

SE

Created by	B Borhan
Last edited time	@June 16, 2025 11:55 AM
Tag	Year 3 Term 2

Resources

- Slide from ma'am (https://github.com/borhan008/academic_files/tree/main/3-2/5.%20Software%20Engineering%20and%20Information%20System%20Design/Slides)
- Atkia apu's note
- Mahir Labib Dihan sir's note, (https://drive.google.com/file/d/1d_XuqXV4zqYAv8yN8puBCQNII_zcw1jl/view)
- Diagrams
 - <https://www.studocu.com/en-us/document/fairleigh-dickinsonuniversity/digital-marketing-strategies/dfd-case-study-with-solutionsupdated/77995749>
 - <https://sadamutmspace.blogspot.com/2015/06/meet-4-exercise-on-dfd-read-casebelow.html>

Software Engineering is an engineering branch associated with development of software product using well-defined scientific principles method and procedure.

Aspects	Software Engineering	System Engineering
Definition	Software Engineering is an engineering branch associated with development of software product using well-defined scientific principles method and procedure.	System engineering is an interdisciplinary field that focuses on integrating and managing complex system.
Scope	Focuses on software components	Holistic approach to entire system
Objective	To develop, maintain and improve software product	To ensure system functionality and integrating various parts
Method	Scrum, Agile, Devops	MBSE
Deliverable	Software Application	Fully functional system
Application	IT and Software Industry	Broad application in aerospace, automobile
Stakeholder Interaction	Software users, developers and IT Personals	A broader range of stakeholders from various fields

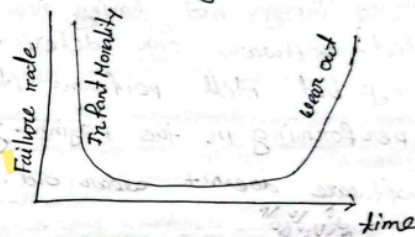
Aspect	Software	Program
Dependency	Software is mainly dependent on the <u>operating system</u> .	Programs are mainly dependent on the compiler.
Categories	Various software categories include application software, system software, computer programming tools, etc.	There are no such categories of programs.
Size	The size of software generally ranges from megabytes (Mb) to gigabytes (Gb).	The program size generally ranges from kilobytes (Kb) to megabytes (Mb).
Developer Expertise	Software is usually developed by people having expert knowledge and experience as well as are trained in developing software and	Programs are usually developed by the person who is a beginner and has no prior experience.

Aspect	Software	Program
	are also referred to as software developers.	
Nature	Software's can be a program that generally runs on computer.	Programs cannot be a software.
Necessity for Computer Functionality	If software's are not present in computers, then computer is useless.	If programs are not present in computer, then also computer can function well because of operating system.
Download	Software's can be downloaded on computer using internet without any need of program.	Program cannot run on computer without any software present in computer.
Features	Features of software includes security, safety, dependability, correctness, etc.	Features of program includes reliable, cost effectiveness, maintainability, profitability, etc.
Development Time	It requires more time to create software than program.	It requires less time to create program than software.
Examples	Examples of software includes Adobe Photoshop, Google Chrome, PowerPoint, Adobe Reader, etc.	Examples of program includes Web browsers, word processors, video games, etc.

~~Q~~ What do you mean by wear-out in case of software?

= When something is no longer of any use, it reaches the 'wear-out' state. That is, it cannot perform the function it was built for. For example a printer reaches wear out state it can't print any more but in case of software "it doesn't wear out" this is the unique characteristics of softwares compared to hardware

hardware can be wear out for excessive temperature, dust, vibration, improper use and so on. With the time failure of the hardware mining and hardware just stop functioning.



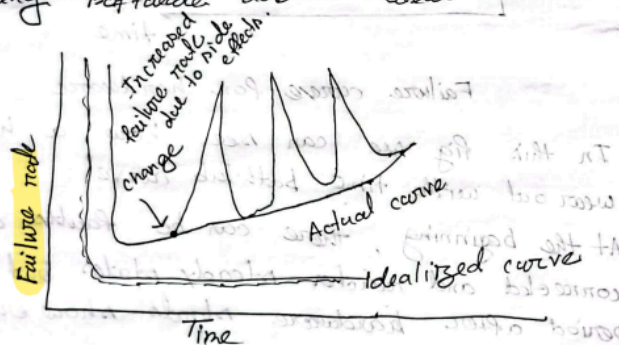
Failure curve for hardware

In this fig we can see how a hardware wear out with time bathtub curve

At the beginning, there can be faults and gets corrected and reaches steady states but then certain period after hardware starts show error and wear out.

Software doesn't wear out. Software also shows high failure rate at its infant state, then it gets modifications and the defects get corrected and software updated, thus it comes to the idealized state. This state continues.

However, after fulfilling one demand another one will rise. And in this way, alternative software will implement of current user demand replace a software. Though not having new features is not a defect. Software can deteriorate due to change but still performs its operation as it was performing in the beginning. That's why software doesn't wear out.



Failure curve for software

Software product - 2 type

1) Generic products: It means it is designed in general for public use. A software that isn't designed for a specific purpose or person.

Example: PC Software such as graphics programs, project management tools.

