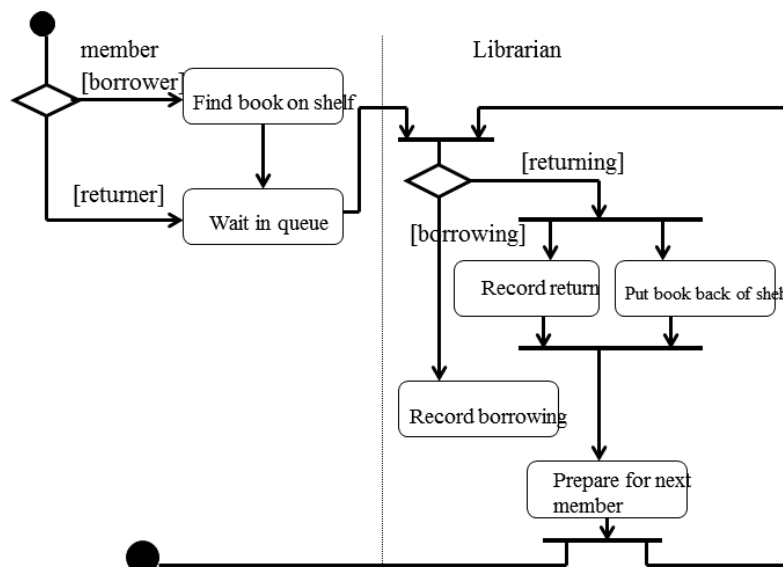


Fig 1.5 Activity Diagram of a Library

Interaction Models

- Interaction Models help to identify user requirements.

- Interaction Modeling helps to understand the communication process between system-to-system interaction
- Modeling system interaction helps us understand and improve the system performance and dependability.
- Use case diagrams and sequence diagrams may be used for interaction modelling.

Use Case Modeling

“A *use case* specifies the behavior of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.”

- *The UML User Guide, [Booch,99]*

“An *actor* is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that outside users may have with the system.”

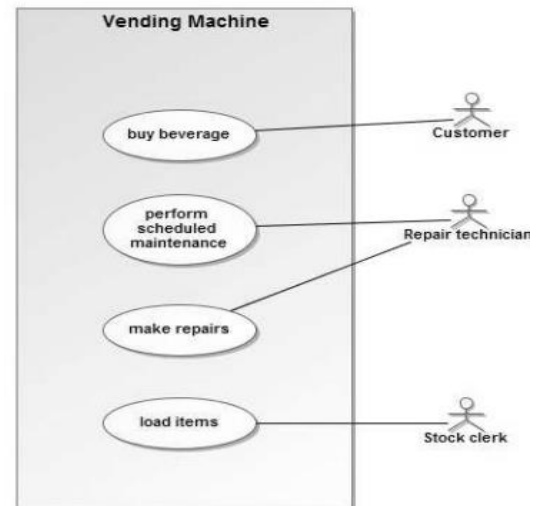
- *The UML Reference Manual, [Rumbaugh,99]*

- Use cases were developed originally to support requirements elicitation and now incorporated into the UML.
- Each use case represents a discrete task that involves external interaction with a system.
- Actors in a use case may be people or other systems.
- Represented diagrammatically to provide an overview of the use case and in a more detailed textual form.
- An actor is a direct external user of a system
- An object or a set of objects that communicates directly with the system but that is not part of the system
- Modelling the actors helps to define a system by identifying the objects within the system and those on its boundary
- An actor is directly connected to the system
- An indirectly connected object is not an actor and should not be included as part of the system model
- Any interaction with an indirectly connected object must pass through actors

Examples

- Customer and Repair Technician are actors of a vending machine
- Traveler, Agent and Airline are actors of a travel agency system
- User and Administrator are actors for a computer database system
- Actors can be persons, devices and other systems (anything that interacts directly with the system)
- the Dispatcher of repair technicians from a service bureau is not an actor of a vending machine

- UML has a graphical notation for
- summarizing use cases into use case diagrams
- A rectangle contains the use cases for a system with the actors listed on the outside
- The name of the system is written near a side of the rectangle
- A name within an ellipse denotes a usecase
- A “stick man” icon denotes an actor with the name placed below the icon
- Solid lines connect use cases to participating actors



Sequence Models

Sequence models show the sequence of object interactions that take place between the actors and the objects within a system

Sequence diagrams are part of the UML and are used to represent the sequence model.

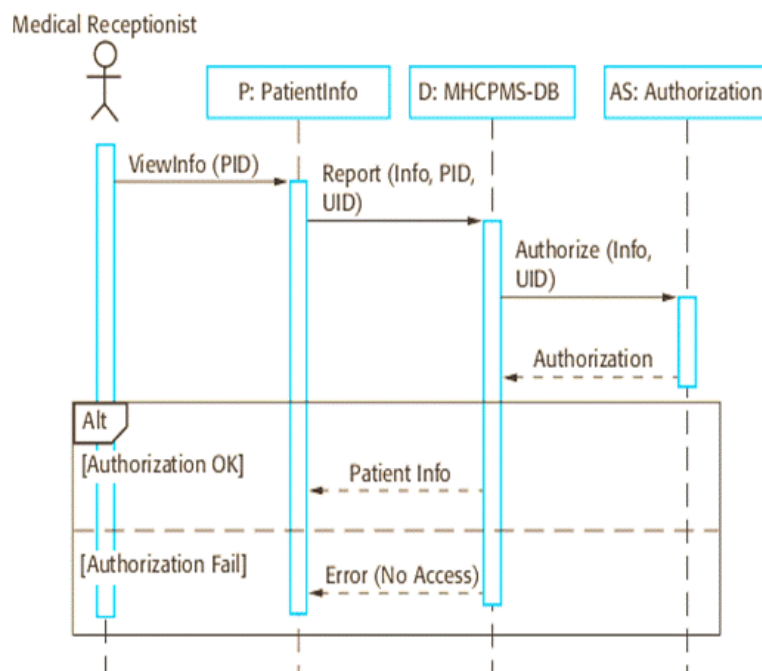
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Time is represented vertically so models are read top to bottom
- Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction

- A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system

Drawing Sequence Diagrams

- Determine the context of the sequence diagram
- Identify the **object** that are participate in the sequence
- Set of the **lifeline** for each object
- Lay out of **messages** from the top to the bottom of the diagram based on the order in which they sent
- Add the execution occurrence to each object's lifeline
- Validate** the sequence diagram

Sequence Diagram of a Patient Information System



1. The medical receptionist triggers the *ViewInfo* method in an instance *P* of the *PatientInfo* object class, supplying the patient's identifier, *PID*. *P* is a user interface object, which is displayed as a form showing patient information.
2. The instance *P* calls the database to return the information required, supplying the receptionist's identifier to allow security checking.
3. The database checks with an authorization system that the user is authorized for this action.
4. If authorized, the patient information is returned and a form on the user's screen is filled in. If authorization fails, then an *error message* is returned.

Structural Models

Structural models show the organization of a system in terms of the components that make up that system and their relationships.

Structural models 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.

Structural models are created during discussion and designing the system architecture

Class Diagrams

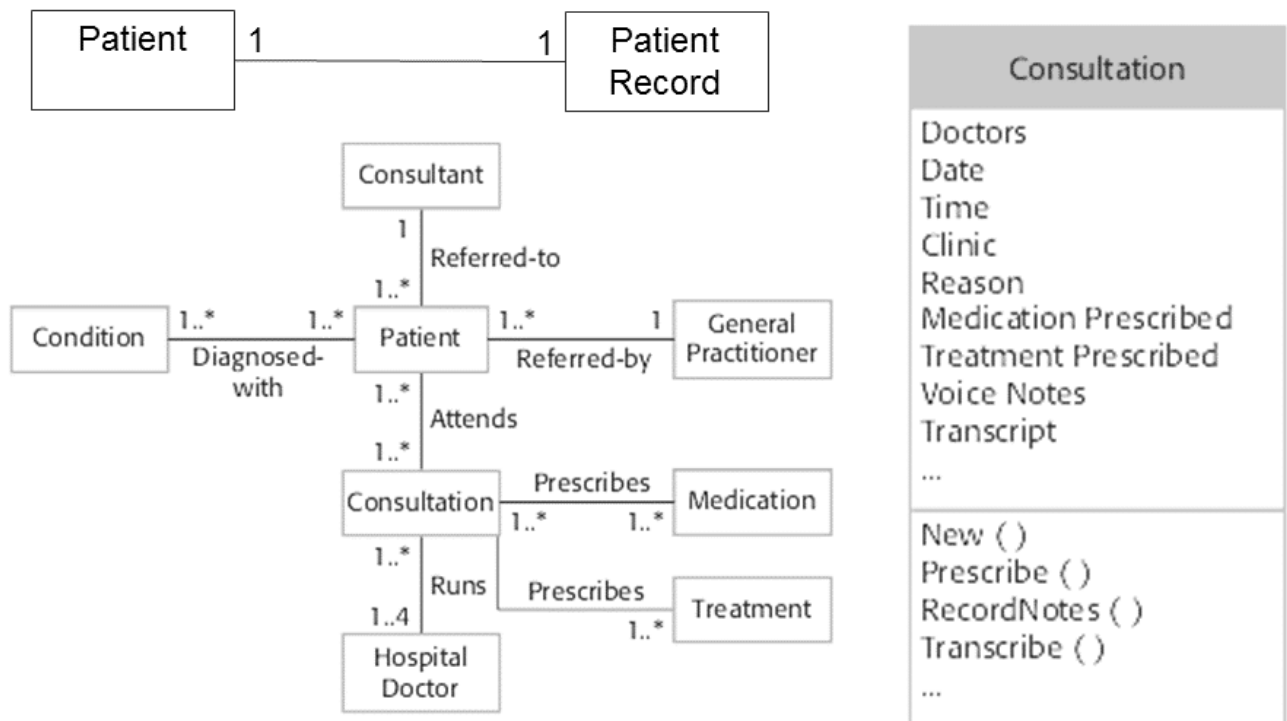
Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.

An object class can be thought of as a general definition of one kind of system object.

An association is a link between classes that indicates that there is some relationship between these classes.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

UML Classes and Associations



Consultation
Doctors
Date
Time
Clinic
Reason
Medication Prescribed
Treatment Prescribed
Voice Notes
Transcript
...
New ()
Prescribe ()
RecordNotes ()
Transcribe ()
...

Generalizations

Complexity is managed by Generalization Technique.

Instead of detailed characteristics of every event that we experience, we generalise these experiences into general classes (animals, cars, houses, etc.) and learn the characteristics of these classes.

As different members of these classes have some common characteristics (e.g. squirrels and rats are rodents), it will be easier to understand (and design) similar events by co-relating the events and rebuilding the scenario.

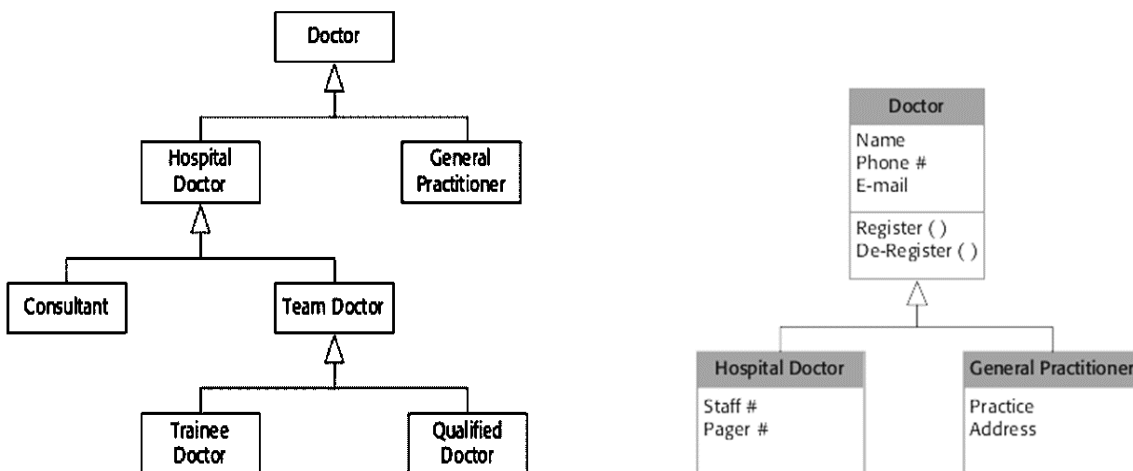
In modeling systems, one of the common technique is to identify the features of classes with scope for generalization. If changes are proposed, then there will be no need to look at all classes in the system to see if they are affected by the change.

In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

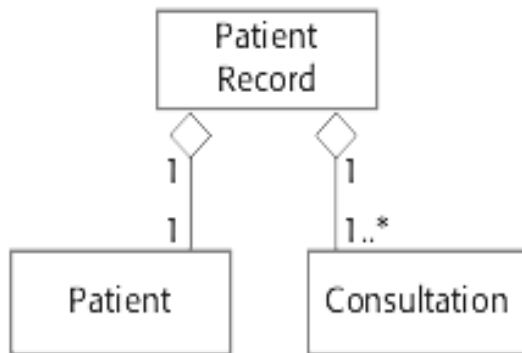
In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.

The lower-level classes are subclasses and inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

A Generalization Heirarchy



Object Class Aggregation Models



Behavioral Models

Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

These stimuli may be of two types:

- i. Data Some data arrives that has to be processed by the system, (Data Driven)
- ii. Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case. (Event Driven)

Data Driven Modeling

Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.

Data-driven models show the sequence of actions involved in processing input data and generating an associated output.

They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

Data Flow Modeling

Data-flow models are used to show how data flows through a sequence of processing steps. For example, a processing step could be to filter duplicate records in a customer database.

The data is transformed at each step before moving on to the next stage.

These processing steps or transformations represent software processes or functions when data-flow diagrams are used to document a software design.

They are simple and intuitive

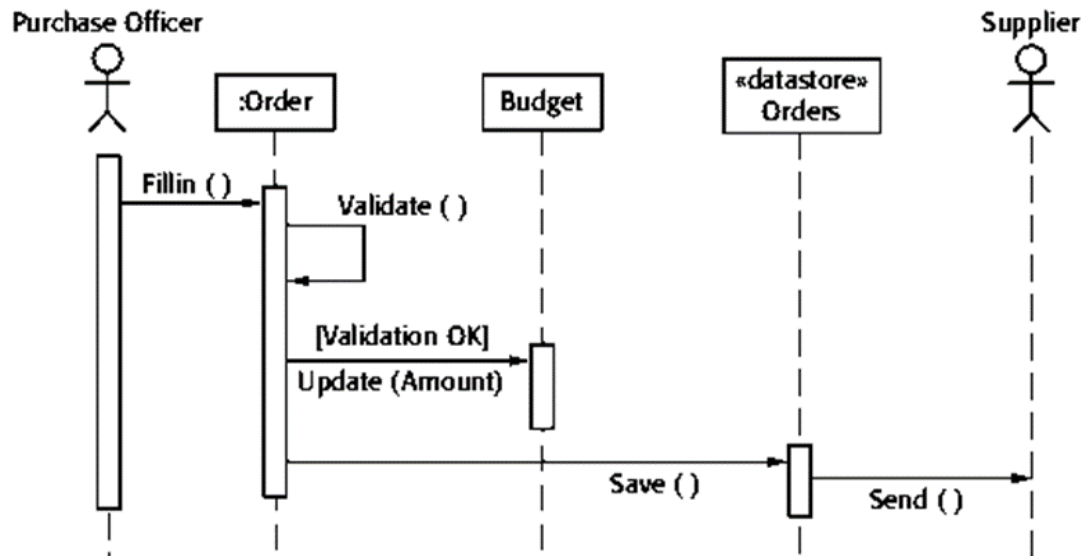


Fig Order Processing Sequence Diagram

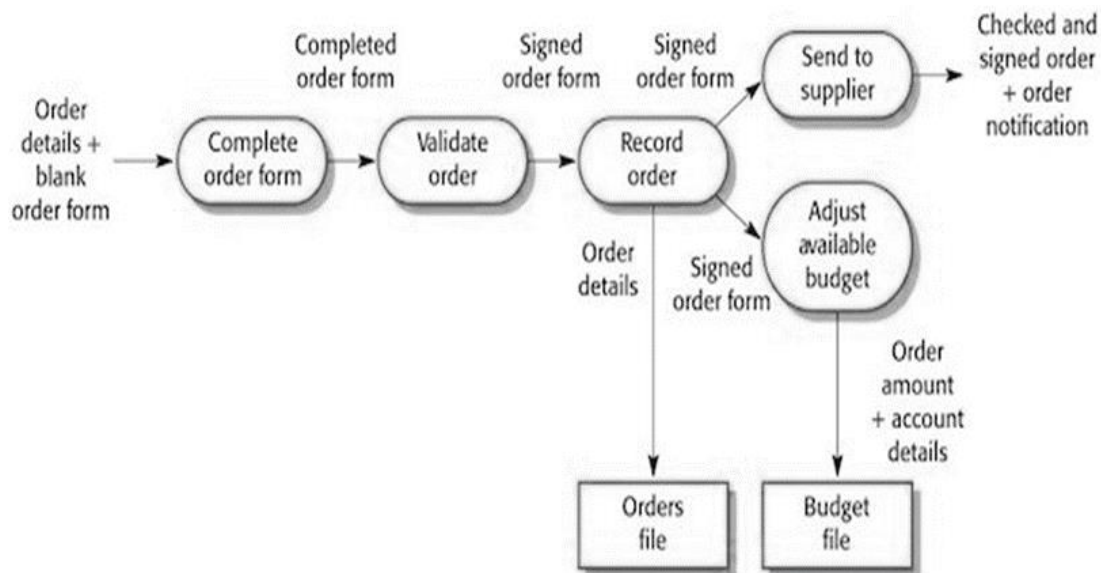


Fig Order Processing Sequence Diagram

Event-driven Modeling

Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.

