UNIT-1

Software Engineering

1.4 Software:

Definition:

" It is a set of instruction that when executed provide desired features, functions and performance"

(Or)

" It is a data structure that enable the programs to adequately manipulate information"

(Or)

" It is a documents that describe the operation and use of programs"

1.5 Software Characteristics:

* The characteristics of software are different from characteristics of hardware:

(1) Software is developed (Or) engineered; it is not manufactured in the classical sense

* In both software development and hardware manufacturing **high quality** is achieved using good design

* The manufacturing phase for hardware can introduce quality problems, which are **non-existent** (Or easily corrected) for software

* Both the activities **dependent on people**, but the relationship between people applied and work accomplished is entirely different

* Both activities require the construction of a **"Product"**, but the approaches are different

* Software engineering costs are concentrated in engineering

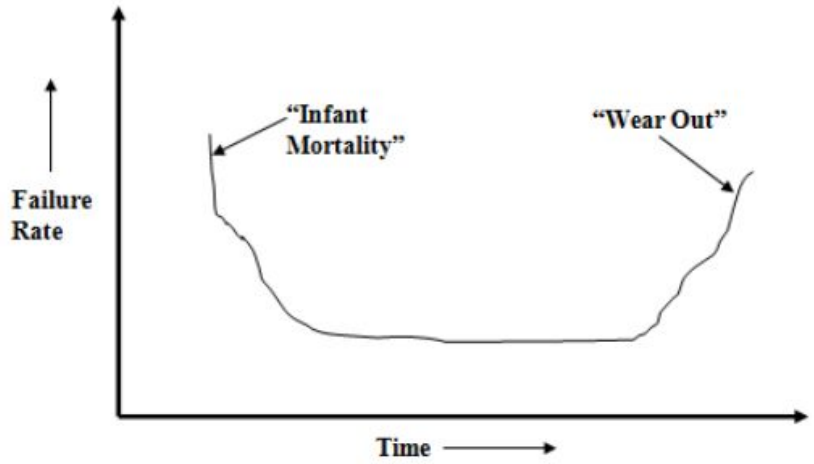
(2) Software doesn't "wear out"

* The hardware exhibits relatively **high failure rates** early in its life, these failures are caused due to design (Or) manufacturing defects

* Once defects are corrected the failure rate drops to a steady-state level, for some period of time

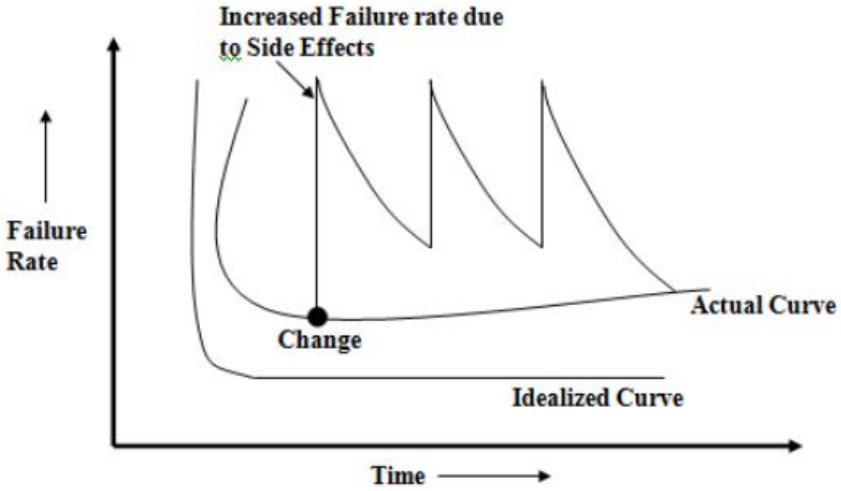
* As time passes, hardware components suffer from the cumulative effects of Dust, Vibration, abuse, temperature extremes and other environmental maladies

* The failure rate rises again, stated simply the hardware begins to **"wear out"**. This is shown below using **"Bath Tub Curve"**



* Software is not susceptible to the environmental maladies

* The failure rate curve for software, should take the form of "Idealized Curve"



* Early in the life of a program, undiscovered errors causes high failure rates
* Once these errors are corrected (without introducing other errors), the curve Flattens
* So, it is clear that;

=> Software **doesn't wear out**, but it does **deteriorate**

* Software will undergoes changes during its life time
* The errors will be introduced as changes are made. This causes the failure rate to spike
* Before the curve can return to the original steady state failure, another changes is requested causing the curve to spike again
* So due to changes the software is deteriorating
* The software maintenance involves more complexity than hardware maintenance because,

=> When hardware components wear out, it is replaced by spare part, but there are no software parts

(3) Although the industry is moving toward component – based construction, most software continues to be custom built

* In hardware world, component reuse is a natural part of the engineering process
* But in software world it has to be achieved on a broad scale
* A software component should be designed and implemented that can be reused in different programs

Example:

* Today's user interfaces built with reusable components that enable;

=> Creation of windows
=> Pull-down Menus &
=> Wide variety of interaction mechanism

1.7 Software Myths:

* It is **beliefs about software**
* The process used to built it, can be traced to the earliest days of computing
* The myth – have a number of attributes that have made them insidious [i.e. proceeding inconspicuously but harmfully]
* For instance myths appear to be reasonable statements of facts [Sometimes containing elements of truth]

Three software Myths:

1. Management Myths **2. Customer Myths** **3. Practitioner's Myth**

(1) Management Myths:

* Managers in most disciplines are often under pressure to maintain budget, keep schedules from slipping and improve quality
* A software manager often grasp at belief in a software myth, if that belief will lessen the pressure

Myth 1:

We already have a book that is full of standards and procedures for building software…..Won't that provide my people with everything they need to know?

Reality:

=> The book of standards may well exists……But is it used?
=> Are software practitioners aware of its existence?
=> Does it reflect modern software engineering practices?
=> Is it complete? Is it adaptable?
=> Is it streamlined to improve time to delivery while still maintaining a Focus on quality?

* In many cases the answer to all of these questions is **NO**

Myth 2:

If we get behind schedule, we can add more programmers and catch up [Some times called 'Mongolian horde Concept"]

Reality:

* Adding people to a late software project makes it later
* As new people are added, people who were working must spend time in educating the new comers, there by reducing the amount of time spent on productive development effort
* People can be added, but only in a planned and well coordinated manner

Myth 3:

If I decide to out source the software project to a third party, I can just relax and let that firm build it

Reality:

* If an organization does not understand, how to manage and control software projects internally, it will invariably struggle when it out sources software projects

(2) Customer Myths:

* The customer who requests computer software may be,

=> a person
=> a technical group
=> a marketing / sales department (Or)
=> an outside company

* In many cases customer believes myths about software
* Myths lead to **false expectations** (by the customer) and ultimately, **dissatisfaction** with the developer

Myth 1:

A general statement of objectives is sufficient to begin writing programs – We can fill in the detail later

Reality:

* An ambiguous statement {i.e. having two meanings} leads to a serious of problems
* But Unambiguous statements are developed only through effective and continuous communication between customer and developer
* So comprehensive and stable statements of requirements is not always possible

Myth 2:

Project requirements continually change, but changes can be easily accommodated because software is flexible

Reality:

* When requirements changes are requested early [before design (Or) code has been started] cost impact is relatively small
* When requirement changes are requested after design (Or) code has been started cost impact is too high
* The change can cause upheaval [i.e. Violent change (Or) disturbance] that require additional resources and major design modification

(3) <u>Practitioner's Myth:</u>

Myth 1:

Once we write the program and get it to work our job is done

Reality:

* Industry data indicate that between 60 and 80 percent of all efforts expended on software will be expanded after it is delivered to the customer for the first time

Myth 2:

Until I get the program running – I have no way of assessing its quality

Reality:

* Apply any one of the effective software quality mechanism, from the beginning of the project
* Software quality reviews are more effective than testing for finding certain classes of software errors

Myth 3:

The only deliverable work product for a successful project is the working program

Reality:

* A work program is part of software configuration, that includes many elements
* But documentation only provides a foundation for successful engineering and support for software

Myth 4:

Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down

Reality:

* Software engineering is not about creating documents, it about creating quality

1.6 The Changing nature of Software

* There are seven broad categories of computer software, that continuing challenges for software engineers are:

(1) System Software: It is a collection of programs **written to service other programs**

=> Some system software processes complex, but determinate information structure and some other processes largely indeterminate data

Example:

=> Compilers, editors and file management utilities

Characteristics:

=> Heavy interaction with computer hardware
=> Heavy usage by multiple users
=> Complex data structures
=> Multiple external interfaces
=> Concurrent operation

(2) Application Software:

* It is a **stand alone program**, that solve a specific business need
* This software process business (Or) Technical decision making. In addition it is used to control business functions in real time

Example:

=> Point – of – sale transaction processing
=> Real – time manufacturing process control

(3) Engineering / Scientific Software:

* This software used in various applications such as;

=> Astronomy
=> Molecular biology
=> Automated Manufacturing etc.

* Modern application within the scientific / engineering area is moving away from conventional numerical algorithm
* Computer aided design, system simulation and other interactive application, begun to take on real – time

(4) Embedded Software:

* This Software resides **within a product (Or) System**
* It is used to **implement and control** features and functions for the end user and for the system itself

Example:

=> Keypad control for a microwave oven
=> Digital Functions in an automobile such as fuel control, dash board Displays and braking system etc.,

(5) Product-line Software:

* It is designed to **provide a specific capability**, for use by many different customers
* This software **focuses on esoteric market place** and address mass consumer market

Example:

=> Word processing
=> Spread sheets
=> Computer Graphics
=> Multimedia and entertainment etc.,

(6) Web Applications:

* This Software span a wide array of applications

* Web applications are evolving into sophisticated computing environment, that not only provide stand alone features, computing functions and content to end users, but also integrate corporate database and business application

(7) Artificial Intelligence Software:

* This software makes use of **non numerical algorithms**, to solve complex problems
* Applications with in this area include;
    => Robotics
    => Expert Systems
    => Pattern recognition
    =>Theorem proving and game playing

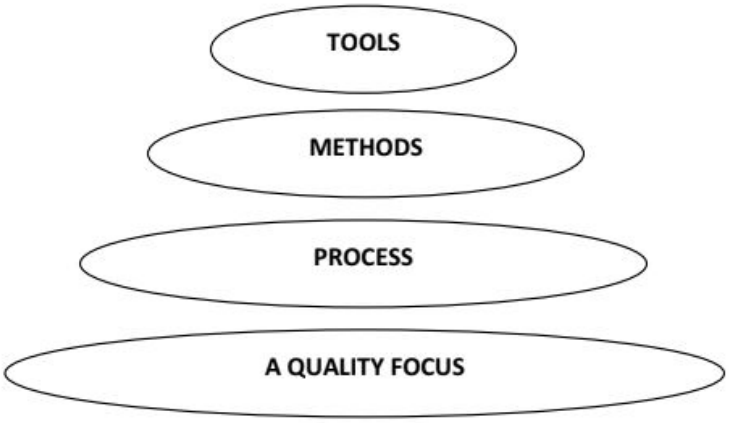
****

Note:

Legacy Software's:

* This software system developed decades ago [i.e., **older programs**] and it have been continually modified to meet changes in business requirements and computing platforms
* The Proliferation of such systems is causes headache for large organizations who find them costly to maintain and risky to evolve

1.8 Software Engineering – A Layered Technology:



(i) Quality Focus:

* Any engineering approach [including software engineering] must rest on an organizational commitment to quality

* Total quality management promote a continuous process improvement, this leads to development of effective approaches to software engineering

* The bedrock that supports software engineering is a quality focus

(ii) Process:

* The process layer is the foundation for software engineering

* It is the glue that holds the technology layers together and enables rational and timely development of computer software

* It defines a framework that must be established for;

  - Effective delivery of software engineering technology

* The software process forms the basis for management control

* It establishes the context in which

- Technical methods are applied
- Work products are produced [i.e. models, documents, data, reports, forms etc..]
- Milestones are established
- Quality is ensured and change is probably managed

(iii) Methods:

* It contains a broad array of tasks that include;

- Communication
- Requirement analysis
- Design modeling
- Program Construction
- Testing and support

* It depends on a set of basic principles that

- Govern each area of the technology
- Include modeling activities and other descriptive techniques

(iv) Tools:

* It provide automated (Or) semi automated support for the process and the methods

* When tools are integrated information created by one tool can be used by another

1.9 A Process Framework:

* It identifying a small number of framework activities that are applicable to all software projects, regardless of their size (Or) complexity
* The process framework include a set of umbrella activities, that are applicable across the entire software process

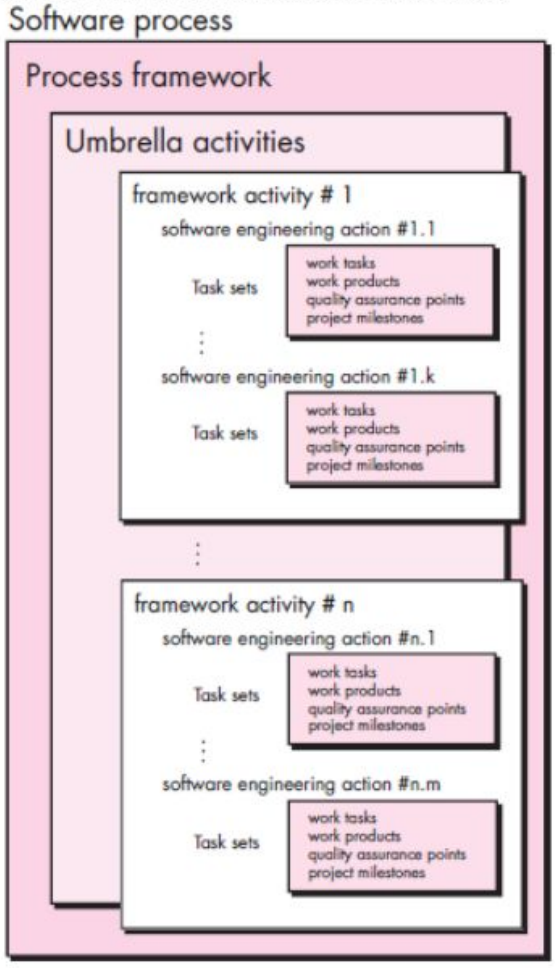
Framework Activity:

* It contains a set of software engineering actions [A collection of related tasks that produces a major software engineering work product]

Example:

Design is a software engineering action

* Each action contain individual work tasks
* The work tasks accomplish some part of the work implied by the action



1.10 Generic Framework Activities:

(1) Communication:

* It involves heavy communication and collaboration with the customer
* Also it includes requirement gathering and related activities

(2) Planning:

* It describes the

- Technical tasks to be conducted
- The risks that are expected
- Resources that will be required
- Work products to be produced
- Work Schedule

(3) Modeling:

* It describes the creation of models – that allow the developer and the customer to understand software requirements and design for those requirements

(4) Construction:

* It combines
  - Code generation [either manual (Or) automated]
  - Testing [Required uncovering errors in the code]

(5) Deployment:

* The software is delivered to customer
* Customer evaluates the delivered product
* Customer provides feedback based on the evaluation
* These generic framework activities can be used during the;
  - Development of small programs
  - Creation of large web applications
  - Engineering of large complex computer based systems
* The software process is different in each case, but framework activities remain same

1.11 Umbrella Activities:

(1) Software project tracking and control:

* Allow the software team to progress against the project plan
* If necessary take action to maintain schedule

(2) Risk Management:

* Find risks that may affect the outcome of the project (Or) quality of the product

(3) Software quality assurance:

* It defines and conducts the activities required to ensure software quality

(4) Formal Technical Reviews:

* Before propagated to the next action (Or) activity, remove uncover errors

(5) Measurements:

* It defines and collects process, projects and product measures that help the team in delivering software
* It can be used in conjunction with other framework and umbrella activities

(6) Software configuration management:

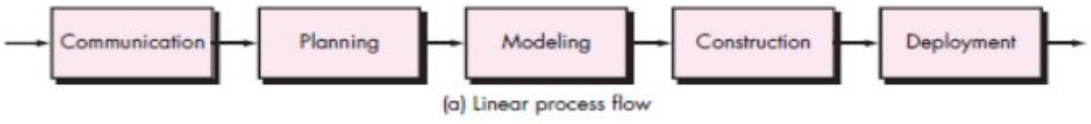
* It manages the effects of change throughout the software process

(7) Reusability management:

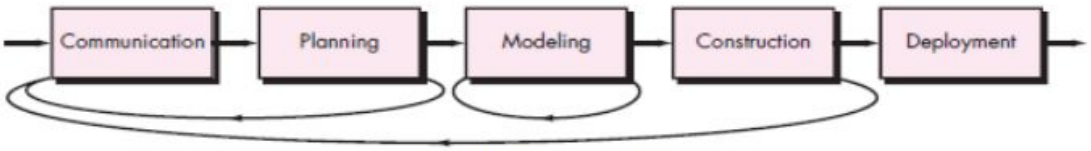
* It establishes mechanism to achieve reusable components
* It defines criteria for work product reuse

(8) Work product preparation and production:

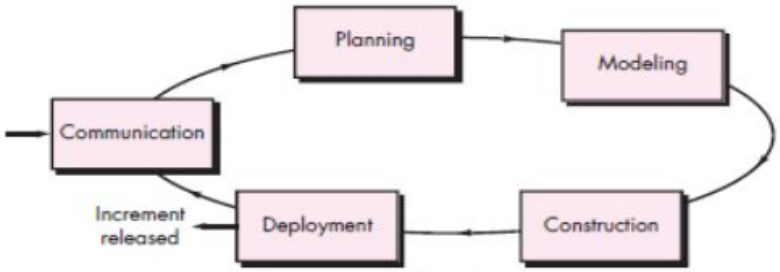
* It includes the activities required to create work products such as,
  - => Models
  - => Documents
  - => Logs, forms and lists



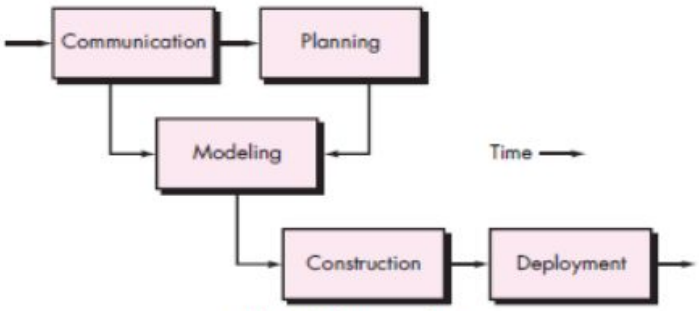
(a) Linear process flow



(b) Iterative process flow



(c) Evolutionary process flow



(d) Parallel process flow

2.1 THE WATER FALL MODEL

* It is sometimes called the classic life cycle

* It suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progress through

=> Planning

=> Modeling

=> Construction and

=> Deployment

Problems encountered in waterfall model:

(1) Real projects rarely follow the sequential flow. As a result changes cause confusion as the project team proceeds

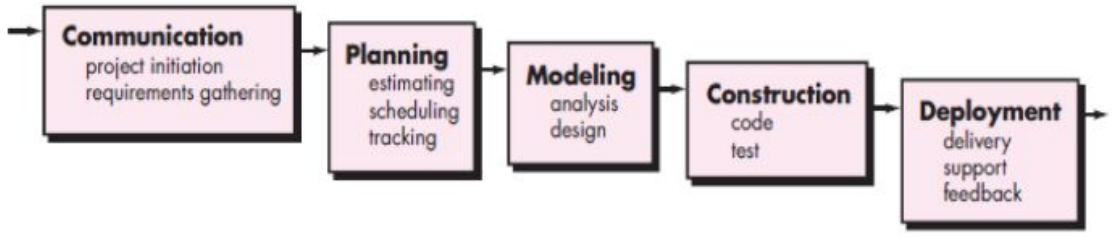
(2) It is difficult for the customer to state all requirements explicitly

(3) The customer must have patience

* The linear nature of the water fall model leads to " Blocking State" in which some project team members must wait for other members of the team to complete dependent task

* The water fall model can serve as a useful process model in situations where

=> Requirements are fixed and work is to proceed to completion in a linear manner



2.2 INCREMENTAL PROCESS MODELS

* The incremental model combines elements of the water fall model applied in an iterative fashion

* This model applies linear sequences in a staggered fashion as calendar time progresses

* Each linear sequence produces deliverable "increments" of the software

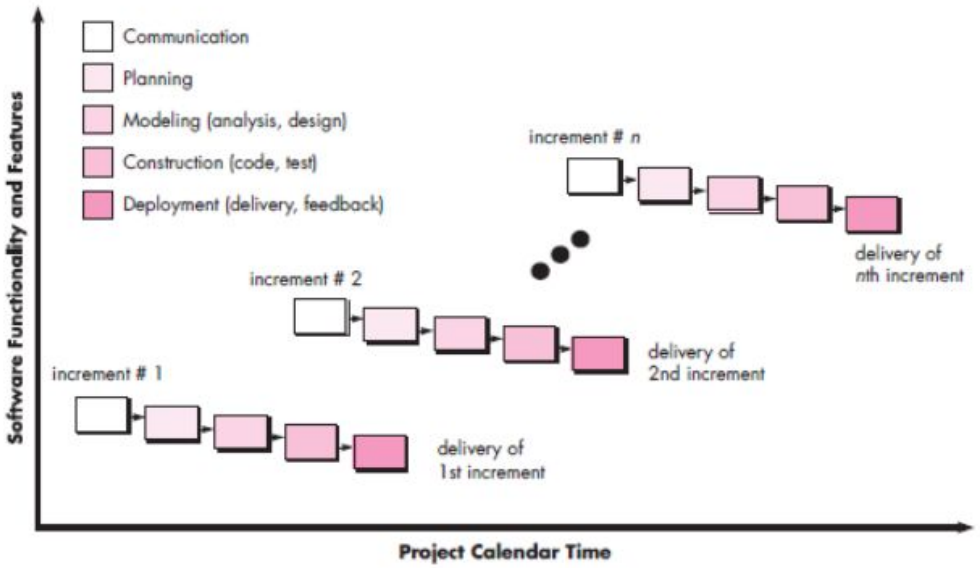
Main Idea:

* When an incremental model is used the first increment is called " CORE PRODUCT"

* i.e. the basic requirements are addressed but main supplementary features remain undelivered

* The core product is used by customer (Or) undergoes detailed evaluation

* As a result of evaluation a plan is developed for next increment



 => Communication

 => Planning

 => Modeling {Analysis, Design}

 => Construction {Code, Test}

 => Deployment {Delivery, Feedback}

* The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of the additional features and functionality

* This process is repeated for each increment delivery, until the complete product is developed

* Unlike prototyping model, the incremental model focuses on the delivery of an operational product with each increment

* This model is particularly useful, when staffing is unavailable for a complete implementation by the business deadline that has been established for the project

* Early increments can be implemented with fewer people. If core product is well received additional staff can be added to implement the next increment

* Increments can be planned to manage technical risks

* For example a major availability of new hardware is under development, whose delivery date is uncertain

* So plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end-users without inordinate delay

Prototype Model:

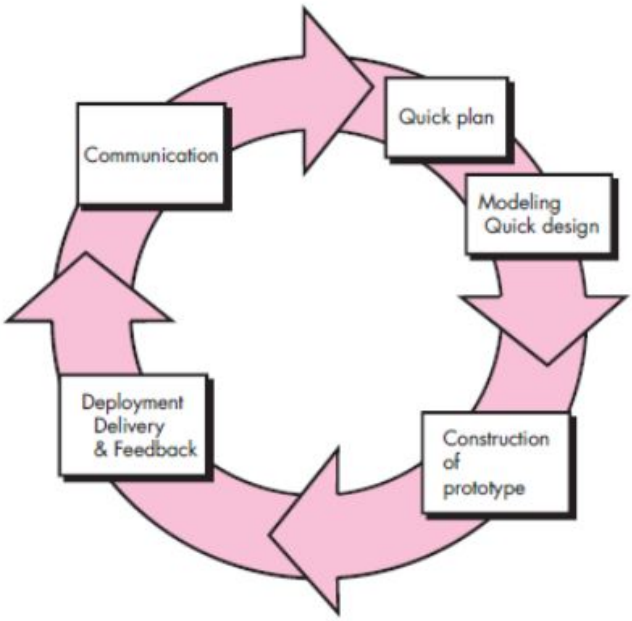
* The prototype model may offer the best approach, if

=> Customer defines a set of objectives for software, but does not identify detailed in put, processing (Or) output requirements

=> In other cases, the developer may be unsure of the efficiency of an algorithm (Or) the form that human-machine interaction should take

* The prototyping model assists the software engineer and the customer to better understand

=> What is to be built, when requirements are fuzzy?



Communication:

* The prototype model begins with communication

* The software engineer and customer meet and define the overall objectives for the software

=> Identify what ever requirements are known

=> Outline areas where further definition is mandatory

Quick design:

* The quick design focuses on a representation of those aspects of the software that will be visible to the customer / end user

Example:

* Human interface Layout (Or0 Output display formats

Construction of prototype:

* Ideally the prototype serves as a mechanism for identifying software requirements

* If the working prototype is built, the developer uses the existing programs fragments that ensure working programs to be generated quickly

Deployment:

* The prototype is deployed and evaluated by the customer

* Feedback is used to refine requirements for the software

Drawbacks:

(i) The customer sees what appears to be a working version of the software, unaware that the prototype is held together

(ii) The developer makes the implementation compromises in order to get a prototype working quickly

(iii) An inefficient algorithm may be implemented simply to demonstrate capability

Spiral Model:

Definition:

* It is evolutionary software process model that combines the iterative nature of prototyping with the controlled and systematic aspect of the water fall model

Two main features

(i) A **cyclic approach** for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk

(ii) An **anchor point** mile stones for ensuring stake holders commitment to feasible and mutually satisfactory system solutions

Main idea:

* A spiral model is divided into a set of frame work activities, defined by the software engineering team

* Each frame work activities represent one segment of spiral path

* As evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction beginning at the centre

* The first circuit around the spiral might result in development of a product specification

* Subsequent passes might be used to develop a prototype and more sophisticated versions of the software

* Each pass through the planning region results in adjustments to the project plan

* Cost and Schedule are adjusted based on feedback derived from the customer after delivery

* Using spiral model software is developed in a series of evolutionary release

* During each iteration the release might me a paper model (Or) prototype

* During later iterations, increasingly more complete versions of the engineered system are produced

* Unlike other process model that end when software is delivered, the spiral model can be adapted to apply throughout the life of the software

* In the above fig first circuit represents concept development project

* Once concept is developed into actual product, process proceeds outwards on spiral, A new product development project commences

* Later circuit around the spiral might be used to represent "product enhancement project"

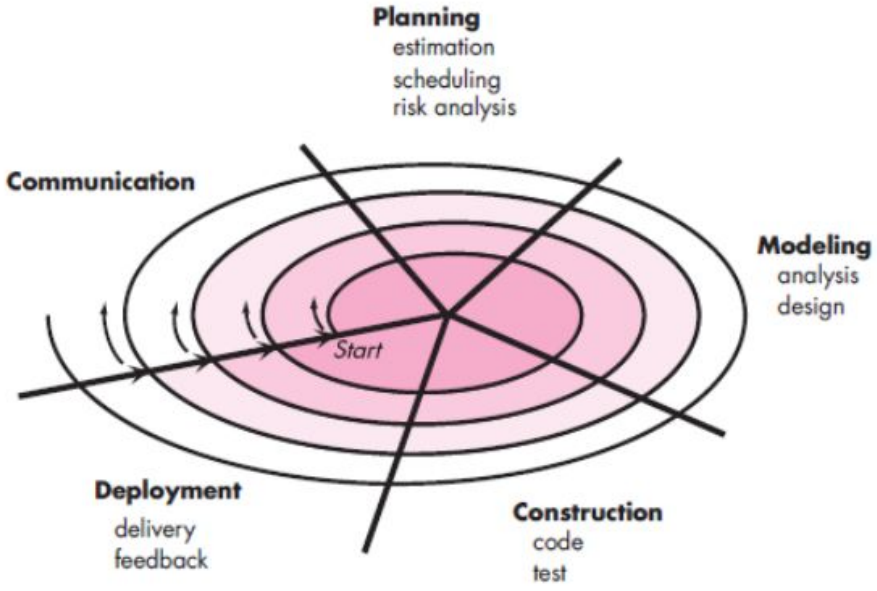
* Whenever a change is initiated the process starts at the appropriate entry point

Advantages:

(i) It is a realistic approach to the development of large-scale system and software

(ii) It enables the developer to apply the prototyping approach at any stage in the evolution of the product

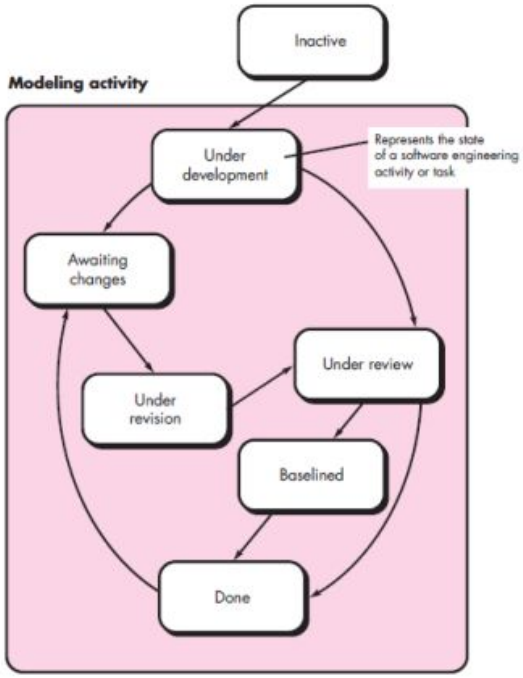
(iii) It maintains the systematic stepwise approach suggested by classic life cycle and incorporates it into iterative framework



CONCURRENT DEVELOPMENT MODEL

* It is also called as "**Concurrent Engineering**"

* It can be represented schematically as a series of framework activities, software engineering actions and tasks and their associated states



* At any given time the modeling activity may be in any one of the states

* All activities exist concurrently, but reside in different states

Example:

* Initially in a project, the communication activity has completed its first iteration and exists in awaiting change state

* When initial communication was completed, the modeling activity exists in none state, makes a transition from none state into the under development state

* If customer indicates changes in requirements, the modeling activity moves from under development into awaiting change state

* A series of events will trigger transition from state to state for each of the software engineering activities, actions (Or) tasks

* During early stages of design [a software engineering action that occurs during the modeling activity] an inconsistency in the analysis model will trigger the analysis action from the done state into the awaiting change state

Advantages:

(1) The concurrent process model provides an accurate picture of the current state of a project

(2) It defines the software engineering activities, actions and tasks as a network, instead of sequence of events

(3) Each activity, action (Or) tasks on network exists simultaneously with other activities

THE UNIFIED PROCESS

* It is an attempt to draw on the best features and characteristics of conventional software process models

* But characteristics them in a way that implements many of the best principles of agile software development

* The unified process emphasizes the important role of software architecture and help the architect focus on,

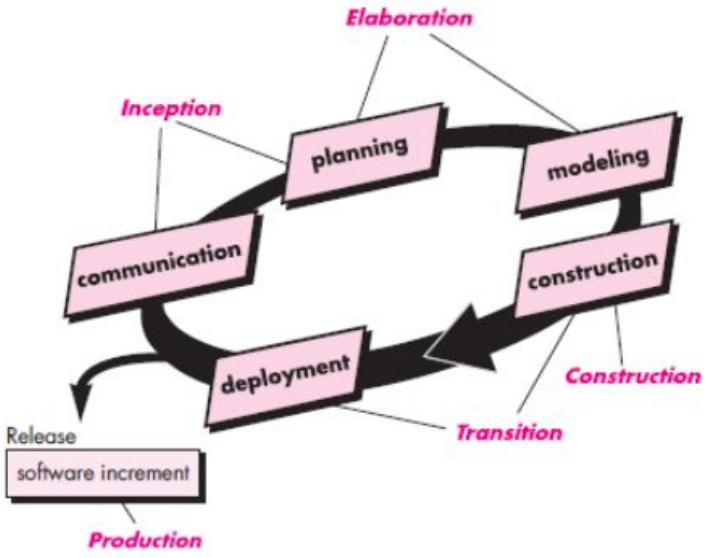
=> Right goals

=> Understandability

=> Reliance to future changes and

=> Reuse

* It suggests a process flow that is iterative and incremental providing the evolutionary feel



Phases of the Unified Process:

1. Inception Phae 2. Eloboration Phase 3. Construction Phase 4. Production Phase

(1) Inception Phase:

* This phase combines both customer communication and planning activities

* By combining these the

=> Business requirements for the software are identified
=> A rough architecture for the system is proposed
=> A plan for the iterative, incremental nature of the ensuring project is Developed

* In general a use-case describes a sequence of actions that are performed by an actor [e.g. a person, a machine]
* Use-case helps to identify the scope of the project and provide a basis for project planning

(2) Elaboration Phase:

* It refines and expands the preliminary use-cases that were developed on inception phase
* It expands the architectural representation to include five different views of the software:
  => Use-case model
  => Analysis Model
  => Design Model
  => Implementation Model
  => Deployment Model
* The modification to the plan may be made at this time

(3) Construction Phase:

* This phase develops (Or) acquires the software components that will make each use-case operational for end users
* All necessary and required features and functions of software release are implemented in source code
* After implementing components, unit tests are designed and executed for each
* In addition

(4) Transition Phase:

* The software is given to end-users for Beta testing
* The users feed back the reports both defects and necessary changes
* This phase allow software team creates the necessary support information

Example:

=> User Manuals
=> Trouble shooting guides
=> Installation procedures

(5) Production Phase:

* During these phase the
  => Ongoing use of the software is monitored
  => Support for the operating environment [infrastructure] is provided
  => Defect reports and requests for changes are submitted and evaluated

Personal and team process models

PSP

What Is Personal Software Process (PSP)?

• The Personal Software Process (PSP) shows engineers how to

- manage the quality of their projects
- make commitments they can meet
- improve estimating and planning
- reduce defects in their products PSP emphasizes the need to record and analyze the types of errors you make, so you can develop strategies to eliminate them.

Personal Software Process

• Because personnel costs constitute 70 percent of the cost of software development, the skills and work habits of engineers largely determine the results of the software development process.

• Based on practices found in the CMMI, the PSP can be used by engineers as a guide to a disciplined and structured approach to developing software. The PSP is a prerequisite for an organization planning to introduce the TSP.

Levels of Personal Software Process :
Personal Software Process (PSP) has four levels-

1. **PSP 0 –**
   The first level of Personal Software Process, PSP 0 includes Personal measurement , basic size measures, coding standards.
2. **PSP 1 –**
   This level includes the planning of time and scheduling .
3. **PSP 2 –**
   This level introduces the personal quality management ,design and code reviews.
4. **PSP 3 –**
   The last level of the Personal Software Process is for the Personal process evolution.

TSP

What Is Team Software Process (TSP)?

• The Team Software Process (TSP), along with the Personal Software Process, helps the high performance engineer to - ensure quality software products - create secure software products - improve process management in an organization

PSP Framework Activities	TSP Framework Activities
Planning	Launch high level design
High level Design	Implementation
High level Design Review	Integration
Development	Test
Postmortem	postmortem

Immature Software Organizations

- Software processes are generally improvised
- If a process is specified, it is not rigorously followed or enforced
- The software organization is reactionary
- Managers only focus on solving immediate (crisis) problems
- Schedules and budgets are routinely exceeded because they are not based on realistic estimates
- When hard deadlines are imposed, product functionality and quality are often compromised
- There is no basis for judging process quality or for solving product or process problems
- Activities such as reviews and testing are curtailed or eliminated when projects fall behind schedule

To balance all these pressures and conflicting forces and handling software development projects a team has to be self-directed.

A self-directed team should have these qualities:

- Understands product and business goals
- Produces their own plans for addressing the goals
- Makes their personal commitments
- Directs their own projects
- Consistently uses processes and methods that they select

- Manages quality.