2) Customized products

Software that is designed for specific purpose or customer to meet their own needs.

Example: embedded control systems, air traffic control software etc.

What is legacy software?

Legacy software implies that the software is out of date or in need of replacement. However, it may be in good working order so that business or individual owner doesn't want to upgrade the software.

Legacy Software must be changed for

i) adapted to meet the needs of new computing environments or technology

ii) enhanced to implement new business requirements.

iii) re-architected to make it durable.

iv) extended to make it interoperable.

① Difference between open world computing and open source?	
Open source distribution	Open world computing
① Open source refers to a type of software program whose source code is made available to the users or developer.	① Open world computing is a concept that involves creating computer systems and software environment that are not tightly controlled or restricted.

② characteristics: Any one can view, modify and distribute the code.	② characteristics It is designed to adapt to changing requirements and scenarios.
③ open source specifically refers to the accessibility and openness of source code.	③ open world computing is a general term that indicates computing environments flexibility and adaptability.
④ It relates to the collaborative development and distribution of software.	④ It deals with handle uncertainty in dynamic environment.
⑤ Example: Linux O.S, Mozilla Firefox browser, python Programming Language	⑤ Example (hypothetical) Artificial intelligence system, certain types of robotics that need to operate unpredictable environment, autonomous vehicles.

Importance of Software engineering

Software engineering is the branch of computer science that deals with the design, development, testing and maintenance of software application.

Importance

- (1) Reliability: Software engineering ensures that the software is reliable and operates properly under various condition.
- (2) Efficiency: It helps optimize software for performance and energy efficiency. Efficient software can run on a wide range of hardware.
- (3) Maintainability: Well structure code is easier to maintain. S.E allows developers to make updates, fix bugs and add new features.
- (4) Cost effectiveness: Proper SE practices help manage project with minimum costs.
- (5) Customer satisfaction: SE focuses on user require and expectation. Satisfied customers are more likely

to continue using Software products.

- (6) Global collaboration: It increases global collaboration and code sharing.
- (7) Handling big project: Some companies use S.E methods to handle large project without any issues.
- (8) Decrease time: Using SE decrease a lot of time.

⑧ Software engineering a layered technology-driven
 = S.E is fully layered technology, to develop software
 we need to go from one to another layer. All
 layers are connected to each other and each layer
 demands the fulfillment of the previous layer.

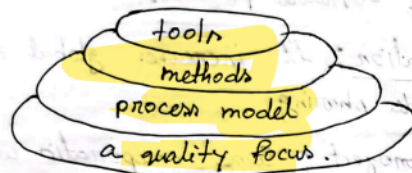


Fig: S.E engineering layer

✓ 1) A quality focus:

It defines the continuous process improvement principles of software. It provides security to the software so that data can be accessed by only an authorized person, no outsider can access data. It also focus maintainability and usability.

✓ 2) process model:

It is the foundation or base layer of SE. It binds all layer together which enables the development of software before the deadline. process model covers all the activities, action required to be carried out for software development.

✓ 3) Methods: It has the information of all the tasks which includes communication, requirement analysis, design, testing, construction, modeling.

✓ 4) Tools: Tools are integrated means information created by one tool can be used by another.

Software process: A framework for the activities, actions and tasks that are required to build high quality software.

✓ A process is a collection of activities, actions and tasks that are performed when some work product is to be created.

Activity: involves development of the software modifying an existing system (communication with stakeholders)

Action: An Action is a set of tasks that produce a major work product. (architectural design)

task:

A task focuses on a small but well-defined objective. (conducting a unit test)

A software process framework is a collection of task sets.

Common Activities in software processes

There are many software processes but all involve:

- (i) Specification: defining what the system should do
- (ii) Design & implementation: defining the organisation of the system and implementing the system
- (iii) Validation: checking that it does what the customer wants.
- (iv) Evolution: changing the system according to customers requirements.

Generic Process model activities

Activity	What Project Team does?
Communication	Communicates with client
Planning	Schedules the project development
	Estimates the project cost
Modeling	Analyzes the client requirements
	Designs algorithm and flowchart of different software components
Construction	Implements and tests different software components
Deployment	Delivers the software product to client
	Takes feedback to improve the quality of the product if required

Framework activities (common process should have these 5 phase)

Framework activities refer to the fundamental tasks and processes that are commonly applicable throughout the software development life cycle (SDLC). Activities are:

- 1) Communication: It is the first and foremost thing for the development of software. Communication is necessary to know the actual demand of the client. It includes customers & stakeholders.
- 2) Planning: It basically means drawing a map for reduced the complication of development. It includes work plan, describes technical risks, list of resource requirement etc.
- 3) Modeling: In this process, a model is created according to the client for better understanding. The software model is prepared by:
i) Analysis of requirements & ii) Design.
- 4) Construction: It includes the coding and testing of the problem. It also include fixing bugs and confirming.

5) Deployment: It includes the delivery of software to the client for evaluation and feedback. (complete or non complete). On the basis of their feedback, we modify the product for the supply of better product.