Event-driven modeling shows how a system responds to external and internal events.

It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

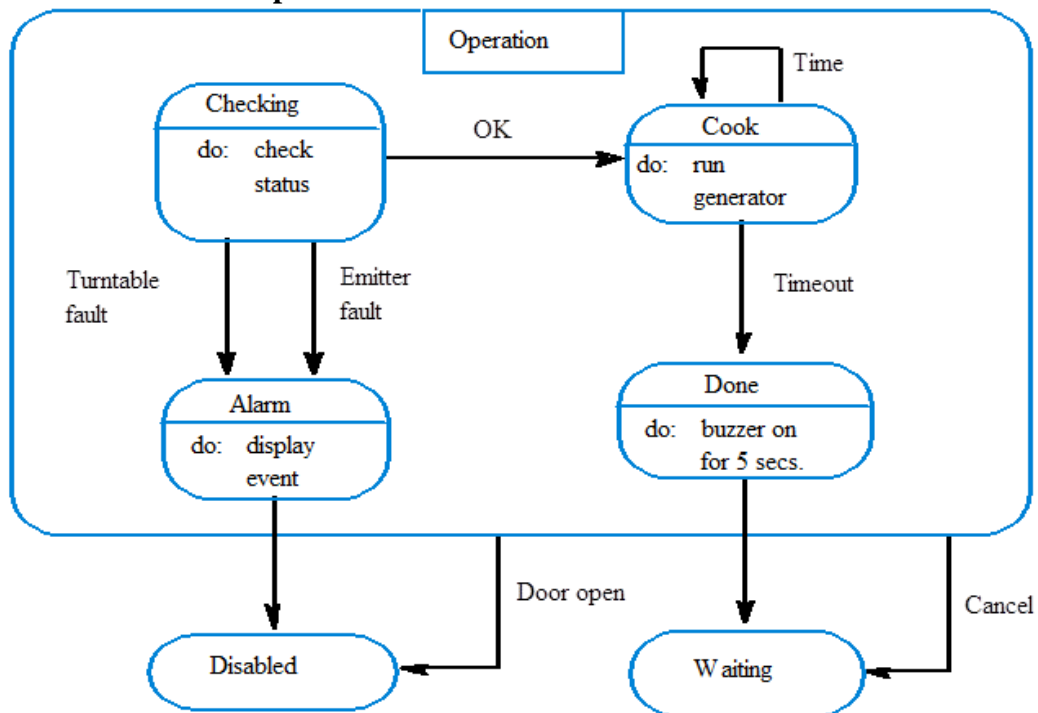
State Machine Models

These model the behaviour of the system in response to external and internal events. They show the system's responses to stimuli so are often used for modelling real-time systems.

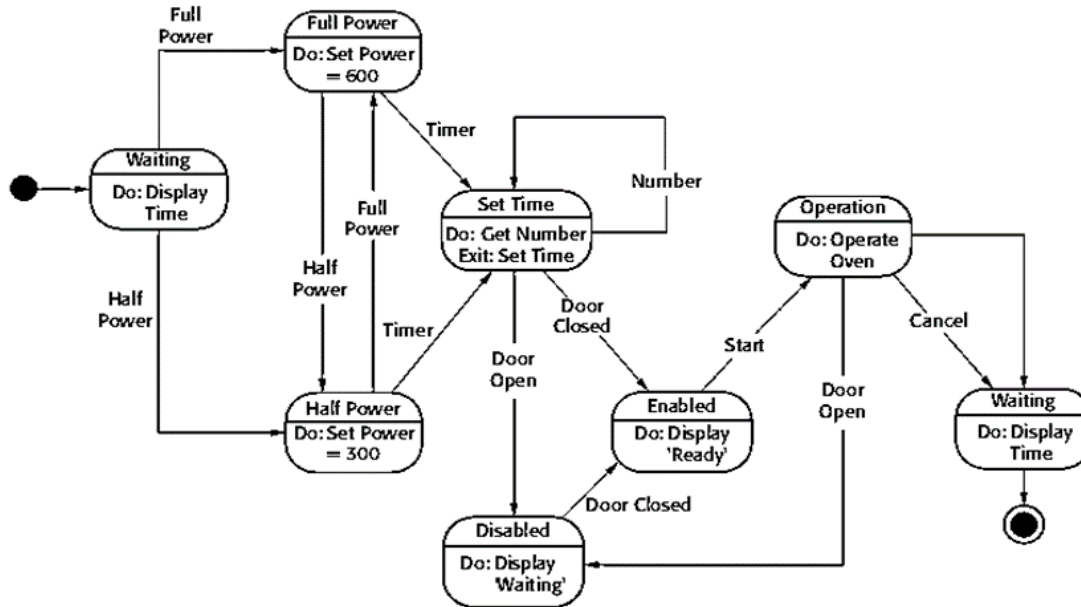
State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.

Statecharts are an integral part of the UML and are used to represent state machine models.

Microwave Oven Operation



State Diagram of a Microwave Oven



States And Stimuli For The Microwave Oven

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Model-driven Engineering

Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process. The programs that execute on a hardware/software platform are then generated automatically from the models.

Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Model Driven Architecture

Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.

Pros

- Allows systems to be considered at higher levels of abstraction
- Generating code automatically means that it is cheaper to adapt systems to new platforms.

Cons

- Models for abstraction and not necessarily right for implementation.
- Savings from generating code may be outweighed by the costs of developing translators for new platforms.

Types of Models

A computation independent model (CIM)

These model the important domain abstractions used in a system. CIMs are sometimes called domain models.

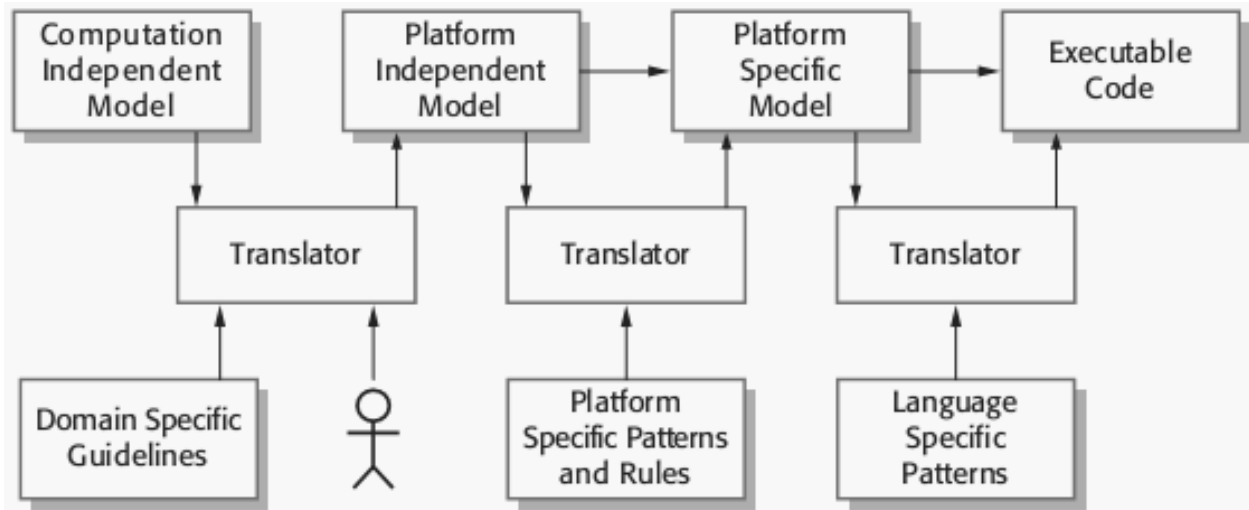
A platform independent model (PIM)

These model the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.

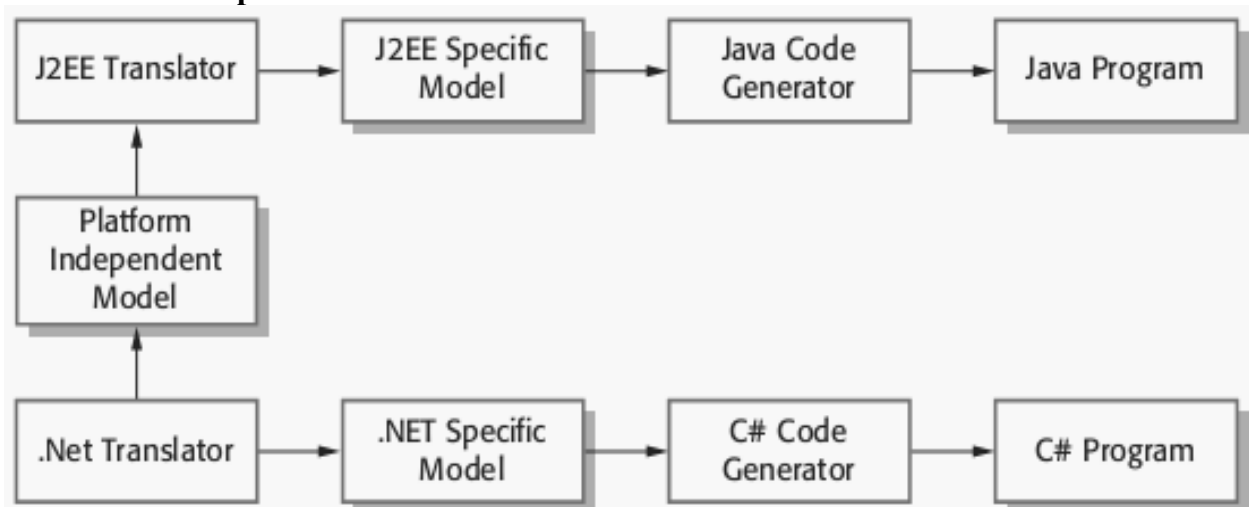
Platform specific models (PSM)

These are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail.

MDA Transformations



MDA Platform Specific Models



Agile Methods and MDA

The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods.

The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering.

If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required.

Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.

This is possible using a subset of UML 2, called Executable UML or xUML.

Features of Executable UML

To create an executable subset of UML, the number of model types has therefore been dramatically reduced to these 3 key types:

Domain models that identify the principal concerns in a system. They are defined using UML class diagrams and include objects, attributes and associations.

Class models in which classes are defined, along with their attributes and operations.

State models in which a state diagram is associated with each class and is used to describe the life cycle of the class.

The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL), or may be expressed using UML's action language.

Design and Implementation

Introduction

In software engineering the goal is to build a software product or to enhance an existing one

Designing software is a process

An effective process

- Provides guidelines for efficient development of quality software
- Reduces risk and increases predictability
- Captures and presents best practices
 - Learn from experiences
 - Mentor for New Recruits
 - Extension of training material
- Promotes common vision and culture
- Enables applying tools
- Provides efficient information exchange,

Rational Unified Process (RUP)

The Rational Unified Process (RUP) (Krutchen, 2003) is an adaptable process model that has been derived from work on the UML and the associated Unified Software Development Process.

It brings together elements from all of the generic software development process models RUP enforces good practice in specification and design and supports prototyping and incremental delivery.

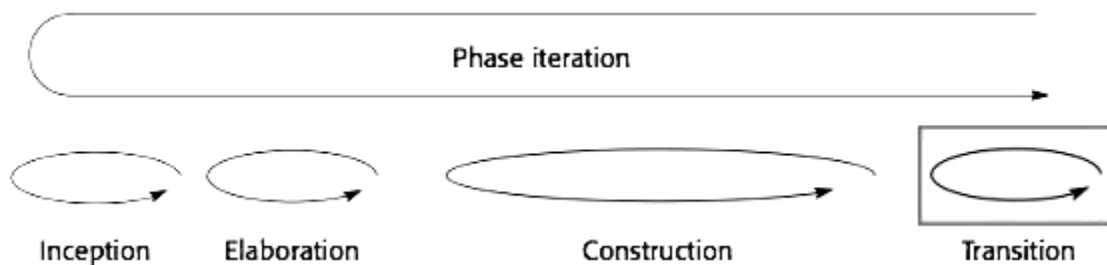
The RUP recognizes that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Most descriptions of the RUP attempt to combine the static and dynamic perspectives in a single diagram. I think that makes the process harder to understand, so I use separate descriptions of each of these perspectives.

Dynamic perspective

The RUP is a phased model that identifies four discrete phases in the software process. However, unlike the waterfall model where phases are equated with process activities, the phases in the RUP are more closely related to business rather than technical concerns.



It is a software engineering process, aimed at guiding software development organizations in their endeavors to develop effective software efficiently

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. **Develop software iteratively** : Plan increments of the system based on customer priorities and develop the highest priority system features early in the development process.
2. **Manage requirements** : Explicitly document the customer's requirements and keep track of changes to these requirements. Analyse the impact of changes on the system before accepting them.
3. **Use component-based architectures** : Structure the system architecture into components, as discussed earlier in this chapter.
4. **Visually model software** : Use graphical UML models to present static and dynamic views of the software.

5. **Verify software quality.** : Ensure that the software meets the organizational quality standards.
6. **Control changes to software.** : Manage changes to the software using a change management system and configuration management procedures and tools.

The RUP may not be a suitable process for all types of development e.g. embedded software development. However, it does represent an approach that potentially combines the three generic process models discussed in section 2.1. The most important innovations in the RUP are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

Design and Implementation

Software design and implementation is the stage in the software engineering process at which an executable software system is developed.

Software design and implementation activities are invariably inter-leaved.

Software design is a creative activity to identify software components and their relationships, based on a customer's requirements.

Implementation is the process of realizing the design as a program.

Build or Buy

In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.

For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.

When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

System Context And Interactions

Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. Understanding of the context also lets the developer establish the boundaries of the system. Setting the system boundaries helps designor decide what features are implemented in the system being designed and what features are in other associated systems.

Context and Interaction Models

A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.

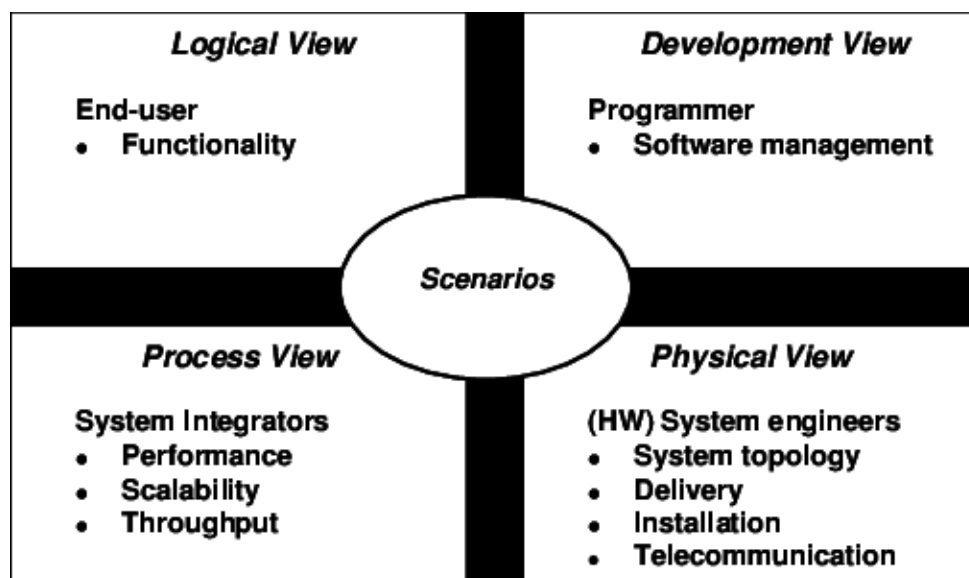
An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

Architectural Design

Once interactions between the system and its environment have been understood, we use this information for designing the system architecture.

we identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.

The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.



The Object Model

A general view of program structure shared by UML and object-oriented programming languages like Java and C++

Computation takes place in *objects* that:

- store data and implement behaviour
- are *linked* together in a network
- communicate by sending *messages*
- are described by *classes*

Object Oriented Design Process

There are a variety of different object-oriented design processes that depend on the organization using the process.

Common activities in these processes include:

- Define the context and modes of use of the system;
- Design the system architecture;
- Identify the principal system objects;
- Develop design models;
- Specify object interfaces.

Process illustrated here using a design for a wilderness weather station.

Approaches to Object Identification

Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).

Base the identification on tangible things in the application domain.

Use a behavioural approach and identify objects based on what participates in what behaviour.

Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Examples of Design Models

Sequence models that show the sequence of object interactions.

State machine models that show how individual objects change their state in response to events.

Other models include use-case models, aggregation models, generalization models, etc.