Team software process builds and maintains self-directed teams. A successful self-directed team requires capable and skilled team members. Their commitment, discipline, and skills come together to produce high-quality software. Therefore, high-quality software products are a team effort. TSP creates an environment that supports disciplined and self-directed teamwork.

UNIT-2

Software Engineering

Functional and Non-Functional Requirements

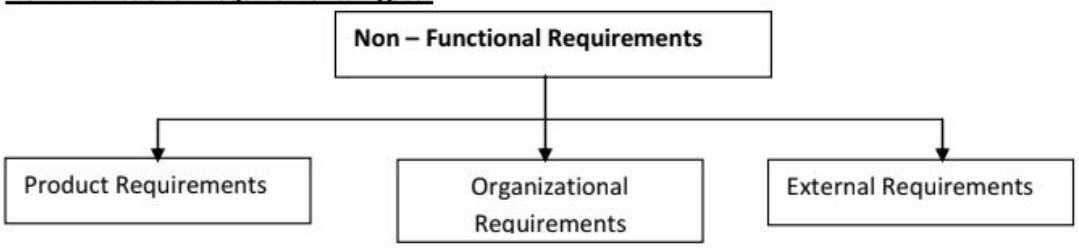
Functional Requirements:

* These are statements of services the system should provide
  => how the system should react to particular inputs and
  => how the system should behave in particular situations
* In some cases, the functional requirements may also explicitly state
  => What the system should not do
* The functional requirements definition of a system should be both
  => Complete [i.e. It means that all services required by the user should be Defined]
  => Consistent [i.e. it means that requirements should not have contradictory definitions]

2.7 Non – Functional Requirements:

* These are constraints on the services (Or) functions offered by the system
* They include
  => Timing Constraints
  => Constraint on development process
  => Standards and so on…
* Some non-functional requirements may be process rather than product requirements
* Customer imposes these process requirements for two reasons;
  => System Quality
  => System Maintainability

Non – Functional Requirements Types:



(i) Product Requirements:

* These requirements results from the need for the delivered product, to behave in a particular way

Example:

* Requirements on how fast the system must execute and how much memory it requires
* Reliability Requirements [i.e. acceptable failure rate]
* Portability Requirements

(ii) Organizational Requirements:

* These requirements are consequence of organizational policies and procedures

Example:

* Implementation requirements such as programming language (Or) design method used
* Delivery Requirements which specify when the product and its documentation to be delivered

(iii) External Requirements:

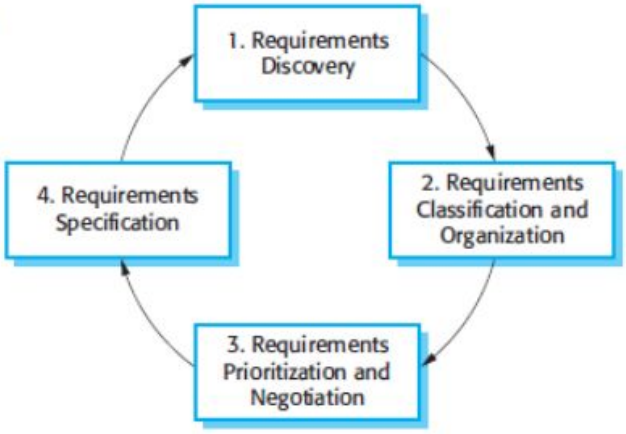
* This requirements arise from factors external to the system and its development process

Example:

* Interoperability Requirements which specify how the system interacts with systems in other organizations
* Legislative Requirements, which ensure that the system operates within the law

Requirements elicitation and analysis

The requirements elicitation and analysis process



A process model of the elicitation and analysis process is shown in Figure

1. ***Requirements discovery*** this is the process of interacting with stakeholders of the system to discover their requirements.

2. ***Requirements classification and organization*** This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters.

3. ***Requirements prioritization and negotiation*** Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation.

4. ***Requirements specification*** The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

Requirements discovery:

Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.

Interviewing:

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions.

Interviews may be of two types:

1. **Closed interviews**, where the stakeholder answers a pre-defined set of questions.
2. **Open interviews,** in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with

Requirements engineering processes

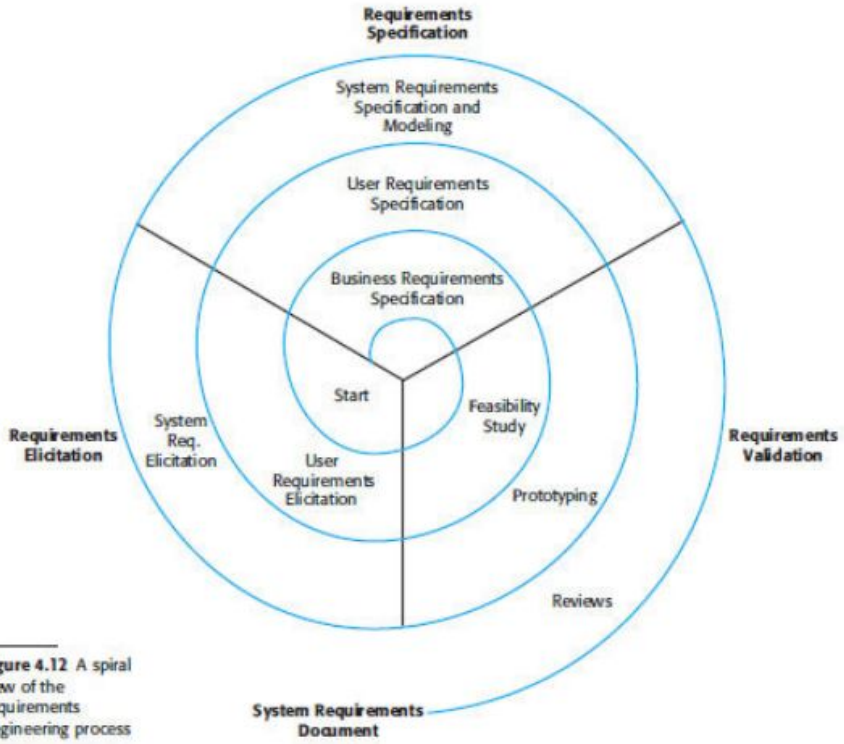


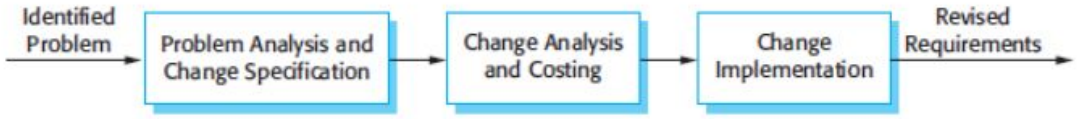
Figure 4.12 A spiral view of the requirements engineering process

Requirements management

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing

Requirements management planning

Figure
Requirements change management



1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.

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. The traceability policy should also define how these records should be maintained.

4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements validation:

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

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. ***Validity checks*** A user may think that a system is needed to perform certain functions.

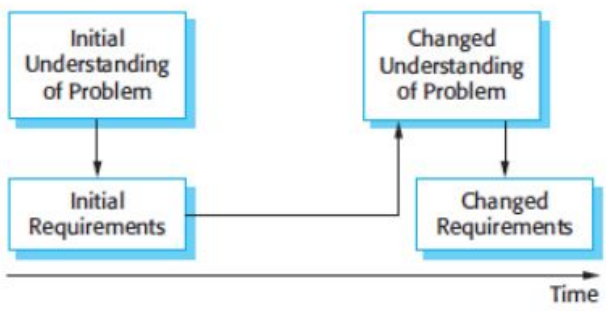
2. ***Consistency checks*** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. ***Completeness checks*** The requirements document should include requirements that define all functions and the constraints intended by the system user.

4. ***Realism checks*** Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented.

5. ***Verifiability*** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

Figure
Requirements evolution



There are a number of requirements validation techniques that can be used individually or in conjunction with one another:

1. ***Requirements reviews*** The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.

2. ***Prototyping*** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

3. ***Test-case generation*** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Requirements Specification Document:

Structure of SRS:

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 non-functional 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 non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, 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.

Software Requirements Specification

Version 1.0

Web Publishing System

Submitted in partial fulfillment
Of the requirements

<<Any comments inside double brackets such as these are *not* part of this SRS but are comments upon this SRS example to help the reader understand the point being made.

Refer to the SRS Template for details on the purpose and rules for each section of this document.

This work is based upon the submissions of the Spring 2004 CS 310. The students who submitted these team projects were Thomas Clay, Dustin Denney, Erjon Dervishaj, Tiffanie Dew, Blake Guice, Jonathan Medders, Marla Medders, Tammie Odom, Amro Shorbatli, Joseph Smith, Jay Snellen, Chase Tinney, and Stefanie Watts. >>

Table of Contents

List of Figures

1.0. Introduction

1.1. Purpose

The purpose of this document is to present a detailed description of the Web Publishing System. It will explain the purpose and features of the system, the interfaces of the system, what the system will do, the constraints under which it must operate and how the system will react to external stimuli. This document is intended for both the stakeholders and the developers of the system and will be proposed to the Regional Historical Society for its approval.

1.2. Scope of Project

This software system will be a Web Publishing System for a local editor of a regional historical society. This system will be designed to maximize the editor's productivity by providing tools to assist in automating the article review and publishing process, which would otherwise have to be performed manually. By maximizing the editor's work efficiency and production the system will meet the editor's needs while remaining easy to understand and use.

More specifically, this system is designed to allow an editor to manage and communicate with a group of reviewers and authors to publish articles to a public website. The software will facilitate communication between authors, reviewers, and the editor via E-Mail. Preformatted reply forms are used in every stage of the articles' progress through the system to provide a uniform review process; the location of these forms is configurable via the application's maintenance options. The system also contains a relational database containing a list of Authors, Reviewers, and Articles.

1.3. Glossary

Term	Definition
Active Article	The document that is tracked by the system; it is a narrative that is planned to be posted to the public website.
Author	Person submitting an article to be reviewed. In case of multiple authors, this term refers to the *principal author*, with whom all communication is made.
Database	Collection of all the information monitored by this system.
Editor	Person who receives articles, sends articles for review, and makes final judgments for publications.
Field	A cell within a form.
Historical Society Database	The existing membership database (also HS database).
Member	A member of the Historical Society listed in the HS database.
Reader	Anyone visiting the site to read articles.
Review	A written recommendation about the appropriateness of an article for publication; may include suggestions for improvement.
Reviewer	A person that examines an article and has the ability to recommend approval of the article for publication or to request that changes be made in the article.
Software Requirements Specification	A document that completely describes all of the functions of a proposed system and the constraints under which it must operate. For example, this document.
Stakeholder	Any person with an interest in the project who is not a developer.
User	Reviewer or Author.

1.4. References

IEEE. *IEEE Std 830-1998 IEEE Recommended Practice for Software Requirements Specifications.* IEEE Computer Society, 1998.

1.5. Overview of Document

The next chapter, the Overall Description section, of this document gives an overview of the functionality of the product. It describes the informal requirements and is used to establish a context for the technical requirements specification in the next chapter.

The third chapter, Requirements Specification section, of this document is written primarily for the developers and describes in technical terms the details of the functionality of the product.

Both sections of the document describe the same software product in its entirety, but are intended for different audiences and thus use different language.

2.0. Overall Description

2.1 System Environment

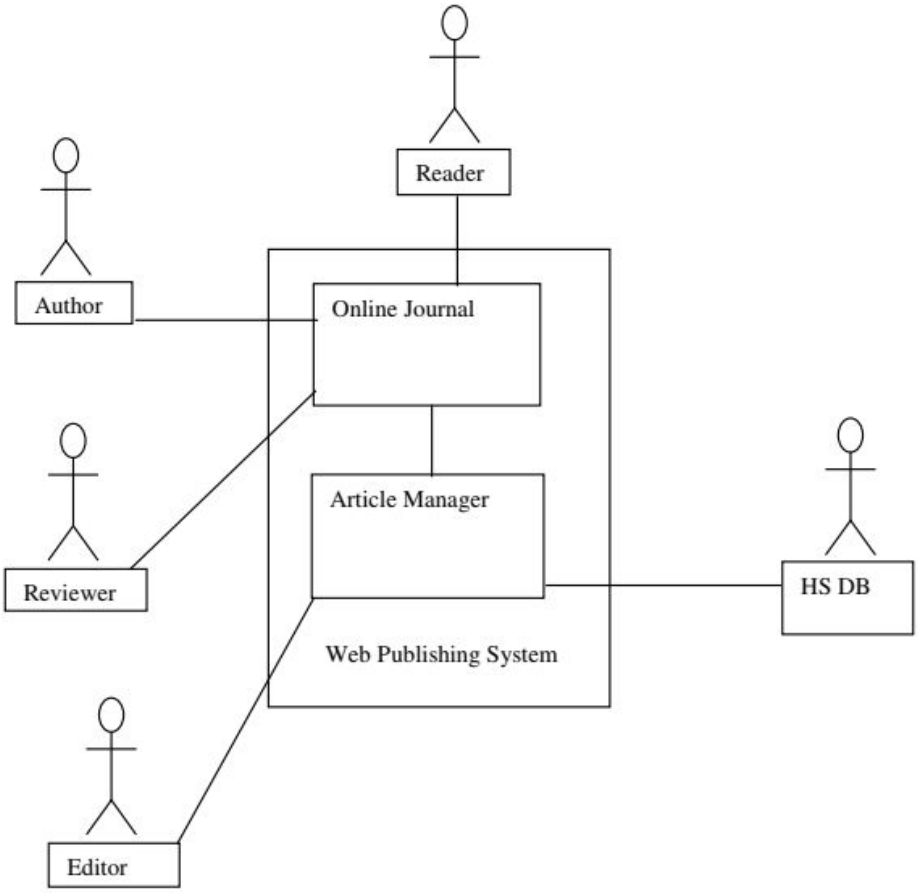


Figure 1 - System Environment

The Web Publishing System has four active actors and one cooperating system. The Author, Reader, or Reviewer accesses the Online Journal through the Internet. Any Author or Reviewer communication with the system is through email. The Editor accesses the entire system directly. There is a link to the (existing) Historical Society.

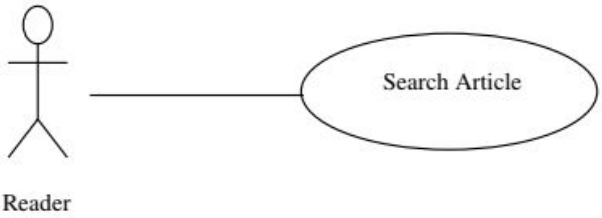
<< The division of the Web Publishing System into two component parts, the Online Journal and the Article Manager, is an example of using domain classes to make an explanation clearer. >>

2.2 Functional Requirements Specification

This section outlines the use cases for each of the active readers separately. The reader, the author and the reviewer have only one use case apiece while the editor is main actor in this system.

2.2.1 Reader Use Case

Use case: **Search Article**
Diagram:



Brief Description
The Reader accesses the Online Journal Website, searches for an article and downloads it to his/her machine.

Initial Step-By-Step Description
Before this use case can be initiated, the Reader has already accessed the Online Journal Website.

1. The Reader chooses to search by author name, category, or keyword.
2. The system displays the choices to the Reader.
3. The Reader selects the article desired.
4. The system presents the abstract of the article to the reader.
5. The Reader chooses to download the article.
6. The system provides the requested article.

Xref: Section 3.2.1, Search Article

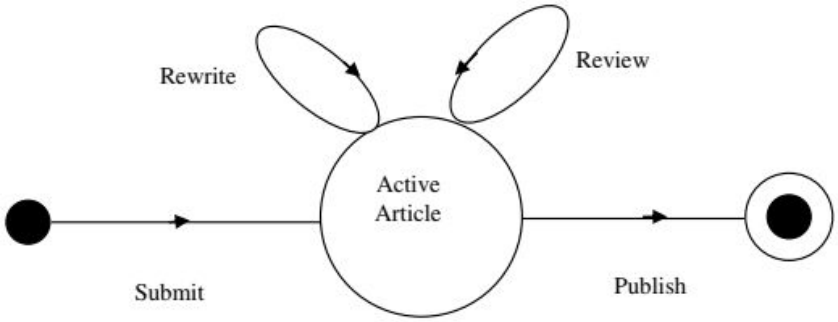


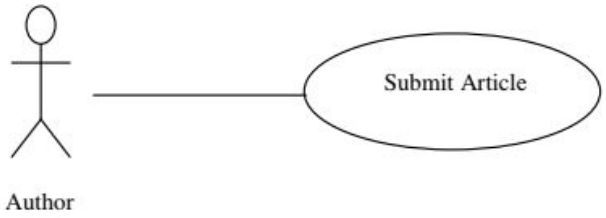
Figure 2 - Article Submission Process

The *Article Submission Process* state-transition diagram summarizes the use cases listed below. An Author submits an article for consideration. The Editor enters it into the system and assigns it to and sends it to at least three reviewers. The Reviewers return their comments, which are used by the Editor to make a decision on the article. Either the article is accepted as written, declined, or the Author is asked to make some changes based on the reviews. If it is accepted, possibly after a revision , the Editor sends a copyright form to the Author. When that form is returned, the article is published to the Online Journal. Not shown in the above is the removal of a declined article from the system.

2.2.2 Author Use Case

In case of multiple authors, this term refers to the *principal author*, with whom all communication is made.

Use case: Submit Article
Diagram:



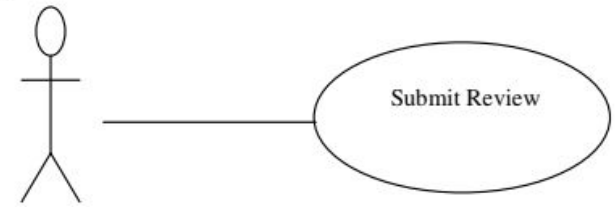
Brief Description
The author either submits an original article or resubmits an edited article.

Initial Step-By-Step Description
Before this use case can be initiated, the Author has already connected to the Online Journal Website.

1. The Author chooses the *Email Editor* button.
2. The System uses the *sendto* HTML tag to bring up the user's email system.
3. The Author fills in the Subject line and attaches the files as directed and emails them.
4. The System generates and sends an email acknowledgement.

Xref: Section 3.2.2, Communicate

2.2.3 Reviewer Use Case
Use case: Submit Review
Diagram:



Brief Description
The reviewer submits a review of an article.

Initial Step-By-Step Description
Before this use case can be initiated, the Reviewer has already connected to the Online Journal Website.

1. The Reviewer chooses the *Email Editor* button.
2. The System uses the *sendto* HTML tag to bring up the user's email system.
3. The Reviewer fills in the Subject line and attaches the file as directed and emails it.
4. The System generates and sends an email acknowledgement.

Xref: Section 3.2.2, Communicate

2.2.4 Editor Use Cases

The Editor has the following sets of use cases:

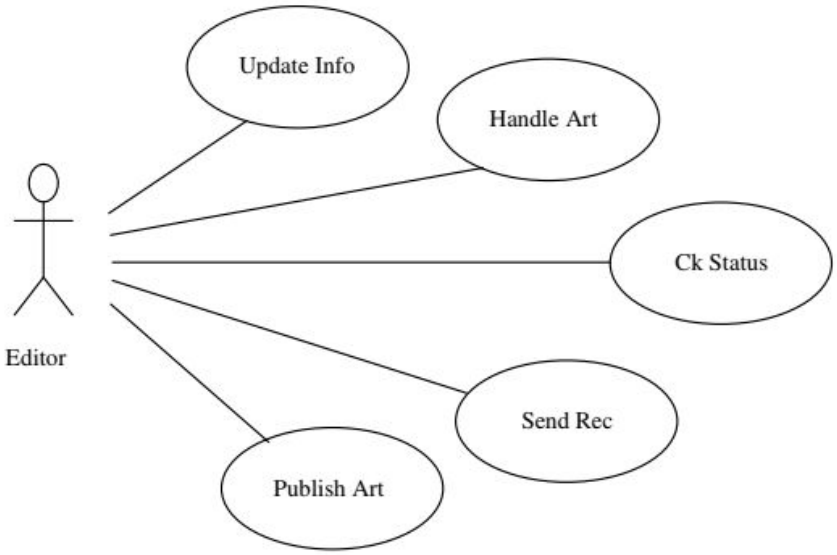
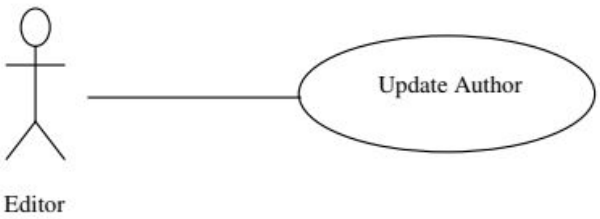


Figure 3 - Editor Use Cases

Update Information use cases

Use case: Update Author
Diagram:



Brief Description
The Editor enters a new Author or updates information about a current Author.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager.

1. The Editor selects to *Add/Update Author*.
2. The system presents a choice of adding or updating.
3. The Editor chooses to add or to update.

4. If the Editor is updating an Author, the system presents a list of authors to choose from and presents a grid filling in with the information; else the system presents a blank grid.
5. The Editor fills in the information and submits the form.
6. The system verifies the information and returns the Editor to the Article Manager main page.

Xref: Section 3.2.3, Add Author; Section 3.2.5 Update Person

Use case: Update Reviewer
Diagram:

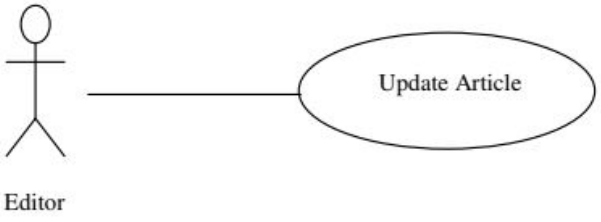
Brief Description
The Editor enters a new Reviewer or updates information about a current Reviewer.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager.

1. The Editor selects to *Add/Update Reviewer*.
2. The system presents a choice of adding or updating.
3. The Editor chooses to add or to update.
4. The system links to the Historical Society Database.
5. If the Editor is updating a Reviewer, the system and presents a grid with the information about the Reviewer; else the system presents list of members for the editor to select a Reviewer and presents a grid for the person selected.
6. The Editor fills in the information and submits the form.
7. The system verifies the information and returns the Editor to the Article Manager main page.

Xref: Section 3.2.4, Add Reviewer; Section 3.2.5, Update Person

Use case: Update Article
Diagram:



Brief Description
The Editor enters information about an existing article.

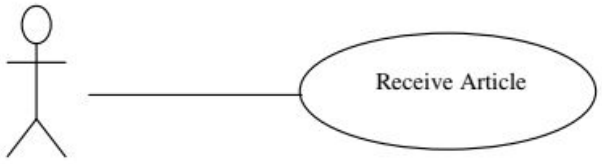
Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager.

1. The Editor selects to *Update Article*.
2. The system presents s list of active articles.
3. The system presents the information about the chosen article.
4. The Editor updates and submits the form.
5. The system verifies the information and returns the Editor to the Article Manager main page.

Xref: Section 3.2.6, Update Article Status

Handle Article use cases

Use case: Receive Article
Diagram:



Editor

Brief Description
The Editor enters a new or revised article into the system.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager and has a file containing the article available.

1. The Editor selects to *Receive Article*.
2. The system presents a choice of entering a new article or updating an existing article.
3. The Editor chooses to add or to update.
4. If the Editor is updating an article, the system presents a list of articles to choose from and presents a grid for filling with the information; else the system presents a blank grid.
5. The Editor fills in the information and submits the form.
6. The system verifies the information and returns the Editor to the Article Manager main page.

Xref: Section 3.2.7, Enter Communication

Use case: Assign Reviewer
This use case extends the *Update Article* use case.
Diagram:

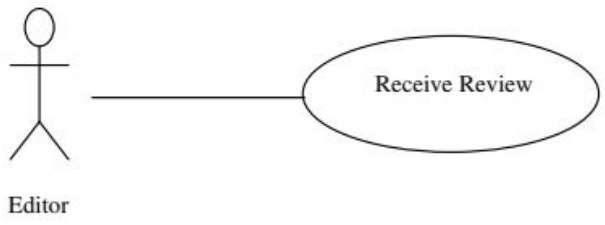
Brief Description
The Editor assigns one or more reviewers to an article.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the article using the *Update Article* use case.

1. The Editor selects to *Assign Reviewer*.
2. The system presents a list of Reviewers with their status (see data description is section 3.3 below).
3. The Editor selects a Reviewer.
4. The system verifies that the person is still an active member using the Historical Society Database.
5. The Editor repeats steps 3 and 4 until sufficient reviewers are assigned.
6. The system emails the Reviewers, attaching the article and requesting that they do the review.
7. The system returns the Editor to the *Update Article* use case.

Xref: Section 3.2.8, Assign Reviewer

Use case: Receive Review
This use case extends the *Update Article* use case.
Diagram:



Brief Description
The Editor enters a review into the system.

Initial Step-By-Step Description

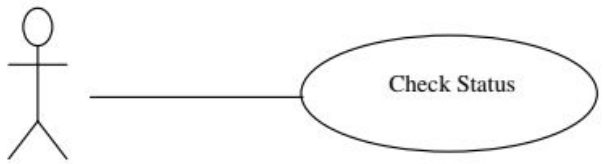
Before this use case can be initiated, the Editor has already accessed the article using the *Update Article* use case.

1. The Editor selects to *Receive Review*.
2. The system presents a grid for filling with the information.
3. The Editor fills in the information and submits the form.
4. The system verifies the information and returns the Editor to the Article Manager main page.

Xref: Section 3.2.7, Enter Communication

Check Status use case:

Use case: Check Status
Diagram:



Brief Description
The Editor checks the status of all active articles.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the main page of the Article Manager.

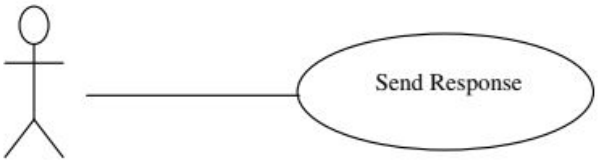
1. The Editor selects to *Check Status*.
2. The system returns a scrollable list of all active articles with their status (see data description in section 3.3 below).
3. The system returns the Editor to the Article Manager main page.

Xref: Section 3.2.9, Check Status

Send Recommendation use cases:

Use case: Send Response
This use case extends the *Update Article* use case.

Diagram:



Brief Description
The Editor sends a response to an Author.

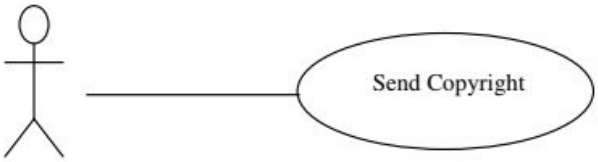
Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the article using the *Update Article* use case.

1. The Editor selects to *Send Response*.
2. The system calls the email system and puts the Author's email address in the Recipient line and the name of the article on the subject line.
3. The Editor fills out the email text and sends the message.
4. The system returns the Editor to the Article Manager main page.

Xref: Section 3.210, Send Communication

Use case: Send Copyright
This use case extends the *Update Article* use case.

Diagram:



Brief Description
The Editor sends a copyright form to an Author.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the article using the *Update Article* use case.

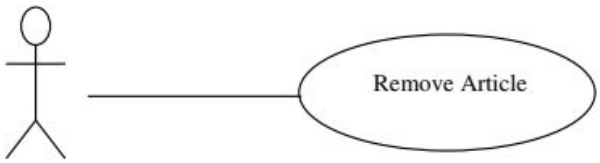
1. The Editor selects to *Send Copyright*.
2. The system calls the email system and puts the Author's email address in the Recipient line, the name of the article on the subject line, and attaches the copyright form.
3. The Editor fills out the email text and sends the message.

4. The system returns the Editor to the Article Manager main page.

Xref: Section 3.2.10, Send Communication

Use case: Remove Article
This use case extends the *Update Article* use case.

Diagram:



Brief Description
The Editor removes an article from the active category.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the article using the *Update Article* use case.

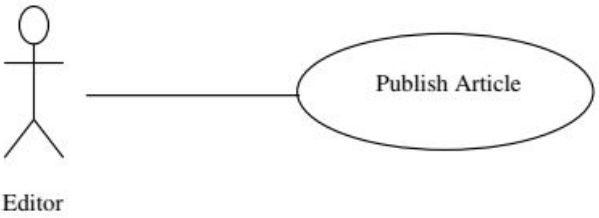
1. The Editor selects to remove an article from the active database.
2. The system provides a list of articles with the status of each.
3. The Editor selects an article for removal.
4. The system removes the article from the active article database and returns the Editor to the Article Manager main page.

Xref: Section 3.2.12, Remove Article

Publish Article use case:

Use case: Publish Article
This use case extends the *Update Article* use case.

Diagram:



Brief Description

The Editor transfers an accepted article to the Online Journal.

Initial Step-By-Step Description
Before this use case can be initiated, the Editor has already accessed the article using the *Update Article* use case.

1. The Editor selects to *Publish Article*.
2. The system transfers the article to the Online Journal and updates the search information there.
3. The system removes the article from the active article database and returns the Editor to the Article Manager home page.

Xref: Section 3.2.11, Publish Article

<< Since three of the actors only have one use case each, the summary diagram only involves the Editor. Adapt the rules to the needs of the document rather than adapt the document to fit the rules. >>

2.3 User Characteristics

The Reader is expected to be Internet literate and be able to use a search engine. The main screen of the Online Journal Website will have the search function and a link to "Author/Reviewer Information."

The Author and Reviewer are expected to be Internet literate and to be able to use email with attachments.

The Editor is expected to be Windows literate and to be able to use button, pull-down menus, and similar tools.

The detailed look of these pages is discussed in section 3.2 below.

2.4 Non-Functional Requirements

The Online Journal will be on a server with high speed Internet capability. The physical machine to be used will be determined by the Historical Society. The software

developed here assumes the use of a tool such as Tomcat for connection between the Web pages and the database. The speed of the Reader's connection will depend on the hardware used rather than characteristics of this system.

The Article Manager will run on the editor's PC and will contain an Access database. Access is already installed on this computer and is a Windows operating system.

3.0. Requirements Specification

3.1 External Interface Requirements

The only link to an external system is the link to the Historical Society (HS) Database to verify the membership of a Reviewer. The Editor believes that a society member is much more likely to be an effective reviewer and has imposed a membership requirement for a Reviewer. The HS Database fields of interest to the Web Publishing Systems are member's name, membership (ID) number, and email address (an optional field for the HS Database).

The *Assign Reviewer* use case sends the Reviewer ID to the HS Database and a Boolean is returned denoting membership status. The *Update Reviewer* use case requests a list of member names, membership numbers and (optional) email addresses when adding a new Reviewer. It returns a Boolean for membership status when updating a Reviewer.

3.2 Functional Requirements

The Logical Structure of the Data is contained in Section 3.3.1.

3.2.1 Search Article

Use Case Name	Search Article
XRef	Section 2.2.1, Search Article SDD, Section 7.1
Trigger	The Reader assesses the Online Journal Website
Precondition	The Web is displayed with grids for searching
Basic Path	1. The Reader chooses how to search the Web site. The choices are by Author, by Category, and by Keyword. 2. If the search is by Author, the system creates and presents an alphabetical list of all authors in the database. In the case of an article with multiple authors, each is contained in the list. 3. The Reader selects an author. 4. The system creates and presents a list of all articles by that author in the database. 5. The Reader selects an article.

	6. The system displays the Abstract for the article. 7. The Reader selects to download the article or to return to the article list or to the previous list.
Alternative Paths	In step 2, if the Reader selects to search by category, the system creates and presents a list of all categories in the database. 3. The Reader selects a category. 4. The system creates and presents a list of all articles in that category in the database. Return to step 5. In step 2, if the Reader selects to search by keyword, the system presents a dialog box to enter the keyword or phrase. 3. The Reader enters a keyword or phrase. 4. The system searches the Abstracts for all articles with that keyword or phrase and creates and presents a list of all such articles in the database. Return to step 5.
Postcondition	The selected article is downloaded to the client machine.
Exception Paths	The Reader may abandon the search at any time.
Other	The categories list is generated from the information provided when article are published and not predefined in the Online Journal database.

3.2.2 Communicate

Use Case Name	Communicate
XRef	Section 2.2.2, Submit Article; Section 2.2.3, Submit Review SDD, Section 7.2
Trigger	The user selects a *mailto* link.
Precondition	The user is on the *Communicate* page linked from the Online Journal Main Page.
Basic Path	This use case uses the *mailto* HTML tag. This invokes the client email facility.
Alternative Paths	If the user prefers to use his or her own email directly, sufficient information will be contained on the Web page to do so.
Postcondition	The message is sent.
Exception Paths	The attempt may be abandoned at any time.
Other	None

3.2.3 Add Author

Use Case Name	Add Author
XRef	Section 2.2.4, Update Author SDD, Section 7.3
Trigger	The Editor selects to add a new author to the database.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The system presents a blank grid to enter the author information. 2. The Editor enters the information and submits the form. 3. The system checks that the name and email address fields are

	not blank and updates the database.
Alternative Paths	If in step 2, either field is blank, the Editor is instructed to add an entry. No validation for correctness is made.
Postcondition	The Author has been added to the database.
Exception Paths	The Editor may abandon the operation at any time.
Other	The author information includes the name mailing address and email address.

3.2.4 Add Reviewer

Use Case Name	Add Reviewer
XRef	Section 2.2.4, Update Reviewer SDD, Section 7.4
Trigger	The Editor selects to add a new reviewer to the database.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The system accesses the Historical Society (HS) database and presents an alphabetical list of the society members. 2. The Editor selects a person. 3. The system transfers the member information from the HS database to the Article Manager (AM) database. If there is no email address in the HS database, the editor is prompted for an entry in that field. 4. The information is entered into the AM database.
Alternative Paths	In step 3, if there is no entry for the email address in the HS database or on this grid, the Editor will be reprompted for an entry. No validation for correctness is made.
Postcondition	The Reviewer has been added to the database.
Exception Paths	The Editor may abandon the operation at any time.
Other	The Reviewer information includes name, membership number, mailing address, categories of interest, and email address.

3.2.5 Update Person

Use Case Name	Update Person
XRef	Sec 2.2.4 Update Author; Sec 2.2.4 Update Reviewer SDD, Section 7.5
Trigger	The Editor selects to update an author or reviewer and the person is already in the database.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The Editor selects Author or Reviewer. 2. The system creates and presents an alphabetical list of people in the category. 3. The Editor selects a person to update. 4. The system presents the database information in grid form for modification. 5. The Editor updates the information and submits the form. 6. The system checks that required fields are not blank.

Alternative Paths	In step 5, if any required field is blank, the Editor is instructed to add an entry. No validation for correctness is made.
Postcondition	The database has been updated.
Exception Paths	If the person is not already in the database, the use case is abandoned. In addition, the Editor may abandon the operation at any time.
Other	This use case is not used when one of the other use cases is more appropriate, such as to add an article or a reviewer for an article.

3.2.6 Update Article Status

Use Case Name	Update Article Status
XRef	Section 2.2.4, Update Article SDD, Section 7.6
Trigger	The Editor selects to update the status of an article in the database.
Precondition	The Editor has accessed the Article Manager main screen and the article is already in the database.
Basic Path	1. The system creates and presents an alphabetical list of all active articles. 2. The Editor selects the article to update. 3. The system presents the information about the article in grid format. 4. The Editor updates the information and resubmits the form.
Alternative Paths	In step 4, the use case *Enter Communication* may be invoked.
Postcondition	The database has been updated.
Exception Paths	If the article is not already in the database, the use case is abandoned. In addition, the Editor may abandon the operation at any time.
Other	This use case can be used to add categories for an article, to correct typographical errors, or to remove a reviewer who has missed a deadline for returning a review. It may also be used to allow access to the named use case to enter an updated article or a review for an article.