Umbrella Activities

Umbrella activities are that take place during a software development process for improved project management and tracking. These activities often run in parallel with other specific development tasks and are essential for the success of the project. They serve as a kind of umbrella because they cover and coordinate a broad range of activities.

i) Software project tracking and control:

allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

ii) Formal technical reviews:

examines SE work products to uncover and remove errors before they are propagated to next activity. At each level of the process errors are evaluated and fixed.

III) Software quality assurance:

Perform actions to ensure the product's quality.

IV) Software configuration management

Managing of configuration process when any change in the software occurs.

V) Work product preparation and production

The activities to create models, documents, logs, forms and lists are covered under

VI) Reusability management:

It defines criteria for work product reuse. Reusable work items should be backed up, reusable software components should be archived.

VII) Measurement

defines and collects process, project and product measures that assist the team in delivering software that meets stakeholders' needs.

VIII) Risk management:

The risks that may have an effect on project outcome or quality can be analyzed.

HOOKER'S General principles (V.V.E)

① The Reason It All exists: to provide value to its user

② Keep it simple, stupid!: All design should be as simple as possible, but no simpler.

③ Maintain the vision: A clear vision is essential to the success of a software project.

④ What you produce, Others will consume:

always specify, design and implement knowing someone else will have to understand what you are doing.

⑤ Be Open to the Future:

Never design yourself into a corner!

⑥ Plan ahead for Reuse: Planning ahead

for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

⑦ Think!: Placing clear, complete thought before action almost always produces better results.

1. Professional Ethics

a. Code of Ethics: PCP JM PCS

- i. **Public:** Software engineers shall act consistently with the public interest.
- ii. **Client and Employer:** Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
- iii. **Product:** Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
- iv. **Judgement:** Software engineers shall maintain integrity and independence in their professional judgment.
- v. **Management:** Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
- vi. **Profession:** Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
- vii. **Colleagues:** Software engineers shall be fair to and supportive of their colleagues.
- viii. **Self:** Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Software Process model:

A software process model is an abstract representation of a process.

It presents a description of a process from some particular perspective. It describes the sequence of different phases of entire life time of a product from initial to the final state. It some times calls product life cycle.

✓ Process flow:

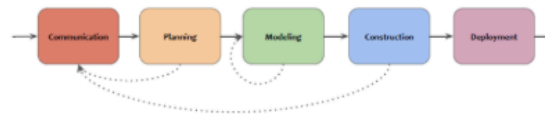
A process flow is a series of steps that S.E and development team go through to achieve ~~goal~~ goal.

iii. 4 unique process flows

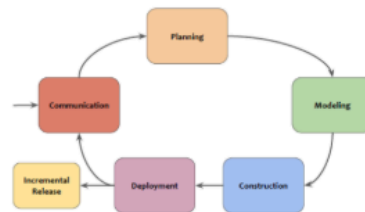
1. **Linear Process Flow:** Linear process flow executes each of the 5 (five) activities sequentially.



2. **Iterative Process Flow:** Iterative process flow repeats one or more activities before proceeding to the next activity.



3. **Evolutionary Process Flow:** Evolutionary process flow executes activities in a circular fashion. Each cycle leads to a more complete version of the software.



4. **Parallel Process Flow:** Parallel process flow executes one or more activities in parallel with other activities.



Linear process flow	Iterative process flow
① A linear process flow executes each of the five framework activities in sequence.	① An iterative process flow repeats one or more of the activities before proceeding to the next activity.
② Each phase must be complete before moving to the next phase in order.	② Doesn't follow order.
③ Not flexible at all.	③ Much flexible.
④ Changes & revise is difficult.	④ easy.
⑤ used when requirements are fixed.	⑤ Not fixed.
⑥ Waterfall model.	⑥ Scrum.

process pattern: (V.I.J)
 A process pattern describes a process-related problem that is encountered during S.E work. It can be used to describe a problem and solution associated with framework activities in some situations.

Types of process pattern:

1. **Task Pattern** : Problems associated with a software engineering **action or work task and relevant to successful SE proactive** are defined by task pattern. Example: Requirement gathering.
2. **Stage pattern**: problems associated with a **framework activity** for a process are described by stage pattern. Example : Establishing communication
3. **Phase Pattern**: Phase pattern defines **the sequence of framework activities** that occur with the process, even the overall of activities are iterative in nature. Example : Spiral Model or Prototyping

Template for describing process pattern

1. Pattern name: meaningful
2. Intent : Objectives
3. Pattern type:
4. Initial Context : pre-requisite or conditions
5. Problems :
6. Solution:
7. Resulting context
8. Related pattern
9. Known uses & example

Example from GFG:

Pattern Name: Prototyping Model Design

Intent: Requirements are not clear. So aim is to make an model iteratively to solidify the exact requirements.

Type: Phase Pattern

Initial Context: Before going to the prototyping these basic conditions should be made

1. Stakeholder has some idea about their requirements i.e. what they exactly want
2. Communication medium should be established between stakeholder and software development team to ensure proper understanding about the requirements and future product
3. Initial understanding about other factors of project like scope of project, duration of project, budget of project etc.

Problem: Identifying and Solidifying the hazy and nonexistent requirements.

Solution: A description of the prototyping should be presented.

Resulting Context: A prototype model which can give a clear idea about the actual product and that needs to be agreed by stakeholder.

Related Patterns: Requirement extraction, Iterative design, customer communication, Iterative development, Customer assessment etc.