Sub-system models show

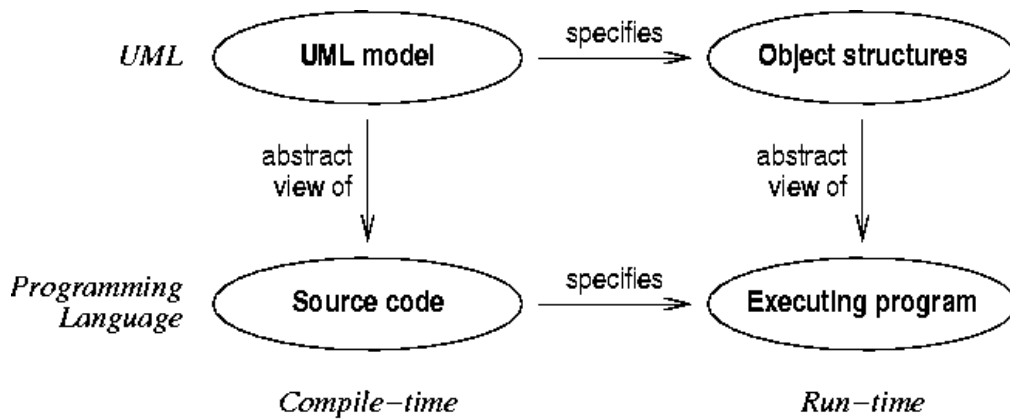
logical groupings of objects into coherent subsystems.

how the design is organized into logically related groups of objects.

In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organization of objects in the system may be different.

Design Models and Code

UML is based on the same object model as object-oriented programming languages



```

interface WeatherStation {

    public void WeatherStation ();

    public void startup ();
    public void startup (Instrument i);

    public void shutdown ();
    public void shutdown (Instrument i);

    public void reportWeather ();

    public void test ();
    public void test ( Instrument i );

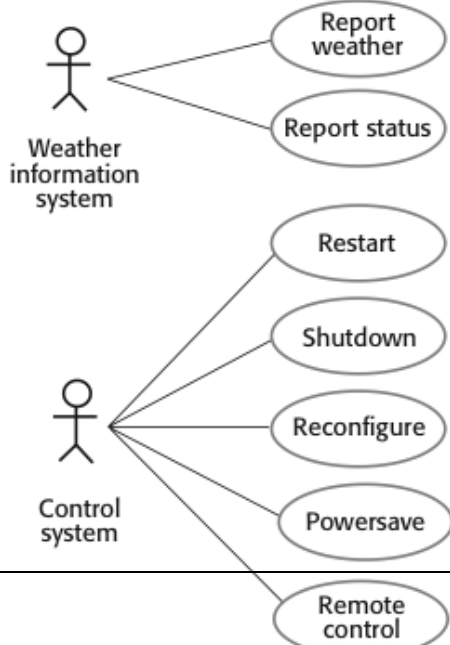
    public void calibrate ( Instrument i);

    public int getID ();

} //WeatherStation

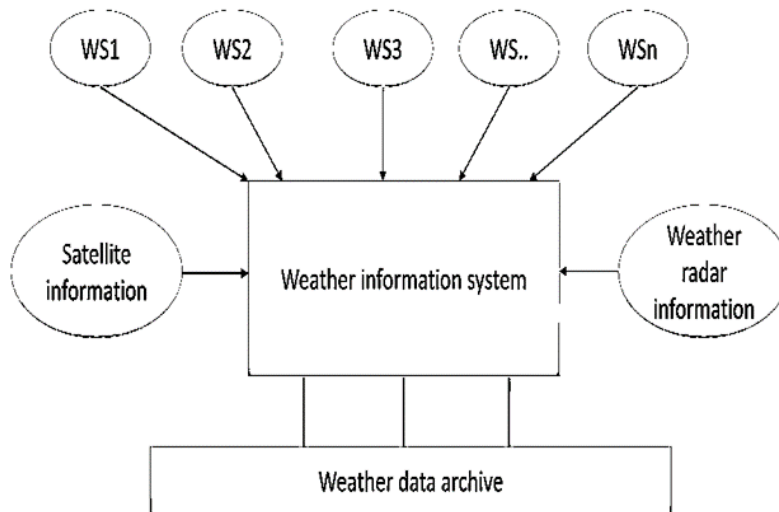
```

Example : Weather Station Information

Requirements:	Use Case Diagram
Collect weather information from instruments at regular intervals	 <pre> graph LR WIS[Weather information system] --- R1(Report weather) WIS --- R2(Report status) CS[Control system] --- R3(Restart) CS --- R4(Shutdown) CS --- R5(Reconfigure) CS --- R6(Powersave) CS --- R7(Remote control) </pre>
Transmit this information, on request, to the weather information system over the satellite link	
Store information if communications are not available	
Monitor external conditions and shut down power generation/instruments if threat of	

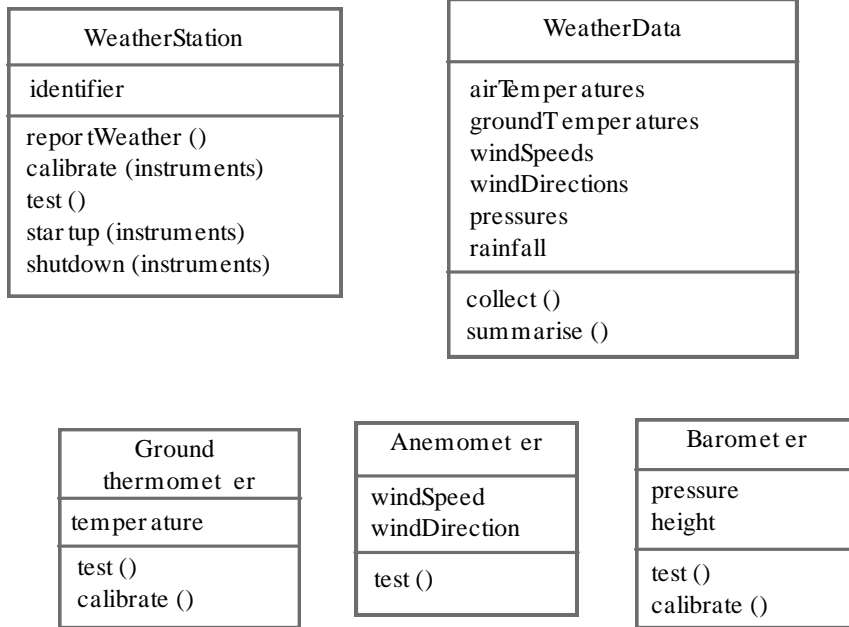
damage from extreme weather	
Run regular diagnostic tests to assess overall health of system	

Context Model for weather station

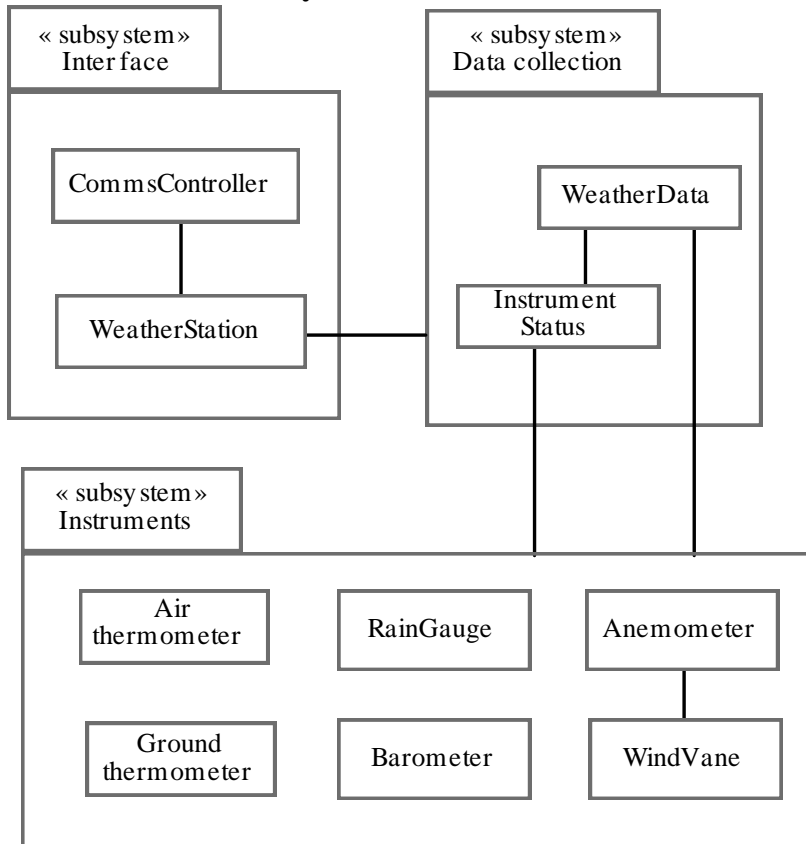


- ✓ **Anemometer** – wind speed measurement
- ✓ **Barometer** – air pressure measurement
- ✓ **Ground and air thermometers**
- ✓ **Rain/precipitation gauge**
- ✓ **Sunshine gauge**
- ✓ **Visibility gauge**

Weather Station Class Models



Weather Station Subsystems



Design Patterns

A design pattern is a 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 descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

A design pattern is a descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

Design patterns represent the best practices used by experienced object-oriented software developers.

Design patterns are solutions to general problems that software developers faced during software development.

These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Design patterns are optimized, reusable solutions to the programming problems that we encounter every day.

A design pattern is not a class or a library that we can simply plug into our system; It is a template that has to be implemented in the correct situation.

It's not language-specific.

A good design pattern should be implementable in most programming languages, depending on the capabilities of the language.

Any design pattern when correctly implemented , it can be a Great Solution,

However, if implemented in the wrong place, it can be disastrous and create many problems.

Benefits of patterns

Design reuse

Uniform design vocabulary

Enhance understanding, restructuring, & team communication

Basis for automation

Transcends language-centric biases/myopia

Abstracts away from many unimportant details

Patterns, Architectures and Frameworks

There can be confusion between patterns, architectures and frameworks.

Let's try to distinguish them:

Architectures model software structure at the highest possible level, and give the overall system view. An architecture can use many different patterns in different components

Patterns are more like small-scale or local architectures for architectural components or sub-components

Frameworks are partially completed software systems that may be targeted at a particular type of application. These are tailored by completing the unfinished components.

Pattern Elements

Name : A meaningful pattern identifier.

Problem description : Explains the problem and its context

Solution description : Not a concrete design but a template for a design solution that can be instantiated in different ways.

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

Types of Design Patterns

There are 3 types of pattern ...

Creational: address problems of creating an object in a flexible way. Separate creation, from operation/use.

Structural: These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

Behavioral: These design patterns are specifically concerned with communication between objects. Patterns address problems of assigning responsibilities to classes. Suggest both static relationships and patterns of communication (use cases)

S.N.	Pattern & Description
1	<p>Creational Patterns</p> <p>These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.</p>
2	<p>Structural Patterns</p> <p>These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.</p>
3	<p>Behavioral Patterns</p> <p>These design patterns are specifically concerned with communication between objects.</p>
4	<p>J2EE Patterns</p> <p>These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.</p>

Creational Design Patterns

Creational patterns often used in place of direct instantiation with constructors. They make the creation process more adaptable and dynamic. In particular, they can provide a great deal of flexibility about which objects are created, how those objects are created, and how they are initialized.

Singleton

When an application wants to have one and only one instance of any class per JVM, in all possible scenarios without any exceptional condition.

Factory

This is most suitable where there is some complex object creation steps are involved. To ensure that these steps are centralized and not exposed to composing classes, factory pattern should be used.

Abstract factory

Whenever you need another level of abstraction over a group of factories, you should consider using abstract factory pattern.

Structural Design Patterns

These design patterns show you how to glue different pieces of a system together in a flexible and extensible fashion. Structural patterns help you guarantee that when one of the parts changes, the entire structure does not need to change.

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Decorator

This is used to add additional features or behaviors to a particular instance of a class, while not modifying the other instances of same class.

Behavioral Design Patterns

A behavioral pattern abstracts an action you want to take from the object or class that takes the action. By changing the object or class, you can change the algorithm used, the objects affected, or the behavior, while still retaining the same basic interface for client classes.

Command

Command pattern is a behavioral design pattern which is useful to abstract business logic into discrete actions which we call commands. This command object helps in loose coupling between two classes where one class (invoker) shall call a method on other class (receiver) to perform a business operation.

Visitor

When you want a hierarchy of objects to modify their behavior but without modifying their source code.

Memento

Memento design pattern provides ability to capture(save) an object's state and then restore back this captured state when required by the system.

State

State Design Pattern allows the behavior of an object to vary based on its state. I.e. whenever the object's state changes, its behavior changes as per its new state. To the observer it appears as if the object has changed its class.

Using Design Patterns

To use patterns in our design, we need to recognize that any design problem may have an associated pattern that can be applied.

Tell several objects that the state of some other object has changed (Observer pattern).

Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).

Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).

Some general guidelines for using design patterns are :

Is there a pattern that addresses my problem?

Does the pattern provide an acceptable solution?

Is there a simpler solution? (pattern overuse)

Is the context of the pattern consistent with my problem?

Are the consequences of using the pattern acceptable?

Are there forces in my environment that conflict with the use of the pattern?

Software Implementation

Software Implementation is often the most important step in the software process cycle. Some of the issues of implementation are

Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.

Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

Software Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.

The only significant reuse of software was the reuse of functions and objects in programming language libraries.

Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.

An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse Level

The abstraction level : At this level, you don't reuse software directly but use knowledge of successful abstractions in the design of your software.

The object level : At this level, you directly reuse objects from a library rather than writing the code yourself.

The component level : Components are collections of objects and object classes that you reuse in application systems.

The system level : At this level, you reuse entire application systems

Reuse Costs

The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.

Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration Management

Configuration management is the name given to the general process of managing a changing software system.

The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

Version management : Versioning is the mechanism to manage systems changes
Complex systems developed, installed, and maintained in series of versions to simplify testing and support

Alpha version – incomplete testing version

Beta version – end-user testing version

Production release version – formally distributed to users or made operational

Maintenance release – bug fixes, small changes

where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.

System integration, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.

Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Host Target Development

Most software is developed on one computer (the host), but runs on a separate machine (the target).

More generally, we can talk about a development platform and an execution platform. A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.

Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Development Platform Tools

Some of the development platform tools are,

An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.

A language debugging system.

Graphical editing tools, such as tools to edit UML models.

Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.

Project support tools that help you organize the code for different development projects.

Software development tools are often grouped to create an integrated development environment (IDE).

An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.

IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools

Deployment Factors

If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another

Open Source Development

If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another

The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.

Other important open source products are Java, the Apache web server and the MySQL database management system

Open Source Issues

Should the product that is being developed make use of open source components?

Should an open source approach be used for the software's development?

More and more product companies are using an open source approach to development.

Their business model is not reliant on selling a software product but on selling support for that product.

They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open Source Licensing

A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.

Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.

Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.

Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.

The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.

The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

License Management

- Establish a system for maintaining information about open-source components that are downloaded and used.
- Be aware of the different types of licenses and understand how a component is licensed before it is used.
- Be aware of evolution pathways for components.
- Educate people about open source.
- Have auditing systems in place.

- Participate in the open source community.

Implementation Process

The process of software implementation consists of the following steps

1. Acquisition
2. Development
3. Testing
4. Documentation
5. Software versioning
6. Data porting and conversion
7. System deployment
8. Training
9. Maintenance

Acquisition refers to the decision of either build or buy the software after due diligence

Program development is time consuming. It may account for as much as one-third of development labor. One-third to one-half of project development schedule

Quality assurance is the process of Process of ensuring information system meets minimum quality standards as Determined by users, implementation staff, and management. It also involves Identification of gaps or inconsistencies in system requirements. QA integrated into project throughout SDLC and the Cost of fixing errors rise as project progresses

System testing involves testing hardware devices, testing and debugging computer programs, and testing information processing procedures.

An important part of testing is the production of prototypes of displays, reports, and other output.

It is important to involve end users in the testing stage to recognize errors, and to provide feedback.

Software Implementation consists of smooth transition from old system to New System. This involves porting old data to new system or converting old data format to new formats. Major forms of conversion are

Parallel: both old and new systems are operated until IS team and management agrees to convert

Pilot: one department or work site serves as a tester.

Phased: only parts of the new system or only a few departments, offices, or plant locations at a time are converted

Plunge: direct abandonment of old system and conversion to new system.

Data Conversion : Data needed at system startup

Files or databases of system being replaced

Manual records

Files or databases of other systems

User feedback during normal system operation

Reuse of existing databases

Reloading database contents

Creating new databases

Installation

After development and testing, system must be put into operation. New system can be installed and quickly made operational, However, in case of enhancement or defect removal, the following points need to be taken care of

Overlapping systems turned off

Both systems concurrent for brief time

Advantage – simplicity and fewer logistics issues to manage

Disadvantage – risk due to no backup

Important planning considerations include

Costs of operating both systems in parallel,

Detecting and correcting errors in new system,

Potentially disrupting the company and IS operations
Training personnel and customers with new procedures

System Documentation

One of the most important aspect of the project is documentation. In fact, the success of a project depends entirely on the quality of documentation. The documentation should cover but not limited to

- Descriptions of system functions, architecture, and construction details
- User manuals for maintenance personnel and future developers
- Source code, Analysis and design models, Operational details (System Manual) as applicable
- Failure to maintain system documentation compromises value of a system

User Documentation

Special attention should be paid for user documentation which will be used by end users and system operators. It should include the details about

- Descriptions of how to interact with and maintain the system
- Startup and shutdown
- Keystrokes, mouse, or command functions to perform specific functions
- Program function for specific business procedures
- Common errors and correction techniques

Module 3: Agile Software Development

The Agile Manifesto: Values and Principles

Learning Objectives

By studying The Agile Manifesto: Values and Principles.