3.2.7 Enter Communication

Use Case Name	Enter Communication
XRef	Section 2.2.4, Receive Article; Section 2.2.4, Receive Review SDD, Section 7.7
Trigger	The Editor selects to add a document to the system.
Precondition	The Editor has accessed the Article Manager main screen and has the file of the item to be entered available.
Basic Path	1. The Editor selects the article using the *3.2.6, Update Article Status* use case. 2. The Editor attaches the file to the grid presented and updates the respective information about the article.

	3. When the Editor updates the article status to indicate that a review is returned, the respective entry in the Reviewer table is updated.
Alternative Paths	None
Postcondition	The article entry is updated in the database.
Exception Paths	The Editor may abandon the operation at any time.
Other	This use case extends *3.2.6, Update Article Status*

3.2.8 Assign Reviewer

Use Case Name	Assign Reviewer
XRef	Section 2.2.4, Assign Reviewer SDD, Section 7.8
Trigger	The Editor selects to assign a reviewer to an article.
Precondition	The Editor has accessed the Article Manager main screen and the article is already in the database. .
Basic Path	1. The Editor selects the article using the *3.2.6, Update Article Status* use case. 2. The system presents an alphabetical list of reviewers with their information. 3. The Editor selects a reviewer for the article. 4. The system updates the article database entry and emails the reviewer with the standard message and attaches the text of the article without author information. 5. The Editor has the option of repeating this use case from step 2.
Alternative Paths	None.
Postcondition	At least one reviewer has been added to the article information and the appropriate communication has been sent.
Exception Paths	The Editor may abandon the operation at any time.
Other	This use case extends *3.2.6, Update Article Status*. The Editor, prior to implementation of this use case, will provide the message text.

3.2.9 Check Status

Use Case Name	Check Status
XRef	Section 2.2.4, Check Status SDD, Section 7.9
Trigger	The Editor has selected to check status of all active articles.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The system creates and presents a list of all active articles organized by their status. 2. The Editor may request to see the full information about an article.
Alternative Paths	None.
Postcondition	The requested information has been displayed.

Exception Paths	The Editor may abandon the operation at any time.
Other	The editor may provide an enhanced list of status later. At present, the following categories must be provided: 1. Received but no further action taken 2. Reviewers have been assigned but not all reviews are returned (include dates that reviewers were assigned and order by this criterion). 3. Reviews returned but no further action taken. 4. Recommendations for revision sent to Author but no response as of yet. 5. Author has revised article but no action has been taken. 6. Article has been accepted and copyright form has been sent. 7. Copyright form has been returned but article is not yet published. A published article is automatically removed from the active article list.

3.2.10 Send Communication

Use Case Name	Send Communication
XRef	Section 2.2.4, Send Response; Section 2.2.4, Send Copyright SDD, Section 7.10
Trigger	The editor selects to send a communication to an author.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The system presents an alphabetical list of authors. 2. The Editor selects an author. 3. The system invokes the Editor's email system entering the author's email address into the *To:* entry. 4. The Editor uses the email facility.
Alternative Paths	None.
Postcondition	The communication has been sent.
Exception Paths	The Editor may abandon the operation at any time.
Other	The standard copyright form will be available in the Editor's directory for attaching to the email message, if desired.

3.2.11 Publish Article

Use Case Name	Publish Article
XRef	Section 2.2.4, Publish Article SDD, Section 7.11
Trigger	The Editor selects to transfer an approved article to the Online Journal.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The system creates and presents an alphabetical list of the active articles that are flagged as having their copyright form returned. 2. The Editor selects an article to publish.

	3. The system accesses the Online Database and transfers the article and its accompanying information to the Online Journal database. 4. The article is removed from the active article database.
Alternative Paths	None.
Postcondition	The article is properly transferred.
Exception Paths	The Editor may abandon the operation at any time.
Other	Find out from the Editor to see if the article information should be archived somewhere.

3.2.12 Remove Article

Use Case Name	Remove Article
XRef	Section 2.2.4, Remove Article SDD, Section 7.12
Trigger	The Editor selects to remove an article from the active article database.
Precondition	The Editor has accessed the Article Manager main screen.
Basic Path	1. The system provides an alphabetized list of all active articles. 2. The editor selects an article. 3. The system displays the information about the article and requires that the Editor confirm the deletion. 4. The Editor confirms the deletion.
Alternative Paths	None.
Postcondition	The article is removed from the database.
Exception Paths	The Editor may abandon the operation at any time.
Other	Find out from the Editor to see if the article and its information information should be archived somewhere.

3.3 Detailed Non-Functional Requirements

3.3.1 Logical Structure of the Data

The logical structure of the data to be stored in the internal Article Manager database is given below.

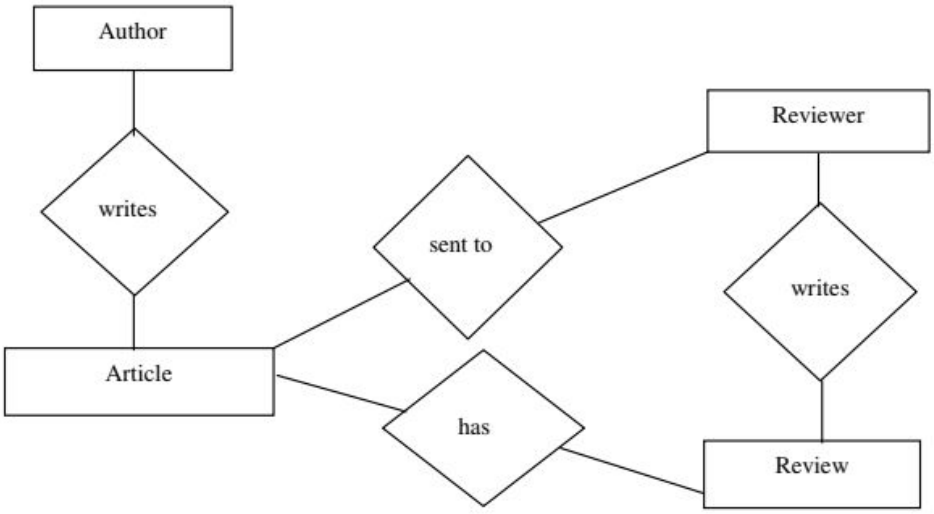


Figure 4 - Logical Structure of the Article Manager Data

The data descriptions of each of these data entities is as follows:

Author Data Entity

Data Item	Type	Description	Comment
Name	Text	Name of principle author	
Email Address	Text	Internet address	
Article	Pointer	Article entity	May be several

Reviewer Data Entity

Data Item	Type	Description	Comment
Name	Text	Name of principle author	
ID	Integer	ID number of Historical Society member	Used as key in Historical Society Database
Email Address	Text	Internet address	
Article	Pointer	Article entity of	May be several
Num Review	Integer	Review entity	Number of not returned reviews
History	Text	Comments on past performance	
Specialty	Category	Area of expertise	May be several

Review Data Entity

Data Item	Type	Description	Comment
Article	Pointer	Article entity	
Reviewer	Pointer	Reviewer entity	Single reviewer
Date Sent	Date	Date sent to reviewer	
Returned	Date	Date returned; null if not	

		returned	
Contents	Text	Text of review	

Article Data Entity

Data Item	Type	Description	Comment
Name	Text	Name of Article	
Author	Pointer	Author entity	Name of principle author
Other Authors	Text	Other authors is any; else null	Not a pointer to an Author entity
Reviewer	Pointer	Reviewer entity	Will be several
Review	Pointer	Review entity	Set up when reviewer is set up
Contents	Text	Body of article	Contains Abstract as first paragraph.
Category	Text	Area of content	May be several
Accepted	Boolean	Article has been accepted for publication	Needs Copyright form returned
Copyright	Boolean	Copyright form has been returned	Not relevant unless Accepted is True.
Published	Boolean	Sent to Online Journal	Not relevant unless Accepted is True. Article is no longer active and does not appear in status checks.

The Logical Structure of the data to be stored in the Online Journal database on the server is as follows:

Published Article Entity

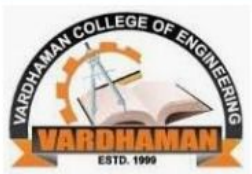
Data Item	Type	Description	Comment
Name	Text	Name of Article	
Author	Text	Name of one Author	May be several
Abstract	Text	Abstract of article	Used for keyword search
Content	Text	Body of article	
Category	Text	Area of content	May be several

3.3.2 Security

The server on which the Online Journal resides will have its own security to prevent unauthorized *write/delete* access. There is no restriction on *read* access. The use of email by an Author or Reviewer is on the client systems and thus is external to the system.

The PC on which the Article Manager resides will have its own security. Only the Editor will have physical access to the machine and the program on it. There is no special protection built into this system other than to provide the editor with *write* access to the Online Journal to publish an article.

Index



Course Code	Course Name	Academic Year	UG	L	T	P	C
A4603	Software Engineering	2020-2021	III B. Tech CSE I Semester	3	0	2	4

Software Engineering
Common to CSE & IT

Lecture-5 [14 & 15 -July-2020]

Agile Process

Faculty
Mr S Venu Gopal
Associate Professor
Department of CSE

Today's Session: Outlines

- What is Agility?
- Agility and The Cost of Change.
- What is an Agile Process? 3 Assumptions
- Extreme Programming (XP)
  - XP (5) Values
  - XP Framework Activities
    - Planning (User Stories, Values, Iteration plan)
    - Design (KIS, CRC Cards, Spike Solutions, Prototypes)
    - Coding (Pair Programming, Refactoring)
    - Test (Unit Test, Integration Test, Acceptance Test)

What is Agility?

- Agility is the **quickness** and **readiness** of movement.
- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Organizing a team so that it is in control of the work performed

Agility in context of software engineering

- The agile process forces the development team to focus on software itself rather than design and documentation.
- The aim of agile process is to deliver the working model of software quickly to the customer
- **For example:** Extreme programming is the best known of agile process.

Agile Principles (12)

- The twelve principles of agile development include:

- **Customer satisfaction through early and continuous software delivery** – Customers are happier when they receive working software at regular intervals, rather than waiting extended periods of time between releases.

- **Accommodate changing requirements throughout the development process** – The ability to avoid delays when a requirement or feature request changes.

- **Frequent delivery of working software** – Scrum accommodates this principle since the team operates in software sprints or iterations that ensure regular delivery of working software.

Agile Principles (12)

- **Collaboration between the business stakeholders and developers throughout the project** – Better decisions are made when the business and technical team are aligned.

- **Support, trust, and motivate the people involved** – Motivated teams are more likely to deliver their best work than unhappy teams.

- **Enable face-to-face interactions** – Communication is more successful when development teams are co-located

Agile Principles (12)

- **Working software is the primary measure of progress** – Delivering functional software to the customer is the ultimate factor that measures progress

- **Agile processes to support a consistent development pace** – Teams establish a repeatable and maintainable speed at which they can deliver working software, and they repeat it with each release.

- **Attention to technical detail and design enhances agility** – The right skills and good design ensures the team can maintain the pace, constantly improve the product, and sustain change.

Agile Principles (12)

- **Simplicity** – Develop just enough to get the job done for right now.

- **Self-organizing teams encourage great architectures, requirements, and designs** – Skilled and motivated team members who have decision-making power, take ownership, communicate regularly with other team members, and share ideas that deliver quality products.

- **Regular reflections on how to become more effective** – Self-improvement, process improvement, advancing skills, and techniques help team members work more efficiently.

Agility and The Cost of Change

Accommodate a change

Functions may be extended

Modify Scenarios

specification can be edited

- **But What if we fast-forward a number of months?**

Team is in the middle of validation testing

Stockholder is requesting a major change

Major modification in the Architectural Design

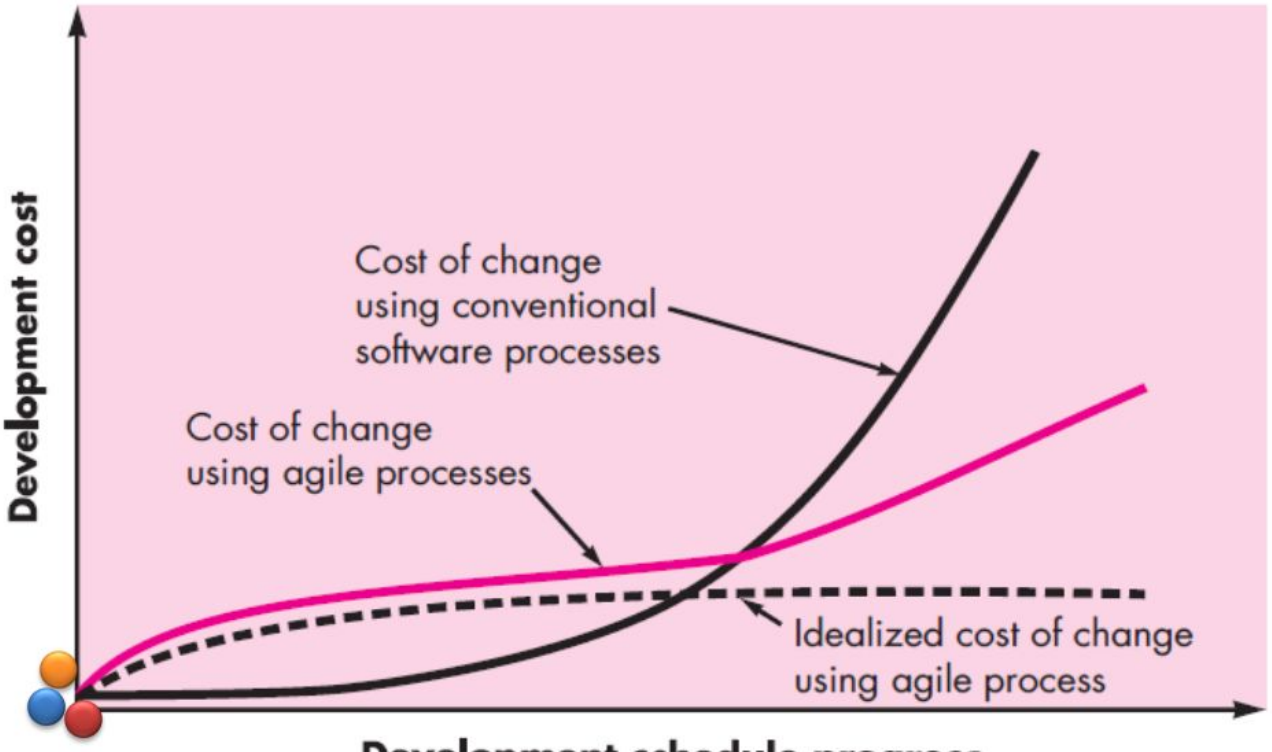
Adding new components to software

Modifying existing 3 components

The design of new test case

Agility and The Cost of Change

- **Diagram:**



What is an Agile Process?

- Agile software process is characterized in a manner that addresses a number of key assumptions. **[3 Major Assumptions]**
- **Agile Processes** Are based on three key assumptions:

1. It is difficult to predict in advance which requirements or customer priorities will change and which will not

2. For many types of software design and construction activities are interleaved (construction is used to prove the design)

3. Analysis, design, and testing are not as predictable from a planning perspective as one might like them to be

Manifesto for Agile Software Development

Agile Development

- Manifesto

1

Individuals and interactions over process and tools.

The best tools will not help if the team doesn't work together. Start small and grow if needed.

2

Working software over comprehensive documentation.

The structure of the system and the rationales for the design should be documented.

Manifesto for Agile Software Development

Agile Development

- Manifesto

3

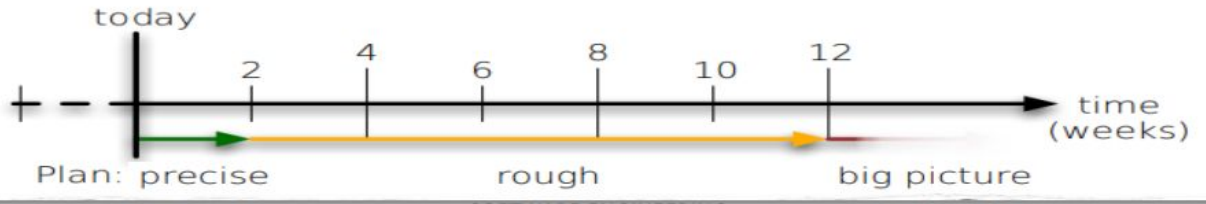
Customer collaboration over contract negotiation.

The contract should specify how the collaboration between the development team and the customer looks like.

A contract which specifies a fixed amount of money that will be paid at a fixed date will likely fail.

4

Responding to change over following a plan.



Extreme Programming (XP)

- Extreme Programming (XP):
- is an agile software development framework that aims to produce higher quality software.
- XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

Extreme Programming (XP)

- **When it is Applicable?**
- The general characteristics where XP is appropriate
  - **Dynamically changing software requirements**
  - **Risks caused by fixed time projects using new technology**
  - **Small, co-located extended development team**
  - **The technology you are using allows for automated unit and functional tests**

Extreme Programming (XP)

- XP Values: Defines a set of five values that establish a foundation for all work performed as part of XP.
  - Communication
  - Simplicity
  - Feedback
  - Courage
  - Respect
- These values is used as a driver for specific XP activities, actions, and tasks.

Extreme Programming (XP)

- **When Applicable :**

➢ Dynamically changing software requirements

➢ Risks caused by fixed time projects using new technology

➢ Small, co-located extended development team

➢ The technology you are using allows for automated unit and functional tests

Extreme Programming (XP) Values

- XP Values: defines a set of five values that establish a foundation for all work performed.
  - Communication
  - Simplicity
  - Feedback
  - Courage
  - Respect

Extreme Programming (XP) (4) Values

- **Communication:**
  - Everyone on a team works jointly at every stage of the project.
  - face to face discussion
  - work together on everything from requirements to code.

- **Simplicity:**
  - keep the design of the system as simple as possible so that it is easier to maintain, support, and revise.
  - address only the requirements that you know about

Extreme Programming (XP)

- **Feedback:**
  - Through constant feedback about their previous efforts, teams can identify areas for improvement and revise their practices
  - Your team builds something, gathers feedback on your design and implementation, and then adjust your product going forward.

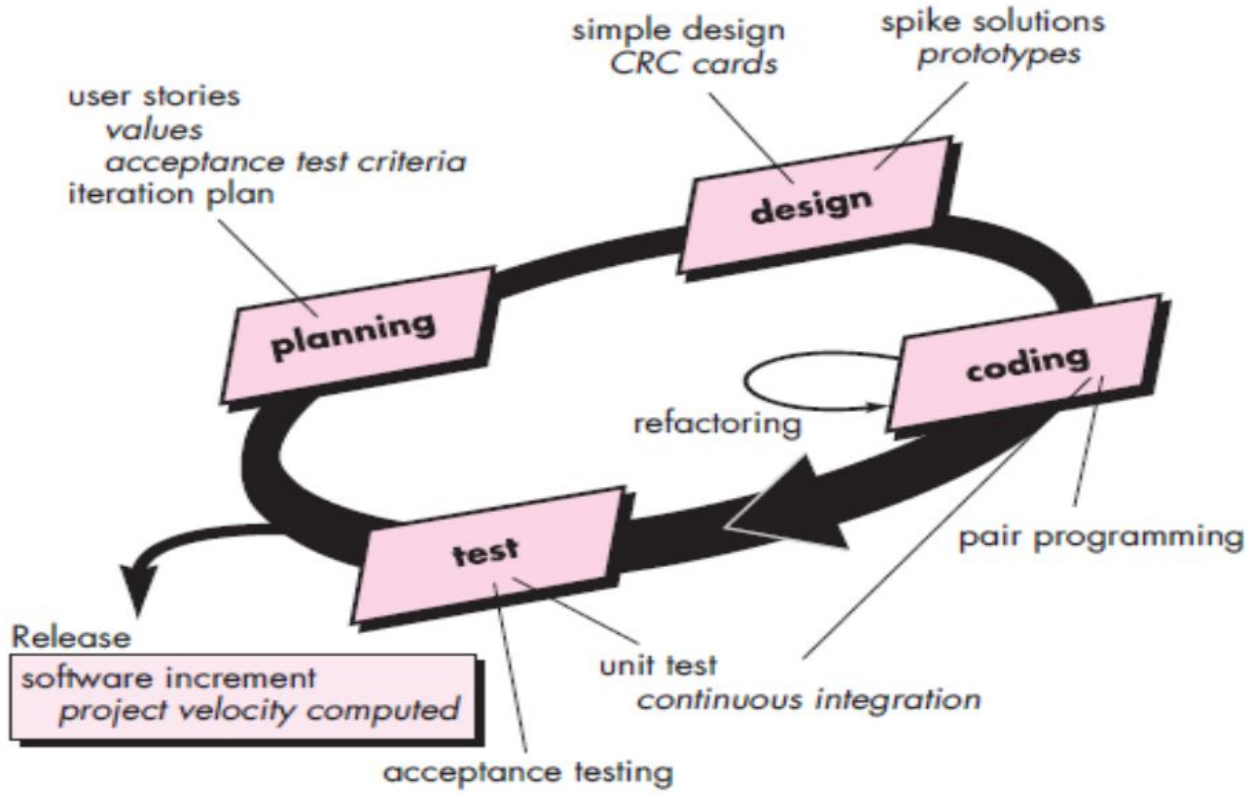
Extreme Programming (XP)

- **Courage:**
  - "effective action in the face of fear"
  - shows a preference for action based on other principles so that the results aren't harmful to the team.
  - will tell the truth about progress and estimates.
  - We will adapt to changes when ever they happen.

- **Respect:**
  - The members of your team need to respect each other in order to communicate with each other,
  - provide and accept feedback that honors your relationship, and to work together to identify simple designs and solutions.

XP Process (Four Framework Activities)

- XP



XP Process (Four Framework Activities)

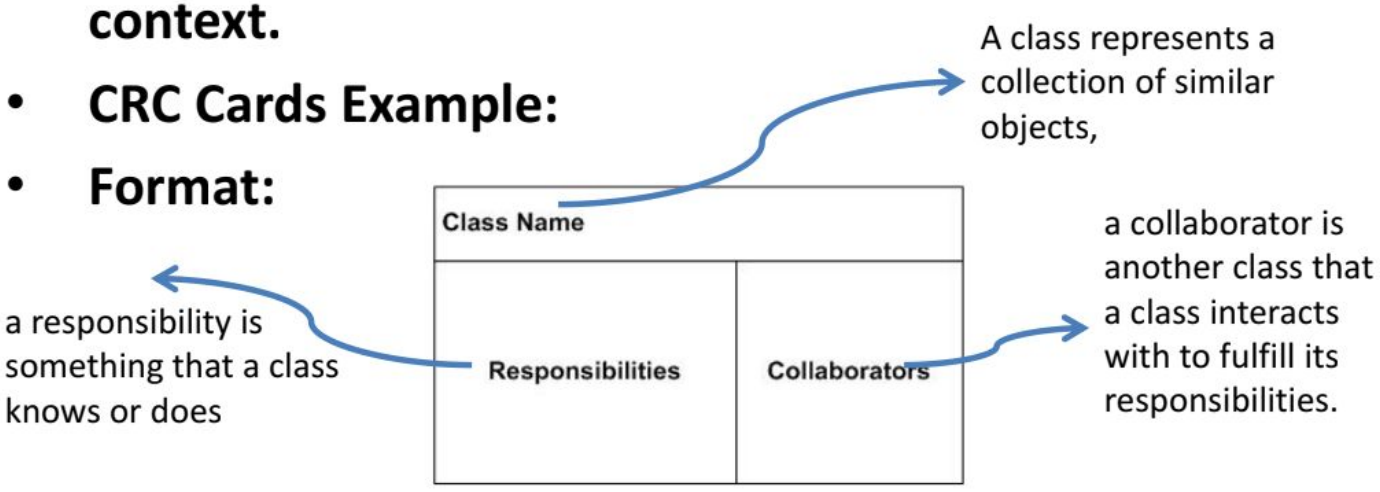
- **Planning:** also called " planning game"
- Begins with listening
- Listening leads to the creation of a set of "Stories" is called ("User Stories") Ex: https://miro.com/ https://www.figma.com/
- Story it similar to Use cases in written by the customer and is placed on an index card.
- The customer assign a Values (i.e. a priority)

User Type	Epic	User Story
Mobile User	Registration	As a user, I can register for the application by entering my email, password, and confirming my password
		As a user, I will receive a confirmation email once I have registered for the application
		As a user, I can register for the application through Facebook
		As a user, I can upload a profile photo and add my name to my account
	Login	As a user, I can log into the application by entering my email and password
		As a user, I can log into the application through Facebook, if I previously registered with it
		As a user, I can reset my password if I have forgotten my password
	My Account	As a user, I can view my personal information
		As a user, I can edit my profile photo
		As a user, I can edit my email. I will receive a confirmation email to my new email address.
		As a user, I can logout of the application from my account
Web User	Registration	As a user, I can register for the application by entering my email, password, and confirming my password
		As a user, I will receive a confirmation email once I have registered for the application
		As a user, I can register for the application through Facebook
		As a user, I can upload a profile photo and add my name to my account

24

XP Process (Four Framework Activities)

- **Design:**
- **it follows the KIS (Keep it Simple)**
- **Here it encourages the use of CRC Cards (Class Responsibility Collaborators) in Object Oriented context.**
- **CRC Cards Example:**
- **Format:**



XP Process (Four Framework Activities)

- CRC Cards Example: online Tool: https://echeung.me/crcmaker/

Class_ATM	
•Start up when switch is turned on •Shut down when switch is turned off •Start a new session when card is inserted by customer •Provide access to component parts for sessions and transactions	OperatorPanel CashDispenser CustomerConsole Session

XP Process (Four Framework Activities)

- ATM Case Study:
- CRC Cards Example:

http://www.math-cs.gordon.edu/courses/cs211/ATMExample/CRCCards.html#CashDispenser

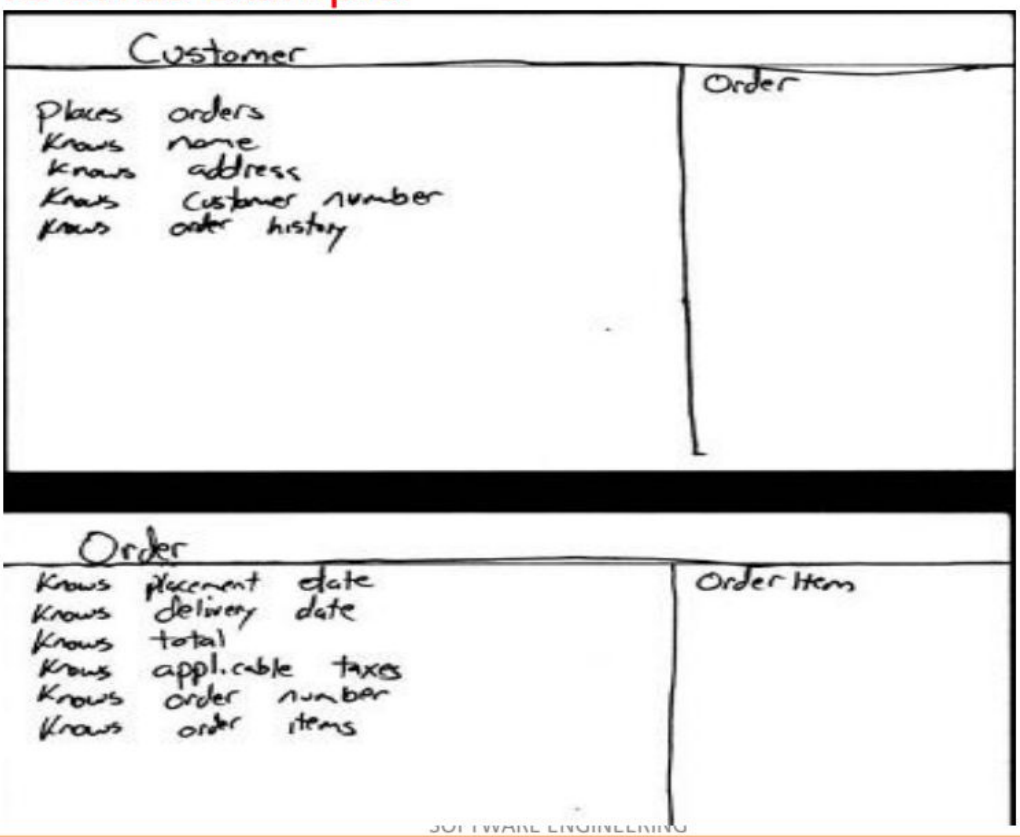
XP Process (Four Framework Activities)

- CRC Cards Example:

Class CardReader	
Responsibilities	**Collaborators**
Tell ATM when card is inserted	ATM
Read information from card	Card
Eject card	
Retain card	

XP Process (Four Framework Activities)

- CRC Cards Example:



XP Process (Four Framework Activities)

- **Spike Solutions:**
- **If any difficult in design , XP recommends the immediate creation of an operational prototype of that portion of the design , is called a "Spike Solution"**

"A spike solution is a very simple program to explore potential solutions."

A spike is a product-testing method

XP Process (Four Framework Activities)

- **Coding:**
  - **Pair programming : two people work together at one computer workstation to create code for a story.**
  - **" Two heads are better than one"**
  - **Refactoring** is the process of clarifying and simplifying the design of existing code, without changing its behavior.

- Release : Release to the Custome

XP Process (Four Framework Activities)

- **Testing:**
- **Unit Test : a Part of testing**
  - part of the product's source code and checks the results.
- **Integration Testing:**
  - in which components (software and hardware) are combined to confirm that they interact according to expectations and requirements.
- **Acceptance Test :** " Customer Test" are specified by the customer after reviewable by the customer.

Pair Programming Tools

1. Motepair by Atom
2. Tuple
3. Teletype for Atom
4. Microsoft Visual Studio Live
5. CodePen
6. Codeanywhere
7. USE Together
8. Remote Collab for Sublime
9. CodeSandbox Live
10. Cloud9
11. Codeshare : https://codeshare.io/
12. Brackets
13. Coda

1. Motepair by Atom
2. Tuple
3. Teletype for Atom
4. Microsoft Visual Studio Live
5. CodePen
6. Codeanywhere
7. USE Together
8. Remote Collab for Sublime
9. CodeSandbox Live
10. Cloud9
11. Codeshare
12. Brackets
13. Coda

Thank You

Session Break 10:40 AM to 11:10 AM

User Stories Template:

What is a user story?

User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system. They typically follow a simple template:

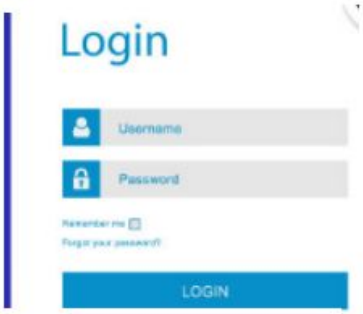
- **As a** < type of user >,
- **I want** < some goal >
- **so that** < some reason >.

An example:

- **As a** bank customer
- **I want** to withdraw money from an ATM
- **So that** I'm not constrained by opening hours or lines at the teller's

Example:

Login Page User Stories:



- **As a** user I want to login.
- As a registered_user I can authenticate against my profile.
- **I want** to send a email after login
- **So that** i can delete my unwanted emails from the Inbox.

UNIT-3

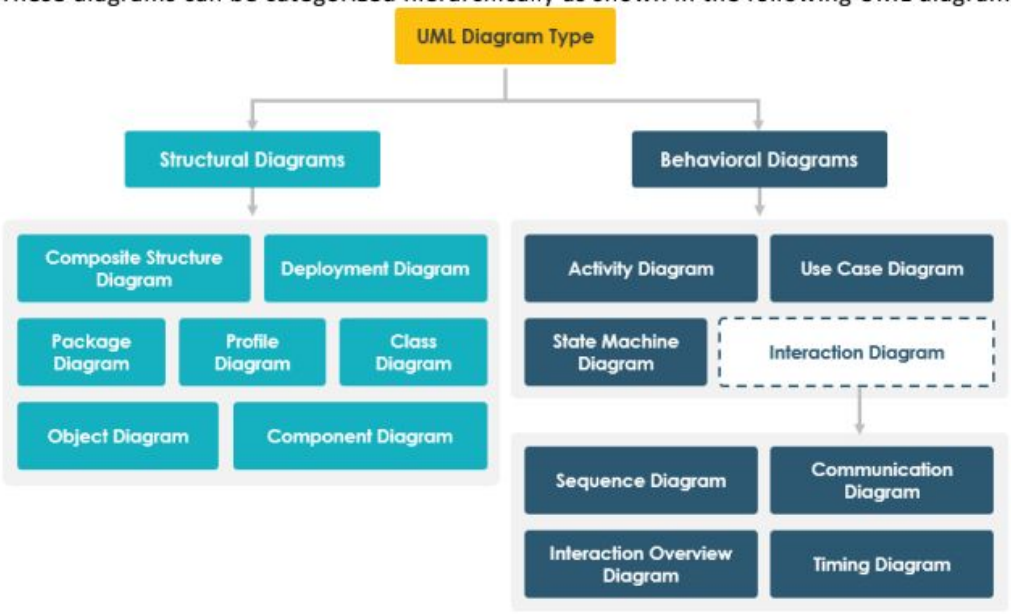
Software Engineering

The Unified Modeling Language is a standardized general-purpose modeling language and nowadays is managed as a de facto industry standard by the **Object Management Group (OMG)**. UML includes a set of graphic notation techniques to create visual models for software-intensive systems.

In UML 2.2 there are 14 types of UML diagrams, which are divided into two categories:

- 7 diagram types represent structural information
- Another 7 represent general UML diagram types for behavioral modeling, including four that represent different aspects of interactions.

These diagrams can be categorized hierarchically as shown in the following UML diagram map:



(You may click on individual UML diagram above to open the corresponding UML diagram guide)

Why the UML is necessary

How the UML came to be

The Unified Modeling Language (UML) is one of the most exciting and useful tools in the world of system development. Why? The UML is a visual modeling language that enables system builders to create blueprints that capture their visions in a standard, easy-to understand way, and provides a mechanism to effectively share and communicate these visions with others.

Communicating the vision is of utmost importance. Before the advent of the UML, system development was often a hit-or-miss proposition. System analysts would try to assess the needs of their clients, generate a requirements analysis in some notation that the analyst understood (but not always the client), give that analysis to a programmer or team of programmers, and hope that the final product was the system the client wanted.

How the UML Came to Be

The UML is the brainchild of Grady Booch, James Rumbaugh, and Ivar Jacobson. Dubbed "the Three Amigos," these gentlemen worked in separate organizations through the 1980s and early 1990s, each devising his own methodology for object-oriented analysis and design. Their methodologies achieved preeminence over those of numerous competitors. By the mid- 1990s, they began to borrow ideas from each other, so they decided to evolve their work together.

In 1994 Rumbaugh joined Rational Software Corporation, where Booch was already working. Jacobson enlisted at Rational a year later. The rest, as they say, is history. Draft versions of the UML began to circulate throughout the software industry, and the resulting feedback brought substantial changes. Because many corporations felt the UML would serve their strategic purposes, a UML consortium sprung up. Members included DEC, Hewlett-Packard, Intellicorp, Microsoft, Oracle, Texas Instruments, Rational, and others. In 1997 the consortium produced version 1.0 of the UML and submitted it to the Object Management Group (OMG) in response to the OMG's request for a proposal for a standard modeling language. The consortium expanded, generated version 1.1, and submitted it to the OMG, who adopted it in late 1997. The OMG took over the maintenance of the UML and produced two more revisions in 1998. The UML has become a de facto standard in the software industry, and it continues to evolve. Versions 1.3, 1.4, and 1.5 have come into being, and OMG recently put its stamp of approval on version 2.0. The earlier versions, referred to generically as version 1.x, have been the basis of most models and most UML modeling books. Throughout this book, I'll show you differences between the old and the new.

What is the Unified Modeling Language?

When discussing UML, we need to establish one important point right up front.

The **Unified Modeling Language** is a **notation;** that is a set of diagrams and diagram elements that may be arranged to describe the design of a software system. UML is not a **process**, nor is it a **method** comprising a notation and a process.