Known Uses & Examples: When stakeholder requirements are unclear and uncertain, prototyping is recommended.

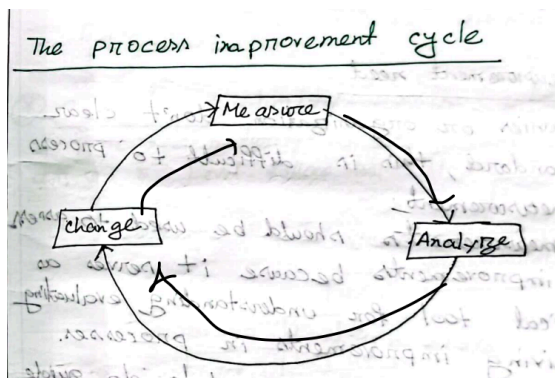
Process improvement means understanding existing processes and changing these processes to increase product quality on/and reduce costs and development time.

Two types of process improvement:

(i) Process maturity approach:

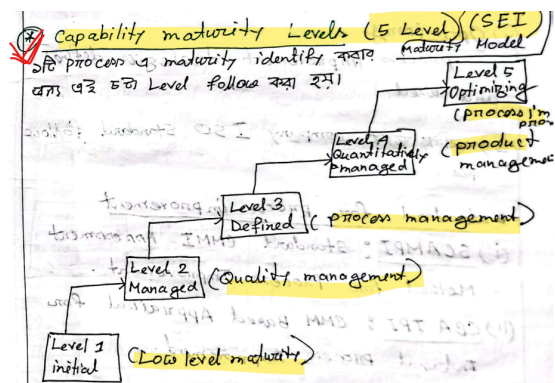
प्रत्येक company का अपना standard है।
यह बिना कहे improve नहीं हो सकता।
The process maturity approach which focuses on improving process and project management and introducing good software engineering practice.

(ii) Agile approach: flexible



Process Improvement Cycle:

1. Measure - Collect data on the current process to establish a baseline.
2. Analyze - Study the collected data to identify performance issues and improvement opportunities.
3. Change - Implement modifications to the process based on analysis findings.



** The sei capability maturity model :-




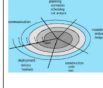
- ① level-1 : initial
 ⇒ essentially uncontrolled. (processes are not controlled)
- ② repeatable :
 product management procedures defined & used.
- ③ defined :
 process management procedures & strategies defined & used.
- ④ managed :
 quality management strategies defined & used.
- ⑤ optimizing :
 process improvement strategies defined & used.

Software Development Life Cycle (SDLC) consists of the following **seven stages**:

1. **Planning**: Identify the problem, assess the need for a new or alternative system, set project scope, and create a detailed project plan.
2. **Requirement Analysis**: Gather and prioritize business requirements through collaboration between users and IT specialists.
3. **Design**: Create technical blueprints including architecture and system models to outline how the system will function.
4. **Implementation**: Convert design documents into actual code by building the architecture, database, and programs.
5. **Integration & Testing**: Ensure the system meets business requirements through defined test conditions and system testing.
6. **Deployment**: Distribute the system to users, provide training, and implement using methods like parallel, plunge, pilot, or phased approach.
7. **Maintenance**: Provide ongoing support, fix bugs (corrective), adapt to changes (adaptive), and improve performance (perfective).

	Waterfall	Incremental	Prototyping	Spiral
Definition	Different activities are executed in a sequential and systematic manner.	Involves development and deployment of a series of versions of the software product, known as increments.	A prototype is built, tested and then reworked as necessary until an acceptable outcome is achieved from which the complete system or product can be developed.	Development of the software product through a series of versions of that product. Deals with the uncertainties in software project by incorporating different risk analysis techniques throughout the process.
Project size	Small	More or less fixed and clear	Vague and likely to change	
Requirements	Well-defined and known before the starting of the project.			
Phase 1	The process begins with communication, where requirements are collected from the client and documented.	A simple functioning system, known as core product, which handles basic requirements is first developed and delivered.	One or more prototypes of the software product are built with currently known client requirements before the development of final product.	The first loop may result in the development of a basic prototype of the final product.
Phase 2	In planning phase, the time and financial constraints of the project are estimated, resulting in a schedule and a budget.	Client feedback is collected after each incremental delivery to incorporate in the next increment	The client evaluates the prototype and provides feedback as well as additional requirements which get incorporated in next prototype.	The subsequent loops may result in the gradual development of more mature versions of the product.
Phase 3	Then, a design of the software product is crafted in modeling phase based on gathered requirements and keeping project constraints in mind.	Multiple increments are delivered by adding functionalities, as per the client requirements, until the final version of the product is released.	This workflow is repeated until the prototype evolves into a complete product, acceptable to the client.	This spiral continues until an acceptable software product is built and delivered to the client.
Phase 4	In construction phase, essential code is generated and tested to build the final product.			