- You will Understand and Appreciate the rationale for Agile Software Development Methods
- You will Understand and Appreciate the Agile Manifesto: Values and Principles, and
- You will Understand and Appreciate the differences between Agile and Plan Driven Development

Introduction

Agile Movement is towards developing software and software systems in a rapid way. In this chapter the Agile Manifesto: Values and Principles, we study the rationale for Agile Software Development Methods, the Agile Manifesto: Values and Principles, and the differences between Agile and Plan Driven Development

Need for Agile Software Development

Software is a part of every system and such a software shall be released on-time, help users to fulfill their expectations, and also, ensure business value. These critical requirements can be addressed by adapting practices of rapid development and delivery of software so that we can address technical disruptions, emergence of competing products and services, new opportunities and markets, and align to changing social and economic conditions. However making it possible is not so easy but quite a challenging one because of changes that have become common phenomena which come in the form of – demand for do it fast in days or weeks; expectations of customer, users, and team members to understand their ideas and incorporate them; more aggressive demand to give out workable product as soon as possible; and that too without any need for cumbersome and rigorous processes including sign-offs. In order to address these challenges, software development community and organizations are always seeking for superior approaches,

and as days are progressing these superior approaches have basis in repaid development and delivery of software approaches.

Agile Software Development is one such rapid development and delivery of software approach that helps to address a critical requirement in software product development by making it increasingly possible to release software product on time, making it relevant and useful, and incorporate changes even in later stages in software development life cycle stages. Agile practices will help in release of workable product by increasing possibility of higher value to business and users.

However mindset and rigorous practice of processes over years have ‘conditioned’ software development community and organizations that have ‘institutionalized’ process based culture where all roles in software development and delivery are reflecting and exhibiting mindset of worrying only about the project by forgetting the product in every acts and deeds which show-off in the form of - customer needs everything that would have been addressed yesterday, conflicts and negotiations that are more inclined towards making someone accountable by exhibiting culture of ‘If it is your problem, fix it at your cost’ than focusing on addressing the concern; and attitude that intensively focuses where in no-one wants to pay for anything which hinders the very thought process of adding innovation and value.

Though this is predominantly representing existing situations, the context is increasingly compelling software development and organizations to look-out solutions for ‘what to do now to stay and be relevant?!’. Failure to find solutions or exhibition of ‘corporate ego and arrogance’ may result in abandonment of product by customers and bankruptcy of well-to do companies faster than our imagination. Hence to stay in business and continue to be relevant there is no other option than accepting that change is bound to be here, understanding that project and customer-vendor mindset is “not working”, and customer and user needs greater product and then we shall move forward to establish culture of worrying about delivering the workable product by getting away from irrelevant project development, process, and documentation. Agile Software Development provides more optimistic path to achieve these goals and objectives.

Agile Model

Unlike other software development models, assumes that the change is constant so adapt to change rather than control change

This is very practical and major movement away from conventional software development models which assume that Requirements will not rapidly change ones we specify them and Object Oriented models that work with objective of need to design for the future and need to design for reuse

Agile Manifesto and Four Key Ideas in Agile

To produce better software product by reducing the time gap between doing some activity and gaining feedback.

In order to implement agile manifesto, agile development approach uses four key ideas:

- Individuals and interactions are preferred over processes and tools
- Working software is preferred over comprehensive Documentation
- Customer collaboration is preferred over contract negotiation
- Responding to change is preferred over following a plan

And agile manifesto and agile software development uses following Key Principles.

Key Principles of Agile Manifesto and Agile Software Development

Agile manifesto and agile software development uses following Key Principles.

- **#1 Principle: Customer Involvement:** Customer Should be Closely involved throughout the Development Process Their role is provide and priorities new system requirements and to evaluate the iterations of the system.
- **#2 Principle: Incremental Delivery:** The Software Developed in Increments with the customer specifying the requirements to be included in each increment.
- **#3 Principle: People Not Process:** The Skills of Development Team should be recognized and exploited. Team member should be left to develop their own ways of working without prescriptive process.
- **#4 Principle: Embrace Change:** Expect the System Requirements to Change and so design system to accommodate these changes.
- **#5 Principle: Maintain Simplicity:** Focus on Simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate the complexity from the system.

Agile Methods: Where is it Working Well?

Agile methods work very well in small and medium scale software development and teams. Agile methods are found to be effective under situations mentioned below:

- Product Development where software company is developing a small or medium sized product for sale
- Customer System Development within an organization where there is clear commitment from the customer involved in the development process and where there are not a lot of external rules and regulations that affect the software

Agile methods seem to be less effective in project, teams, and software that are large in size, and also, where systems are safety and mission critical. Agile methods face issues and failures when

- Putting in a fixed price contract where managing cost & effort becomes more important over business value. Agile methods since focus on value more than cost and effort frequently fail to stick to cost and effort.
- Expecting all of the scope to be delivered: Agile methods focus on workable product and on-time delivery. Hence in agile we can notice negotiating with scope by reducing or increasing scope when it is found to be of value can be quite a common characteristic.
- Pushing the team hard – making unrealistic demands: Agile uses group creativity and co-creation and as such team motivation is critical for success. When we push the team hard by putting unrealistic demands, we can see that it may result in demotivated team. Demotivated team frequently fails to deliver value and innovation.
- Focusing on project execution efficiency as against product value delivered: Project efficiency is all about sticking to scope, effort, schedule, and budget. Agile accepts changes and hence scope always undergoes change even in later stages of the software development life cycle and it may consume more effort and budget. Hence agile methods may not be of great value in fixed-budget and fixed-scope projects.

Plan-Driven and Agile Development

Learning Objectives

By studying Plan-driven and Agile Development:

- You will understand and appreciate the differences between Agile Development and Plan-driven Approaches
- You will understand on range of questions to be answered so as to effectively balance Agile Development and Plan-driven Approaches

Introduction

Plan-driven development focuses on sticking to scope, using allocated effort and budget, sticking to on-time delivery, and compliance with process. It believes that quality product is result of plan-driven project management and process management.

When compared with plan-driven approach, agile development focuses on providing value and delivering workable product. Accepting changes and varying scope, using more effort and cost than allocated are seen frequently in agile development. However agile believes in fixed-schedule ensuring on-time delivery. In brief, agile development believes in workable product by willingly accepting changes with higher value against that of plan-driven development wherein ones plan including processes is accepted then changes are discouraged. In this section, we study the differences between Agile Development and Plan-driven Approaches, and understand on range of questions to be answered so as to effectively balance Agile Development and Plan-driven Approaches.

The Differences between Agile Development and Plan-driven Development Approaches

Following table provides the differences Agile Development and Plan-driven Development Approaches:

Table 3.1 : Difference between Agile Development and Plan-driven Development Approach

Agile Development Approach	Plan-driven Development Approach
<p>“Producing Workable Product”:</p> <ul style="list-style-type: none"> • Creativity of People and their 	<p>“Process Centric Delivery”</p> <ul style="list-style-type: none"> • Compliance with Processes shall be

<p>Decision shall be considered as the basis to produce Workable Product</p> <ul style="list-style-type: none"> • Continuous Incremental Integration and Deployment of the Product as fast as possible 	<p>considered as the basis to produce “Quality Product”</p> <ul style="list-style-type: none"> • Well planned Product Integration and Deployment of the Product only after User Acceptance of the Product
<p>“Design and Implementation” centric</p> <ul style="list-style-type: none"> • Requirement Specification, Designing, Implementation, And Verification & Validation Take Place “Simultaneously” 	<p>“Stage-wise Implementation”</p> <ul style="list-style-type: none"> • Well Defined Stages with Clearly Defined Inputs, Entry Criteria, Outputs with Baselined Output Of One Stage becomes Input To Begin The Next Stage.
<p>“Active Participation in All Tasks”</p> <ul style="list-style-type: none"> • All Stake Holders are Accountable and Actively Engage throughout • Engage to add value with Simplicity 	<p>“Well-defined Role and Role Specific Tasks”</p> <ul style="list-style-type: none"> • Each Role is Accountable for a Specific Role and Work Product • Work only for Delivering the Work Product they are Responsible for
<p>“Active Participation of People throughout and Group Creativity”</p> <ul style="list-style-type: none"> • Accountable for Delivery of the Workable End-Product • Group Thinking, Seamless Communication and Active Participation 	<p>“Role Specific Involvement and Analytical Thinking”</p> <ul style="list-style-type: none"> • Accountable for the Delivery of a Specific Phase-end Work Product • Analytical Thinking, Role Specific Communication and Participation
<p>“No Explicit Testing Phase”</p> <ul style="list-style-type: none"> • Development and Testing are Simultaneous Activities • Verification and Validation takes place Simultaneously with the Creation of Work Product 	<p>“Explicit Testing Phase”</p> <p>Development and Testing are Separate Phases</p> <p>Verification and Validation takes place only “after Creation” of Work Product</p> <p>Testing is the Role of Specific Roles only</p>

<ul style="list-style-type: none"> • All are involved in Testing • Stake Holders Decide on Product Readiness 	<p>User Acceptance Testing is the basis for Deciding on Product Readiness</p>
<p>“Relevant Documentation based on Need felt by Stake Holders”</p> <ul style="list-style-type: none"> • Emphasis on Relevant Documentation only • Stake Holders decide on Need, Type, and Depth of Documentation 	<p>“Intensive Documentation based on Stringent Requirements of the Phase in Life Cycle”</p> <ul style="list-style-type: none"> • Emphasis on Intensive and In depth Documentation • Project Specific tailored Process ‘forces’ on Rigorous and Mandatory Documentation

Effectively Balancing Agile Development and Plan-driven Approaches

Most projects include Balancing of practices from Agile Development and Plan-driven Approaches. To arrive at effective combined approach we need to answer a range of Technical, Human, and Organizational Questions. We have to ask this range of questions and get proper professional and corroborated answers so as to effectively balance Agile Development and Plan-driven Approaches.

Questions and what we seek for: A Range of Technical, Human, and Organizational Questions to Define a Combined Project Specific Approach

<p>Question #1: Need for Very Detailed Specification and Design</p>	
<ul style="list-style-type: none"> • Is it Important to have a <u>very detailed specification and design</u> before moving to Implementation? • If <p>=> Probably need to use Plan-driven Approach</p>	<p>YES</p>
<p>Question #2: Incremental Delivery with Rapid Feedback from Customer</p>	
<ul style="list-style-type: none"> • Is an Incremental delivery strategy, where you deliver the software to the customer and get rapid feedback from them, realistic? • If <p>=> If so, consider using Agile Approach</p>	<p>YES</p>

Question #3: How large is the system that is being developed and team size and where it is located?

- If the system developed is small by a small co-located team who can communicate informally

If YES

=> Consider using Agile Approach

- If the system developed large by a larger team who can communicate formally

If YES

=> Consider using Plan-driven Approach

Question #4: What type of System is being developed?

- Does the system requires a lot of analysis and detailed design before implementation?

If YES

=> Consider using Plan-driven Approach

Question #5: What is the expected System Life Time?

- Does the expected system life time is long and requires a lot of detailed documentation to communicate with maintenance and support team?

If YES

=> Consider using Plan-driven Approach

- Does the expected system life time is small with rudimentary and incomplete documentation with no need of maintenance and support the workable product?

If YES

=> Consider using Agile Approach

Question #6: What Technologies are available to support system development?

- If technologies that are available are robust, stable, and with enough expertise?

If YES

=> Consider using Plan-driven Approach

- If technologies are bleeding edge and whose stability and robustness are yet to be evaluated

<p>If YES</p> <p>=> Consider using Agile Approach</p>
<p>Question #7: How is the Development Team Organized?</p> <ul style="list-style-type: none"> • Distributed, Outsourced, and need intensive documentation and formal communication? <p>If YES</p> <p>=> Consider using Plan-driven Approach</p> <ul style="list-style-type: none"> • Co-located, homogeneous, and rudimentary and basic documentation and interactive communication? <p>If YES</p> <p>=> Consider using Agile Approach</p>
<p>Question #8: Are their cultural issues that may affect system development?</p> <ul style="list-style-type: none"> • Traditional Engineering Organizations with a need of intensive documentation and formal communication? <p>If YES</p> <p>=> Consider using Plan-driven Approach</p> <ul style="list-style-type: none"> • Non-traditional, deployment oriented organizations with continuous live updates and replacements that require rudimentary and basic documentation and interactive communication? <p>If YES</p> <p>=> Consider using Agile Approach</p>
<p>Question #9: How good are the designers and programmers in the development team?</p> <ul style="list-style-type: none"> • If the designers and programmers in the development team are with higher skills, capability, competence, and adaptability? <p>If YES</p> <p>=> Consider using Plan-driven Approach</p> <ul style="list-style-type: none"> • Non-traditional, deployment oriented organizations with continuous live updates and replacements that require rudimentary and basic documentation and interactive communication?

If	YES
=> Consider using Agile Approach	
Question #10: Is the System subject to external regulation?	
<ul style="list-style-type: none"> • Is the System subject to external regulation requiring external regulatory authority to approve with the requirement to produce intensive and formal documentation? 	
If	YES
=> Consider using Plan-driven Approach	
<ul style="list-style-type: none"> • Is the System not subjected to any external regulation and does not require any external regulatory authority to approve? 	
If	YES
=> Consider using Agile Approach	

The above questions are acting as cues and trigger much needed though processes. By using these questions, and by understanding situations and context, we have to make professional judgements to go in depth to develop an insight and then make a proper choice of plan-driven development approach and agile development approach.

Agile Methods: SCRUM Approach

Learning Objectives

By studying Agile Methods- SCRUM Approach:

- You will Understand and Appreciate the SCRUM Approach and
- You will Understand and Appreciate How Key Practices in Extreme Programming relate to the General Principles of Agile Methods

Introduction

Scrum is the most popular framework for agile implementation in projects. Scrum is an iterative incremental framework for managing agile projects. The SCRUM Approach of Agile has roles, terminology, and software process framework with agile specific programming approach referred to as extreme programming.

SCRUM Approach and Roles

Scrum is the most popular framework for agile implementation in projects. Scrum is an iterative incremental framework for managing agile projects. Scrum has following roles:

- Scrum Master – maintains the Scrum processes
- Product Owner – represents the stakeholders
- Team – a group of about 7 people
 - The team does the actual development: analysis, design, implementation, testing, etc.

While using Scrum Approach, we shall be conversant with certain Scrum specific terminology which include the following”

- Product Backlog: All features that have to be developed
- Sprint Backlog: All features planned for the current sprint
- Sprint: An iteration in the Scrum development and its usually few weeks

Scrum Process Framework

Scrum Process Framework starts with “Product Backlog”. “Product Backlogs” is set of features that form the basis to start with agile implementation. There can be changes to the product backlog as we proceed since agile welcomes changes at any stage in software development.

The “Product Backlogs” is split into many “Sprint Backlogs” with each spring backlog having a set of specific features. While deciding on sprint backlog and its scope, importance is given for rigid “2-weeks schedule” (sometimes up to 4 weeks buy never beyond 4 weeks) for implementation of the sprint backlog. All Scrum Roles contribute in deciding “Sprint Backlogs” and by doing estimation.

Each “Sprint Backlog” forms the scope of “Sprint” which is an iteration in the Scrum development and its usually 2-4 weeks. Scrum focuses on design-and-program by using “Extreme Programming” approach. At the end of each Sprint we get workable Configuration Item which shall be maintained in configuration management tool.

As and when we complete a sprint it is integrated, workable product is built, and rolled out. We require automation with proper build and release strategy towards this.