In theory you can apply aspects of the notation according to the steps prescribed by any process that you care to choose - traditional **waterfall, extreme programming, RAD** - but there are processes that have been developed specifically to complement the UML notation. You'll read more about the complementary process(es) later in this chapter.

Why use UML?

Hidden inside that specific question there's a more generic question, which is **"Why use a formal analysis and design notation, UML or otherwise?"** Let's start to answer that question by drawing an analogy.

In a software context, this means that formal design becomes increasingly important as a function of the size and complexity of the project; in particular, as a function of the number of people involved. Based on that analogy, and wider project experience, we could conclude that a formal design notation is important in:

- Establishing a blueprint from the application
- Estimating and planning the time and materials
- Communicating between teams, and within a team
- Documenting the project

Of course, we've probably all encountered projects in which little or no formal design has been done up-front (corresponding with the first three bullet points in that list); in fact more projects than we care to mention! Even in those situations, UML notation has been found to be invaluable in documenting the end result (the last bullet point in that list).

Now that we've answered the generic question, let's return to the specific question of **why use UML?**

Well it's become something of an **industry standard,** which means that there's a good chance of finding other people who understand it. That's very important in terms of the **communication** and **documentation** bullet points in our list. Also if you or anyone else in the team does not understand it, there's a good chance of finding relevant training courses, or books like this one.

That's very pragmatic reasoning and perhaps more convincing than a more academic (or even commercial) argument such as:

*"**The application of UML has a proven track record in improving the quality of software systems**."*

What is UML?

The Unified Modeling Language is the industry standard language for specifying, visualising, constructing and documenting the artifacts of software systems.

Who created UML? When? Why?

Leading O-O design experts Grady Booch, James Rumbaugh, Ivar Jacobson each had their own competing design languages in the early 1990s. They decided to come together and design a unified language.

Is UML a standard?

UML 1.0 was adopted by the Object Management Group (www.omg.org) as a standard in 1997. The OMG is a consortium of (major) software vendors who aim to "create a component-based software marketplace by hastening the introduction of standardized object software" and "... to provide a common framework for application development".

Modeling in s/w development

The designing of software applications before coding. A model gives assurance that functionality is complete and correct end-user needs are met scalability, robustness, security, extendibility equirements are met before implementation.

Principles of Modeling

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

2. Every model may be expressed at different levels of precision.

3. The best models are connected to reality.

4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

1. Why We Model

· The importance of modeling

· Four principles of modeling

· The essential blueprints of a software system

· Object-oriented modeling

What, then, is a model? Simply put,

A model is a simplification of reality.

A model provides the blueprints of a system. Models may encompass detailed plans, as well as more general plans that give a 30,000-foot view of the system under consideration. A good model includes those elements that have broad effect and those minor elements that are not relevant to the given level of abstraction. Every system may be described from different aspects using different models. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

Why do we model? There is one fundamental reason.
We build models so that we can better understand the system we are developing.
Through modeling, we achieve four aims.
How the UML addresses these four things
1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling. However, it's definitely true that the larger and more complex the system, the more important modeling becomes, for one very simple reason:

Object-Oriented Modeling

For example, consider a simple three-tier architecture for a billing system, involving a user interface, middleware, and a database. In the user interface, you will find concrete objects, such as buttons, menus, and dialog boxes. In the database, you will find concrete objects, such as tables representing entities from the problem domain, including customers, products, and orders.

In the middle layer, you will find objects such as transactions and business rules, as well as

higher-level views of problem entities, such as customers, products, and orders.

NOTE: Visualizing, specifying, constructing, and documenting object-oriented systems are exactly the purpose of the Unified Modeling Language.

Introducing the UML

· Overview of the UML

· Three steps to understanding the UML

· Software architecture

· The software development process

The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software intensive system.

The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems. the

UML is not difficult to understand and to use. Learning to apply the UML effectively starts with forming a conceptual model of the language, which requires learning three major elements: the UML's basic building blocks, the rules that dictate how these building blocks may be put together, and some common mechanisms that apply throughout the language.

The UML is only a language and so is just one part of a software development method. The UML is process independent, it should be used in a process that is use case driven, iterative, and incremental.

An Overview of the UML

The UML is a language for

- **Visualizing**
- **Specifying**
- **Constructing**
- **Documenting**

The artifacts of a software-intensive system.

The UML Is a Language :-

A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A ***modeling*** language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modeling language such as the UML is thus a standard language for software blueprints.

The UML Is a Language for Visualizing:-

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms. In such cases, the programmer is still doing some modeling, He or she may even sketch out a few ideas on a white board or on a napkin. However, there are several problems with this. **First**, communicating those conceptual models to others is error-prone unless everyone involved speaks the same language. Typically, projects and organizations develop their own language, and it is difficult to understand what's going on if you are an outsider or new to the group. **Second**, there are some things about a software system you can't understand unless you build models that transcend the textual programming language. **Third**, if the developer who cut the code never wrote down the models that are in his or her head, that information

would be lost forever or, at best, only partially recreatable from the implementation, once that developer moved on.

Writing models in the UML addresses the third issue: An explicit model facilitates communication. Some things are best modeled textually; others are best modeled graphically. Indeed, in all interesting systems, there are structures that transcend what can be represented in a programming language. The UML is such a graphical language. This addresses the second problem described earlier. The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model.

The UML Is a Language for Specifying :-

specifying means building models that are unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

The UML Is a Language for Constructing :-

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database. Things that are best expressed graphically are done so graphically in the UML, whereas things that are best expressed textually

are done so in the programming language.

This mapping permits **forward engineering**: The generation of code from a UML model into a programming language. The **reverse engineering** is also possible: You can reconstruct a model from an implementation back into the UML.

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code.

These artifacts include

- Requirements

- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

Depending on the development culture, some of these artifacts are treated more or less formally than others. Such artifacts are not only the deliverables of a project, they are also critical in controlling, measuring, and communicating about a system during its development and after its deployment. The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests. Finally, the UML provides a language for modeling the activities of project planning and release management.

Where Can the UML Be Used?

The UML is intended primarily for software-intensive systems. It has been used effectively for

such domains like.

- Enterprise information systems
- Banking and financial services
- Telecommunications
- Transportation
- Defense/aerospace
- Medical electronics
- Scientific
- Distributed Web-based services

The UML is not limited to modeling software. In fact, it is expressive enough to model nonsoftware systems, such as workflow in the legal system, the structure and behavior of a patient healthcare system, and the design of hardware.

A Conceptual Model of the UML :-

To understand the UML, you need to form a conceptual model of the language, and this requires learning **three major elements**: the UML's basic building blocks.

Building Blocks of the UML :-

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things
2. Relationships
3. Diagrams

1.Things in the UML

There are four kinds of things in the UML:

1.1. Structural things
1.2. Behavioral things
1.3. Grouping things
1.4. An notational things

These things are the basic object-oriented building blocks of the UML. You use them to write well formed models.

1.1. Structural Things

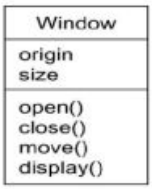
Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

1. Classes	**2. Interface**
3. Collaborations	**4. Use Cases**
5. Active Classes	**6. Components 7. Nodes**

1.a class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations,

as in Figure 1.

Figure 1 Classes



2.an interface is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface, as in Figure 2.

Figure 2 Interfaces

3.a collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name, as in Figure 3.

Figure 3 Collaborations

4. a use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name, as in

Figure 2-4 Use Cases

The remaining **three things 5.active classes, 6.components, and 7.nodes** are all class-like, meaning

they also describe a set of objects that share the same attributes, operations, relationships, and semantics

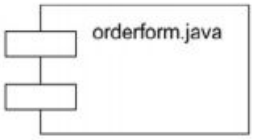
5.active classes is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations, as in Figure 5.

Figure 5 Active Classes

The remaining **two** elements **6.component**, and **7.nodes** are also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

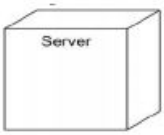
6.component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components, such as COM+ components or Java Beans, such as source code files. A component typically represents the physical packaging of logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name, as in Figure 6.

Figure 6 Components



7.nodes is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name, as in Figure 7.

Figure 7 Nodes



There are also variations on these seven,

NOTE: actors, signals, and utilities (kinds of classes), processes and threads (kinds of active classes), and applications, documents, files, libraries, pages, and tables (kinds of components).

1.2 Behavioral Things

Behavioral things are the dynamic parts of UML models. Representing behavior over time and space. In all, there are two primary kinds of behavioral things.

1. An interaction **2. A state machine**

1. An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to a specific purpose. An interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as in Figure 8.

Figure 8 Messages

2. A state machine is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates, if any, as in Figure 9.

Figure 9 States

1.3.Grouping Things

Grouping things are the organizational parts of UML models. These are the boxes into which a

model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

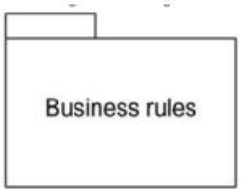
1.packages

1.packages is a general-purpose mechanism for organizing elements into groups. Structural

things, behavioral things, and even other grouping things may be placed in a package. Unlike

components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents, as in Figure 10.Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

Figure 10 Packages

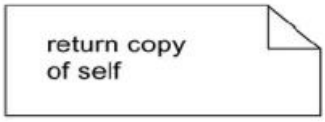


1.4.Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, and remark about any element in a model. There is one primary kind of annotational thing, called a **note**.

A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as in Figure 11.

Figure 11 Notes



Relationships in the UML

There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

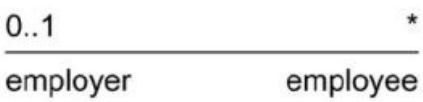
These relationships are the basic relational building blocks of the UML. You use them to write well-formed models.

1. Dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label, as in Figure 12.

Figure 12 Dependencies

2. Association is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names, as in Figure 13.

Figure 13 Associations



3. Generalization is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the

child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent, as in Figure 14.

Figure 14 Generalizations

4. Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship, as in Figure 15.

Figure 15 Realization

NOTE: These four elements are the basic relational things you may include in a UML model. There are also variations on these four, such as refinement, trace, include, and extend (for dependencies).

Diagrams in the UML

A **diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. In theory, a diagram may contain any combination of things and relationships. The UML includes **nine** diagrams:

1. Class diagram

2. Object diagram

3. Use case diagram

4. Sequence diagram

5. Collaboration diagram

6. State chart diagram

7. Activity diagram

8. Component diagram

9. Deployment diagram

1. Class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

2. Object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

3. Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

4. Sequence diagram is an interaction diagram that emphasizes the time-ordering of messages;

An shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system.

5. Collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.

NOTE: Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

6. State chart diagram shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

7. Activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

8. Component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

9. Deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components.

Architecture

The need for viewing complex systems from different perspectives **Visualizing, specifying, constructing, and documenting** a software-intensive system demands that the system be viewed from a number of perspectives. Different stakeholders• end users, analysts, developers, system integrators, testers, technical writers, and project managers• each bring different agendas to a project, and each looks at that system in different ways at different times over the project's life. A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.

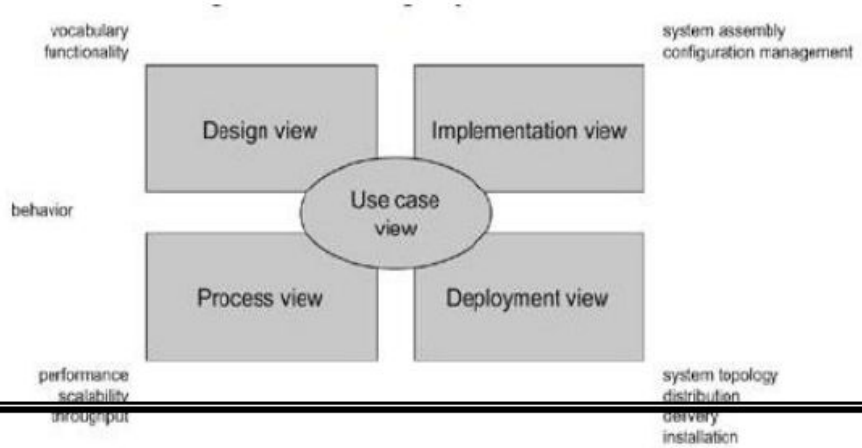
Architecture is the set of significant decisions about

- The organization of a software system.
- The selection of the structural elements and their interfaces by which the system is composed.
- Their behavior, as specified in the collaborations among those elements.
- The composition of these structural and behavioral elements into progressively larger subsystems.
- The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition.

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

As Figure 16 illustrates, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

Figure 16 Modeling a System's Architecture



1. Use case view **2.Design view** **3.Process view**

4. Implementation view **5.Deployment view**

1. The **use case view** of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

2. The **Design view** of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

3. The **Process view** of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

4. The **Implementation view** of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management

of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

5. The **Deployment view** of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture. These five views also interact with one another. nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express every one of these five views and their interactions.

Rules of the UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like.

A *well-formed model* is one that is semantically self-consistent and in harmony with all its related models.

The UML has semantic rules for

Names	What you can call things, relationships, and diagrams
Scope	The context that gives specific meaning to a name
Visibility	How those names can be seen and used by others

Integrity	How things properly and consistently relate to one another
Execution	What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is

common for the development team to not only build models that are well-formed, but also to build models that are

Elided	Certain elements are hidden to simplify the view
Incomplete	Certain elements may be missing
Inconsistent	The integrity of the model is not guaranteed

Common Mechanisms in the UML

A building is made simpler and more harmonious by the conformance to a pattern of common features. The same is true of the UML. It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications

2. Adornments

3. Common divisions

4. Extensibility mechanisms

1. Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might

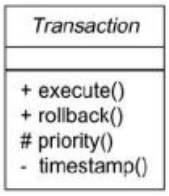
only show a small part of this specification. You use the UML's graphical notation to visualize a system; you use the UML's specification to state the system's details.

2. Adornments

Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.

A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. For example, Figure 16 shows a class, adorned to indicate that it is an abstract class with **two public**, **one protected**, and **one private** operation.

Figure 16 Adornments



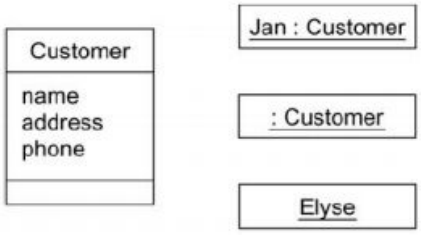
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

3. Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in Figure 17.

Figure 17 Classes And Objects

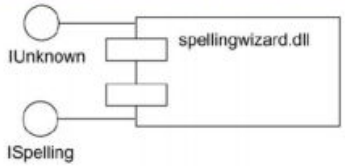


In this figure, there is one class, named **Customer**, together with three objects: **Jan** (which is marked explicitly as being a **Customer** object), **:Customer** (an anonymous **Customer** object), and **Elyse** (which in its specification is marked as being a kind of **Customer** object, although it's not shown explicitly here).

Almost every building block in the UML has this same kind of class/object dichotomy. For example, you can have use cases and use case instances, components and component instances, nodes and node instances, and so on. Graphically, the UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name.

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in Figure 18.

Figure 18 Interfaces and Implementations



In this figure, there is one component named **spellingwizard.dll** that implements two interfaces, **IUnknown** and **ISpelling.** Almost every building block in the UML has this same kind of interface/ implementation dichotomy. For example, you can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.

4. Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possibility of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways. The UML's extensibility mechanisms include

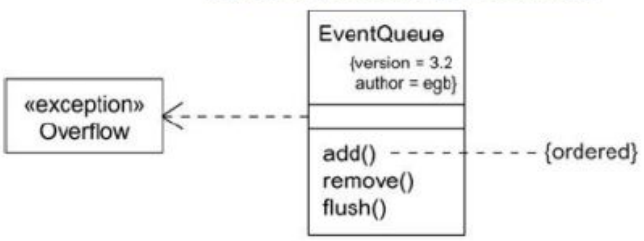
· Stereotypes

· Tagged values

· Constraints

A **Stereotypes** extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first class citizens in your models• meaning that they are treated like basic building blocks• by marking them with an appropriate stereotype, as for the class **Overflow** in Figure 19.

Figure 19 Extensibility Mechanisms



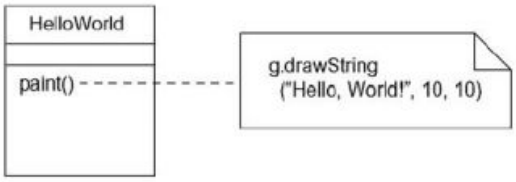
A **Tagged values** extends the properties of a UML building block, allowing you to create new information in that element's specification. For example, adding the version and author to any building block. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. In Figure 19, for example, the class **EventQueue** is extended by marking its version and author explicitly.

A **Constraint** extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the **EventQueue** class so that all additions are done in order. As Figure 19 shows, you can add a constraint that explicitly marks these for the operation **add**.

Collectively, these three extensibility mechanisms allow you to shape and grow the UML to your project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. You can add new building blocks, modify the specification of existing ones, and even change their semantics.

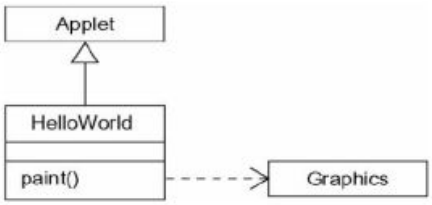
Modeling this application in the UML is straightforward. As Figure :20 shows, you can represent the class HelloWorld graphically as a rectangular icon. Its paint operation is shown here, as well, with all its formal parameters elided and its implementation specified in the attached note.

Figure: 20 Key Abstractions for Hello World



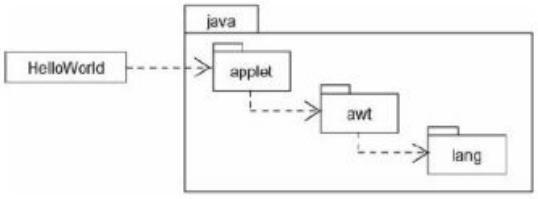
This class diagram captures the basics of the "Hello, World!" application, but it leaves out a number of things. As the preceding code specifies, two other classes• Applet and Graphics• are involved in this application and each is used in a different way. The class Applet is used as the parent of HelloWorld, and the class Graphics is used in the signature and implementation of one of its operations, paint. You can represent these classes and their different relationships to the class HelloWorld in a class diagram, as shown in Figure 21.

Figure 21 Immediate Neighbors Surrounding HelloWorld



As this figure shows, packages are represented in the UML as a tabbed folders. Packages may be nested, and the dashed directed lines represent dependencies among these packages. For example, HelloWorld depends on the package java.applet, and java.applet depends on the package java.awt. You can visualize this packaging in a class diagram, shown in Figure 22.

Figure 22 HelloWorld Packaging



the rules that dictate how those

building blocks may be put together

COMMON MECHANISMS

UML Extensibility Mechanism?

The UML is a general-purpose, tool-supported, and standardized modeling language that is used in order to specify, visualize, construct and document all the elements of a wide range of system intensive processes. It promotes a use case driven, architecture-centric, iterative and incremental process, which is object-oriented and component-based. The UML is broadly applicable to different types of systems, domains, methods and processes, which is why it is such a popular and broadly used language.

However, even though the UML is very well-defined, there might be situations in which you might find yourself wanting to bend or extend the language in some controlled way to tailor it to your specific problem domain in order to simplify the communication of your objective. This is where the UML extension mechanisms come in.

The Three UML Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways.

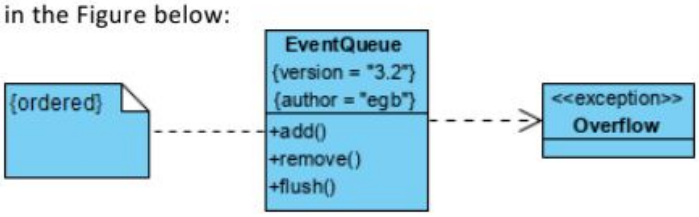
UML defines three extensibility mechanisms to allow modelers to add extensions without having to modify the underlying modeling language. These three mechanisms are:

- Stereotypes
- Constraints
- tagged values

Stereotypes

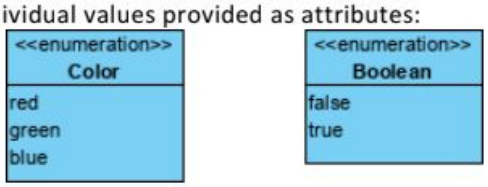
A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. They are used for classifying or marking the UML building blocks in order to introduce new building blocks that speak the language of your domain and that look like primitive, or basic, model elements.

For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first-class citizens in your models, meaning that they are treated like basic building blocks, by marking them with an appropriate stereotype, as for the class Overflow as shown in the Figure below:



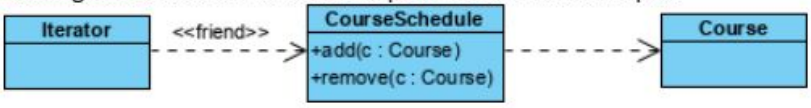
Model New types in UML

Another Example, an enumeration types, such as Coloroolean and Status, can be modeled as enumerations, with their individual values provided as attributes:



Model Special Relationship

A dependency can have a name, although names are rarely needed unless you want to use stereotypes to distinguish different flavors of dependencies. For example:

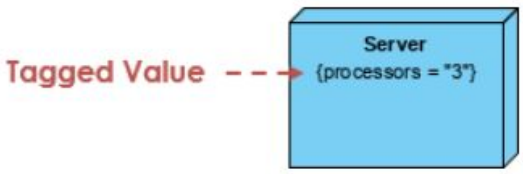


Tagged Values

A tagged value extends the properties of a UML building block, allowing you to create new information in that element's specification. They are properties for specifying keyword-value pairs of model elements, where the keywords are attributes. They allow you to extend the properties of a UML building block so that you create new information in the specification of that element. Tagged Values can be defined for existing model elements, or for individual stereotypes so that

everything with that stereotype has that tagged value. It is important to mention that a tagged value is not equal to a class attribute. Instead, you can regard a tagged value as being metadata, since its value applies to the element itself and not to its instances.

Graphically, a tagged value is rendered as a string enclosed by brackets, which is placed below the name of another model element. The string consists of a name (the tag), a separator (the symbol =), and a value (of the tag) as shown in the Figure below:



You can specify just the value if its meaning is unambiguous, such as when the value is the name of the enumeration.

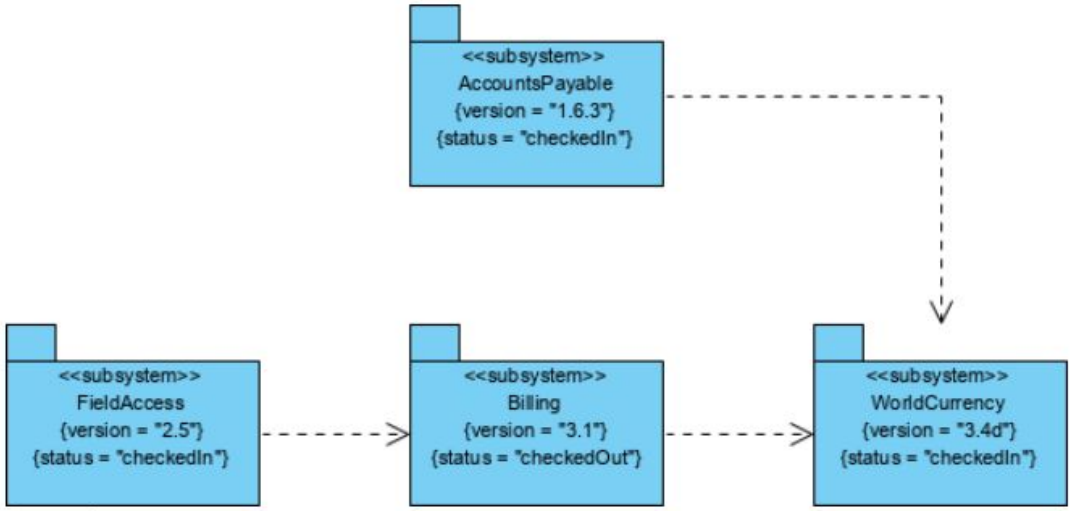
Usage of Tagged Values

One of the most common uses of a tagged value is to specify properties that are relevant to code generation or configuration management. So, for example, you can make use of a tagged value in order to specify the programming language to which you map a particular class, or you can use it to denote the author and the version of a component.

Tagged Value Example – Configuration Management System

Suppose you want to tie the models you create to your project's configuration management system. Among other things, this means keeping track of the version number, current check-in/check out status, and perhaps even the creation and modification dates of each subsystem. Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values. Furthermore, this information is not just a class attribute either. A subsystem's version number is part of its metadata, not part of the model.

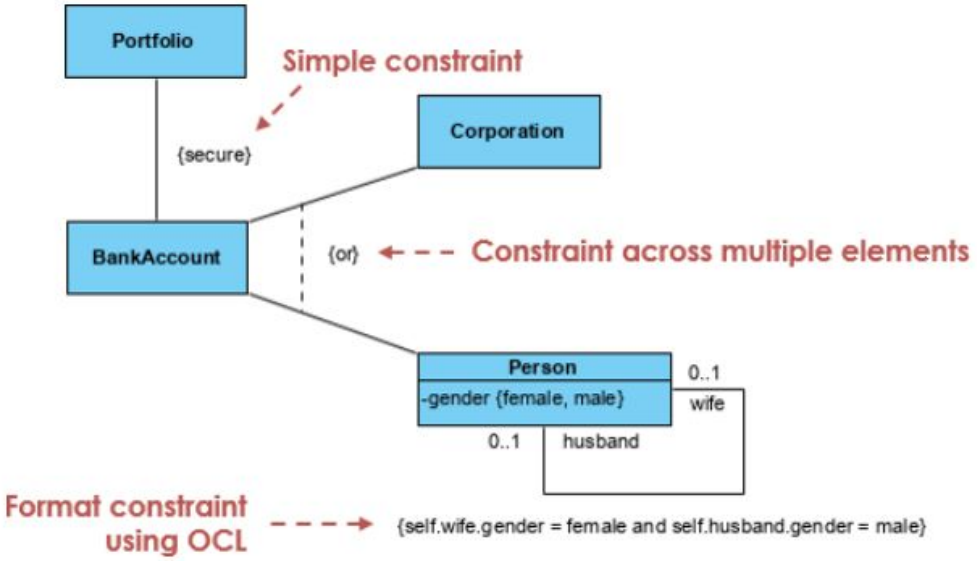
The Figure below shows four subsystems, each of which has been extended to include its version number and status. In the case of the Billing subsystem, one other tagged value is shown – the person who has currently checked out the subsystem.



Constraints

A constraint is an extension of the semantics of a UML element, allowing you to add new rules or to modify existing ones. They allow you to extend the semantics of a UML building block by adding new rules or modifying existing ones. A constraint specifies conditions that must be held true for the model to be well-formed. This notation can also be used to adorn a model element's basic notation, in order to visualize parts of an element's specification that have no graphical cue. For example, you can use constraint notation to provide some properties of associations, such as order and changeability

Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships. The example below shows, you might want to specify that, across a given association, communication is encrypted. Similarly, you might want to specify that among a set of associations, only one is manifest at a time.


Portfolio
Simple constraint
{secure}
Corporation
BankAccount
{or}
Constraint across multiple elements
Person
-gender {female, male}
0..1
wife
0..1
husband
Format constraint
using OCL
{self.wife.gender = female and self.husband.gender = male}

Activity Diagram

- **Objectives**

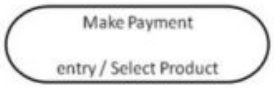
- to describe dynamic aspects of the system.

- Activity diagram is essentially an advanced version of flow chart that modeling the flow from one activity to another activity.

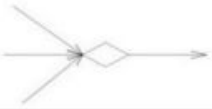
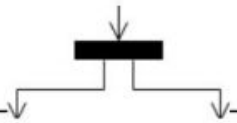
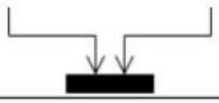
Activity Diagram - Introduction

- An activity diagram is like a flowchart, representing flow of control from activity to activity.
- Activities result in some action.
- Actions involve calling another operation, sending a signal, creating or destroying an object or some pure computation, such as evaluating an expression.

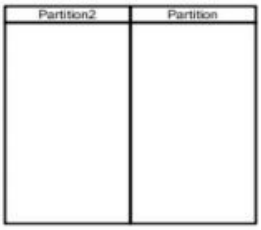
Activity Diagram - Notations

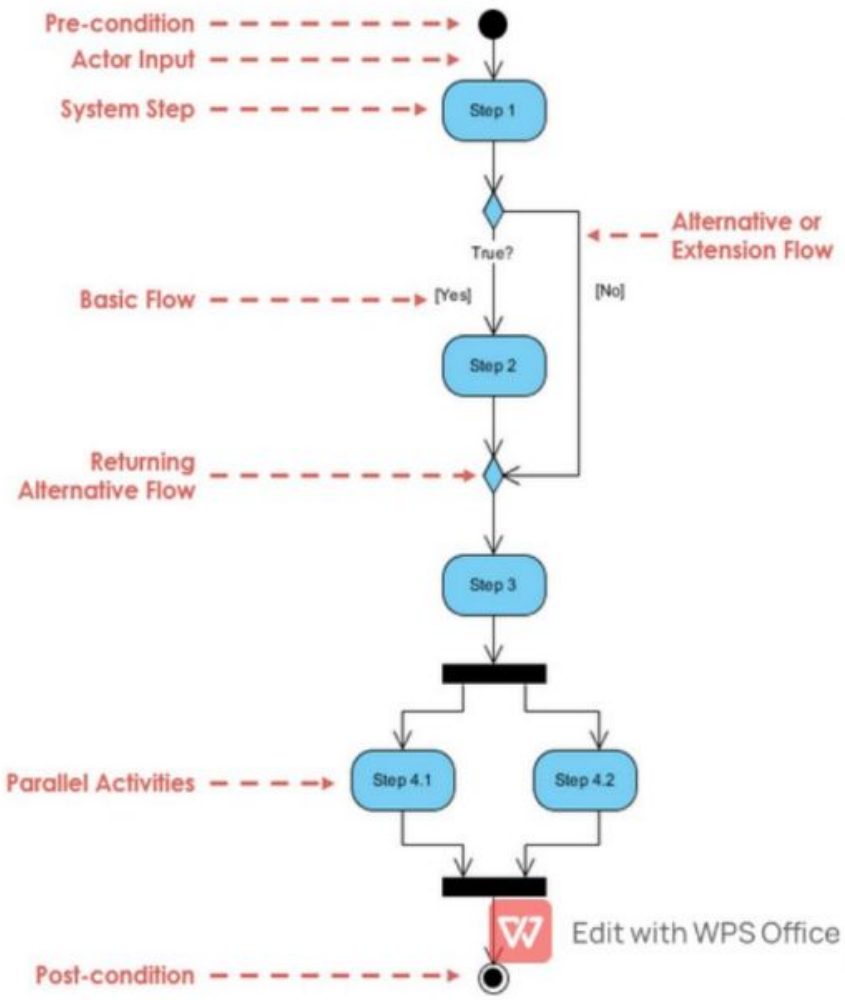
Activity Is used to represent a set of actions	
Action A task to be performed	
Control Flow Action flows, also called edges and paths, illustrate the transitions from one action state to another. They are usually drawn with an arrowed line.	
Object Flow Show the flow of an object from one activity (or action) to another activity (or action).	
Initial State or Starting State Portrays the beginning of a set of actions or activities	
End State or Final State Stop all control flows and object flows in an activity (or action)	

Activity Diagram - Notations

Object Node Represent an object that is connected to a set of Object Flows	
Decision Node Represent a test condition to ensure that the control flow or object flow only goes down one path	
Merge Node Bring back together different decision paths that were created using a decision-node.	
Fork Node Split behavior into a set of parallel or concurrent flows of activities (or actions)	
Join Node Bring back together a set of parallel or concurrent flows of activities (or actions).	