Phase 4	In construction phase, essential code is generated and tested to build the final product.			
Phase 5	Finally, in deployment phase, the product is delivered to the client and necessary maintenance is provided based on the client feedback.			
Advantage	Simple model to use and implement.	Flexible to changing requirements.	Promotes active involvement of the client	Rigorously tackles risks associated with a project

	Easily understandable workflow	Modifications can be made throughout the process		Any type of changes can be incorporated even at a later stage of the process
	Easily manageable as requirements are known before the starting of the project	Errors are mitigated as the product is assessed by the client after each incremental delivery	Client feedback helps to better understand the product and facilitates early detection of error in the product	
		Functioning software product is available at the early stage of the process	Detailed client requirements are not needed to start the project	
		Product can easily be tested because of multiple iterations		
		The initial project cost is lower		
Disadvantage	It may become tough for the client to provide all the requirements beforehand.	Breaking the problem into increments is difficult	Prototyping can slow down the process	Process gets costly and complicated
	Testing and client evaluation are carried out in the last phases resulting in high risk	Total project cost is high	Frequent changes may increase complexity of the system	Requires risk assessment expertise
	Iteration of activities is not promoted which may be crucial for some projects.	A complete planning of the project is required before committing	Client dissatisfaction may lead to scrapping of multiple prototypes	
		Refining requirements in each iteration may affect the software architecture		
				

Phases of the V-Model

Left Side (Development):

1. Requirements Modeling – Understand what the system should do
2. Architectural Design – High-level system design
3. Component Design – Detailed design of each module
4. Code Generation – Actual coding of components

Right Side (Testing):

1. Unit Testing – Test individual components
2. Integration Testing – Test interactions between modules
3. System Testing – Test the complete system against design
4. Acceptance Testing – Validate system against user requirements

Problems with Prototyping (Short):

sem ques

1. Unclear Requirements – Users may keep changing their minds.
2. False Expectations / User Confusion – Users may think the prototype is the final system and misunderstand its purpose.
3. Too Much Focus on UI – Backend logic may be ignored.
4. Delays – Frequent changes can slow down progress.
5. Higher Cost – Repeated updates can be expensive.
6. Not for All Projects – Bad fit for complex or secure systems.

Explain how both the waterfall model of the software process and the prototyping model can be accommodated in the spiral process model. (5)

the spiral process model accommodates the waterfall model by breaking it into iterative phases and accommodates the prototyping model by allowing for the creation and refinement of prototypes in each iteration.

The waterfall model can be accommodated in the spiral process model by having each iteration represent a phase of the waterfall model. For example, the first iteration could focus on requirements gathering and analysis, the second on design, the third on implementation, and so on.

Similarly, the prototyping model can be accommodated by using the spiral model's iterative nature to create and refine prototypes in each iteration, incorporating user feedback and refining the prototype until it evolves into the final product.

Steps

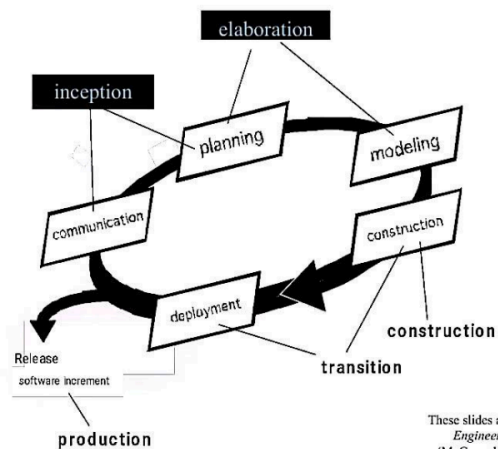
- i. The first loop may result in the development of a basic prototype of the final product.
- ii. The subsequent loops may result in the gradual development of more mature versions of the product.
- iii. This spiral continues until an acceptable software product is built and delivered to the client.

Unified Process: A unified process is a software development process that uses the UML language to represent models of the software system to be developed. It is use-driven, architecture-centric, iterative and incremental. software process.

Unified Modeling Language (UML)

Why Unified Process is said to be process

- 1) Software projects are largely complex
- 2) timely delivered (must)
- 3) many phase



These slides are design
Engineering: A
(McGraw-Hill, 200

Short Summary of Phases:

1. Inception

- Focus: Communication & Planning
- Define project scope using use-case model.
- Identify customer needs, estimate cost/time.
- Create project plan, goals, risks, and description.
- Reviewed against milestone; may cancel/redesign if failed.

2. Elaboration

- Focus: Planning & Modeling
- Detailed evaluation and risk reduction.
- Refine use-case model (~80%), update business case and risks.
- Milestone check again; may cancel/redesign.
- Build executable architecture baseline. ↓

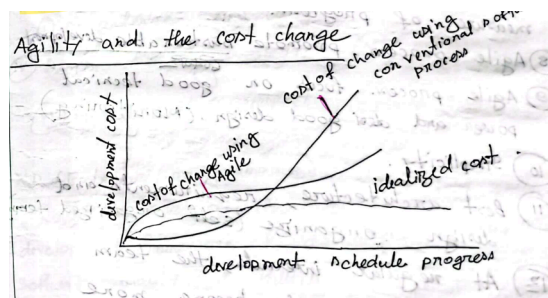
3. Construction

- Focus: Development & Testing
- Build full system/code.
- Perform testing and coding.

4. Transition

- Focus: Deployment
- Release to users/public.
- Move project to production.
- Update docs, conduct beta testing.
- Fix issues based on feedback.