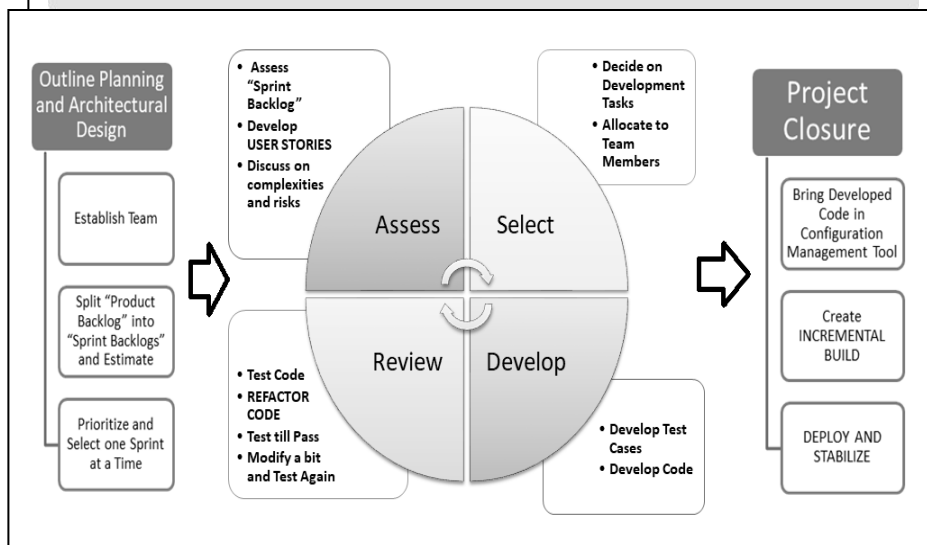
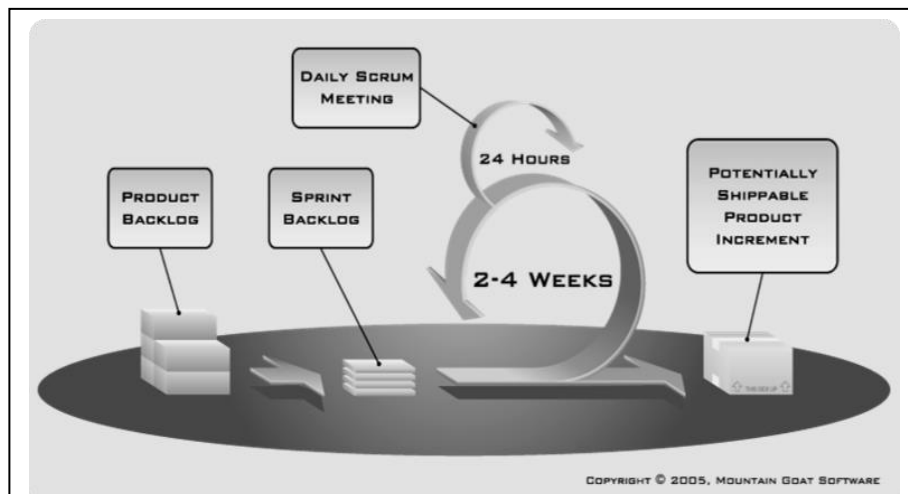


Fig 3.1 Scrum Approach

Stage 1: Outline Planning and Architectural Design

Agile Team is established comprising of developers and testers, typically Select maximum 7 members. The “Product Backlog” received from the client are Broken up into “Sprint Backlogs”. Sprints are of fixed length, normally of 2-4 weeks. The team arrives at a high level plan for the sprint. Team deliberates on Architectural Design, Strategy for Continuous Integration, Unit Testing and Automation.

Stage 2: Implement Sprint Cycle

Sprint Cycle implementation involves following 4 Phases:

- 2.1 Assess Phase
- 2.2 Select Phase
- 2.3 Develop Phase
- 2.4 Review Phase

2.1 Assess Phase

This involves the following:

- Study “Sprint Backlogs”
- Write user stories for each product backlog
 - User Stories are the simplest description of the business requirement that will allow acceptance tests to be created and estimates to be done
 - The Business Customers are the owners of these user stories and are the one point contact to the development team for the queries
 - Use user stories to create time estimates for the release planning meeting

2,2 Select Phase

This involves the following:

- Select features and functionality to be developed with discussion involving the team that works with customer each iteration starts with an Iteration Planning Meeting.
- Arrive at Development Tasks
- Team Member chooses Development Tasks he wants to own
- Decide on “Test-Code-Refactor” based Iterative development

2.3 Develop Phase

This involves the following steps:

- 2.3.1 Test First Approach: write a test and make design decisions
- 2.3.2 Code by using Xtreme Programming with Paired Programming. Write Just Enough Code and Test simultaneously. Automate Unit Testing, and Refactor to keep Code Healthy

Following sections discuss on each of these steps in length.

2.3.1 Test First Approach: Write a Test and Make Design Decisions

XP uses Test-first Development. Test-first Development or Test Driven Development involves incremental Test Development from User Stories by involving User in the Test Development and Validation. These Test Cases are being used as specification for development of code and also, testing and also, using Automated Testing Framework with focus on stronger Unit Testing using Test Automation Tools. For example, in Java programming Junit is used.

Test-first Development or Test Driven Development (TDD), also known as test-first programming or test-first development, is a testing methodology associated with Agile Programming.

Test First Approach is an evolutionary approach to development where you must first write a test that fails before you write new functional code. We write tests not as an afterthought to ensure our code works, but instead as just part of the everyday, every minute way of building software. Instead of writing our detailed design specifications on paper, we write them in code. Instead of first striving to perfectly design a system on paper, we use tests to guide our design. Here every chunk of code is covered by unit tests, which must all pass all the time. This would eliminate unit-level and regression bugs during development.

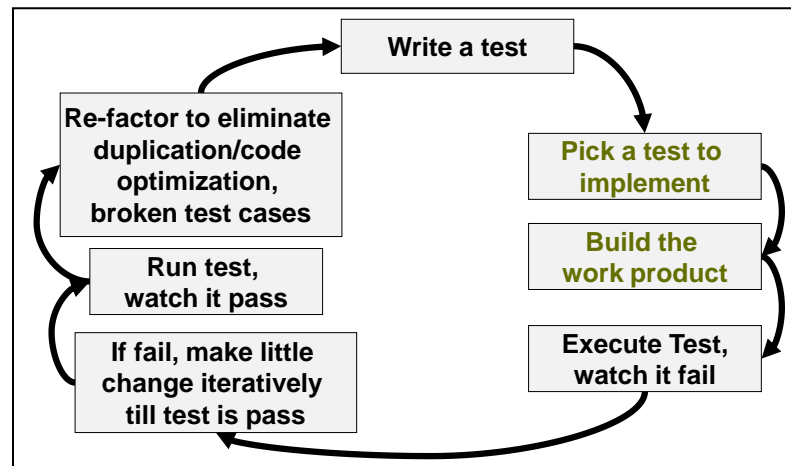


Fig 3.2 Test First or Test Driven Development (TDD) Approach

TDD concentrates on designing and developing one requirement at a time. After the requirement is implemented it is tested. If the test fails, the problem is rectified and is tested again. This process of testing and rectifying occurs until the test becomes successful. All the requirements are designed, developed and tested one by one. Integration will be continual.

Test First Development addresses the following: Ad-hoc unit testing and building infrastructure for regression testing, load testing and stress testing

TDD focuses on Internal Quality Characteristics which are related to quality of code: Correctness, Structured, Modularity, and Documentation.

2.3.2 Code by using Xtreme Programming with Paired Programming

Here we write “Just Enough Code and Test simultaneously”. Automate Unit Testing, and Refactor to keep Code Healthy. Now we shall discuss on Xtreme Programming with Paired Programming concepts in length.

Extreme (XTreme) Programming (XP)

Extreme Programming (XP) is perhaps the most popular agile development methodology that uses good practices of iterative development that provides the highest value for the customer. In order to make it happen it uses a set of values, principles and practices for rapidly developing high-quality software.

XP is characterized by

- User stories

- Pair programming
- Refactoring frequent integration

Following are Key Practices in Extreme Programming relate to the General Principles of Agile Methods:

- Incremental Development: Incremental Development is well supported through small frequent releases of the system with the help of requirements that are represented as Functionalities in the form of User Stories
- Customer Involvement is supported: Representative of End-User of the System and Customer shall Involve, Engage, and Constructively Criticise and also, define & prioritise requirements and developing Acceptance Test for the system
- People but not Processes: People but not Processes is supported through Paired Programming, Collective Ownership of the Code, and Suitable Development Process that discourages long working hours
- Change is Embraced through Regular and Frequent Releases to Customer: Test First Development, Refactoring to avoid Code Degeneration and Continuous Integration of New Functionality
- Maintaining Simplicity is Supported: By constant refactoring that improves code quality and by using simple design that do not unnecessarily anticipate future changes to the system

Extreme Programming Principles or Practices

Following are Extreme Programming Principles and Practices:

Principle or Practice	Description
Incremental Planning	“Requirements” => Recorded as “User Stories” one per “Story Card” => Prioritised => User Stories are split into “Development Tasks” by developers
Small Releases	“Minimal useful Set of Functionalities” that provide “Business Value” is Developed first. “Frequent and Incremental Development and Release”
Simple Design	Good Enough Design that is Just Sufficient to meet the Current Requirements

Test-First Development	Emphasis for Unit Test Automation Framework using Test Scenarios and Providing them as the basis for Validation before commencement of Implementation/ Coding
Refactoring	Code is Continuously Refactored as and when scope for improvement comes. Also, keep the Code Simple and Maintainable
Paired Programming	Developers work in Pairs. Check each others Code and Provide Inputs and Expertise to Improve Upon Each other's Code
Collective Ownership	All Stake Holders including Customer, Developer, Tester, and any other role will seamlessly interact and exchange with no bar on their role by taking collective Ownership and Accountability
Continuous Integration	As soon as the work in the task is complete then it is integrated into whole system. After any such Integration all the Unit Test in the System must Pass
Sustainable Pace	Team is not over stretched so as to work extra hours. This would enhance Code Quality and Productivity since team works with higher motivational levels and lesser stress.
On-site Customer	Representative of End-User of the System and Customer shall Involve, Engage, and Constructively Criticise and also, define & prioritise requirements

Extreme Programming Process

Life Cycle of XP include a lightweight process targeted towards engineering teams and also focused on adaptive to changes.

XP works out very effectively for small and medium sized project teams, i.e., 3-20 members. To make XP to work effectively, communication between development and testing team shall be enabled all the time.

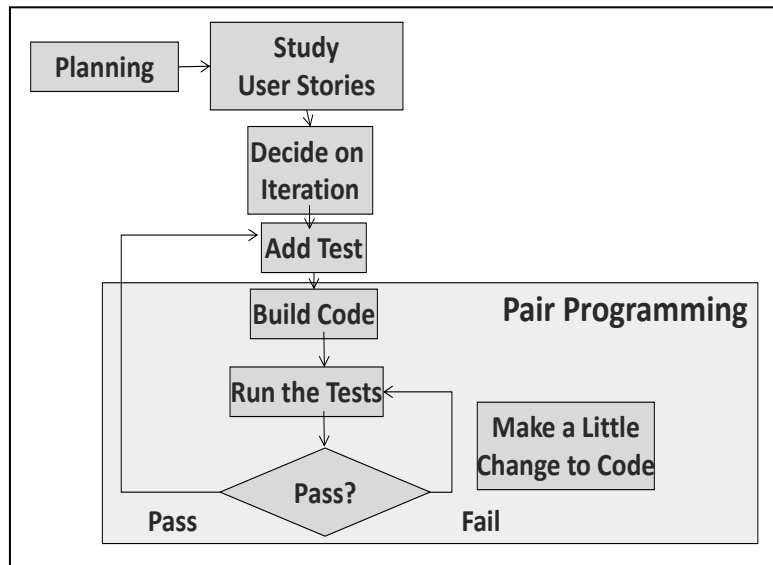


Fig 3.3 Steps in EP Process

Extreme Programming (XP) uses paired programming concept.

Paired Programming and Refactoring

Extreme Programming (XP) uses paired programming concept. In Paired Programming, The coding is usually done in pairs where two programmers periodically switch roles. The pairs also change often. The code review happens continuously. The pairs are determined while iteration planning.

This ensures

- Higher quality code
- Continuous cross training happening
- Improves team communication.

Paired Programming will be effective when used with “Refactoring”. When each iteration is in progress towards small release, there can be request for changes or revisions to user stories. To address these changes or revisions

- Code is refactored, means, in case the developed code needs some changes the same will be done.
- Also, changes will be carried out on test cases as well to align with these changes.
- This cycle continues until the module is developed

2.4 Review Phase

In review phase, we carry put following activities:

- Refactoring is used to Review and Improve Code and to Keep Code healthy
- When each iteration is in progress towards small release, there can be request for changes or revisions to user stories.
- To address these changes or revisions
 - code is refactored, means, in case the developed code needs some changes the same will be done.
 - Also, changes will be carried out on test cases as well to align with these changes.
 - This cycle continues until the module is developed

Stage 3: Project Closure

Continuous Integration and Acceptance Tests by involving Customers is the basis

- Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority.
- Acceptance tests are also used as regression tests prior to a production release.
- A user story is not considered complete until it has passed all its acceptance tests.
- The acceptance test score is published to the developer. It is the developer's responsibility to schedule time in each iteration to fix any failed tests.
- The code that passes the acceptance test is integrated with code of other developers. Like this, final system is evolved

Now we discuss,

- 3.1 Continuous Integration and Small Releases
- 3.2 Acceptance Testing

3.1 Continuous Integration and Small Releases

The Code from each developer is regularly integrated with code of other developers. All unit tests are then run successfully. Code is normally integrated multiple times per day.

Acceptance tests are run daily to reduce occurrences of major integration issues. Frequent small releases happen at the end of each iteration.

This gives an opportunity

- To show case the progress to Customer more frequently.
- Feedbacks from the customer are then incorporated into the subsequent iterations

3.2 Acceptance Testing

Acceptance tests are black box system tests. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are indicators of the completion of a requirement or feature. When all acceptance tests for a requirement or feature are passing, you know you're done. Acceptance tests are also used as regression tests prior to a production release. A user story is not considered complete until it has passed all its acceptance tests. The acceptance test score is published to the developer. It is the developer's responsibility to schedule time in each iteration to fix any failed tests. The code that passes the acceptance test is integrated with code of other developers. Like this, final system is evolved. Tester will execute all the system test cases and log the defects with respect to that module/iteration which are fixed immediately

Advantages of SCRUM in Telecommunication Software (Ref: Rising and Janoff)

Study on Telecommunication Software developed by using Scrum revealed following advantages:

- Advantage #1: The Product Backlog is broken down into a set of manageable and understandable Chunks
- Advantage #2: Unstable Requirements do not hold up progress
- Advantage #3: The whole team has visibility of everything and consequently team communication has improved
- Advantage #4: Customer see on-time delivery of increments an gain feedback on how product works
- Advantage #5: Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed

Scaling Agile Methods

Agile Methods worked conventionally very well with small and medium sized projects, Co-located Team, and team sizes ranging from 7 to 20 members. Scaling agile to larger projects and large organizations so as to get the advantages and benefits of this approach has gained momentum and interest.

“Scaling Agile Methods”: Characteristics of Large Software System Development

Large Software System Development have the following characteristics:

- #1: Geographical Distributed Teams working independently on their Sub-system with no need to communicate with teams working on other sub-systems. Missing whole system view, and possible risks related to integration
- #2: Many of them are “Brown Field Systems” with new system is built to work by connecting with existing systems. These may pose problems of Interoperability, portability, and immune to change
- #3: More System Configuration and Lesser New Incremental Development of Software
- #4: Large System Development are Bound by Stronger Processes with No Flexibility. Hence Software Development effort is more towards Process Management and Compliance with Rules and Regulations
- #5: Large Systems have a diverse set of stake holders who can not be involved in development process

“Scaling Agile Methods”: Two Perspectives

Following are two perspectives on “Scaling Agile Methods” to make it effective for Large Software System Development

- “Scaling up” Perspective: By using Agile Methods small teams can not develop large software system
- “Scaling out” Perspective: Concerned with how agile methods can be introduced across a large organizations with many years of software experience

“Scaling Agile Methods”: Adaption to Cope with Large Systems Engineering

“It is essential to maintain the fundamentals of agile methods characterised by Flexible Planning, Frequent System Releases, Continuous Integration, Test Driven Development, and Good Team Communication” (Leffingwell, 2007)

Critical Adaptions for Large Systems Development

- #1: Do not just focus on Just Enough Code and Test simultaneously. You need to do more up-front design involving describing critical aspects of the system, database schema, work break down across teams, and system documentation
- #2: Establish Cross Team Communication Mechanism by using team structuring, by making use of more advanced conferencing technologies, and knowledge management
- #3: Adapt Effective Configuration Management Tool with System Configuration, System Building, Frequent Release, Change Management, and Live Update and Replacement. Appropriate Integration-Build-Release- Issue Management with the help of automation of Configuration Management is very crucial.

“Scaling Agile Methods”: Difficulties in Adaption to Cope with Large Companies

Following difficulties have been observed in adaption of agile methods to cope with large companies:

- #1 Reluctance of Project Managers to accept Agile Approach: Project Managers who are accustomed to Project Driven and Process Driven Culture feel more comfort and confidence. They are reluctant to accept challenges and risks of adapting to Agile Approach
- #2 Matured Processes do not encourage Agile Approach: Existing Matured Quality Management System with stable and standard processes that are against frequent changes and continuous adaption discourage use of agile approaches
- #3: Skill Variance and Team Attrition does not suite to Agile Approach: Large Companies have greater Skill Variance, have Globally Distributed Teams, and Individuals may move to other companies. Agile requires Uniform Skills and Capability, Homogenous Team Culture, and Co-located Teams
- #4 Organizations with Conventional Systems Engineering Processes are reluctant to use Agile Approaches since they have evolved and reached stability in earnings and have ability to cope with economic recession successfully

"Why do we fail to get best out of Agile“: Top 6 Reasons

Following are top 6 reasons for not getting the best out of agile.