Activity Diagram

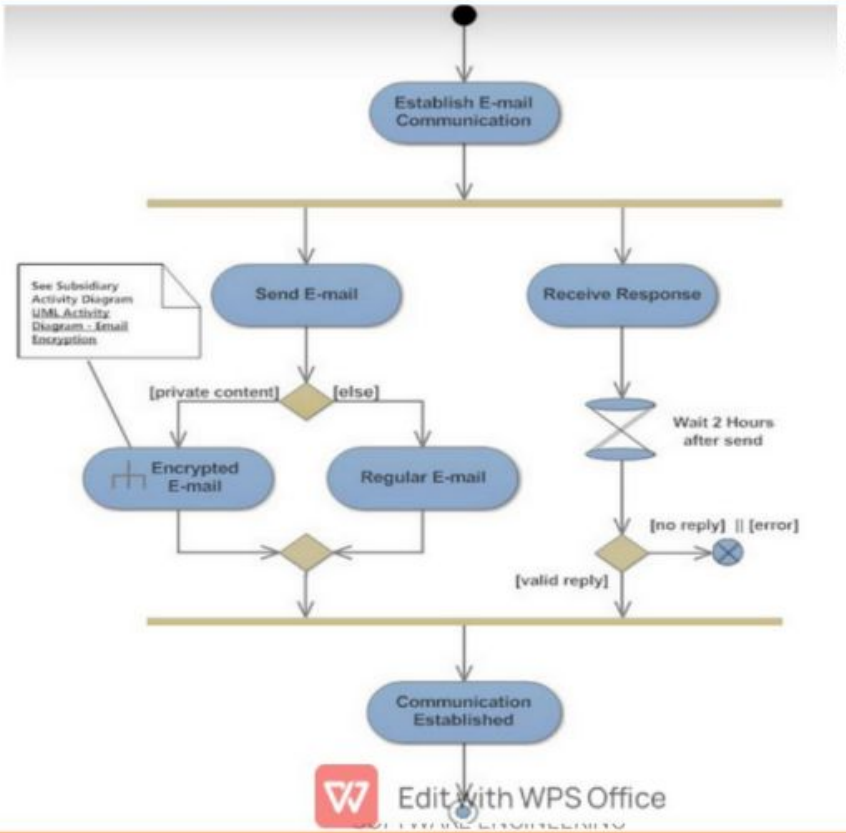
Time Event This refers to an event that stops the flow for a time; an hourglass depicts it.	
Flow Final Accepts inputs, does nothing	
Swimlane and Partition A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread	



Activity Diagram
Basic Example

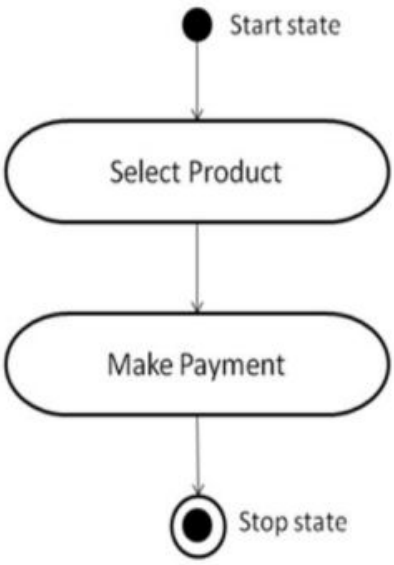
8

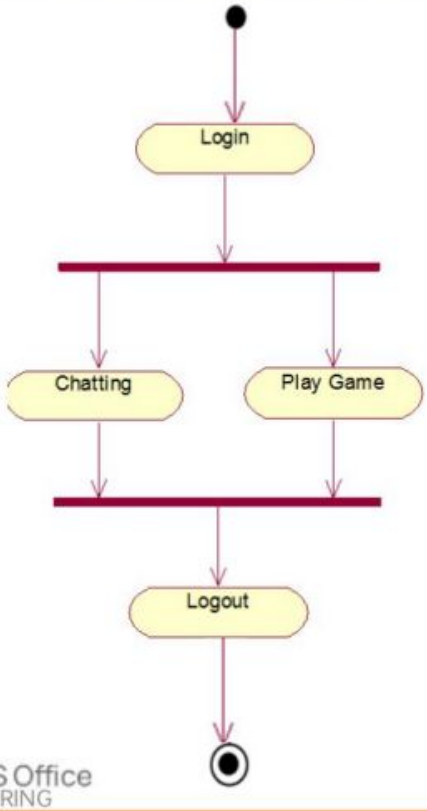
Activity Diagram



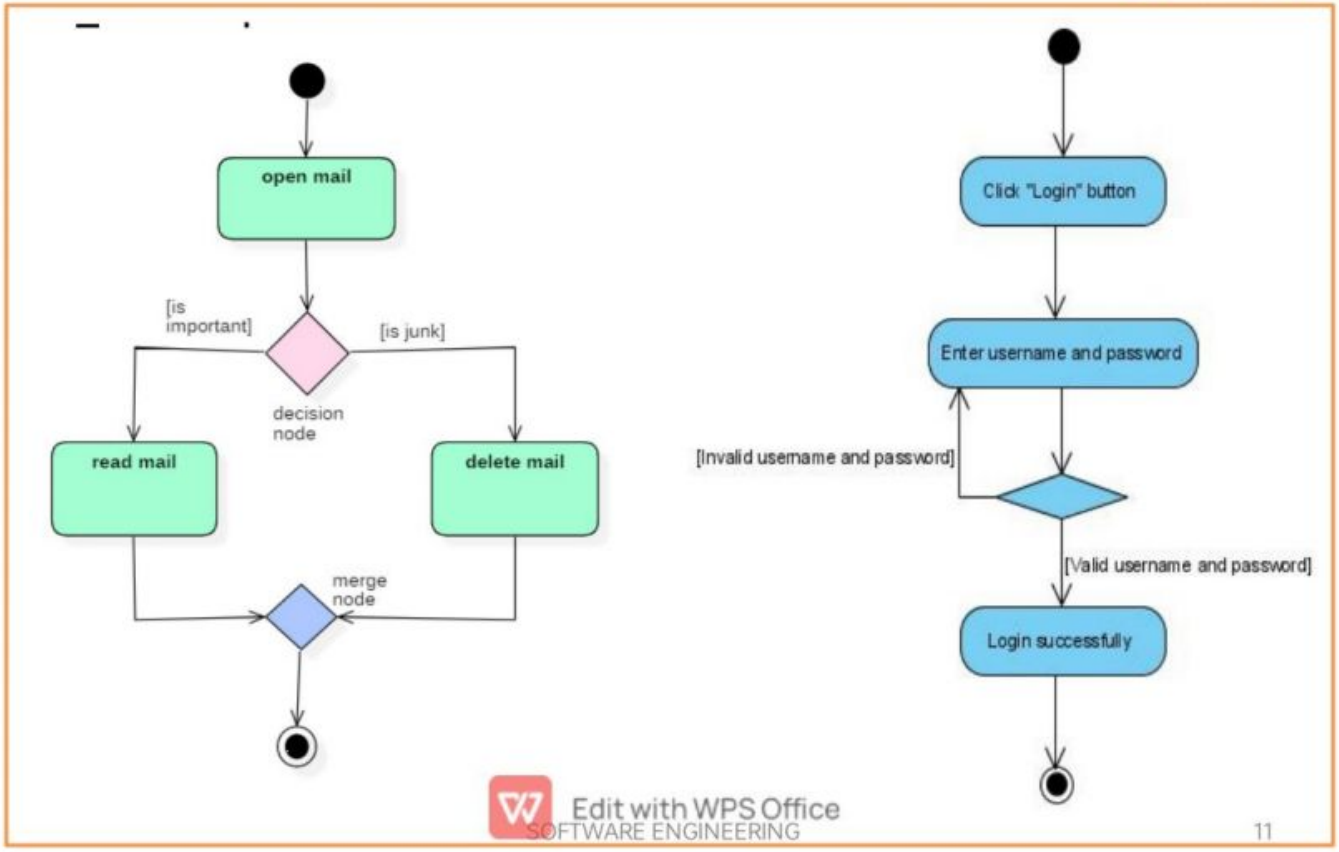
Activity Diagram

- Examples:





Activity Diagram - Examples

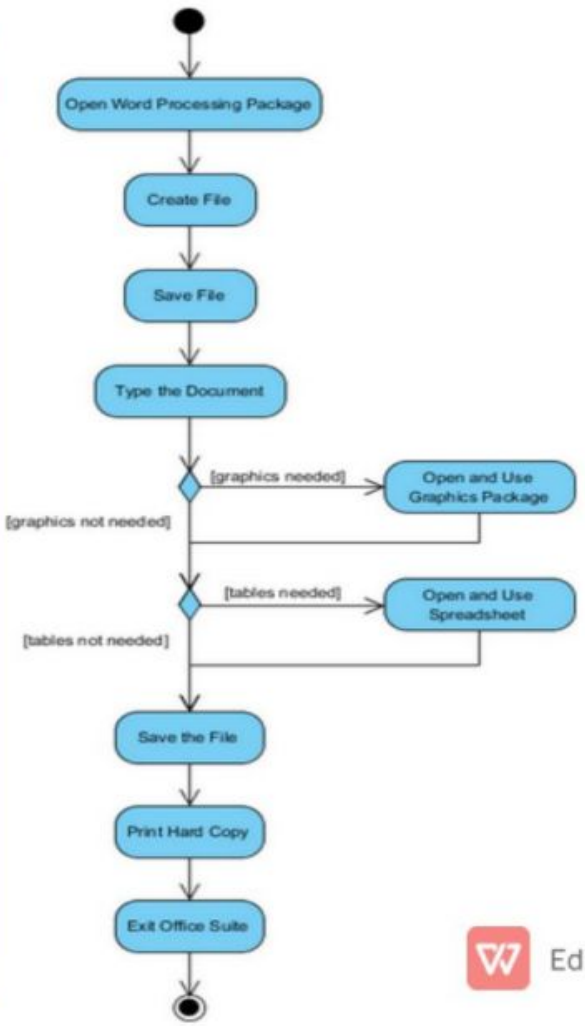


SOFTWARE ENGINEERING 11

Activity Diagram

- **Activity Diagram - "Modeling a Word Processor"**
- The activity diagram example below describes the workflow for a word process to create a document through the following steps:

- Open the word processing package.
- Create a file.
- Save the file under a unique name within its directory.
- Type the document.
- If graphics are necessary, open the graphics package, create the graphics, and paste the graphics into the document.
- If a spreadsheet is necessary, open the spreadsheet package, create the spreadsheet, and paste the spreadsheet into the document.
- Save the file.
- Print a hard copy of the document.
- Exit the word processing package.



Activity Diagram

Activity Diagram - "Modeling a Word Processor"

NGINEERING 13

Activity Diagram

Activity Diagram Example - Process Order

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

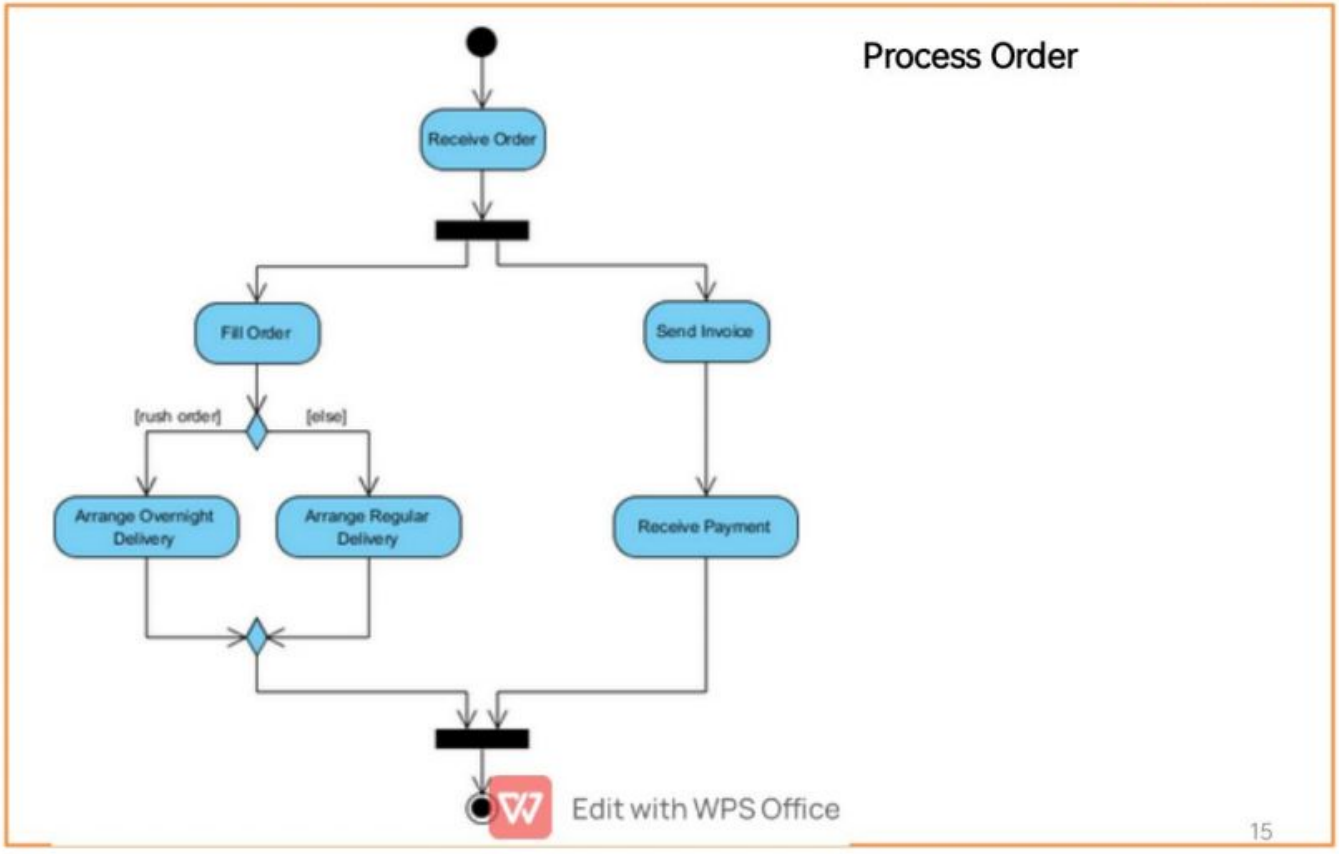
Process Order - Problem Description

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

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

Finally the parallel activities combine to close the order.

Activity Diagram



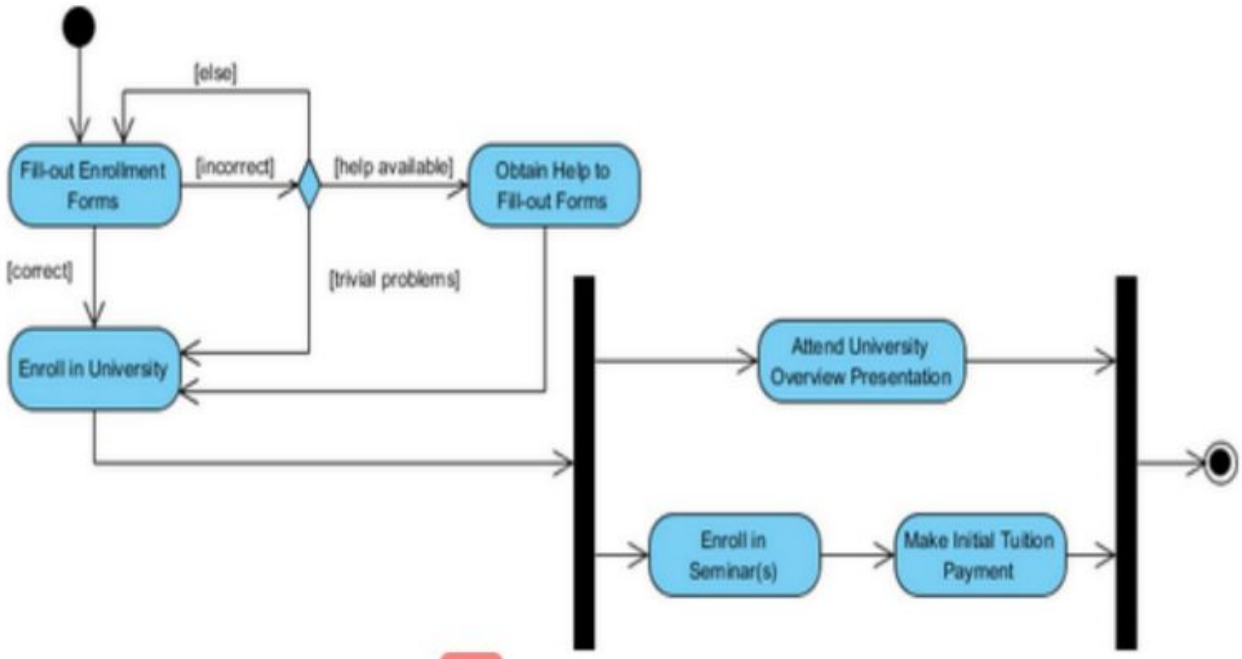
Activity Diagram

Example - Student Enrollment

- **This UML activity diagram example describes a process for student enrollment in a university as follows:**
- An applicant wants to enroll in the university.
- The applicant hands a filled out copy of Enrollment Form.
- The registrar inspects the forms.
- The registrar determines that the forms have been filled out properly.
- The registrar informs student to attend in university overview presentation.
- The registrar helps the student to enroll in seminars
- The registrar asks the student to pay for the initial tuition

Activity Diagram

- Activity Diagram Example - Student Enrollment



Activity Diagram - Swimlanes

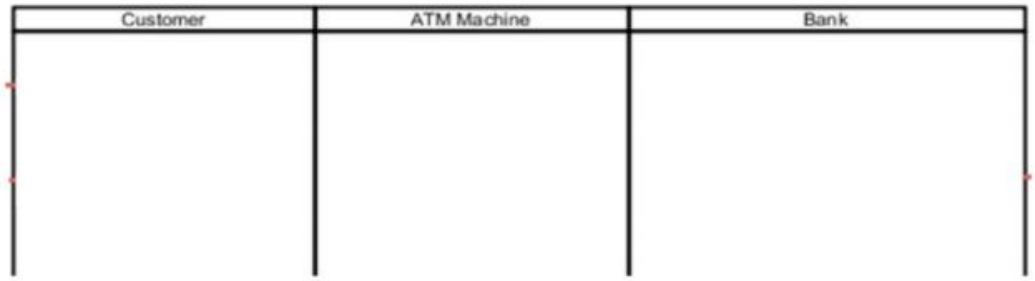
- A **swimlane** is a way to group activities performed by a Actor or department depends on the scenario.

- a swimlane is a visual region in an activity diagram that indicates the element that has responsibility for action states within the region.

Activity Diagram - Swimlanes

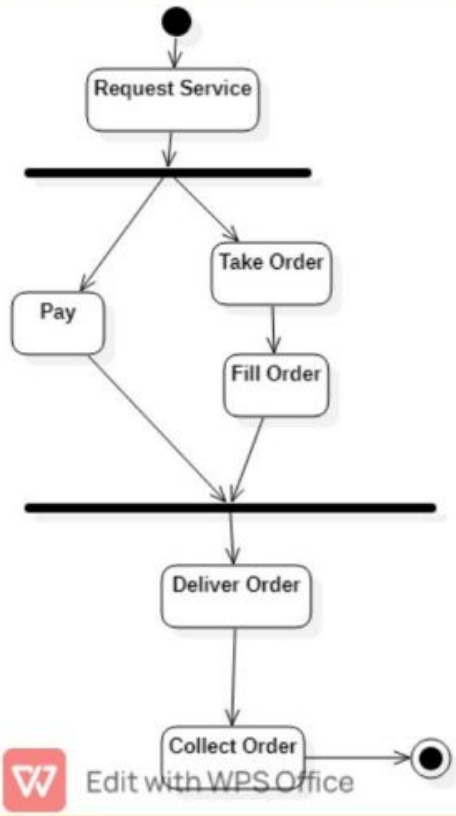
- **Example using Swimlanes:**
- Here is an activity diagram example for ATM.
- Withdraw money from an ATM Account –
- The three involved classes (people, etc.) of the activity are **Customer**, **ATM**, and **Bank**.
- represented **swimlanes** that determine which object is responsible for which activity.

Activity Diagram - Swimlanes



Activity Diagram

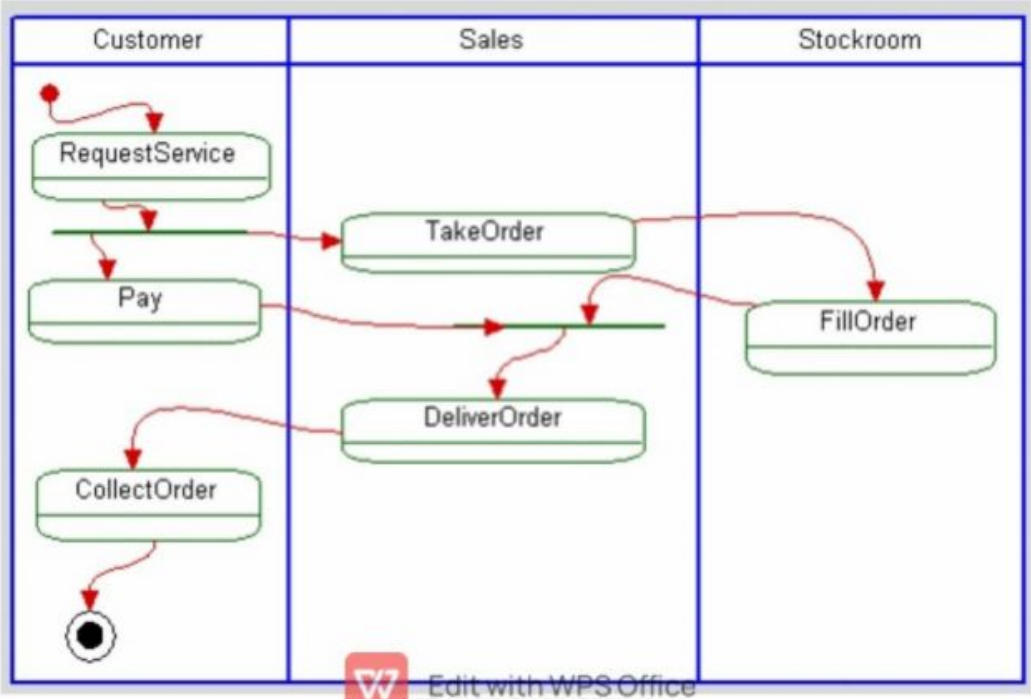
- Activity diagram



21

Activity Diagram

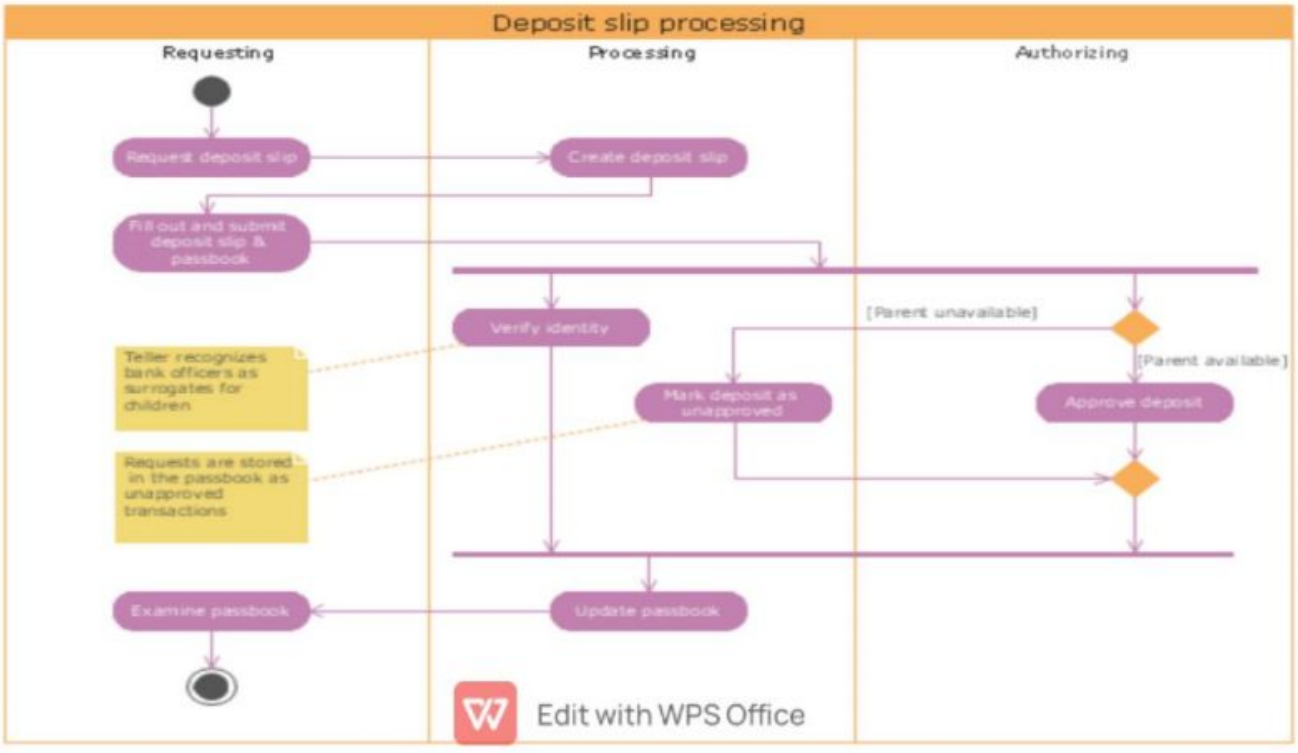
- Activity diagram using the concept of swimlane



Activity Diagram

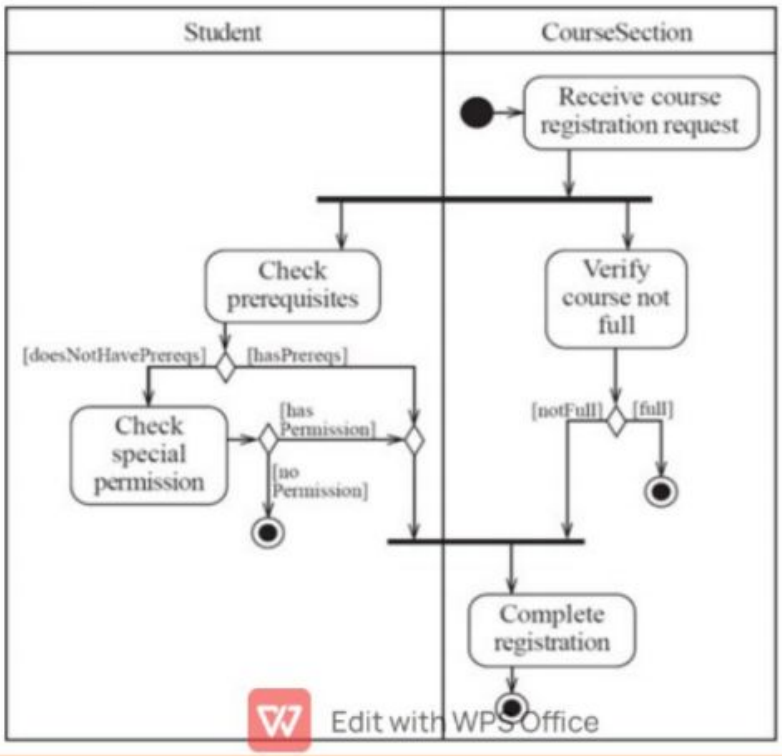
- Activity Diagram for Deposit Slip Processing
- 1. Customer request for deposit slip
- 2. deposit slip issued to customer
- 3. Fill and submit deposit slip & passbook
- 4. verify identity
- 5. while verification mark deposit as unapproved or Approved deposit.
- 6. update passbook
- 7. customer cross check the passbook
- 8. exit from the process

Activity Diagram



Activity Diagram

- Course Registration using Swimlanes



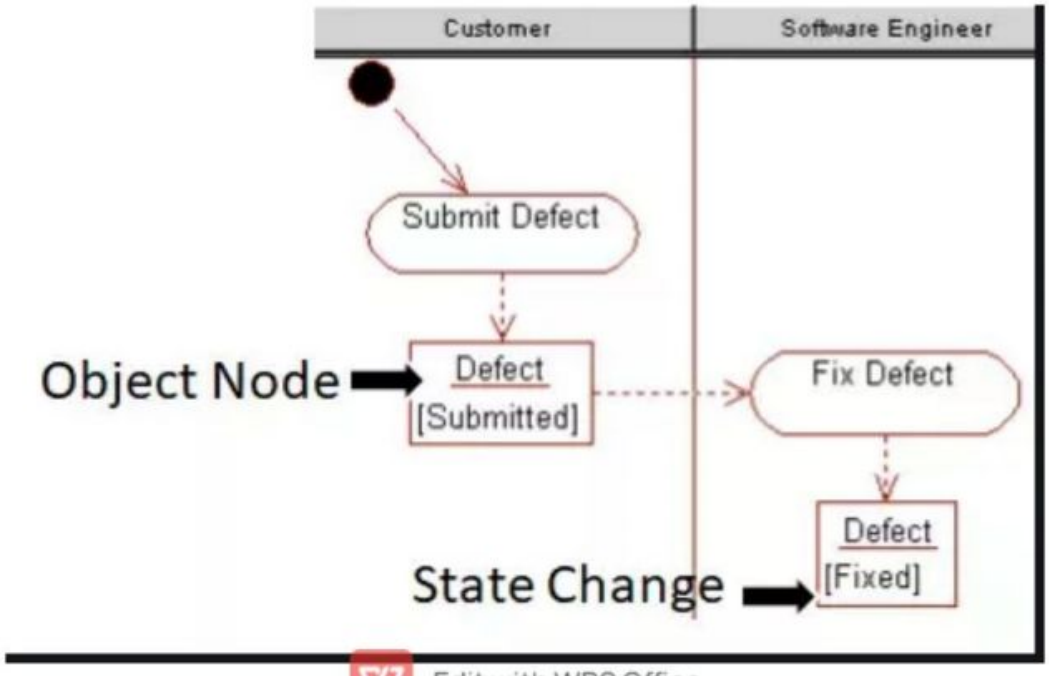
25

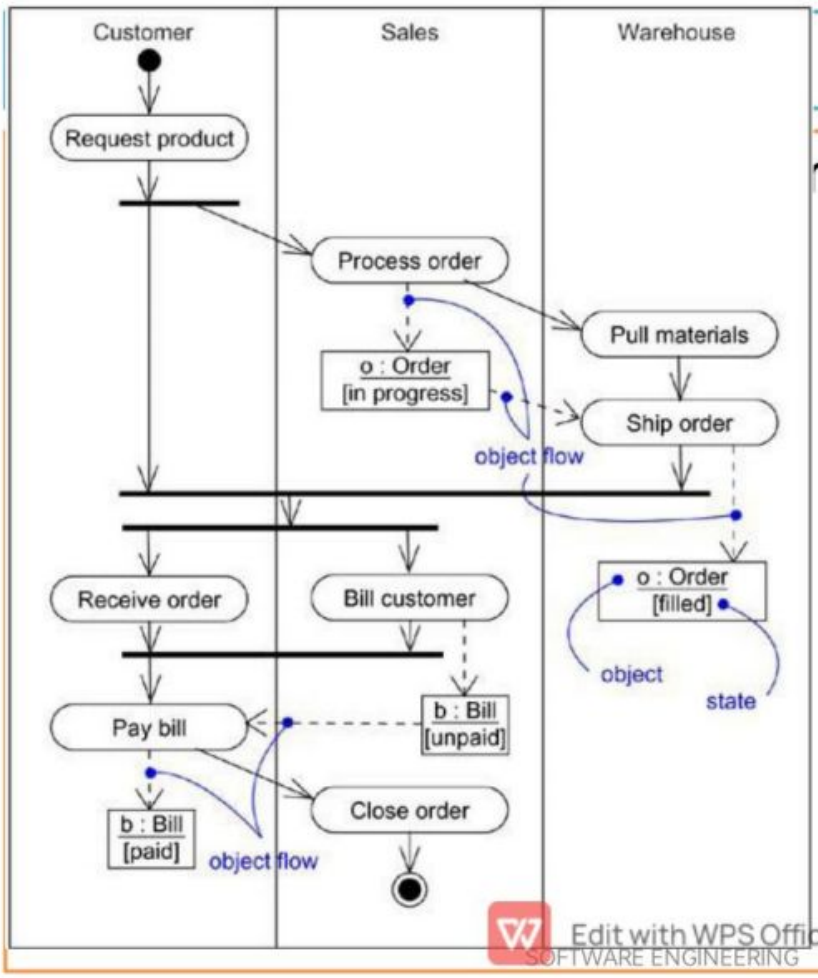
Activity Diagram

- **Object Flow in Activity Diagram:**
- The **object flow** describes the **flow** of **objects** and data within **activities**.
- Edges can be labeled with a name (close to the arrow):
- The **object flow** in an **activity diagram** shows the path of one or more business **objects** between the various **activities**.

Activity Diagram

- Object Flow in Activity Diagram:





Object Flow in Activity Diagram:

4+1 View Architecture

Functional Requirements What the system should provide in terms of service to its users.

Software Module Organization (Hierarchy of Layers, Software Management, Reuse..)

End user

Logical View

Class, Object, Package, Composite Structure, State Machine

Implementation View

Programmers & software managers

Component

Use Case View

Use Case, Activity

Integrator

Process View

Sequence, Communication, Activity, Timing, Interaction Overview

Deployment View

System Engineer

Deployment

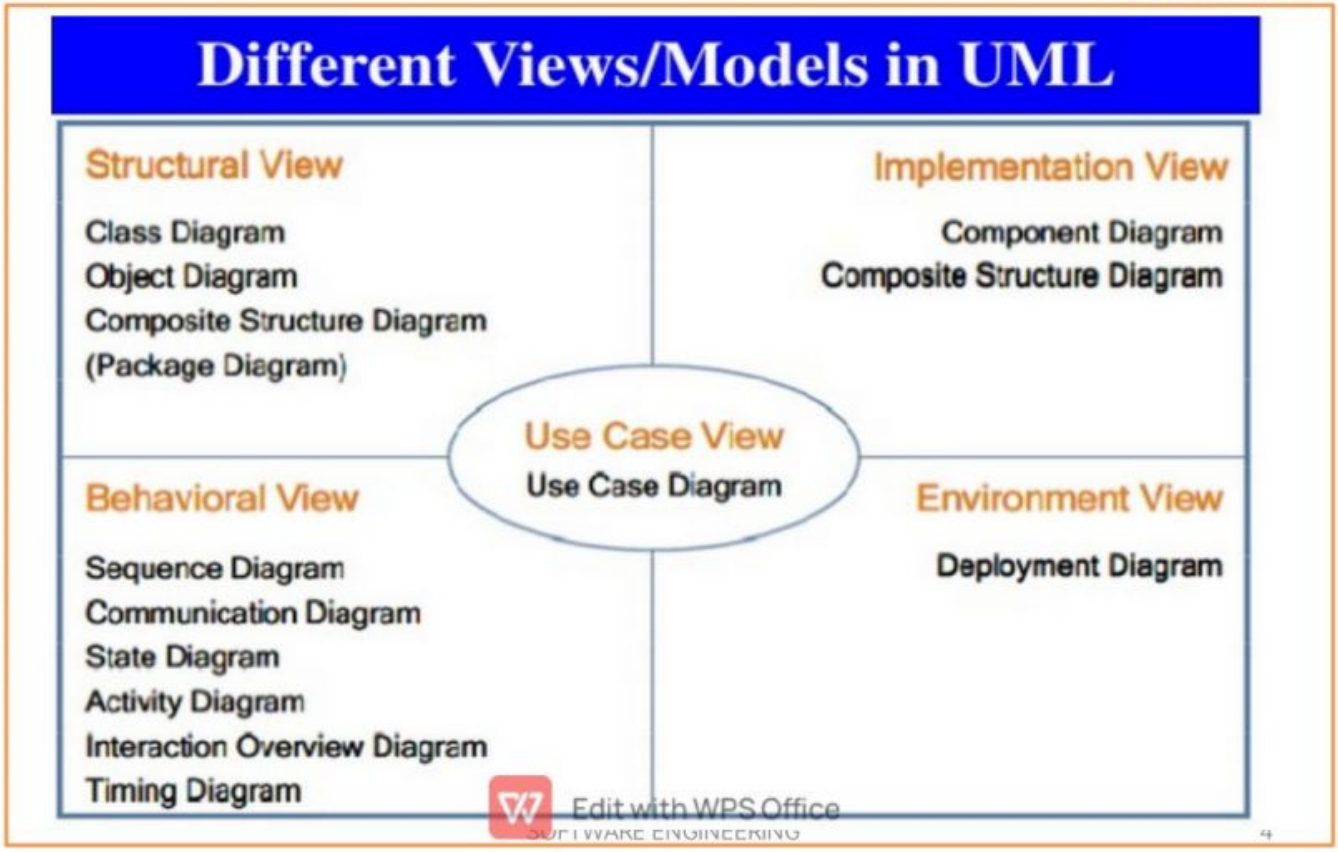
Non-Functional Requirements (Performance, Security, Usability..)

Non-Functional Requirements for Hardware (Topology, Communication)

SOFTWARE ENGINEERING

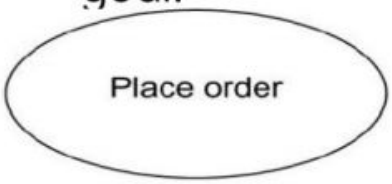
3

4+1 View


Different Views/Models in UML
Structural View
Class Diagram
Object Diagram
Composite Structure Diagram
(Package Diagram)
Implementation View
Component Diagram
Composite Structure Diagram
Use Case View
Use Case Diagram
Behavioral View
Sequence Diagram
Communication Diagram
State Diagram
Activity Diagram
Interaction Overview Diagram
Timing Diagram
Environment View
Deployment Diagram
Edit with WPS Office
SOFTWARE ENGINEERING
4

Things in UML

Use Case: A **use case** is a description of how a person who actually **uses** that process or system will accomplish a goal.



It outlines, from a user's point of view, a system's behavior as it responds to a request.

Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name

Use Case Diagram

- Use case diagrams are considered for high level requirement

analysis of a system.

- use cases are nothing but the system functionalities written in an organized manner.

Use Case Diagram

How to Identify Actor:

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- What other systems use this system?
- Who gets information from this system?
- Who provides information to the system?