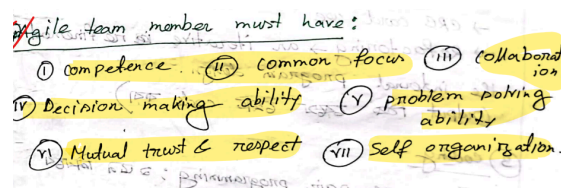
Agility refers to the ability of a software development team or organization to respond quickly and effectively to changing requirements. It also refers to effective communication among all stakeholders.



Principle: Some changes frequently bring business motivation for working sustainably through simple self-reflection

1. Customer Satisfaction

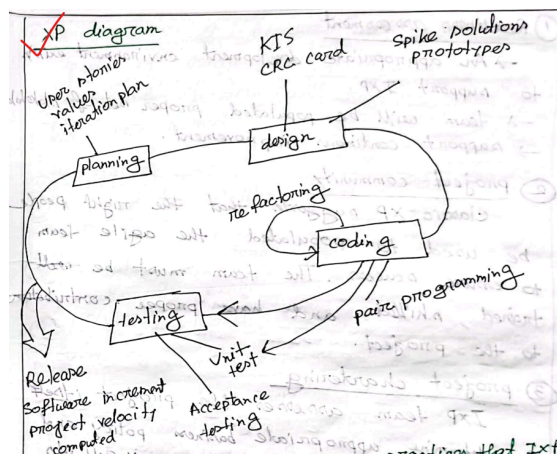
2. Welcome changing requirement
3. frequently deliverable
4. business and developer together
5. supportive and motivative environment
6. face to face conversation
7. working software is the progress
8. sustainable development
9. technical power and good design
10. simplicity
11. self organized team, best architecture, design
12. the team reflects how to become more effective



- i. Extreme Programming
1. Whatever the beneficial software engineering practices are, they should be taken to extreme levels, from day one.
 2. Promotes pair programming.
 3. Advantages
 - a. XP encourages simple code which allows modification at any given time.
 - b. This process model promotes testing codes from day one, resulting in more agile software development.
 - c. XP maintains an energizing and uplifting environment for developers within a project team.
 4. Disadvantage
 - a. The extreme focus on coding can lead to neglecting design, resulting in degradation of software product quality.
 - b. Lack of documentation and monitoring may lead to repetition of similar error in the future.

Basic activity of XP programming

1. XP planning : user-stories, cost and create a release plan, delivery date, progress velocity after first increment
2. XP Design : KISS, CRC card, refactoring
3. Coding : Pair programming, unit test
4. Testing : Unit test, Acceptance test



Industrial XP :

- Organic evolution of XP

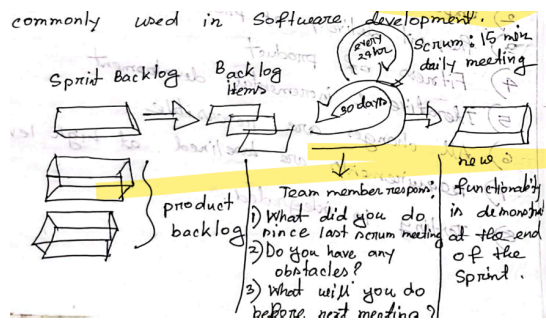
- six practices on XP project works successfully for significant project
 - Readiness assesment : development environment, skill team, support
 - Project community : right people, well-trained, proper contribution
 - Project chartering: business policy ensures
 - Test driven management : measure criteria/metrics to progree
 - Retrospective : specialized technical review
 - Continuous learning

XP	IXP
① <u>small</u>	① <u>large</u>
② <u>limited small & medium organization</u>	② <u>large organization</u>
③ <u>have complete procedure and tool.</u>	③ <u>doesn't have</u>

SCRUM

- Scrum prioritizes simplicity in a project and develops product in a gradual manner with frequent delivery.
- Sprint: Planning → Implementation → Product review → Process review
- The Daily Scrum is a short, daily meeting where team members discuss progress and plan their work for the next 24 hours.
- Advantages
 - less authority and hierarchy
 - adaptability
 - less time to become deliverable
 - more client involvement
- Disadvantages
 - small project team
- 3 Questions
 - What did you do since last scrum meeting ?
 - do you have any obstacles?
 - what will you do before next meeting ?
- Product backlog : list of features collected from customers
- Sprint backlog : priorised backlogs
- backlog item : sorted according to priority
- meeting : 30 days 15 mins

commonly used in Software development.



Scrum	XP
① Scrum team work 2 weeks to 1 month.	① XP team work 1-2 weeks only
② don't allow changes in their timeline.	② allow changes in timeline
③ It doesn't put attention software engineering practice that developer should do.	③ XP do that developers should use tool to ensure better outcome
④ tasks are prioritized by the owner of the product but with flexibility.	④ tasks are prioritized by customer strictly.
⑤ customer involvement in less.	⑤ more

Waterfall Model	V-Model
① Waterfall Model is a sequential development model flows like waterfall	① V-shaped project development model
② testing occur in only construction phase	② Testing occur in every phase
③ phase occur with communication, planning, Modeling construction, deployment.	③ Requirement modeling, Architectural design, component design, code generation
④ Limited opportunity to feedback.	④ early feedback possible
⑤ Higher risk.	⑤ Early identification of risk

prototyping	Spiral
① Rough design	① Well planned design
② Involve the creation of prototype based on feedback.	② Involve repeated cycle of planning testing - ...
③ client involvement greater.	③ depend on the phase of development
④ well suited for unclear requirement project	④ indefinite continuous updated project, high level complexity.
⑤ Low risk management	⑤ more unpredictable uncontrollable

Requirements analysis involves understanding **what the system should do**, how it **interacts with other systems**, and **what limitations or constraints** exist.