- Lack of System Thinking and Contextualization

- Difficulty in moving from “task oriented, process driven mindset” to “solution oriented, workable product oriented mindset”
- Complexity in ensuring seamless communication among agile team members
- Scaling up agile to large projects is not very effective
- Confidence level of delivering the best within a given effort is uncertain
- Decay in agile culture as team progresses

Summary

Agile Software Development is People Centric Approach. It works very well with Small Teams, Small Projects, and Frequent Releases but struggles to adapt to large projects and large companies. In agile, Interactive and Constructive Communication is key to success. In agile, User Stories, Flexible Planning, Frequent System Releases, Continuous Integration, Test Driven Development, Test-Code-Refactor and Good Team Communication are key practices. Agile software development is Design and Implementation Centric approach that encourages frequent releases of workable product. In agile software development, Unit Test Automation and Automation of Configuration Management play crucial role. Agile quite effective and customers and users increasingly like it because of its willingness to accept changes even in late phases of development is acceptable. Agile software development works with “Fixed Time” Approach. However compromises with effort, budget, and requirements. Success of agile software development depends heavily on Skills and Capability of Team. Organizations shall take decision on whether to Adapt Agile Approach or not with care since it is involving tectonic shift in organizational culture.

Module 4: Software Testing

Software Quality Control - Inspection and Testing

Software Engineering is all about producing Producing good Quality software product that

1. Meets users needs,
 2. Is highly reliable
 3. Has MINIMUM number of bugs (preferably 0)
 4. Developed in cost effective way
- and
5. Delivered according to planned schedules.

Term “Quality” as applied to software means different things to different stake holders:



Fig 4.1: Software Quality

However, we can consider Good quality software as one that meets the user’s needs in terms of Correctness (meets Functional Requirements), Reliability (User can depend on it; Operates as expected over specified time) and Robustness (Behaves ‘reasonably’ even under ‘unspecified’ circumstances).

Other important attributes are Efficiency & Performance, Maintainability & Evolvability, Usability / reusability & interoperability, Understandability & Portability .

Thus we see that Software Quality is Not as “ simple and straight forward “ as for other engineering products. It is Problematic because:

- Software specs are usually incomplete & often inconsistent.
- Some quality needs are difficult to specify in an unambiguous way;
- There is a tension between quality requirements of :
 - Customer (efficiency, reliability, etc.)
 - and
 - Developer (maintainability, reusability, etc.);

Further It is not possible for any system to be optimized for all of these attributes – for example, improving robustness may lead to loss of performance. The quality plan should therefore define the most important quality attributes for the software that is being developed. The plan should also include a definition of the quality assessment process - An agreed way of assessing whether some quality, such as maintainability /robustness, is present in the product

The focus may be ‘**fitness for purpose**’ rather than specification conformance.

FITNESS FOR PURPOSE is decided based on answers to questions like:

- Has the software been properly tested?
- Is the software sufficiently dependable to be put into use?
- Is the performance of the software acceptable for normal use?
- Is the software usable?
- Is the software well-structured and understandable?
- Have programming and documentation standards been followed in the development process?

Further, Quality Requirement priorities Vary from Domain to domain.

- Business Software - Data integrity, availability & Security, Transaction Performance & Ease of use
- Web Based Software – Security, Scalability & Content

- PC Software - Ease of use & Convenience
 - Real Time Systems - Precise response time, reliable & Time-critical performance
 - Embedded software - Online response with limited resources
2. Engineering Software - Precision & Accuracy

Next question is how do we produce this quality software?. What are the processes and steps we use to ensure that Quality software is produced? Quality issues need to be managed at different levels to ensure good quality software comes out. At organization level it involves (QA) it involves establishing a framework of processes and standards and Continuous monitoring and validation of these processes to make sure that they ensure quality of developed product. At project level, it involves (QC), it involves establishing a workable Quality plan that sets quality goals and correct application of specific quality processes defined in the plan and Ensuring that the project outputs conform to the specified standards

Quality management Tasks are

- Provide an independent check on the software development process
- Check the project deliverables to ensure that they are consistent with organizational standards and goals
- Done by quality team that is independent from the development team so that:
 - they can take an objective view of the software without being influenced by software development issues

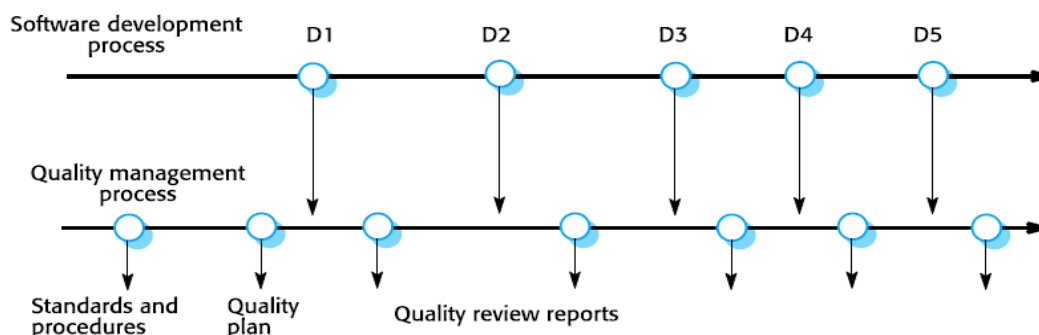


Fig 4.2: Quality management Tasks

Another important issue in ensuring Quality of the software developed is Verification & validation (V & V) .

It is nothing but Checking and analyzing products & processes executed during & after the development ,

to ensure that software being developed meets the specifications and Delivers the expected functionality.

Verification & validation are NOT the same thing. Boehm (1979) differentiated the two with a clear expression as follows :

1. **VERIFICATION** - "Are we building the product right". (The software should conform to its specification)
2. **VALIDATION** - "Are we building the right product". (The software should do what the user really requires.)

V & V – Has TWO principal objectives

- The discovery of defects in a system;
- The assessment of whether or not the system is useful and useable in an operational situation
- Goal is to establish confidence that the software is fit for purpose. (This does NOT mean completely free of defects. Rather, it must be good enough for its intended use)
- The type of use will determine the **degree of confidence** that is needed.

V & V Confidence Depends on following factors:

- Software function
- The level of confidence depends on how critical the software is to an organisation.
- User expectations
- Users may have low expectations of certain kinds of software.
- Marketing environment
- Getting a product to market early may be more important than finding defects in the program.

V & V Process Is a whole life-cycle process applied at each stage in the software process. Verification & validation (V & V). Starts with requirement review and continues through design reviews, code inspection and product testing. It is very expensive, complex, monotonous BUT very important and critical activity that is manpower intensive and it can never be totally automated

TWO approaches for V & V

1. Software inspections

- Concerned with analysis of the static system representation to discover problems (static verification)
- May be supplement by tool-based document & code analysis

2. Software testing

- Concerned with exercising and observing product behaviour (dynamic verification)
- The system is executed with test data and its operational behaviour is observed

Verification & Validation techniques

1. Inspection

Inspection involves program inspections, automated source code analysis, formal verification of programs. It can only check the correspondence between a program & its specifications (verification). It Cannot demonstrate the operational usefulness of the software being developed, Nor can it check emergent properties of the software like performance, reliability etc..

2. Testing

Testing is most widely used and main technique for V & V that involves exercising the program with data. It can reveal the presence of errors NOT their absence. It is the only validation technique for non-functional requirements as the software has to be executed to see how it behaves. It should be used in conjunction with Inspection (static verification) to provide full V&V coverage

Testing & Debugging are Two distinct processes. While Testing is concerned with establishing the existence of defects in a program, Debugging is concerned with locating & repairing these errors.

Planning V & V is Very important since more than half the budget is spent on this process in many cases. Careful planning is required to get the most out of testing and inspection processes. It Should start early in the development process. The plan should identify the balance between static verification and testing. Test planning is about defining standards for the testing process rather than describing product tests

Test Plan ia a link between development & testing, Planning document must contain following Details.

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.

Software Inspections Involve people examining the source representation with the aim of discovering anomalies and defects. It may be used before implementation as there is no need for system execution. It may be applied to any representation of the system (requirements, design, configuration data, test data, etc.). It is shown to be an effective technique for discovering program errors. Success depends on the quality of people performing inspection

Software Testing involves execution of code on the machine using test cases and test data prepared for these cases in advance. It is the only validation technique for non-functional requirements as the software has to be executed to see how it behaves. It Should be used in conjunction with static verification to provide full V&V coverage

Objectives of testing can be stated as :

- To demonstrate to the developer and the customer that the software meets its requirements (Leads to validation Testing)
- To discover faults or defects in software where the behavior is incorrect, undesirable or does not conform to its specification (leads to verification testing)

Testing can not demonstrate that the software is free of defects or it will behave as specified in every situation.

“Testing can only show the presence of errors not their absence”. Hence the goal of testing should be to convince developers & customers that the software is ready for use

Inspection V/s Testing

- Three advantages over testing:
 - a. In testing, one defect ,may mask another so several executions are required; Many different defects may be discovered in a single inspection
 - b. Inspection can be performed on incomplete versions too; But testing needs fully developed product
 - c. Inspection can also consider broader quality attributes like standards compliance, programming style etc., but testing can only look for specific functionality / bug
- BUT disadvantages of inspections are :
 - a. Inspection is highly dependent on the expertise and interest of the people performing inspections
 - b. Inspections can check conformance with a specification but not conformance with the customer’s real requirements.
 - c. Inspections cannot check non-functional characteristics such as: performance, usability, etc

The program inspection Process.

Program inspections are reviews, done to detect defects like logical errors, code anomalies, non-compliance of standards etc.. in the program. It was first developed by IBM in 70's and now fairly widely used method of verification. It is carried out formally by a team of at least 4 people. Team consists of members who represent different view points (Like for example of Author of the code, testing team representative, System expert and moderator). Inspection does not require execution of a system; Hence done before implementation. It may be used on any system element(doc) like Reqt. Design, configuration data, test data,Code.... . it is shown to be an effective technique for discovering program errors. Pre-conditions for inspection are :

- a. A precise specification must be available.
- b. Team members must be familiar with the organisation standards.
- c. Syntactically correct code or other system representations must be available.
- d. An error checklist should be prepared.
- e. Management must accept that inspection will increase costs early in the software process.
- f. Management should not use inspections for staff appraisal i.e. finding out who makes mistakes

Inspection Process (Inspection procedure)

1. System overview presented to inspection team.
2. Code and associated documents like checklists are distributed to inspection team in advance.
3. Inspection takes place and discovered errors are noted.
4. Modifications are made to repair discovered errors.
5. Re-inspection may or may not be required.

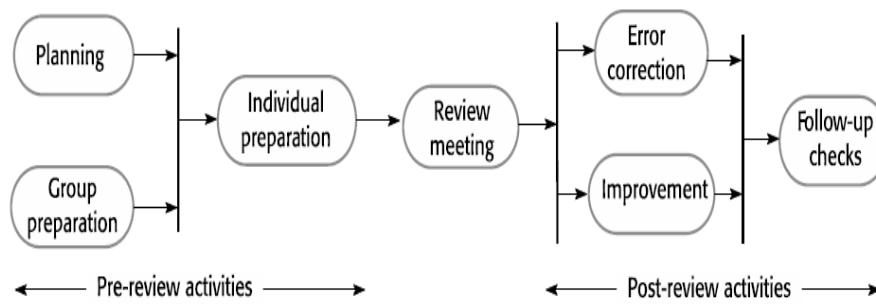


Fig 4.3: Inspection Process

Inspection Checklists are commonly used tool in Inspection. It Contains a list of common errors should be used to drive the inspection. It is Dependent on programming language and reflects the characteristic errors that are likely to arise in the language. In general, the 'weaker' the type checking, the larger the checklist. Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Fault class	Inspection check
Data faults	<ul style="list-style-type: none"> • Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	<ul style="list-style-type: none"> • For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	<ul style="list-style-type: none"> • Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?

Fault class	Inspection check
Interface faults	<ul style="list-style-type: none"> • Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	<ul style="list-style-type: none"> • If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	<ul style="list-style-type: none"> • Have all possible error conditions been taken into account?

Inspection – some remarks : Inspecting 500 lines costs about 40 man/hours effort; (hence it turns out to be less expensive than testing). Since program inspection is a public process of error detection, management needs to be sensitive. As organization gain experience of the inspection process, the process improvements can be attempted. Collection of data on types of errors can lead to process modifications that can cut errors at source

Static Analysis – is a process of examining a program to discover errors WITHOUT executing it (The essence of inspection) . It can be automated for some standard common errors resulting in **STATIC ANALYZERS** - Software tools that scan & parse the text of a program and detect possible faults and anomalies. It can detect whether the statement has been well formed. It can do control flow analysis & even compute the data set needed to test the program. It complements the error detection facilities of a compiler and serves as an effective V & V aid supplementing inspection. Automate static analysis includes Control Flow analysis, Data use Analysis, Interface analysis, Information flow analysis and Path analysis

Another Quality Assurance technique is IMPOSITION OF STANDARDS. Software standards are something set up and established by an authority as a rule or norm for the measure of quantity, functionality, value, or quality. Standardization is the wide use of components, parts, procedures, or processes in which there is regularity, repetition, and a successful practice and predictability. Standards Define the required attributes of a product / process that play an important role in quality management. Standards may be International, National, or organizational or Project standards

WHY USE STANDARDS? It avoids repetition of past mistakes, encapsulates best practices, and provides framework for defining what quality means in a particular setting i.e. that organization's view of quality. Further, it provides continuity by making it easy for new staff to understand the organisation by understanding the standards that are used.

Product and Process Standards

Product standards apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, & coding standards, defining the use of a programming language in coding

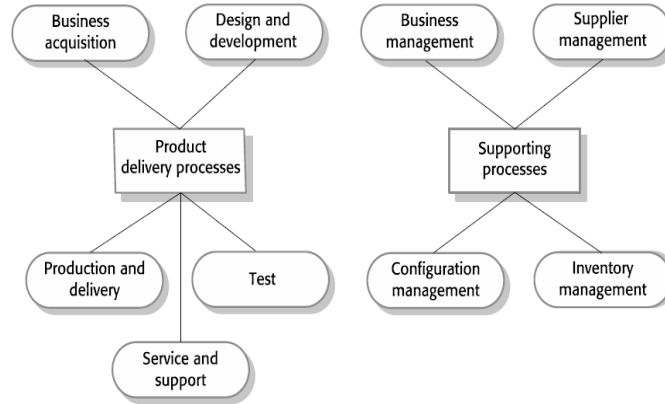
Process standards define the processes to be followed during software development. It may include definitions of specification, design and validation processes, process support tools and a description of the documents that should be written during these processes

Steps involved in Development of Standards are : Involving practitioners in development. Engineers should understand the rationale underlying a standard. Standards and their usage need to be reviewed regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners. Also there must be specialized tool support. Excessive clerical work is the most significant complaint against standards. Web-based forms are not good enough

Standardization has three Drawbacks :

1. Soon become more of document filling ritual
2. Need some automation tools; otherwise consumes too much time in filling forms
3. Needs constant revision; otherwise engineers feel it is irrelevant and just a ritual

ISO 9000 Standards framework : Provides An international set of standards to be used as a basis for developing quality management systems. It applies to organizations that design, develop and maintain products, including software. It is a framework for developing software standards. It sets out general quality principles, describes quality processes in general and lays out the organizational standards and procedures that should be defined. These should be documented in an organizational quality manual



ISO 9001 & Quality Management



Fig 4.4: ISO 9001 & Quality management

ISO 9001 Limitations

The ISO 9001 certification is inadequate because it defines quality to be the conformance to standards. It takes no account of quality as experienced by users of the software. For example: A company could define test coverage standards specifying that all methods in objects must be called at least once. This standard can be met by incomplete software testing that does not include tests with different method parameters. But as long as the defined testing procedures are followed and test records maintained, the company is ISO 9001 certified!!! .

Testing

Testing is the Most widely used and main technique for V & V. it Involves exercising the program with data .

TESTING Can reveal the presence of errors NOT their absence. It is The only validation technique for non-functional requirements as the software has to be executed to see how it behaves.It Should be used in conjunction with static verification to provide full V&V coverage .

Objectives of testing: is to demonstrate to the developer and the customer that the software meets its requirements (Leads to validation Testing) It is two fold.

- To discover faults or defects in software where the behavior is incorrect, undesirable or does not conform to its specification (leads to verification testing)
- Testing can not demonstrate that the software is free of defects or it will behave as specified in every situation.