Use Case Diagram

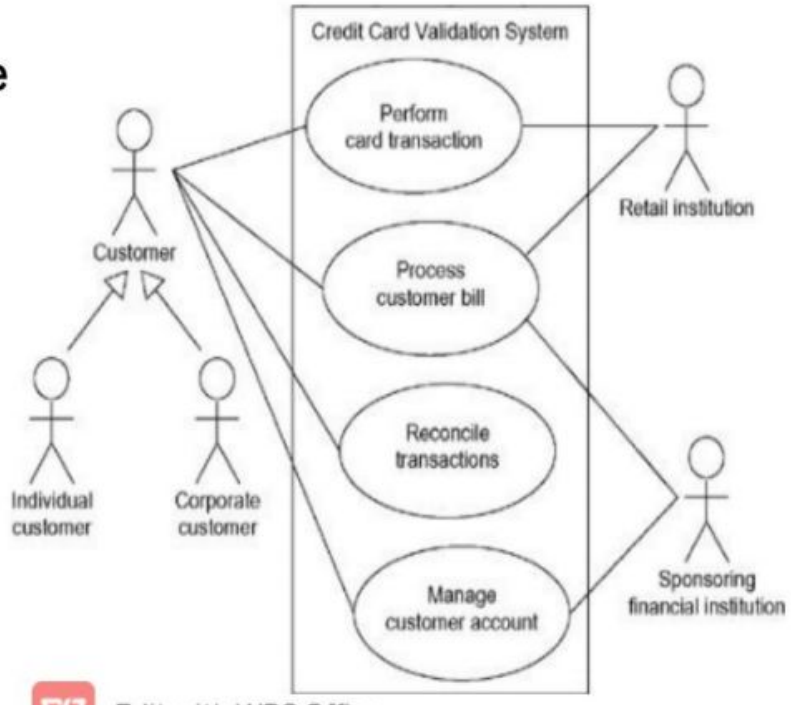
Common Modeling Techniques:

1. Modeling the Context of a System
2. Modeling the Requirements of a System

Use Case Diagram

Common Modeling Techniques:

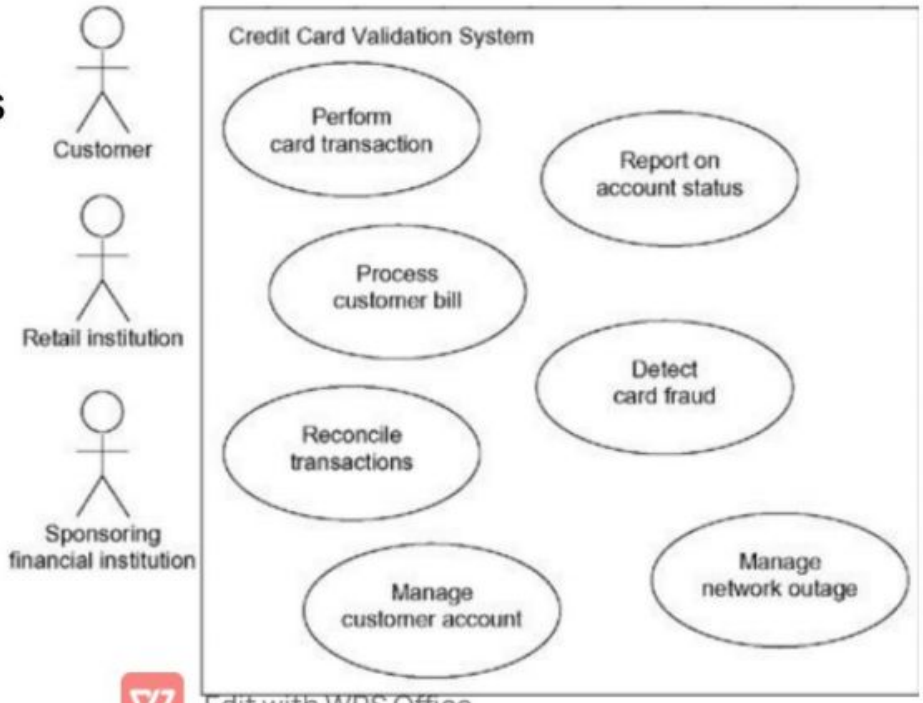
- Modeling the Context of a Syste



Use Case Diagram

Common Modeling Techniques:

Modeling the Requirements of a System



Use Case Diagram

Railway Reservation System

Identify Use Cases:

Use Cases:

Enquire Ticket Availability
Fill Form
Book Ticket
Pay Fare Amount
Print Form
Cancel Ticket
Refund Money

Actors:

Customer
Clerk
Railway System

Use Case Diagram : Railway Reservation System

Use Case Name:	Book Ticket
Actor(s)	Customer (Primary), Railway System (Secondary)
Summary Description	Allows any customer to Book the Seat in their choice (Train)
Priority	Must Have
Status	Medium Level of Details
Pre-Condition	Registered person(Mobile)
Post-Condition(s)	Ticket will be Generated with Seat Number, Location, Station Name, Train Number & Name. SMS will be Forwarded

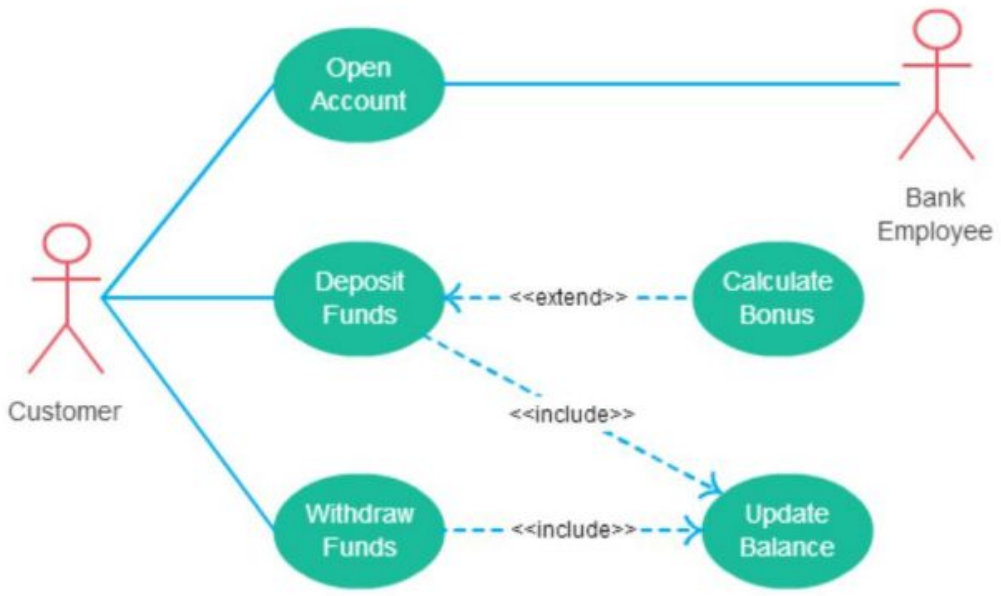
Use Case Diagram

Railway Reservation System

Basic Path:	1. Customer Login with his Account 2. Validated with SMS 3. Search for Trains by entering Source and destination of station points 4. Select departure date and if required select return journey also 5. Select Train number 6. Select number of seats 7. Select payment method 8. Pay amount 9. Ticket generates sends SMS, email and print option will be provided 10. Logout from the website.
Alternative Paths:	Invalid User Name Invalid Password Invalid Phone Number Ticket number Not Generated Insufficient Amount in Customer Account Source and destination invalid

Use Case Diagram

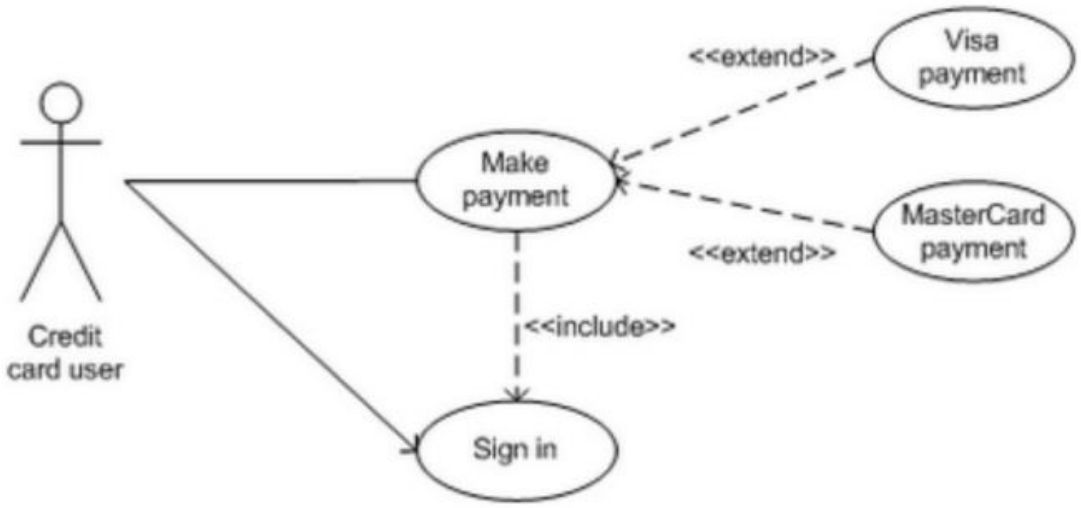
Example: <<include>> and <extend>> Relationships:



Use Case Diagram

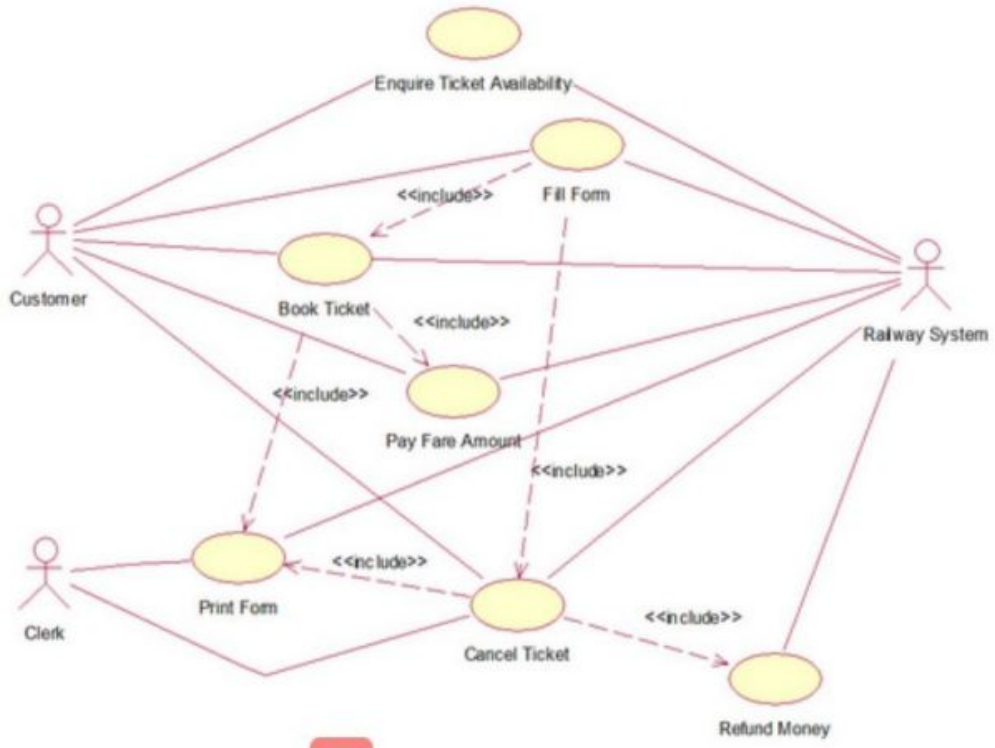
Railway Reservation System

<<include>> and <extend>> Relationship:



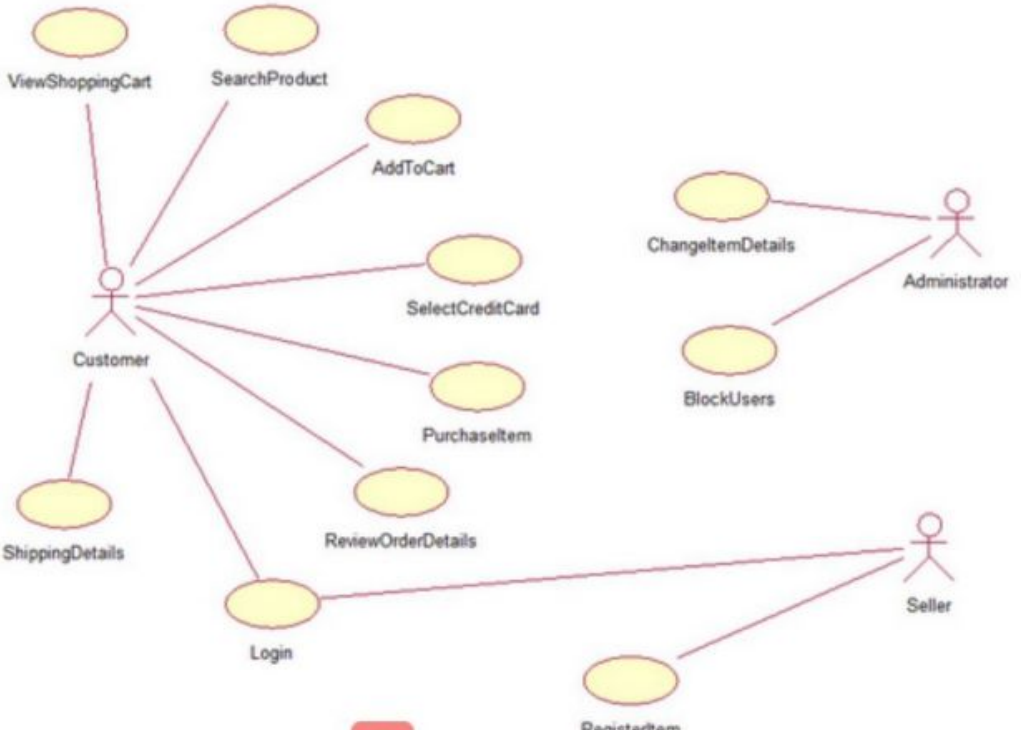
Use Case Diagram

Railway Reservation System



16

Use Case Diagram : Online Bookshop



Thank You

Interaction Diagrams

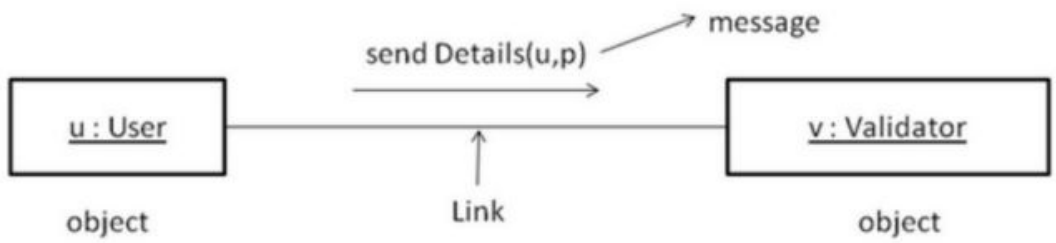
- The dynamic aspects of a system can be modeled using interactions.
- Interactions contain messages that are exchanged between objects.
- A message can be an invocation of an operation or a signal.
- The messages may also include creation and destruction of other objects.

Interaction Diagrams

- Using interaction diagrams, we can model these flows in two(2) ways:
- one is by focusing on how the messages are dispatched across time. (Sequence Diagram)
- the second is by focusing on the structural relationships between objects and then consider how the messages are passed between the objects. (Collaboration Diagram)

Interaction Diagrams

- Let us Consider Example :



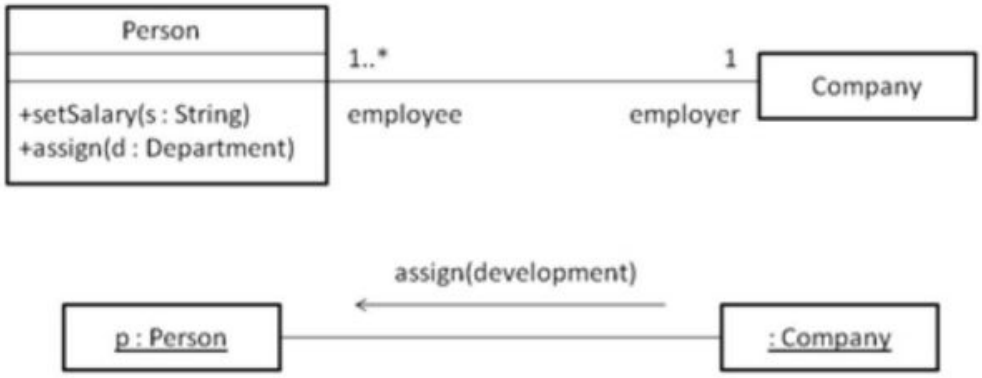
- Object
- Interaction
- Message
- Link

Interaction Diagrams

- **Object :** The objects that participate in an interaction are either concrete things or prototypical things.
- **Interaction :** An interaction is a behavior that contains a set of messages exchanged among a set of objects within a context to accomplish a purpose.
- **Message :** A message is specification of a communication between objects that conveys information with the expectation that the activity will succeed.
- **Link :** A link is a semantic connection among objects. In general, a link is an instance of association.

Interaction Diagrams

- Let Us consider An Example:



Interaction Diagrams

- An interaction diagram represents an interaction, which contains a **set of objects** and the **relationships** between them including the **messages exchanged** between the **objects**.

Interaction Diagrams

- **A sequence diagram** is an interaction diagram in which the focus is on time ordering of messages.
- **Collaboration diagram** is another interaction diagram in which the focus is on the structural organization of the objects.
- Both **sequence diagrams** and **collaboration diagrams** are **isomorphic** diagrams

Course Code	Course Name	Academic Year	UG	L	T	P	C
A4603	Software Engineering	2020-2021	III B. Tech CSE I Semester	3	0	2	4

Software Engineering
CSE A & B

BEHAVIORAL MODELING (UML)
Sequence Diagram

Faculty
Mr S Venu Gopal
Associate Professor
Department of CSE

Interactions Diagrams

- To model flows of control by time ordering (Sequence Diagram)
- To model flows of control by organization (Collaboration Diagram)

Sequence Diagrams

- Notations:
  - Object
  - Object Life Line
  - Activations
  - Call Message 

  - Return Message 

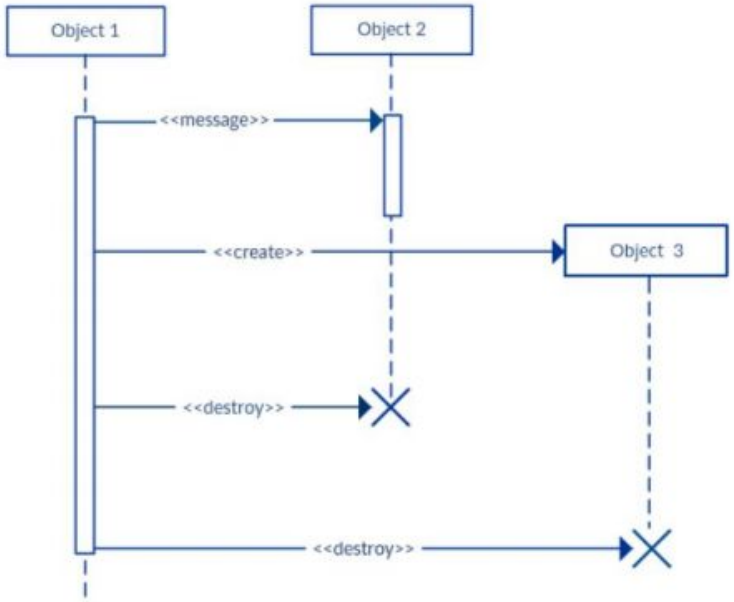
  - Self Message 

  - Create <<create>> 

  - Destroy <<destroy>> 

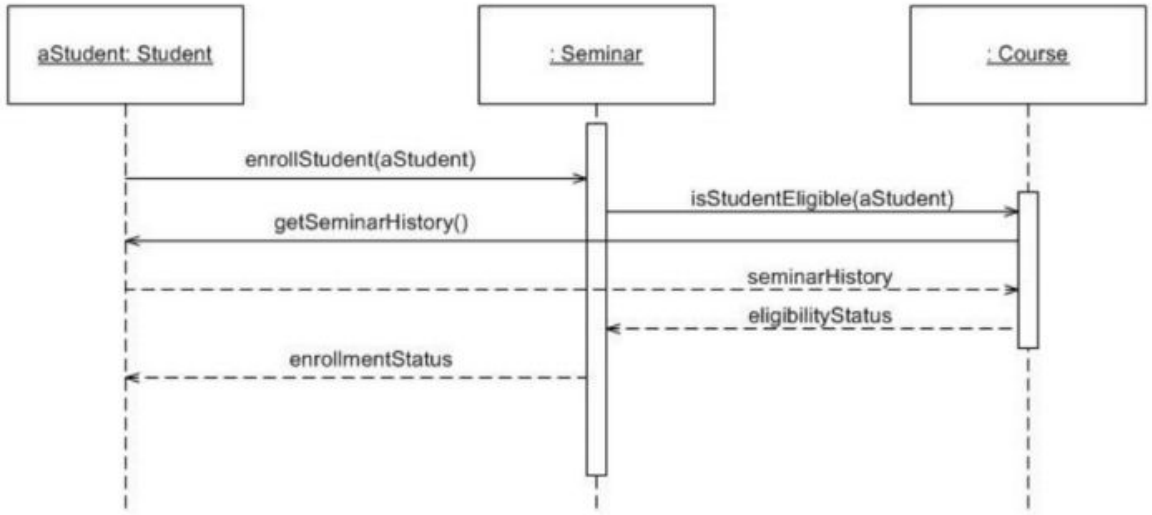
Sequence Diagram

- Basic Example:



Today's Session: Outlines

- Lets consider an example Enrolling in a seminar (method)

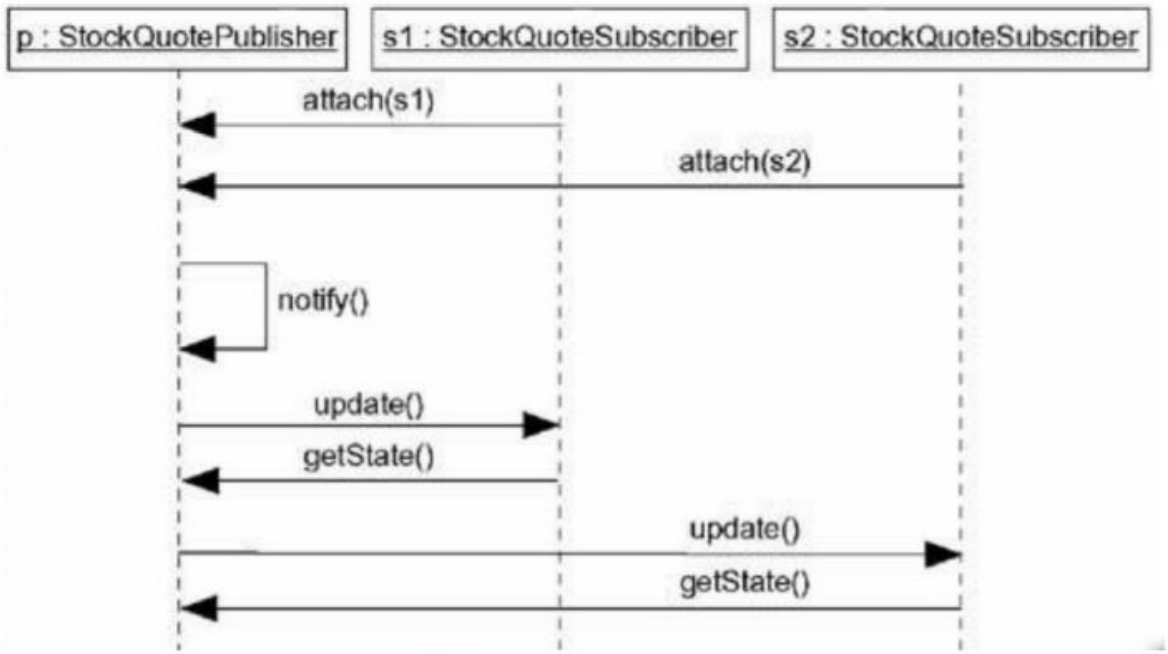


Sequence Diagram

- **Steps:**
  - Identify the objects that take part in the interaction.
  - Identify message as per the situation Starting with the messages that initiate the interaction
  - If require create object
  - When object created dynamically, if require use destroy

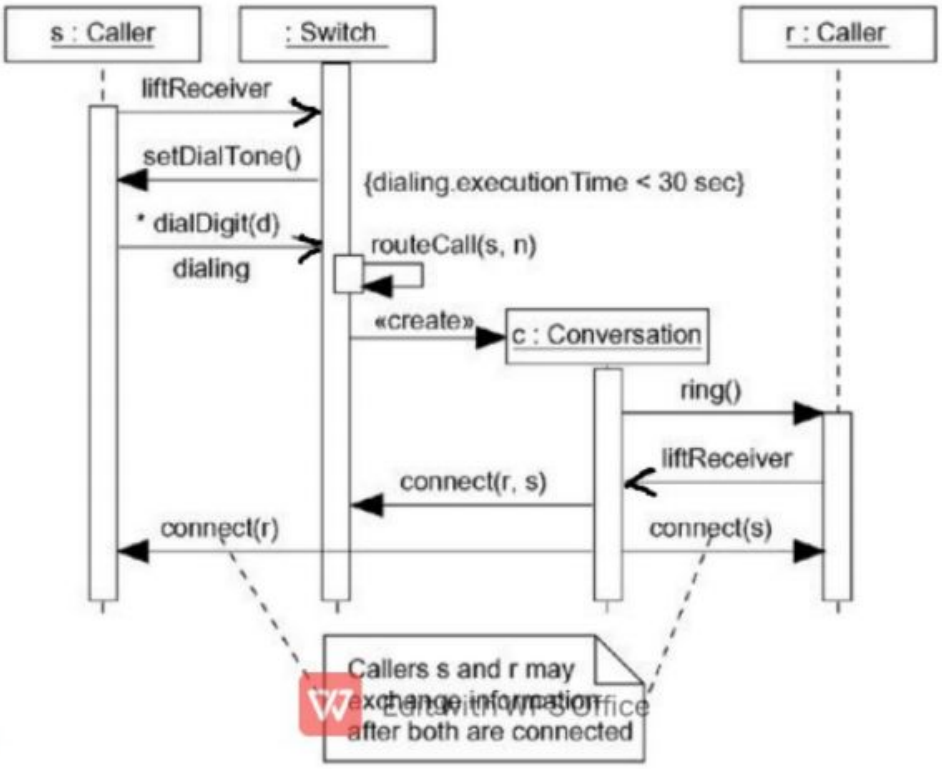
Sequence Diagram

- the context of a publish and subscribe mechanism



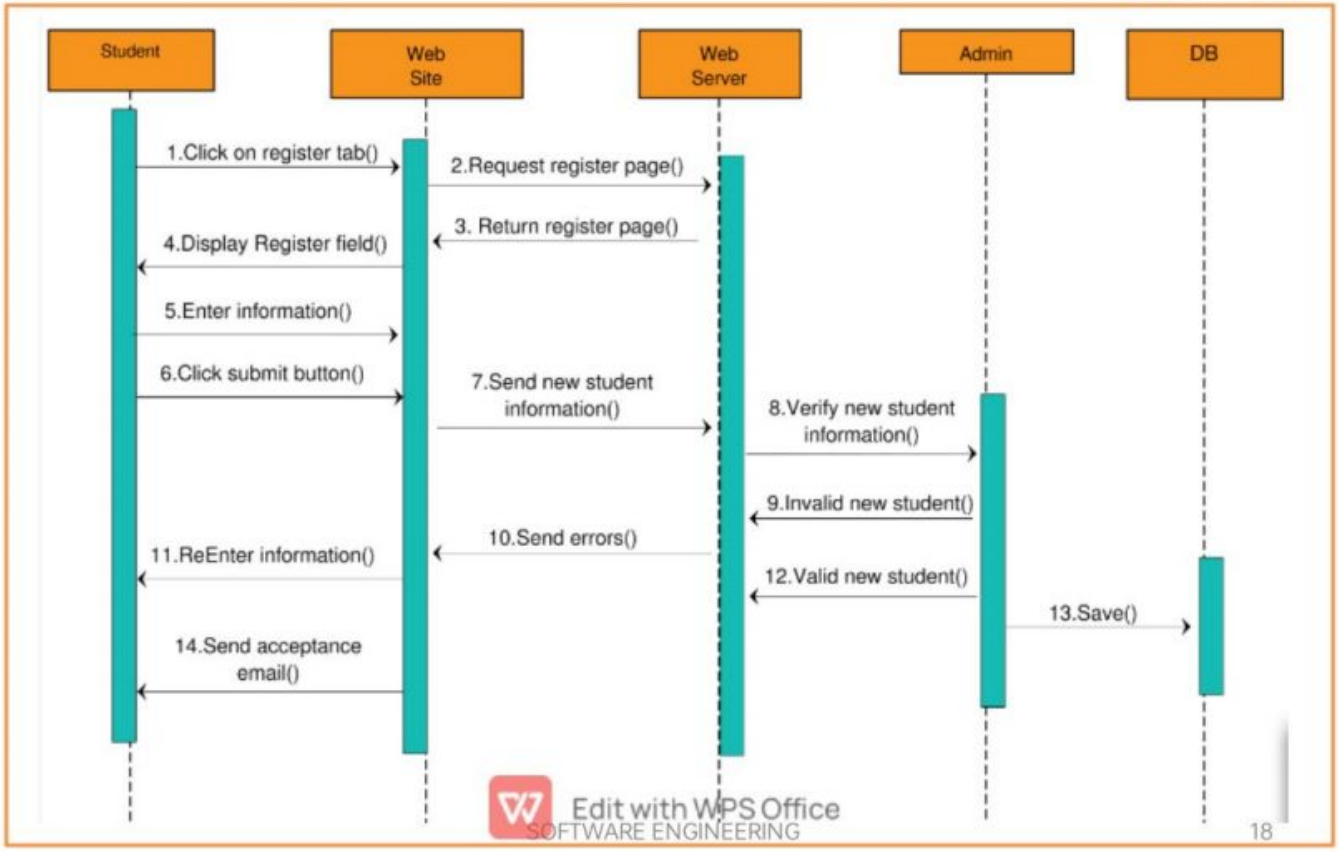
Sequence Diagram

Shows a sequence diagram that specifies the flow of control involved in initiating a simple, two-party phone call.

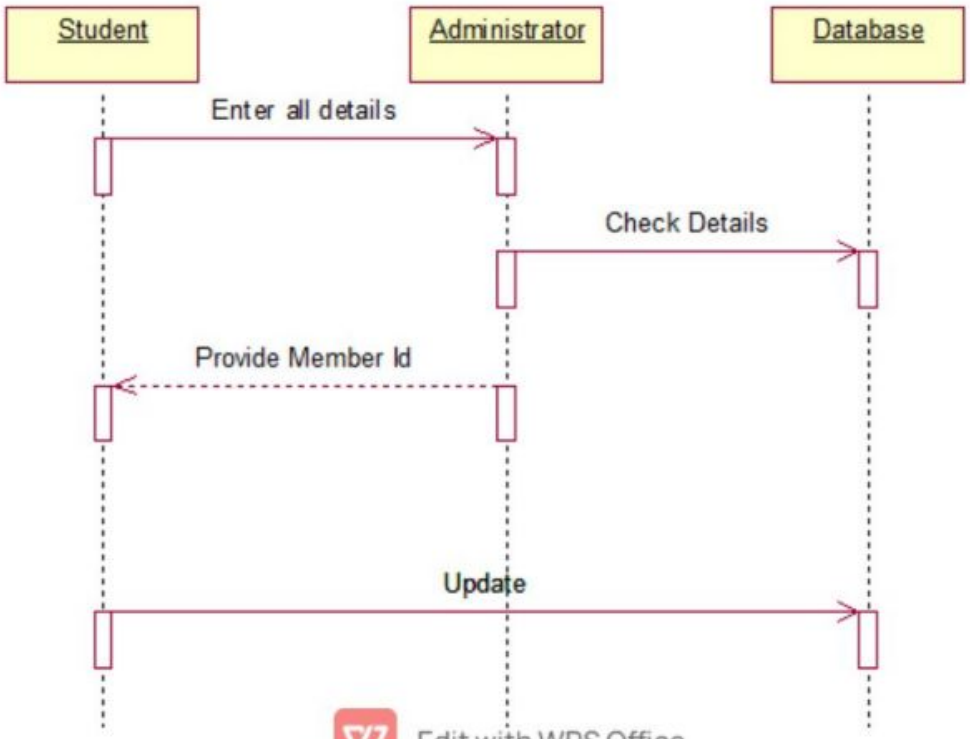


17

Sequence Diagram : Student Registration



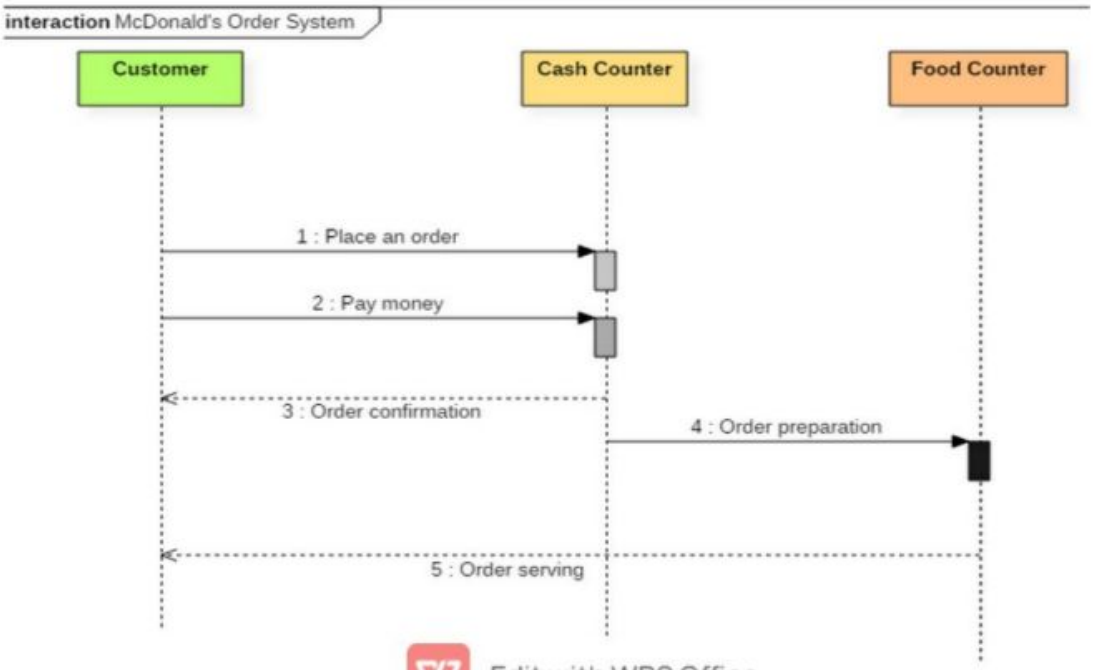
Sequence Diagram : Student Registration



Sequence Diagram : McDonald's Ordering System

Sequence diagram example

The following sequence diagram example represents McDonald's ordering system:



Collaboration Diagram

It also called "communication diagram"

- **Introduction:**
- concerned about object organization.
- The collaboration diagram and sequence diagram shows similar information but in a distinct form.
- It can be used to depict the relationship among various objects within the system.
- Objects collaborate through passing messages to each other.
- Every message inside a collaboration diagram contains a sequence number.