Phases of Requirements Engineering

"I Enjoy Every New Software Version"

Phase	Purpose
Inception	Ask basic questions to understand: - the problem - who needs the solution - the expected solution - communication effectiveness
Elicitation	Gather requirements from all stakeholders (users, customers, developers, etc.)
Elaboration	Build detailed models to capture: - data requirements - functional requirements - behavioral requirements
Negotiation	Prioritize and resolve conflicts among requirements. Agree on a feasible version of the system
Specification	Represent requirements using: - written documents - models - mathematical formulas - use-cases - prototypes "Write More Formally Using Prototypes"
Validation	Review the requirements to find: - errors or misunderstandings - Clarification needed - unclear or missing information - inconsistencies - unrealistic/conflicting expectations "Every Cat Might Ignore Rain"

Types of Requirements

Type	Explanation
Business Requirements	High-level goals, objectives, and needs of the organization.
Stakeholder Requirements	Expectations and needs of each stakeholder group (users, customers, etc.).
Solution Requirements	Specific features and functions that the system must include.
→ <i>Functional Requirements</i>	What the system should do (features, user interactions, business rules).
→ <i>Non-Functional Requirements</i>	Qualities the system should have (performance, security, reliability, etc.). Also called quality attributes .
Transition Requirements	Requirements needed for moving from the current system to the new one (e.g., data migration, training).

✓ Functional Requirements

◆ Definition:

Functional requirements describe **how the system should behave** under specific conditions. They define **features, actions, and rules** that the system must implement.

◆ Key Characteristics:

- Must be **precise** and clear for both developers and stakeholders.
- Describe what the system **should do**.
- Often documented using **use cases**.

◆ Examples of Functional Requirements:

- Business rules
- Transaction corrections, adjustments, cancellations
- Administrative functions
- Authentication
- Authorization levels
- Audit tracking
- Reporting requirements
- External interfaces
- Certification requirements
- Historical data

✓ Non-Functional Requirements (NFRs)

◆ Definition:

Non-functional requirements define the **quality attributes** of a system. They focus on **how well** the system performs, rather than **what** it does.

◆ Key Characteristics:

- Define system **standards, constraints, and qualities**.
- Used to evaluate system performance, usability, and reliability.

◆ Examples of Non-Functional Requirements:

Category	Examples
Performance	Response time, throughput, resource utilization
Capacity	Data volume the system must handle
Availability	Uptime percentage or hours
Reliability	Frequency of failure or errors
Recoverability	Ability to restore from failures
Maintainability	Ease of making updates
Serviceability	Ease of system support
Security	Data protection and access control
Regulatory	Legal or industry compliance
Manageability	Ease of system monitoring and management
Environmental	Constraints based on the operating environment
Data Integrity	Accuracy and consistency of stored data
Usability	User-friendliness and interface design

Functional Requirement	Non Functional Requirement
Describe what the system does	① how the system work
② Define product features	② product properties
③ Focus on user requirement	③ Focus on user expectation
④ defined by user	defined by developer
⑤ mandatory	not mandatory, by derivable
⑦ Authentication, authorization	⑦ usability, accessibility

Elements of the Analysis Model

◆ 1. Scenario-Based Elements

Type	Explanation
Functional	Narratives describing software functions
Use-Case	Interactions between an actor and the system

◆ 2. Class-Based Elements

- Extracted from the scenarios
- Define **data and objects** within the system

◆ 3. Behavioral Elements

- Describe how the system **behaves dynamically**
- Example: **State diagrams**

◆ 4. Flow-Oriented Elements

- Represent the **flow of data**
- Example: **Data Flow Diagrams (DFDs)**

Use Cases

◆ Definition:

A **use case** describes the **interaction between a user (actor)** and the **system** to achieve a **specific goal**.

◆ Main Components:

Element	Description
Actors	Users or systems that interact with the system. They can be primary or secondary .
System	Behaviors/functions the system must support.
Goals	The purpose of the interaction (what the actor wants to achieve).

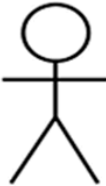
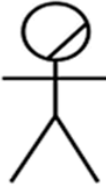





◆ Actor Types:

Actor Type	Role
Primary	Initiates the use case, interacts directly with the system (left side of the diagram).