“Testing can only show the presence of errors not their absence” Hence the goal of testing should be to convince developers & customers that the software is ready for use

Testing Policies constrain the extent and scope of testing by defining approach to testing . Only exhaustive testing can show a program is free from defects. **However, exhaustive testing is impossible,** Hence we adopt Testing policies like

- All functions accessed through menus should be tested;
- Combinations of functions accessed through the same menu should be tested;
- Where user input is required, all functions must be tested with correct and incorrect input

Testing process

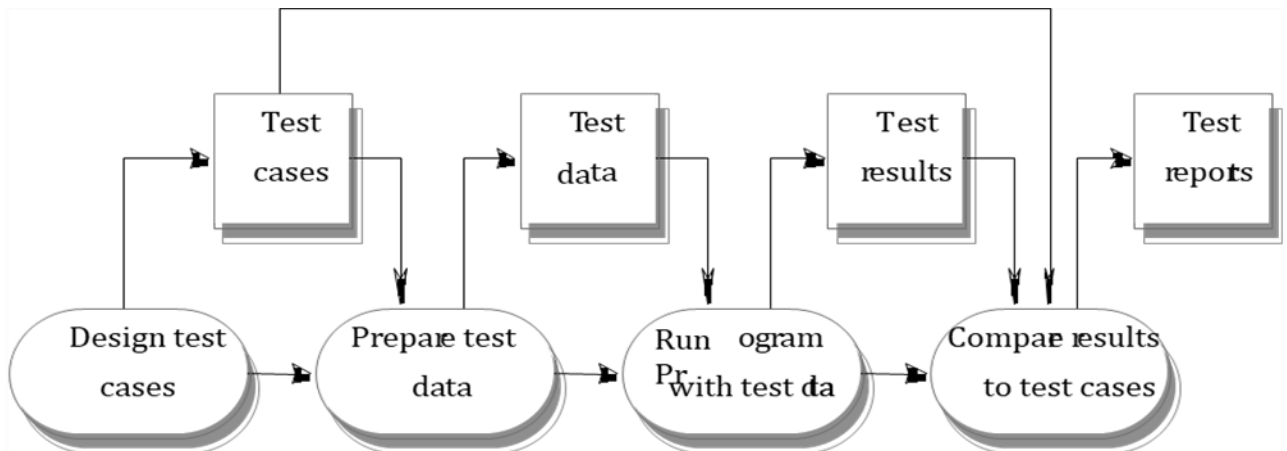


Fig 4.5: Testing process

Testing Phases

1. Component testing is testing of individual program components; It is usually the responsibility of the component developer (except sometimes for critical systems); Tests are derived from the developer's experience.
2. System testing is Testing of groups of components integrated to create a system or sub-system; Independent testing team takes up this responsibility and Tests are based on a system specification.

Testing individual functions / methods are simple. Tests are nothing but a set of calls to these methods with different set of parameters : Right values , Wrong value, Border values and Various combination of right, wrong and border values. Results are matched with expected results.

Component testing is the process of testing individual components in isolation. It is a defect testing process meant to discover errors Components may be:

- Individual functions or methods within an object;
- Object classes with several attributes and methods;
- Composite components with defined interfaces used to access their functionality
 - Developers of the component are responsible for this testing

Complete test coverage of a class involves Testing all operations associated with an object; Setting and interrogating all object attributes; and Exercising the object in all possible states.

Inheritance makes testing all the more difficult: When a super-class provides operations that are inherited by the sub-classes, All of these subclasses must be tested with All inherited operations using All parameters used by these methods because Subclass may make certain presumptions about other operations & attributes / Method may be overridden in sub-class

Composite component testing - Composite components are made up of a several interacting objects, whose functionality is tested basically through the interfaces these objects expose to the external world, Hence testing composite components is primarily concerned with testing those

Interface Testing : Here the Objective is to detect faults due to interface errors or invalid assumptions about interfaces. Different types of interfaces to be tested are:

- Parameter interfaces
 - Data passed from one procedure to another.
- Shared memory interfaces
 - Block of memory is shared between procedures or functions.
- Procedural interfaces
 - Sub-system encapsulates a set of procedures to be called by other sub-systems.
- Message passing interfaces

- Sub-systems request services from other sub-systems

Generally the errors can be classified into 3 categories:

- Interface misuse
 - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
 - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors
 - The called and the calling component operate at different speeds and out-of-date information is accessed

Guidelines for interface testing

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components are activated.

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 such that it Throw up maximum errors with minimum number of test cases. Design approaches generally considered are :

- Requirements-based testing;
- Partition testing;
- Structural testing.

Approach – 1 Requirements-based Testing is a general principle of requirements engineering is that requirements should be testable. Requirements-based testing is a

validation testing technique where you consider each requirement and derive a set of tests for that requirement.

Approach 2 – Partition testing involves dividing Input data and output results of a program often fall into different classes that have common characteristics such as: Positive numbers, Negative numbers, Menu selections etc... Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member. Test cases should be chosen from each partition.

Input partitions for search routine

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the First, Middle and Last elements of the sequence are accessed.
- Test with key element NOT in the sequence

Approach 3 – Structural testing is an approach where test cases are derived from the knowledge of the software's structure & implementation it is also known as white-box / glass-box / clear-box testing. Objective is to exercise all program statements, not all path combinations

Testing is an expensive process phase. Hence **automation** is necessary to reduce time and cost. Testing workbenches provide a range of tools to reduce the time required and total testing costs. Systems such as Junit support the automatic execution of tests. Most testing workbenches are open systems because testing needs are organisation-specific. They are sometimes difficult to integrate with closed design and analysis workbenches.

Software project management

Software project management is an essential part of Software engineering, Concerned with **activities involved in ensuring that software is delivered on time and within budget estimates** in accordance with the requirements of the organisations developing and procuring the software. This is very essential because all projects have budget and time constraints to be met.

Many techniques of engineering project management are equally applicable to software project management.

Technically complex engineering systems tend to suffer from the same problems as software systems. But there are Features unique to software projects like:

1. The product is intangible and uniquely flexible
2. Software engineering is not yet matured like other engineering discipline like mechanical, electrical engineering, etc.
3. The software development process is not standardised.
4. Many software projects are 'one-off' projects.

Though **Software project management** varies from organization to organizations, It **basically involves activities like:**

1. Proposal Planning.
2. Project planning and scheduling.
3. Project monitoring and reviews.
4. Project costing.
5. Risk Management

Proposal Writing: Most projects start with this activity to get management approval for starting a project.

The Proposal describes, The objectives of the project, How the project will be carried out, the Price and schedule estimates . It is a A critical task that needs good skills that can be developed ONLY through practice and experience. No guidelines can be set for this task.

An important aspect of this exercise is to come up with cost / price for the project. It involves Project pricing involves estimating how much the software will cost to develop, taking factors such as staff costs, hardware costs, software costs, etc. into account.

Though pricing is based on estimates, There is NO SIMPLE RELATIONSHIP between development cost and the price charged to the customer. Broader organizational, economic, political and business considerations influence the price charged.

Factors affecting software pricing

Factor	Description
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.

Factor	Description
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.

Pricing Strategies :

1. Under- pricing – (Quoting below the normal price) is done at times to gain a contract and keep staff busy & retain them for future opportunities OR to get entry to new market area.
2. Over Pricing - (Quoting above the normal price) is done at times to cover high risks involved in fixed price contracts.
3. Pricing to win – (Priced according to what the software developer believes the buyer is willing to pay even if this is less than the development costs, The software functionality may be reduced accordingly with a plan to add extra functionality being added in a later release. Additional costs may be added as the requirements change and these may be priced at a higher level to make up the shortfall in the original price.

Project Planning

Project Planning is the most important and probably the most time-consuming project management activity. It is a Continuous activity from initial concept through to system delivery. The Plan drives the development towards the project goals. Plans must be regularly revised as new information becomes available. Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget.

Types of Project Plans

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required..
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

Project Planning Process

```

Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled
loop
    Draw up project schedule
    Initiate activities according to schedule
    Wait ( for a while )
    Review project progress
    Revise estimates of project parameters
    Update the project schedule
    Re-negotiate project constraints and
deliverables
    if ( problems arise ) then
        Initiate technical review and possible
revision
    end if
end loop

```

The Project Plan sets out the resources available to the project; The work breakdown; and a schedule for the work. The Project Plan document should have following chapters:

1. Introduction.
2. Project organisation.
3. Risk analysis.
4. Hardware and software resource requirements.
5. Work breakdown.
6. Project schedule
7. Monitoring and reporting mechanisms.

Activities in a project should be organised to produce tangible outputs for management to judge progress. **Milestones** are the end-point of a process activity. **Deliverables** are project results delivered to customers.

Project Planning Process :

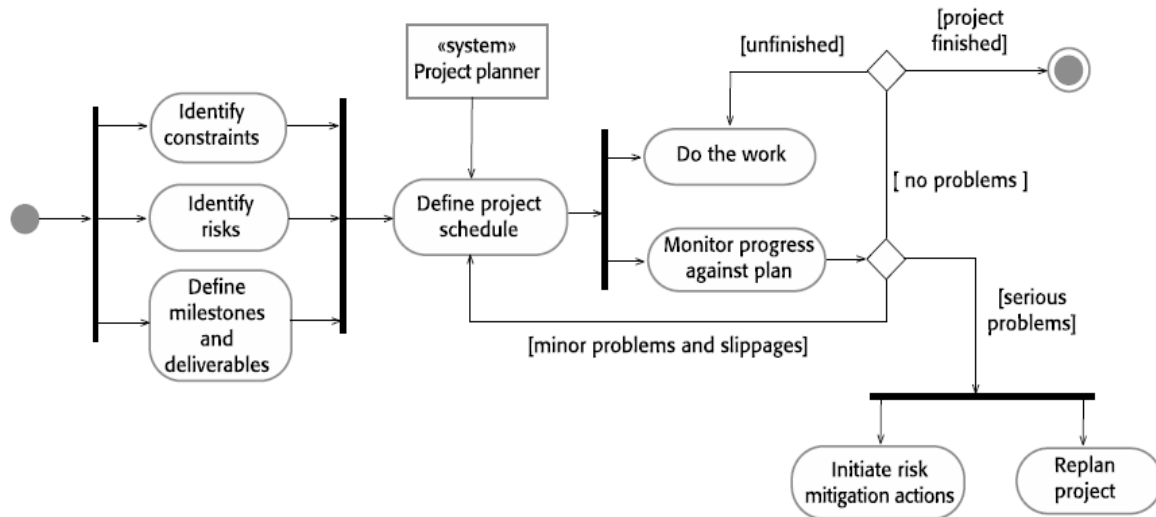


Fig 4.6: Project planning process

Project Scheduling

Project Scheduling is the task of estimating the time and resources needed to complete the activities and organizing them into a coherent sequence. The steps involved are:

1. Split project into tasks
2. Estimate time & resources required to complete each task
3. Organize tasks concurrently to make optimal use of workforce
4. Minimize task dependencies to avoid delays caused by one task waiting for another to complete

The entire exercise is dependent on Manager's experience dependent on project manager's intuition and experience.

The project scheduling process:

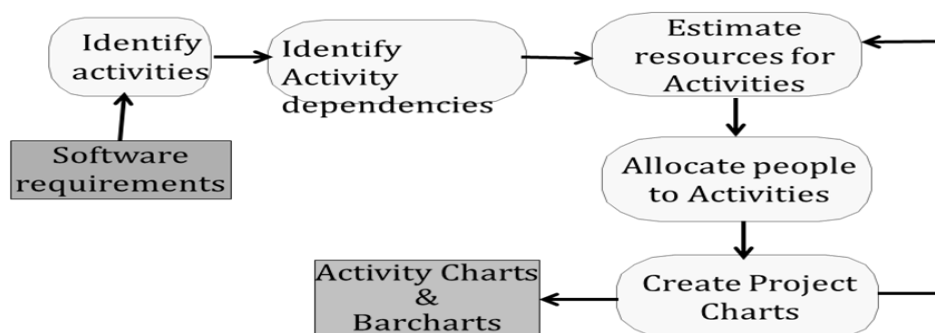


Fig 4.7: Project Scheduling process

Project Activities (tasks) are the basic planning element. Each activity has:

- A duration in calendar days or months,
- An effort estimate, which shows the number of person-days or person-months to complete the work,
- A deadline by which the activity should be complete,
- A defined end-point, which might be a document, the holding of a review meeting, the successful execution of all tests, etc.

- **Schedule representation** is done using graphical notations. They Show project breakdown into tasks. Tasks should not be too small are too large (They should take about a week or two.).

Activity charts show task dependencies and the critical path.

Bar charts show schedule against calendar time

Task duration data for example:

Activity	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

Corresponding BAR CHART

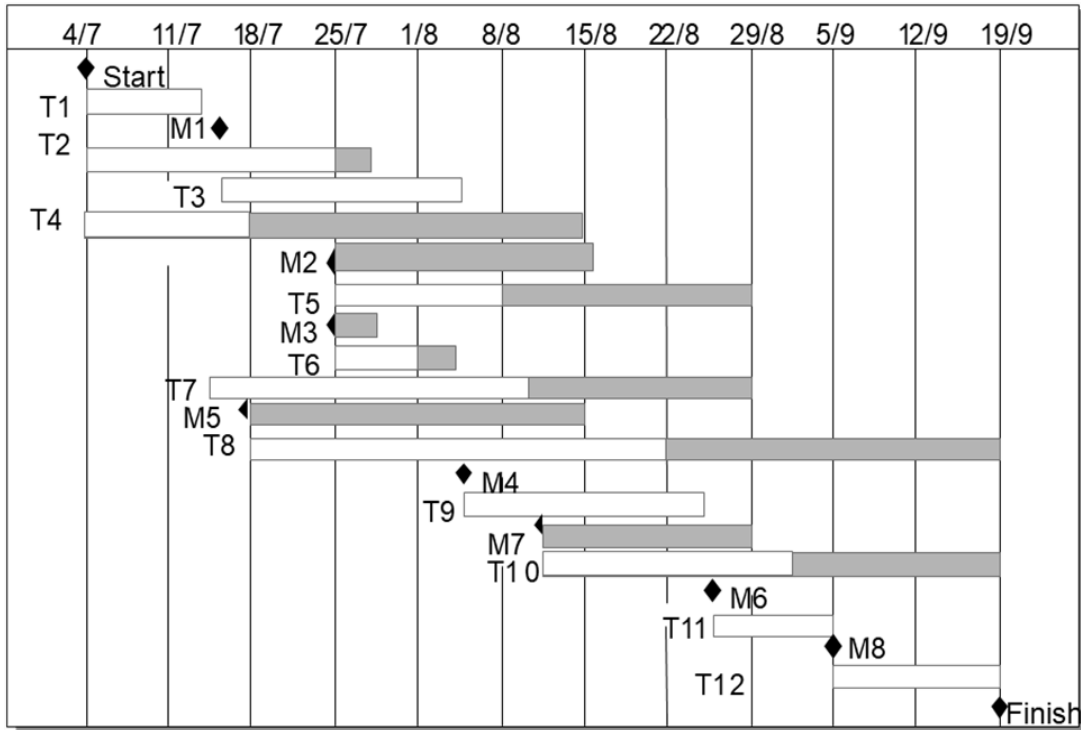


Fig 4.8: GANTT chart

Corresponding Activity Network:

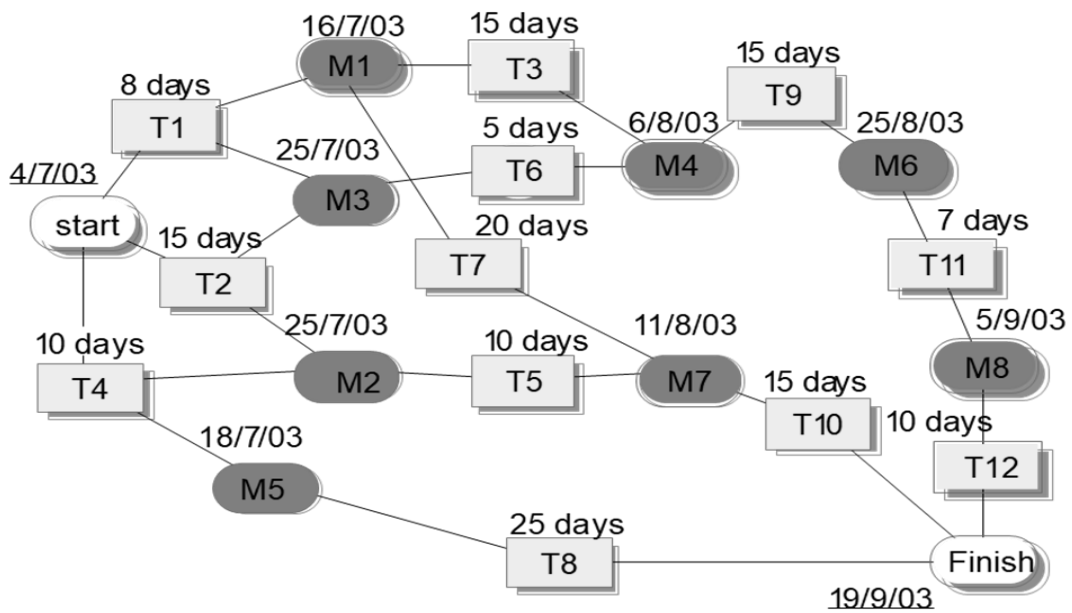


Fig 4.9: Activity Network

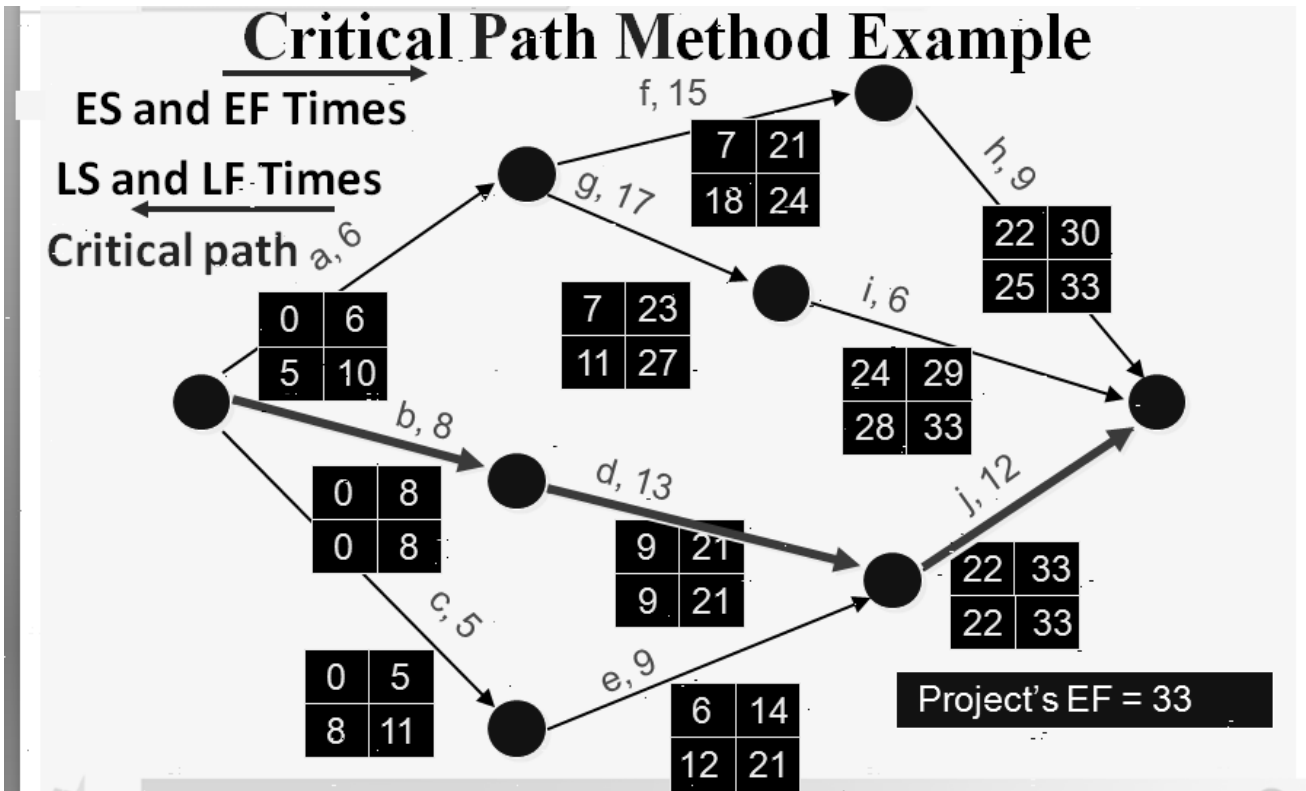


Fig 4.10: Critical Path Method Example

Staff Allocation Chart:

	Week 0	1	2	3	4	5	6	7	8	9	10	11
Jane	T1		T3			T9			T10			T12
Ali	T1		T8									
Geetha	T2		T3	T6	T7			T10				
Maya			T8									
Fred	T4		T8						T11		T12	
Mary				T5								
Hong			T7									
				T6								

Fig 4.11: Staff Allocation Chart

Project Costing and Estimation

Project Costing Involves answering following fundamental questions:

1. How much effort is required to complete an activity?
 2. How much calendar time is needed to complete an activity?
 3. What is the total cost of an activity?
- Project estimation and scheduling are interleaved management activities
 - Three Parameters involved in computing the cost of software development project
 1. Hardware & Software cost (including maintenance)
 2. Travel & training costs
 3. Effort costs

Costing & Pricing :

- Costing
 - The process of estimating that discovers the cost, to the developer, of producing a software system
 - Must be done objectively with the aim of producing the accurate costs incurred during development
 - Must be regularly updated once the project is underway
- Pricing
 - The process of deciding the money to be charged to customer for the software development effort
 - **It is NOT simply cost + profit**
 - Broader organisational, economic, political and business considerations influence the price charged.

Cost Estimation Process:

- Determine size of the product.
- From the size estimate, determine the effort needed.
- From the effort estimate, determine project duration, and cost.

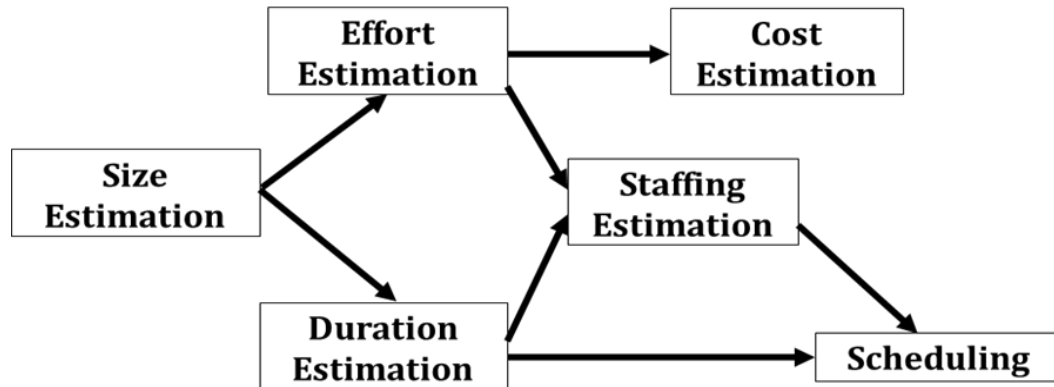


Fig 4.12: Project Costing

Effort Estimation: 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 requirements definition. The software may run on unfamiliar computers or use new technology. The people in the project may be unknown.

Size Estimation is very hard due to the elusive and abstract nature of the product. Different metrics used for measuring the size of the product and in turns the effort involved are

1. Size related measures
 - Based on some output from the software process like Lines of delivered source code (LOC), object code size
2. Function-related measures
 - Based on an estimate of the functionality of the delivered software like function points, object points etc.

Two types of estimation techniques that can be used:

- *Experience-based techniques* - The estimate is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be
2. *Algorithmic cost modelling*

- A *formulaic approach* used to compute the project effort based on estimates of product attributes, Such as size, using process characteristics, like experience of staff involved etc.

Experience-based approaches rely on judgments based on experience of past projects and the effort expended in these projects. Typically: Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.

However, there is some **Problem with experience-based approaches**. New software project may not have much in common with previous projects. Software development changes very quickly. New projects often use unfamiliar techniques such as web services, application system configuration or HTML5. Hence, previous experience may not help you to estimate the effort required.

Algorithmic cost modelling Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers: $\text{Effort} = A \cdot \text{Size}^B \cdot M$

A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M, a multiplier reflecting product, process & people attributes. The most commonly used product attribute is code size. Most models are similar but they use different values for A, B and M.

Estimation Accuracy Estimation depends on size of the software. The size of a software system can only be known accurately when it is finished. Several factors influence the final size. Use of reused systems and components; Programming language; Distribution of system etc... .As the development process progresses then the size estimate becomes more accurate. The estimates of the factors contributing to B and M are subjective and vary according to the judgment of the estimator

Effectiveness of algorithmic models Algorithmic models are a systematic way to estimate the effort required to develop a system. However, they are complex and difficult to use because of many attributes and considerable scope for uncertainty in estimating their values. The practical application of algorithmic cost modelling has been limited to a relatively small number of large companies, mostly working in defence and aerospace systems engineering.

COCOMO Cost Modelling is an empirical model based on project experience. It is a Well-documented, ‘independent’ model which is not tied to a specific software vendor. It has a Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2. COCOMO 2 takes into account different approaches to software development, reuse, etc.

COCOMO 2 GROUP of Models Incorporates a range of sub-models that produce increasingly detailed software estimates. The sub-models in COCOMO 2 are:

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

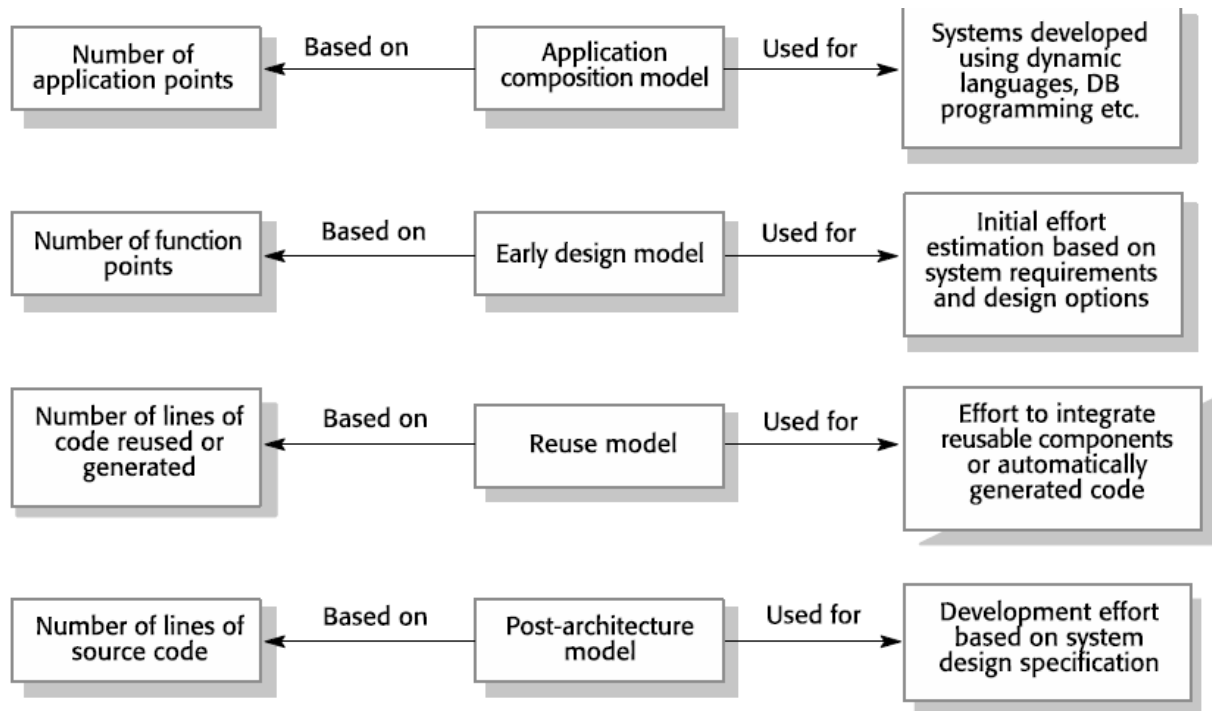


Fig 4.13: COCOMO 2 GROUP of Models

Staffing requirements

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

Project duration and staffing

Apart from effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required. COCOMO 2 formula for calendar time estimation is

$$TDEV = 3 \sqrt[PM]{(PM)(0.33+0.2*(B-1.01))} \text{ Where}$$

PM is the effort computation and

B is the exponent computed as discussed above (B is 1 for the early prototyping model).

This computation predicts the nominal schedule for the project.. The time required is independent of the number of people working on the project.

Software Metrics

Software metric is A standard of measure of a degree to which a software system or process possesses some property. Example : Lines of code in a program, number of persondays required to develop a component etc.. Metric can be used to quantify intangible qualities of software product & process. Metric may also be used to predict product attributes or to control the software process. Product metrics can be used for general predictions or to identify anomalous components.

Examples of Process metric:

1. The time taken for a particular process to be completed - can be the total time devoted to the process, calendar time, or the time spent on the process by particular engineers
2. The resources required for a particular process - might include total effort in person-days, travel costs, or computer resources.
3. The number of occurrences of a particular event - like: the number of defects discovered during code inspection, the number of requirements changes requested, the number of bug reports in a delivered system & av. nbr of LOC modified in response to a requirement change etc..

The computation and use of metric is based on the assumptions that :

- A software property can be measured accurately
- The relationship exists between what we can measure and what we want to know
- We can only measure internal attributes but are often more interested in external software attributes
- This relationship has been formalised and validated
 - It may be difficult to relate what can be measured to desirable external quality attributes

Metric Measurement process:

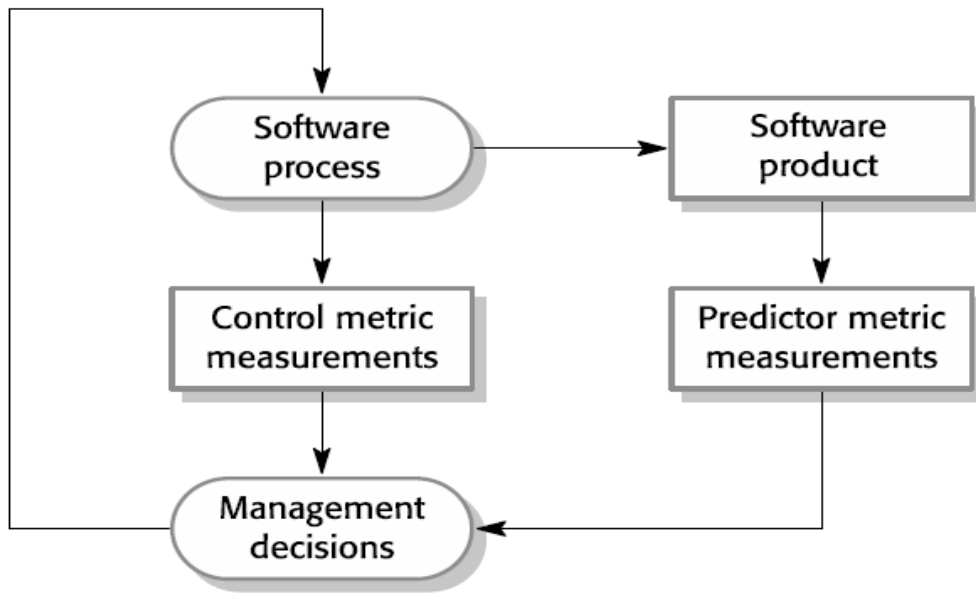


Fig 4.14: Metric measurement process

Metric relationships – Internal & External

External quality attributes

Internal attributes

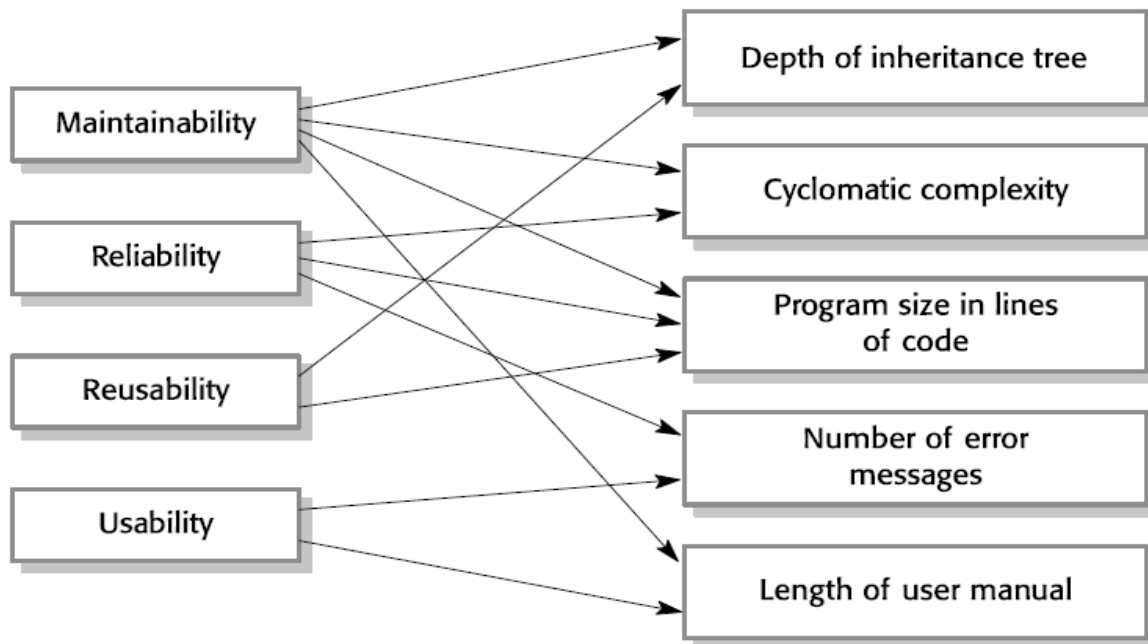


Fig 4.15: Metric Relationships

Why metrics are not commonly used in Industry ?

- Impossible to quantify the return on investment of introducing an organizational metrics program
- No standards for software metrics or standardized processes for measurement and analysis
- In many companies, software processes are not standardized and are poorly defined and controlled
- Most work on software measurement is focused on code based metrics and plan-driven development processes
- Introducing measurement adds additional overhead to processes

Empirical Software Engineering : is a field of academic interest. It is a A research area in which experiments on software systems and the collection of data about real projects has been used to form and validate hypotheses about software engineering methods and techniques. Software measurement and metrics are the basis of empirical software engineering. Research on empirical software engg., has not had a significant impact on software engineering practice. It is difficult to relate generic research to a project that is different from the research study