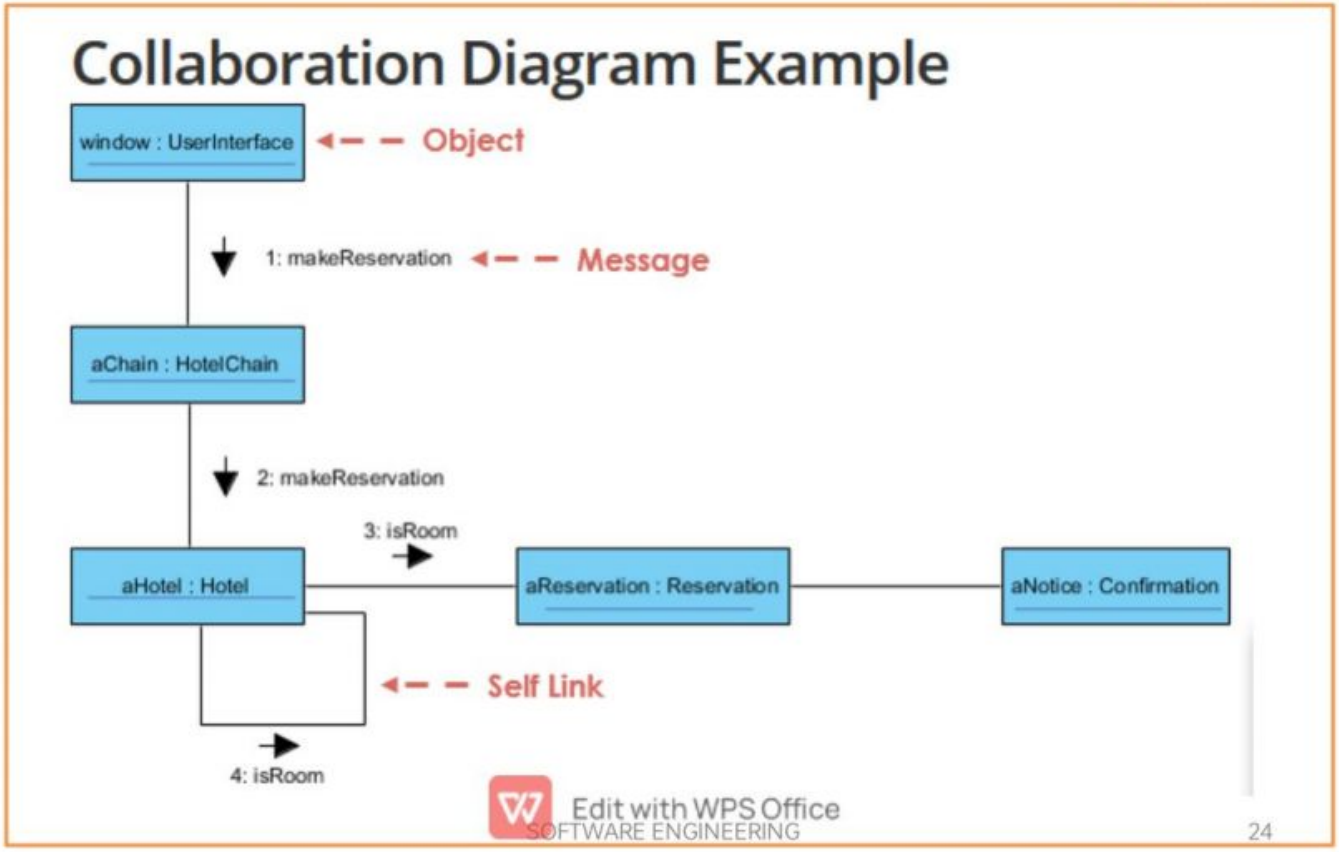
Collaboration Diagram

- **Notations**
  - Objects
  - **Links** : A link can be a relation between objects for which messages are transferred
  - **Messages** : It is communication among objects that transmits information with an expectation that action will arise.

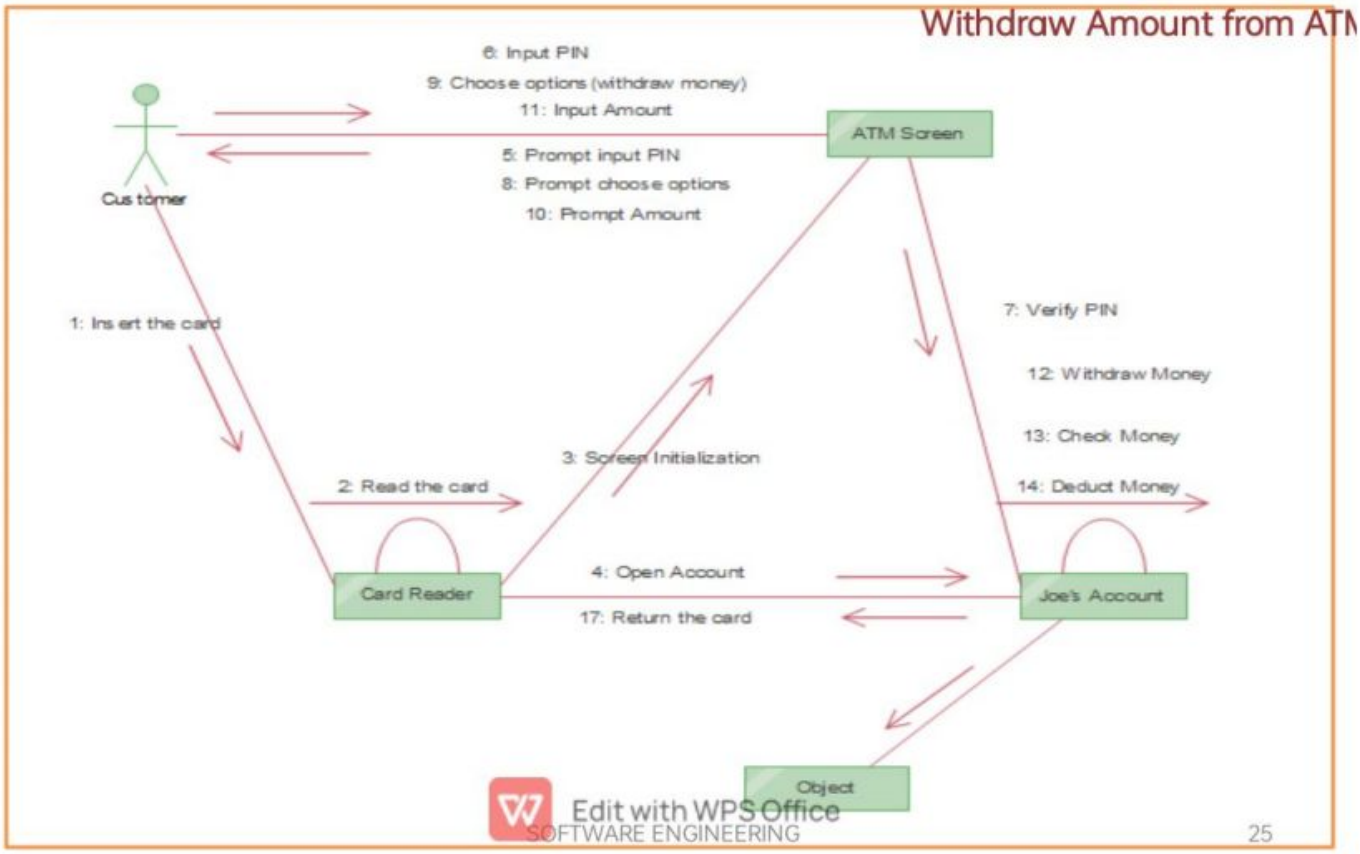
Collaboration Diagram

- **Steps :**
- Analyze the system behavior
- Identify objects
- Establish the links between one object to other object
- Identify the messages according to scenario

Collaboration Diagram

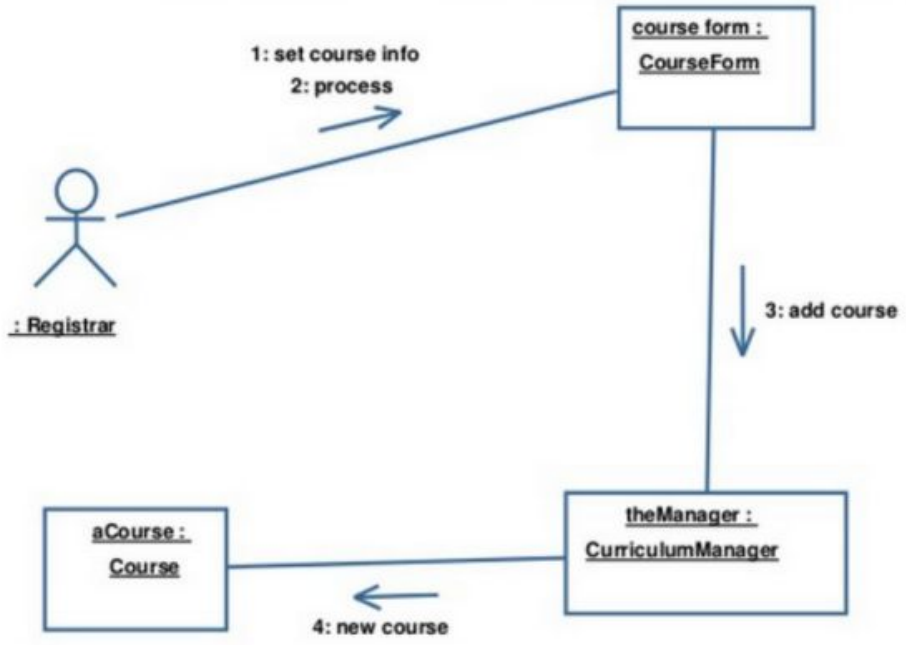


Collaboration Diagram



Collaboration Diagram

- Adding a New Course to Curriculum

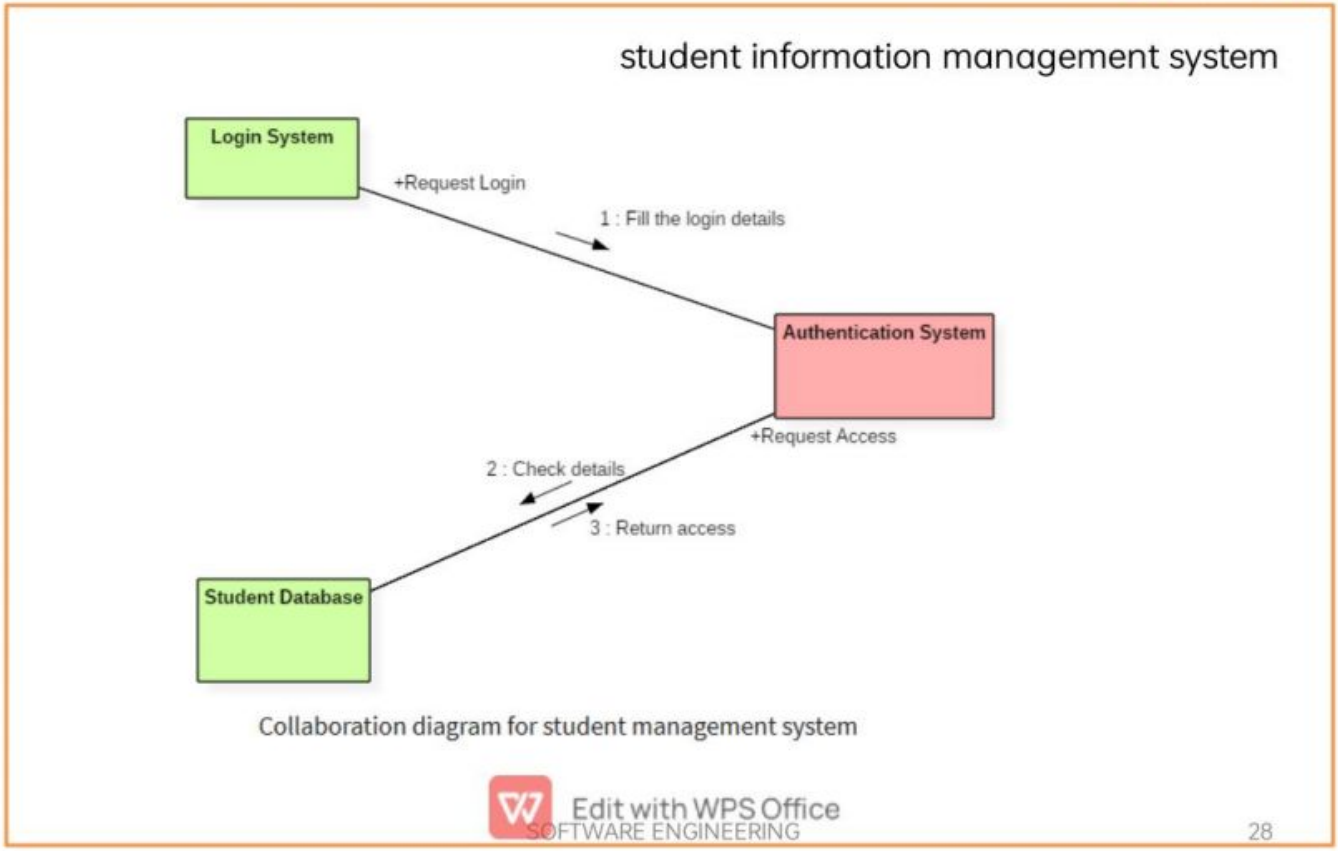


Collaboration Diagram

- student information management system.
- The flow of communication in the above diagram is given by,

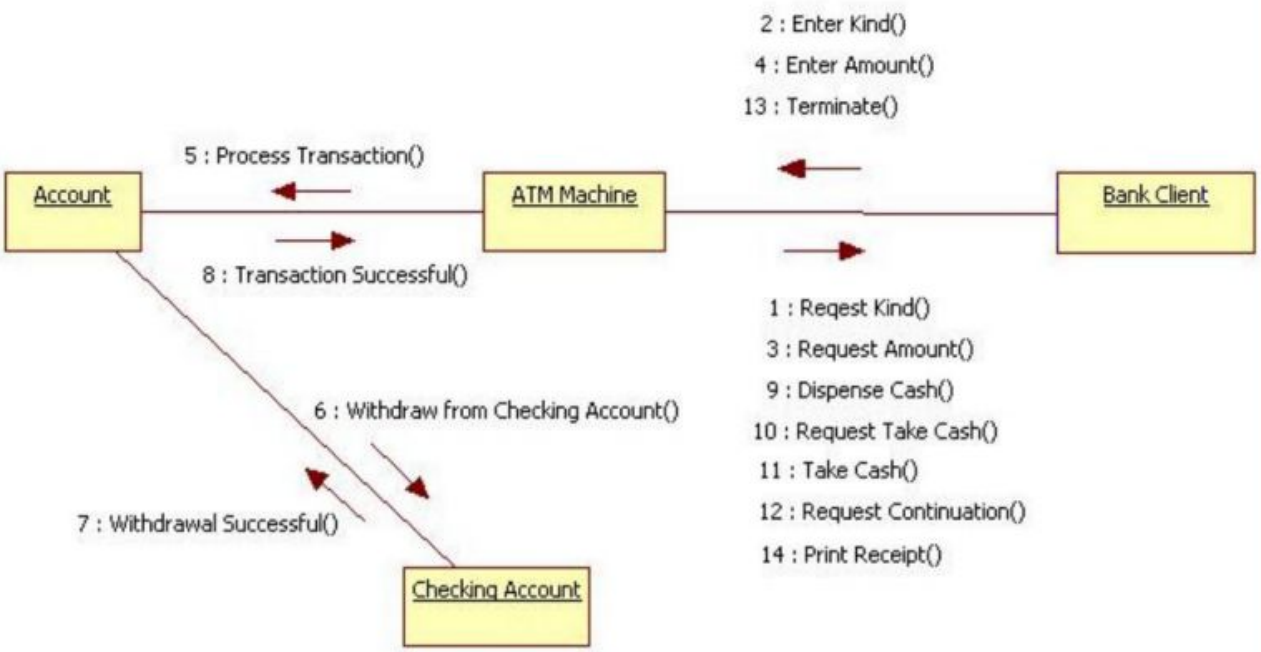
1. A student requests a login through the login system.
2. An authentication mechanism of software checks the request.
3. If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.

Collaboration Diagram



Collaboration diagram for student management system

Collaboration Diagram



Statechart diagram

Or

State Machine Diagram

A state machine is a device that stores the status of an object at any given time.

Statechart Diagram

- **What is a state diagram?**
  - A state diagram is a graphic representation of a state machine. It shows a behavioral model consisting of states, transitions, and actions, as well as the events that affect these.

State Chart Diagram

- **Following are the main purposes of using Statechart diagrams –**
- To model the dynamic aspect of a system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

State Chart Diagram

- A **state diagram**
  - is used to represent the condition of the system or part of the system at finite instances of time

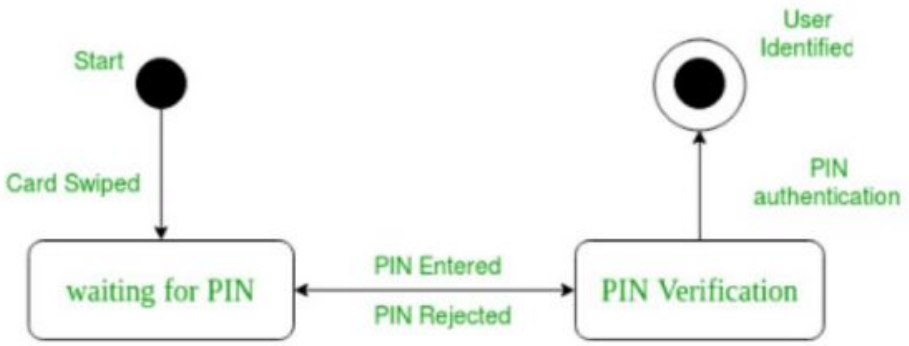
State Chart Diagram

- **Notations:**

Notation Name	Explanation	Graphical Symbol
State	A state is a condition of an object in which it performs some activity or waits for an event. An object may remain in a state for a finite amount of time.	
Transition	A transition represents the change from one state to another:	
Guard Condition	A guard condition is a condition that has to be met in order to enable the transition to which it belongs:	[Guard Condition]
Event	an event is an occurrence of a stimulus that can trigger a state transition.	Event / Action
Initial State	The initial state represents the source of all objects:	
End State	The final state represents the end of an object's existence:	

State Chart Diagram

- Basic Example:



State Chart Diagram

- **State :** A state is a condition of an object in which it performs some activity or waits for an event. An object may remain in a state for a finite amount of time. A state has several properties:

Name	A textual string which distinguishes the state from other states; a state may also be anonymous, meaning that it has no name.
Entry/exit actions	Actions executed on entering and exiting the state.
Internal transitions	Transitions that are handled without causing a change in state.
Substates	The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates.
Deferred events	A list of events that are not handled in that state but are postponed and queued for handling by the object in another state.

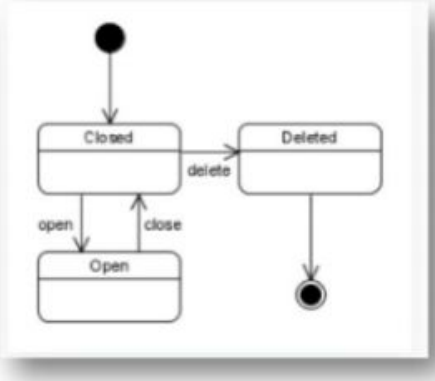
State Chart Diagram

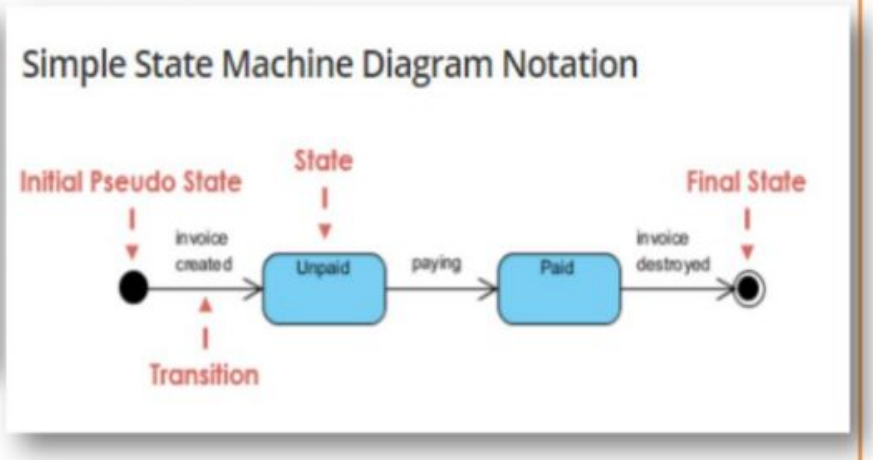
- **Transitions** : A transition is a relationship between two states

Source state	The state affected by the transition; if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied.
Event trigger	The event that makes the transition eligible to fire (providing its guard condition is satisfied) when received by the object in the source state.
Guard condition	A boolean expression that is evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates True, the transition is eligible to fire; if the expression evaluates to False, the transition does not fire. If there is no other transition that could be triggered by the same event, the event is lost.
Action	An executable atomic computation that may directly act upon the object that owns the state machine, and indirectly on other objects that are visible to the object.
Target state	The state that is active after the completion of the transition.

Statechart Diagram

- **State**
- States represent the current status of an object and appear as a rounded rectangle. In the example below, 'Closed', 'Open', and 'Deleted' are all states.



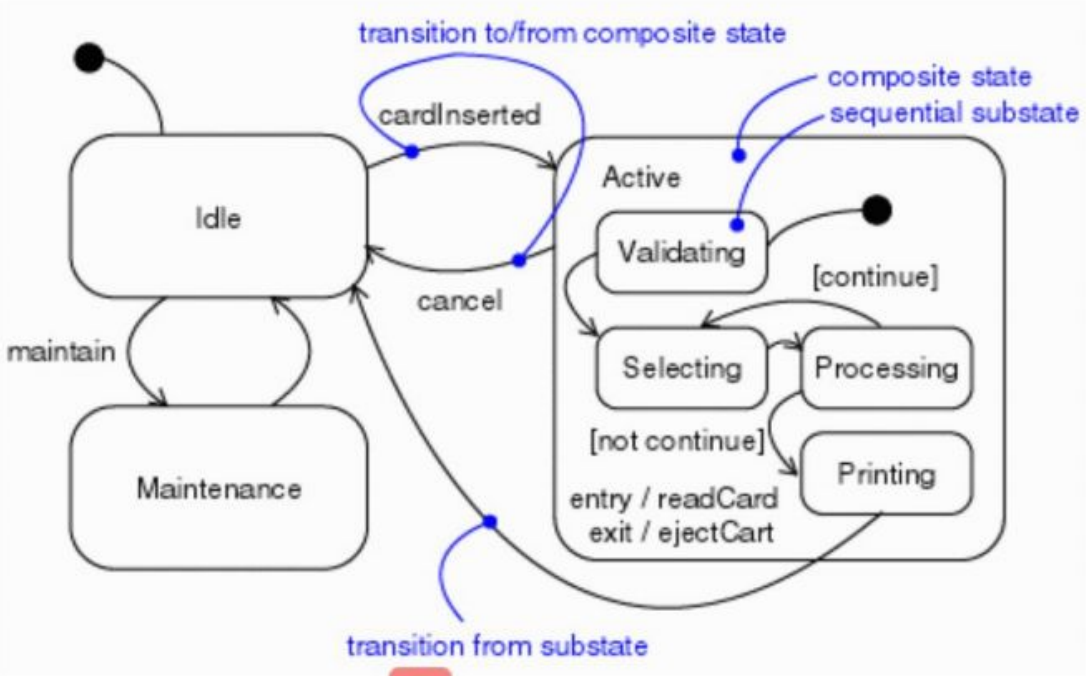


State Chart Diagram

- Some technical terms:
- Substate or Composite State :
  - **A state which has substates (nested states) is called a composite state.**
- Nested State :
  - A nested state machine may have at most one initial state and one final state.

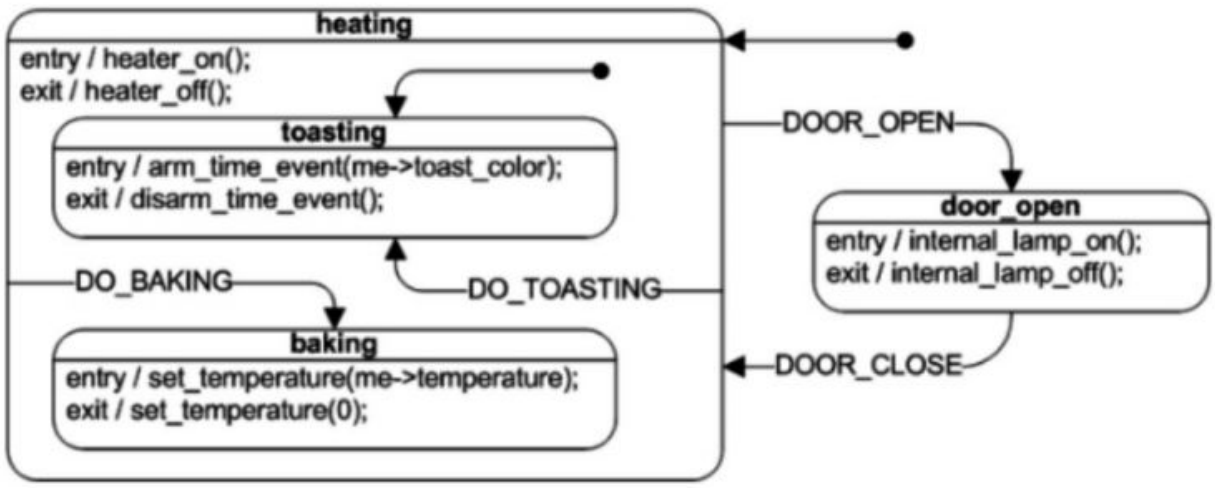
State chart Diagram

- Best Example



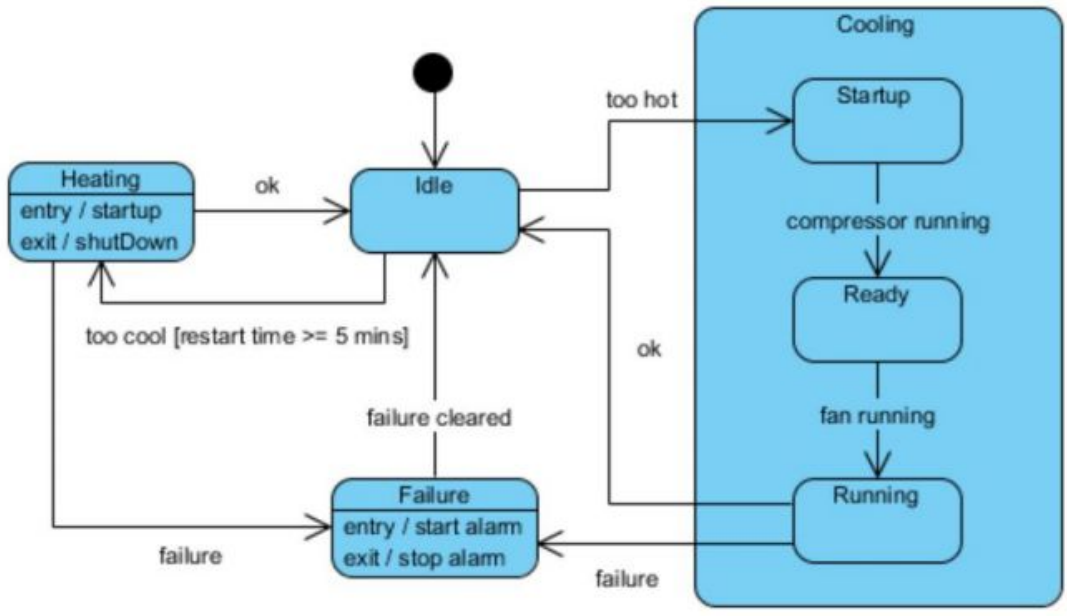
Statechart Diagram

- *Toaster oven state machine with entry and exit actions*



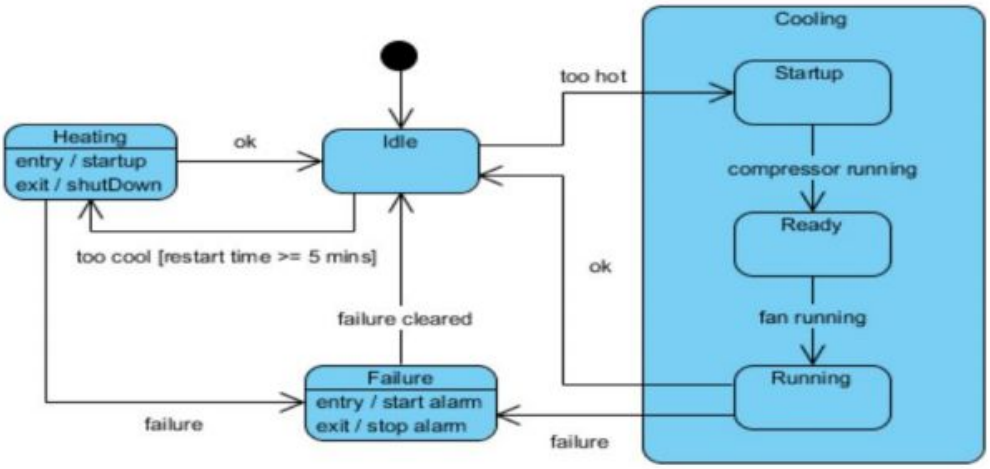
Statechart Diagram

- Substate Example - Heater



Statechart Diagram

- Substate Example - Heater

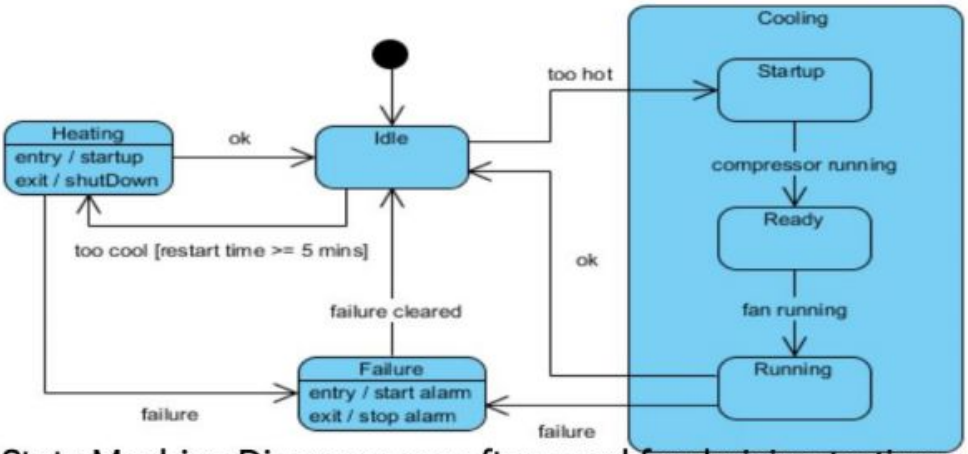


State Machine Diagrams are often used for deriving testing cases, here is a list of possible test ideas:

Idle state receives Too Hot event
Idle state receives Too Cool event

Statechart Diagram

- Substate Example - Heater

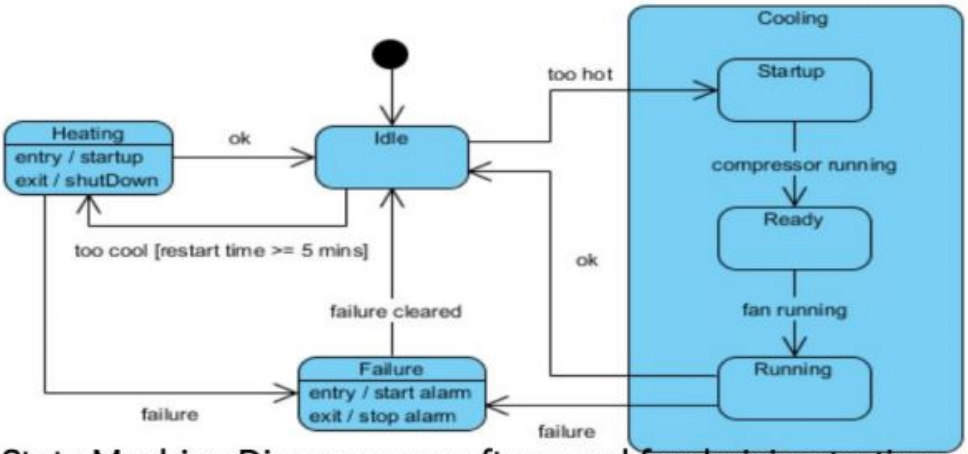


State Machine Diagrams are often used for deriving testing cases, here is a list of possible test ideas:

Cooling/Startup state receives Compressor Running event
Cooling/Ready state receives Fan Running event
Cooling/Running state receives OK event
Cooling/Running state receives Failure event

Statechart Diagram

- Substate Example - Heater

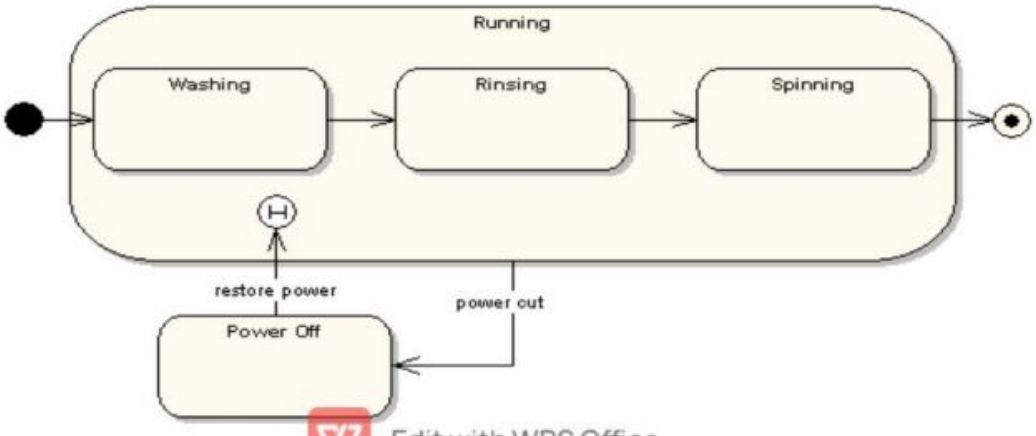


State Machine Diagrams are often used for deriving testing cases, here is a list of possible test ideas:

Failure state receives Failure Cleared event
Heating state receives OK event
Heating state receives Failure event

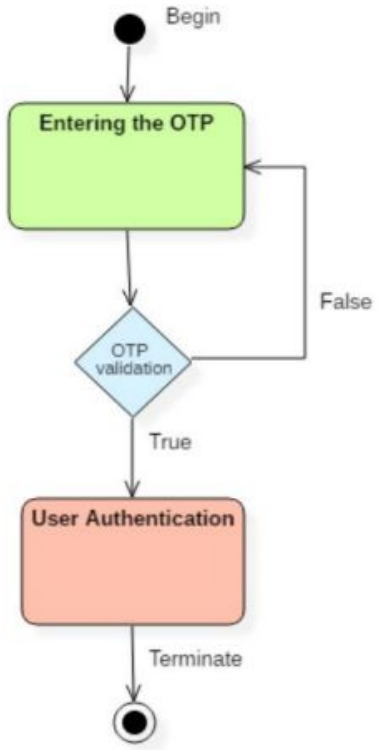
Statechart Diagram

- **History States :** (H)
- A history state is used to remember the previous state of a state machine when it was interrupted.
- The following diagram illustrates the use of history states.
- The example is a state machine belonging to a washing machine



State Machine Diagram

- Example of State Machine
- Following state chart diagram represents the us authentication process.



State machine vs. Flowchart

State machine vs. Flowchart

Statemachine	FlowChart
It represents various states of a system.	The Flowchart illustrates the program execution flow.
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow.

Statechart Diagram

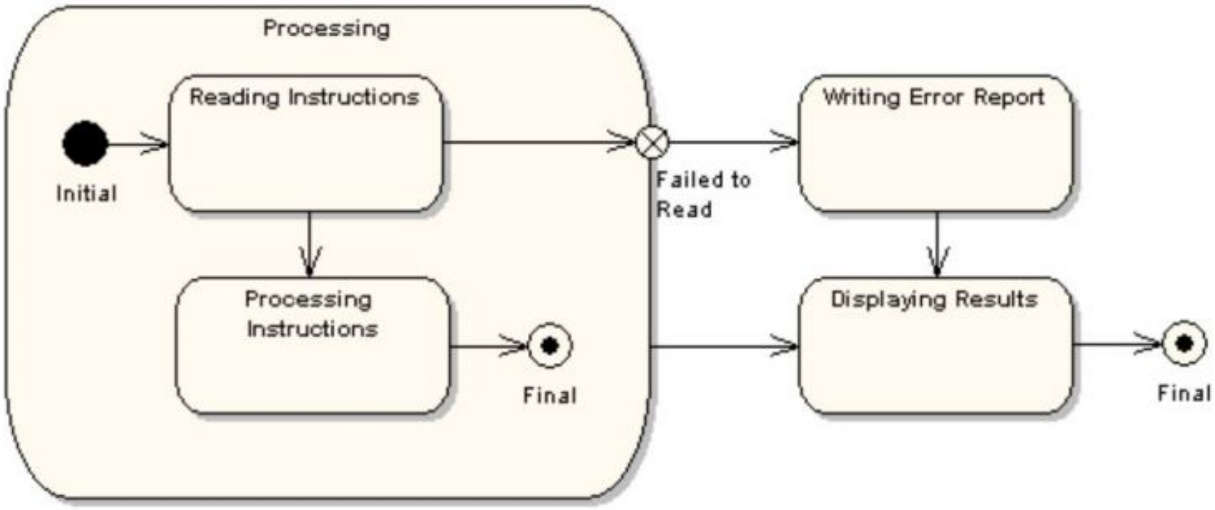
- **Basic information about Statechart Diagram is :**
- Statechart diagrams are also called as state machine diagrams.
- These diagrams are used to model the event-based system.
- A state of an entity is controlled with the help of an event.

Statechart Diagram

- **Exit Point** 
- In a similar manner to entry points, it is possible to have named alternative exit points.
- The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

Statechart Diagram

- Example: **Exit Point**



State Chart Diagram

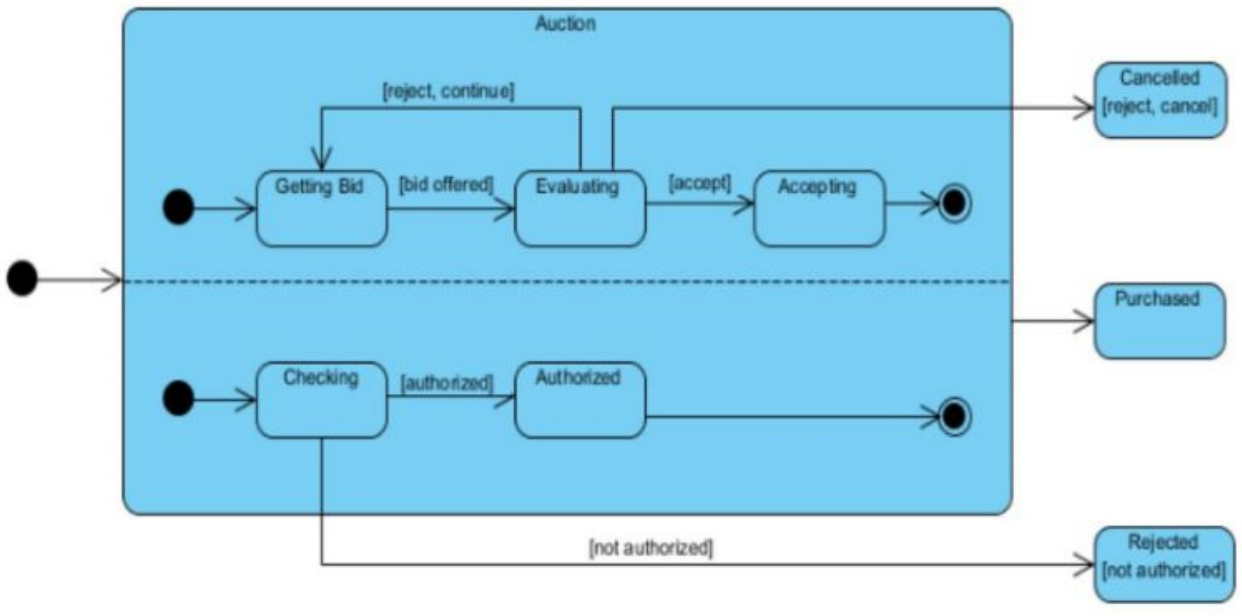
- Concurrent State or Orthogonal composite state:
  - has more than one regions
  - Concurrent Sub-states are independent and can complete at different times and each sub-state is separated from the others by a dashed line

Statechart Diagram

- Concurrent State or Orthogonal composite state:
- Case Study:

Statechart Diagram

- Action Process



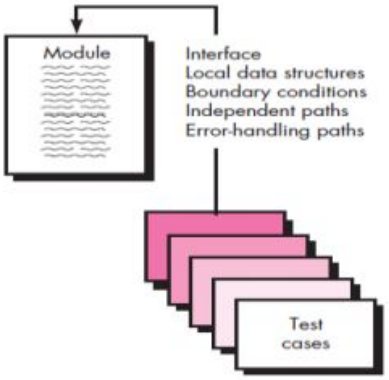
UNIT-5

Software Engineering

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components. Unit-test considerations. Unit tests are illustrated schematically in following figure. The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates

properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested. Fig : Unit Test Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the nth element of an n-dimensional array is processed, when the ith repetition of a loop with i passes is invoked, when the maximum or minimum allowable value is encountered. A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon calls this approach antibugging.



Unit-test procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. The unit test environment is illustrated in following figure.. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Integration Testing:

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design. There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. If a set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. There are two different incremental integration strategies :

Top-down integration. Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner. Referring to the following figure,

depth-first integration integrates all components on a major control path of the program structure. For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

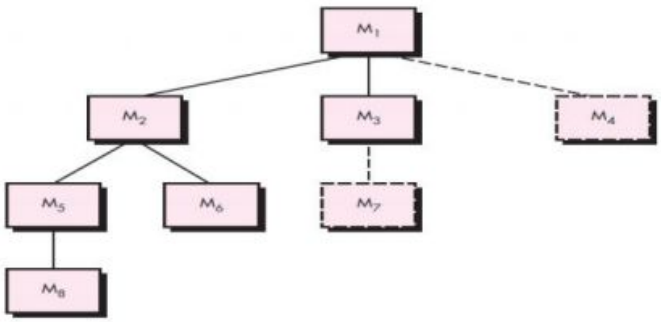


Fig : Top-down integration

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced. Bottom-up integration.

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub function.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in following figure. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

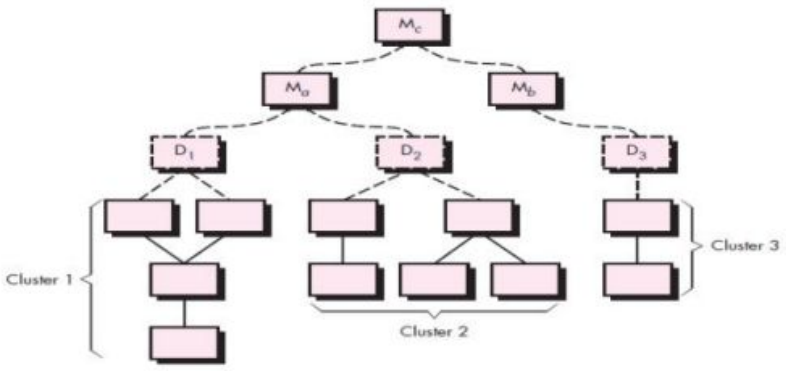


Fig : Bottom-up integration

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

SYSTEM TESTING

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

Recovery Testing Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

Stress Testing Stress tests are designed to confront programs with abnormal situations. Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example,

(1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate,

(2) input data rates may be increased by an order of magnitude to determine how input functions will respond,

(3) test cases that require maximum memory or other resources are executed,

(4) test cases that may cause thrashing in a virtual operating system are designed,

(5) test cases that may cause excessive hunting for disk-resident data are created.

A variation of stress testing is a technique called sensitivity testing. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing. Performance Testing Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted.

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. Deployment Testing Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

A variation of stress testing is a technique called sensitivity testing. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

Performance Testing

Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

Deployment Testing

Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

- White Box Testing or Glass Box Testing
  - **Basis Path Testing**
    - Flow Graph Notation
    - Independent Program Paths
    - Deriving Test Cases
    - Graph Matrices
  - **Control Structure Testing**
    - Coding Testing
    - Data Flow Testing
    - Loop Testing

Basis Path Testing in Software Testing

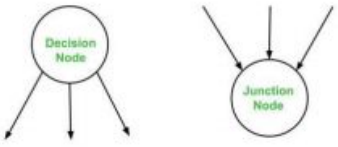
Basis Path Testing is a white-box testing technique based on the control structure of a program or a module. Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing. Therefore, by definition,
Basis path testing is a technique of selecting the paths in the control flow graph, that provide a basis set of execution paths through the program or module.
Since this testing is based on the control structure of the program, it requires complete knowledge of the program's structure. To design test cases using this technique, four steps are followed :

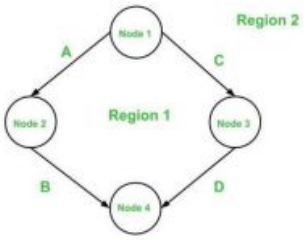
1. Construct the Control Flow Graph
2. Compute the Cyclomatic Complexity of the Graph
3. Identify the Independent Paths
4. Design Test cases from Independent Paths

Let's understand each step one by one.

1. Control Flow Graph –
A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module. A control flow graph (V, E) has V number of nodes/vertices and E number of edges in it. A control graph can also have :

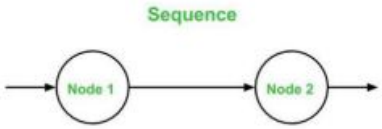
- **Junction Node** – a node with more than one arrow entering it.
- **Decision Node** – a node with more then one arrow leaving it.
- **Region** – area bounded by edges and nodes (area outside the graph is also counted as a region.).



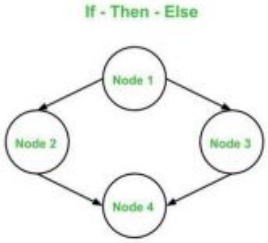


Below are the **notations** used while constructing a flow graph :

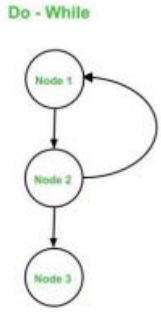
- **Sequential Statements –**



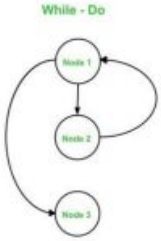
- **If – Then – Else –**



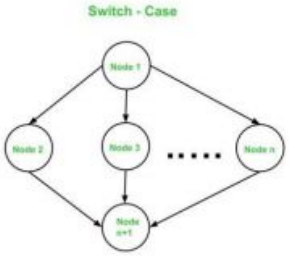
- **Do – While –**



- **While – Do –**



- **Switch – Case –**



Cyclomatic Complexity –
The cyclomatic complexity V(G) is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae :

1. **Formula based on edges and nodes :**

   V(G) = e - n + 2*P

   Where,
   e is number of edges,
   n is number of vertices,
   P is number of connected components.

   For example, consider first graph given above,

   where, e = 4, n = 4 and p = 1

   So,
   Cyclomatic complexity V(G)
   = 4 - 4 + 2 * 1
   = 2

2. **Formula based on Decision Nodes :**

   V(G) = d + P

   where,
   d is number of decision nodes,
   P is number of connected nodes.

For example, consider first graph given above,

where, d = 1 and p = 1
So,
Cyclomatic Complexity V(G)
= 1 + 1
= 2

3. **Formula based on Regions :**

V(G) = number of regions in the graph

For example, consider first graph given above,

Cyclomatic complexity V(G)

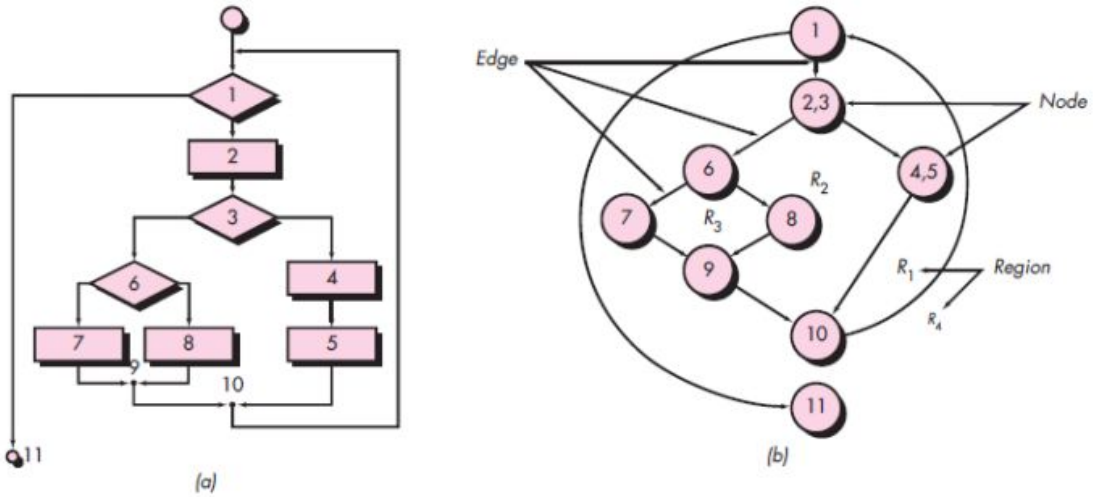
= 1 (for Region 1) + 1 (for Region 2)

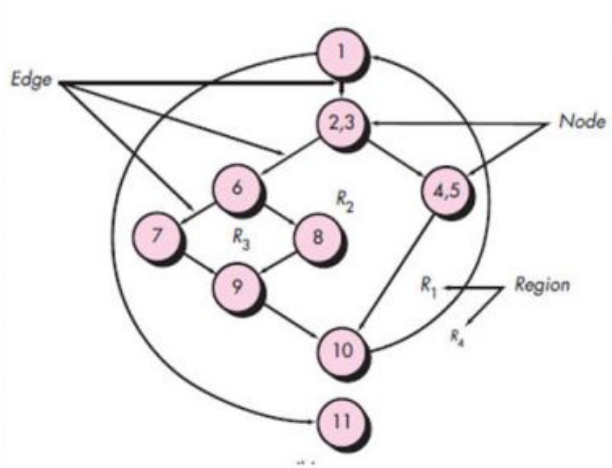
= 2

Hence, using all the three above formulae, the cyclomatic complexity obtained remains same. All these three formulae can be used to compute and verify the cyclomatic complexity of the flow graph.

Example:

(a) Flowchart and (b) flow graph





Independent Program Paths

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

An Independent path is any path through the program that introduces at lest one new set of processing statements or a new condition.

- **Cyclomatic Complexity:**

  is computed one of three ways

  1. The Number of regions of the graph

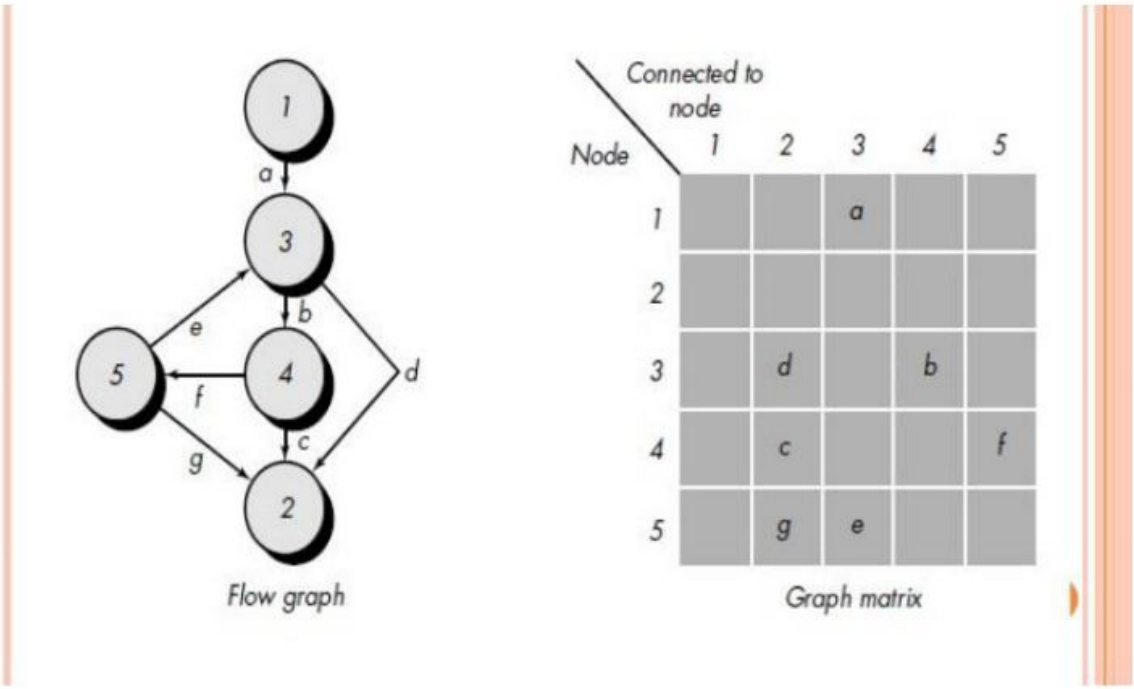
     V(G)=R

  2. Cylomatic complexity V(G) for a flow graph G is defined as

     V(G)=E-N+2

  3. Cylomatic complexity V(G) for a flow graph G is defined as

     V(G)=P+1 where P is the Predicate Node in flow graph



Note –

1. For one function [e.g. Main( ) or Factorial( )], only one flow graph is constructed. If in a program, there are multiple functions, then a separate flow graph is constructed for each one of them. Also, in the cyclomatic complexity formula, the value of 'p' is set depending of the number of graphs present in total.
2. If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula :

```
d = k - 1
```

Here, k is number of arrows leaving the decision node.

Independent Paths :

An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined. The cyclomatic complexity gives the number of independent paths present in a flow graph. This is because the cyclomatic complexity is used as an upper-bound for the number of tests that should be executed in order to make sure that all the statements in the program have been executed at least once.

Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity.

So, the independent paths in above first given graph :

- **Path 1:**

```
A -> B
```

- **Path 2:**
  C -> D

Note –
Independent paths are not unique. In other words, if for a graph the cyclomatic complexity comes out be N, then there is a possibility of obtaining two different sets of paths which are independent in nature.

Design Test Cases :
Finally, after obtaining the independent paths, test cases can be designed where each test case represents one or more independent paths.

Advantages :
Basis Path Testing can be applicable in the following cases:

1. **More Coverage –**
   Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.
2. **Maintenance Testing –**
   When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.
3. **Unit Testing –**
   When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.
4. **Integration Testing –**
   When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.
5. **Testing Effort –**
   Since the basis path testing technique takes into account the complexity of the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

Control Structure Testing

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

1. Condition Testing :

Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect boolean operators, missing parenthesis in a booleans expression, error in relational operators, arithmetic expressions, and so on.

The common types of logical conditions that are tested using condition testing are -

1. A relation expression, like E1 op E2 where 'E1' and 'E2' are arithmetic expressions and 'OP' is an operator.
2. A simple condition like any relational expression preceded by a NOT (~) operator.
   For example, (~E1) where 'E1' is an arithmetic expression and 'a' denotes NOT operator.
3. A compound condition consists of two or more simple conditions, Boolean operator, and parenthesis.
   For example, (E1 & E2) | (E2 & E3) where E1, E2, E3 denote arithmetic expression and '&' and '|' denote AND or OR operators.
1. A Boolean expression consists of operands and a Boolean operator like 'AND', OR, NOT.
   For example, 'A | B' is a Boolean expression where 'A' and 'B' denote operands and | denotes OR operator.

2. Data Flow Testing :

The data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program.

The data flow test approach is depicted as follows suppose each statement in a program is assigned a unique statement number and that the me function cannot modify its parameters or global variables.

For example, with S as its statement number.

DEF (S) = {X | Statement S has a definition of X}

USE (S) = {X | Statement S has a use of X}

If statement S is an if loop statement, then its DEF set is empty and its USE set depends on the state of statement S. The definition of the variable X at statement S is called the line of statement S' if the statement is any way from S to statement S' then there is no other definition of X.

A definition use (DU) chain of variable X has the form [X, S, S'], where S and S' denote statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is line at statement S'.

A simple data flow test approach requires that each DU chain be covered at least once. This approach is known as the DU test approach. The DU testing does not ensure coverage of all branches of a program.

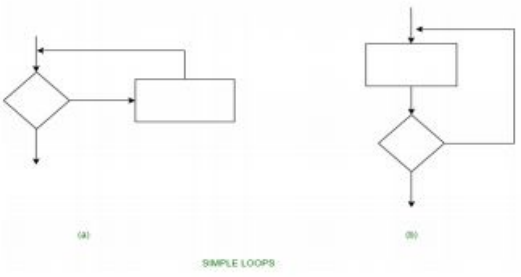
However, a branch is not guaranteed to be covered by DU testing only in rar cases such as then in which the other construct does not have any certainty of any variable in its later part and the other part is not present. Data flow testing strategies are appropriate for choosing test paths of a program containing nested if and loop statements.

3. Loop Testing :

Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction.

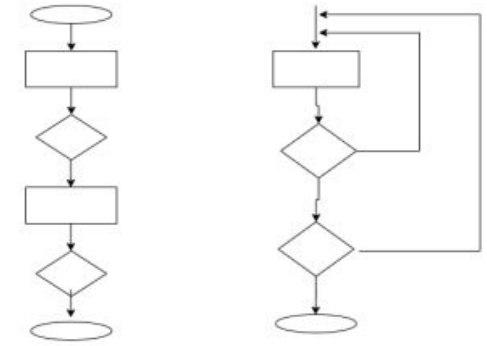
Following are the types of loops.

1. **Simple Loop –** The following set of test can be applied to simple loops, where the maximum allowable number through the loop is n.
    1. Skip the entire loop.
    2. Traverse the loop only once.
    3. Traverse the loop two times.
    4. Make p passes through the loop where p<n.
    5. Traverse the loop n-1, n, n+1 times.



SIMPLE LOOPS

2. **Concatenated Loops –** If loops are not dependent on each other, contact loops can be tested using the approach used in simple loops. if the loops are interdependent, the steps are followed in nested loops.



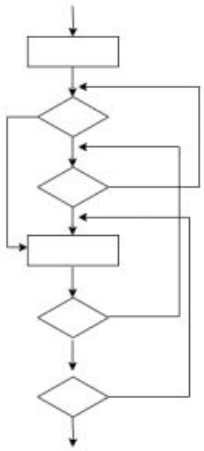
Concatenated Loops

3. **Nested Loops –** Loops within loops are called as nested loops. when testing nested loops, the number of tested increases as level nesting increases.

   The following steps for testing nested loops are as follows -

   1. Start with inner loop. set all other loops to minimum values.
   2. Conduct simple loop testing on inner loop.
   3. Work outwards.
   4. Continue until all loops tested.

4. **Unstructured loops –** This type of loops should be redesigned, whenever possible, to reflect the use of unstructured the structured programming constructs.



Unstructured Loops

Black Box Testing Techniques with Examples

What is Black box testing?

Black box testing refers to a software testing method where the SUT (Software under Test) functionality is tested **without worrying** about its details of **implementation, internal path knowledge** and **internal code structure of the software**.

This method of testing is completely based on the specifications and requirements of the software.

The focus of the black box testing is upon the **output** and **inputs** of the software system rather than the program's internal knowledge.

The system that undergoes this type of testing is considered as the "black box", and it can be any software like a database, website or an Operating System.

What Is The Purpose Of Black Box Testing?

Black box testing focuses on testing the **complete functionality of the system** as well as **its behavior**.
This testing method is also referred to as **behavioral testing and functional testing**.
This testing method is critical during the stages of software testing life cycle like regression testing, acceptance, unit, system, integration and software development.

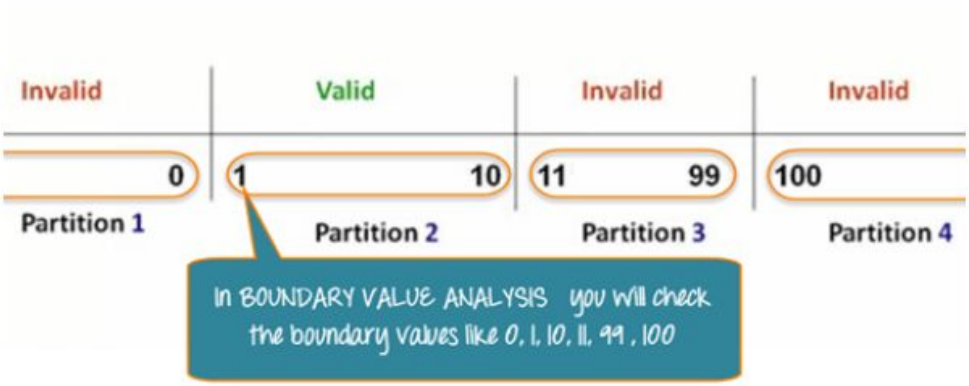
The techniques of Black box testing are beneficial for the end users who wish to perform software verification.

Techniques of Black Box Testing

1. **BVA or Boundary Value Analysis**
2. State Transition Testing
3. Decision Table Testing
4. Graph-Based Testing
5. Error Guessing Technique

The following are the techniques employed while using Black box testing for a software application.

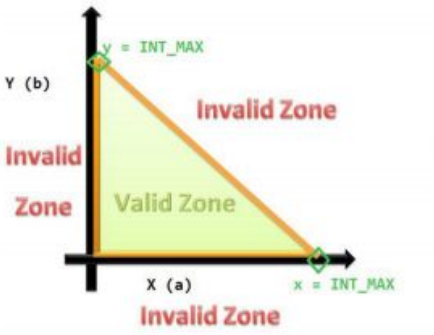
BVA or Boundary Value Analysis:



It is one among the useful and critical Black box testing technique that helps in equivalence partitioning. BVA helps in testing any software having a boundary or extreme values.

This technique is capable of identifying the flaws of the limits of the input values rather than focusing on the range of input value. Boundary Value Analysis also deals with edge or extreme output values.

Equivalence Class Partitioning:

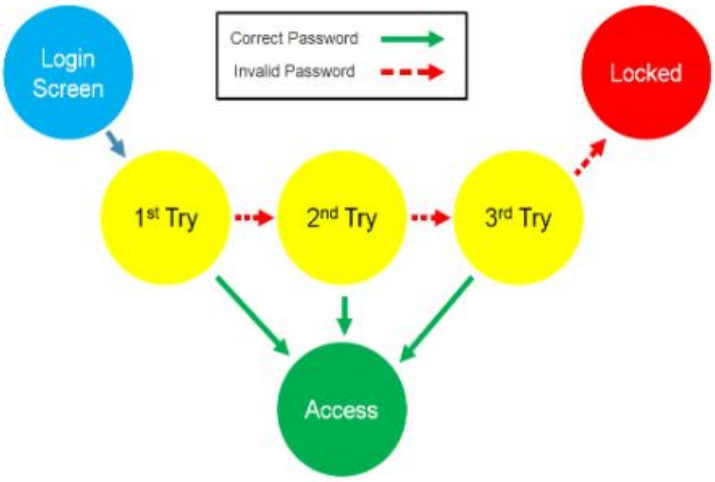


This technique of Black box testing is widely used to write test cases. It can be useful in reducing a broad set of possible inputs to smaller but effective ones.

It is performed through the division of inputs as classes, and each class is given a value.

It is applied when the need for exhaustive testing arises and for resisting the redundancy of inputs.

State Transition Testing



This technique usually considers the state, outputs, and inputs of a system during a specific period.

Based on the type of software that is tested, it **checks for the behavioral changes of a system** in a particular state or another state while maintaining the same inputs.

The test cases for this technique are created by **checking the sequence of transitions** and **state or events among the inputs.**

The whole set of test cases will have the traversal of the expected output values and all states.

Decision Table Testing:

What is Decision Table Testing?

Decision table testing is a software testing technique used to test **system behavior for different input combinations**. This is a systematic approach where the **different input combinations** and their **corresponding system behavior (Output)** are captured in a **tabular form**.

Let's learn with an example.

Example 1: How to make Decision Base Table for Login Screen

Let's create a decision table for a login screen.

The condition is simple if the user provides **correct username and password** the user will be **redirected to the homepage**. If any of the input is wrong, an error message will be displayed.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

- T – Correct username/password
- F – Wrong username/password
- E – Error message is displayed
- H – Home screen is displayed

Interpretation:

- Case 1 – Username and password both were wrong. The user is shown an error message.
- Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
- Case 3 – Username was wrong, but the password was correct. The user is shown an error message.

- Case 4 – Username and password both were correct, and the user navigated to homepage

Why Decision Table Testing is Important?

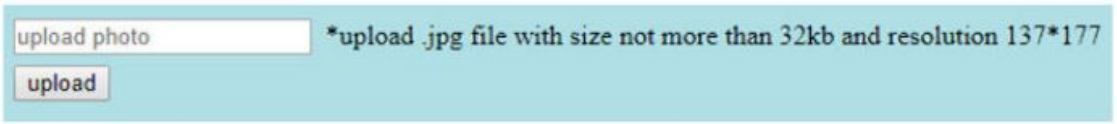
Decision Table Testing is Important because it helps to test different combinations of conditions and provide better test coverage for complex business logic. When testing the behavior of a large set of inputs where system behaviour differs with each set of input, decision table testing provides good coverage and the representation is simple so it is easy to interpret and use.

Example 2: How to make Decision Table for Upload Screen

Now consider a dialogue box which will ask the user to upload photo with certain conditions like –

1. You can upload only '.jpg' format image
2. file size less than 32kb
3. resolution 137*177.

If any of the conditions fails the system will throw corresponding error message stating the issue and if all conditions are met photo will be updated successfully



Let's create the decision table for this case.

Conditions	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8
Format	.jpg	.jpg	.jpg	.jpg	Not .jpg	Not .jpg	Not .jpg	Not .jpg
Size	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb	Less than 32kb	Less than 32kb	>= 32kb	>= 32kb
resolution	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177	137*177	Not 137*177
Output	Photo uploaded	Error message resolution mismatch	Error message size mismatch	Error message size and resolution mismatch	Error message for format mismatch	Error message format and resolution mismatch	Error message for format and size mismatch	Error message for format, size, and resolution mismatch

For this condition, we can create 8 different test cases and ensure complete coverage based on the above table.

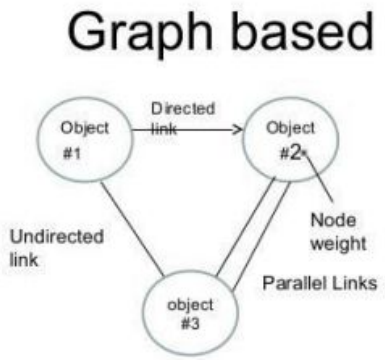
1. Upload a photo with format '.jpg', size less than 32kb and resolution 137*177 and click on upload. Expected result is Photo should upload successfully
2. Upload a photo with format '.jpg', size less than 32kb and resolution not 137*177 and click on upload. Expected result is Error message resolution mismatch should be displayed
3. Upload a photo with format '.jpg', size more than 32kb and resolution 137*177 and click on upload. Expected result is Error message size mismatch should be displayed
4. Upload a photo with format '.jpg', size more than equal to 32kb and resolution not 137*177 and click on upload. Expected result is Error message size and resolution mismatch should be displayed
5. Upload a photo with format other than '.jpg', size less than 32kb and resolution 137*177 and click on upload. Expected result is Error message for format mismatch should be displayed
6. Upload a photo with format other than '.jpg', size less than 32kb and resolution not 137*177 and click on upload. Expected result is Error message format and resolution mismatch should be displayed
7. Upload a photo with format other than '.jpg', size more than 32kb and resolution 137*177 and click on upload. Expected result is Error message for format and size mismatch should be displayed
8. Upload a photo with format other than '.jpg', size more than 32kb and resolution not 137*177 and click on upload. Expected result is Error message for format, size and resolution mismatch should be displayed

In some instances, the inputs combinations can become very complicated for tracking several possibilities.

Such complex situations rely on decision tables, as it offers the testers an organized view about the inputs combination and the expected output.

This technique is identical to the graph-based testing technique; the major difference is using tables instead of diagrams or graphs.

Graph-Based Testing:



This technique of Black box testing involves a graph drawing that depicts the link between the causes (inputs) and the effects (output), which trigger the effects.

This testing utilizes different combinations of output and inputs. It is a helpful technique to understand the software's functional performance, as it visualizes the flow of inputs and outputs in a lively fashion.

Error Guessing Technique:

This testing technique is capable of guessing the erroneous output and inputs to help the tester fix it easily. It is completely based on judgment and perception of the earlier end user experience.

Examples of Black Box Testing

The example given below throws light on how the techniques of this testing can be used to test the specific software with given inputs

While considering a shopping scenario,

- Shop for $500 and receive a discount of 5%
- Shop for $1000 and receive a discount of 7%
- Shop for $1500 or more and receive a discount of 10%

With the help of Equivalence partitioning technique of this testing, it is possible to divide inputs as four partitions, amount less than 0, 0 – 500, 501 – 1000, 1001 – 1500 and so on. The details such as the maximum limit for shopping and the product details will not be considered by this testing technique.

When boundary value is added to the partitions, the boundary values will be 0, 500, 501, 1000, 1001 and 1500. With the BVA technique, the lower and upper values are usually tested, so values like -1, 1 and 499 will be included. Such values will help in explaining the behavior of the input values in software.

According to State Transition Testing technique of Black box testing, when a shopper shops above $1500 two times in a month, their status gets changed from Gold to Platinum, and if he does not shop for the next 2 months, the status gets back to Gold. Using further test cases, it is possible for the tester to such complex track.

Elements of Software Quality Assurance

There are 10 essential elements of SQA which are enlisted below for your reference:

1. Software engineering Standards
2. Technical reviews and audits
3. Software Testing for quality control
4. Error collection and analysis
5. Change management
6. Educational programs
7. Vendor management
8. Security management
9. Safety
10. Risk management

Software Quality Assurance Standards

In general, SQA may demand conformance to one or more standards.

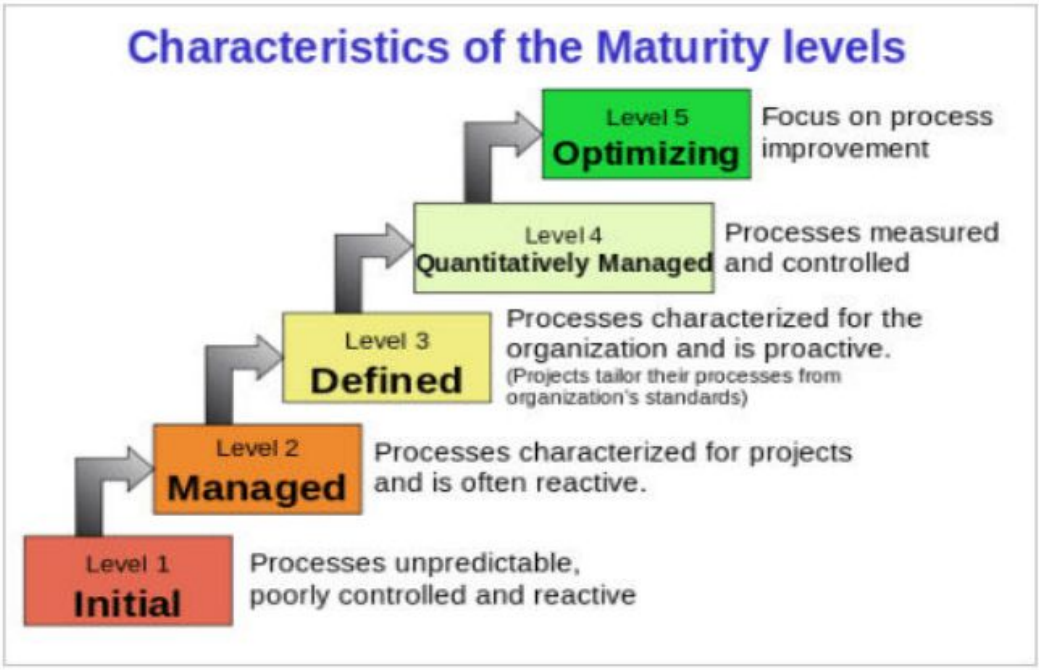
Some of the most popular standards are discussed below:

ISO 9000: This standard is based on seven quality management principles which help the organizations to ensure that their products or services are aligned with the customer needs'.

7 principles of ISO 9000 are depicted in the below image:

CMMI level: CMMI stands for **Capability maturity model Integration**. This model was originated in software engineering. It can be employed to direct process improvement throughout a project, department, or an entire organization.

5 CMMI levels and their characteristics are described in the below image:



An organization is appraised and awarded a maturity level rating (1-5) based on the type of appraisal.