Secondary	Supports the system, used by the system but doesn't initiate interaction (right side of the diagram).
------------------	---

◆ **Representation Formats:**

- **Use Case Specification** (text-based description of steps, actors, outcomes)
- **Use Case Diagram** (visual representation using UML notation)

Symbol Name	Symbol
Actor	
Business Actor	
Use Case	
Business Use Case	
Association	
Dependency	
Generalization	

———— Connection between Actor and Use Case

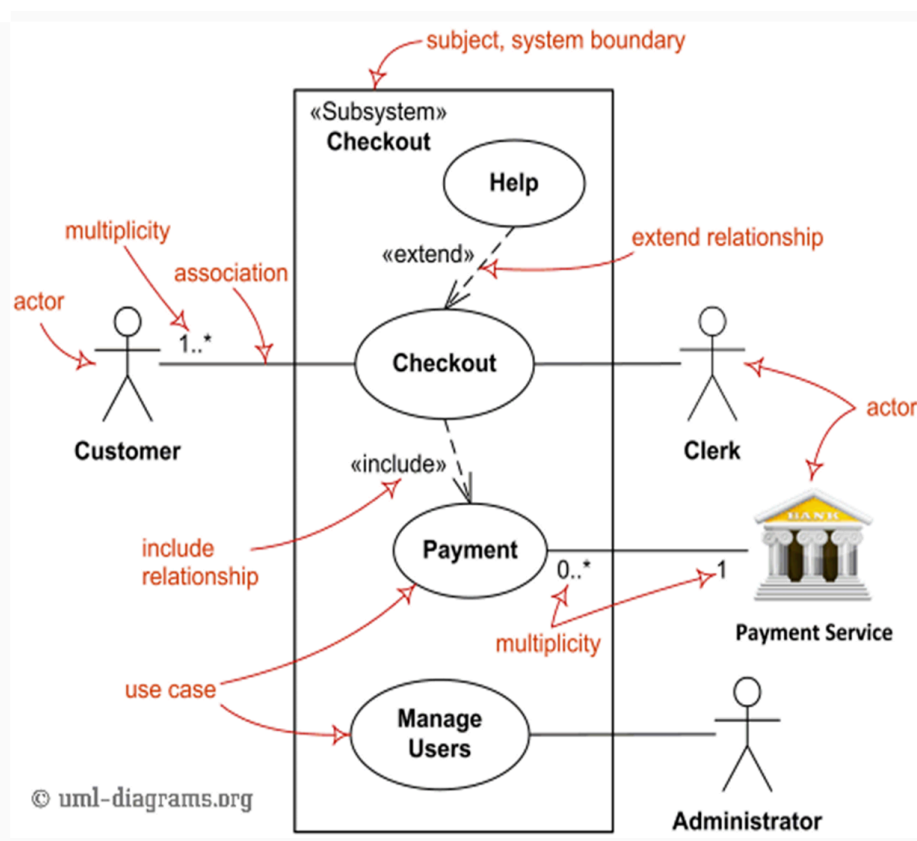
□ Boundary of system

«include»

Include relationship between Use Cases (one UC must call another; e.g., Login UC includes User Authentication UC)

«extend»

Extend relationship between Use Cases (one UC calls Another under certain condition; think of if-then decision points)



Use case:	Distribute Assignments
Actors:	Instructor (initiator)
Type:	Primary and essential
Description:	The Instructor completes an assignment and submits it to the system. The instructor will also submit the due date and the class the assignment is assigned for.
Cross Ref.:	Requirements XX, YY, and ZZ
Use-Cases:	<i>Configure HACS</i> must be done before any user (Instructor or Student) can use HACS

The

Unified Modeling Language (UML) is a standardized **visual modeling language** used in **software engineering** to specify, visualize, construct, and document the artifacts of a software system.

1. Activity Diagram

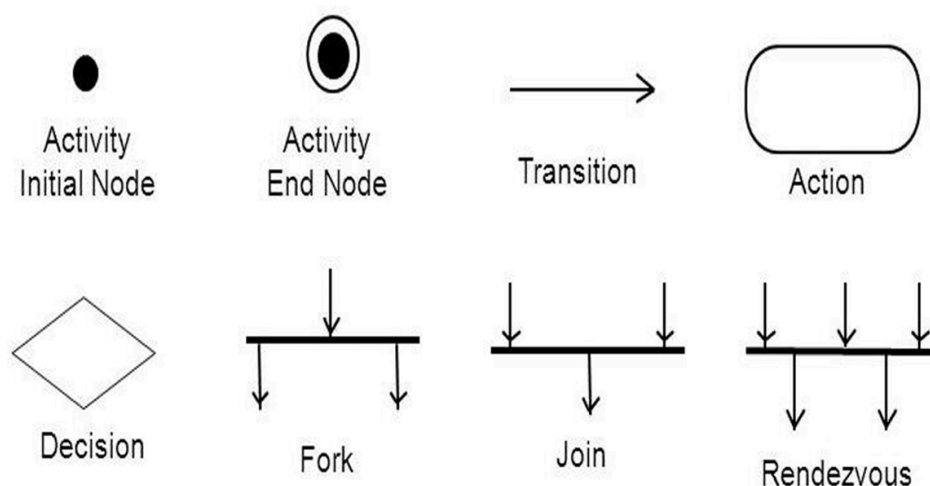
Definition: An activity diagram is a type of UML diagram that illustrates the dynamic aspects of a system by showing the **workflow of control from one activity to another**. It emphasizes the **sequence and conditions** for coordinating lower-level behaviors.

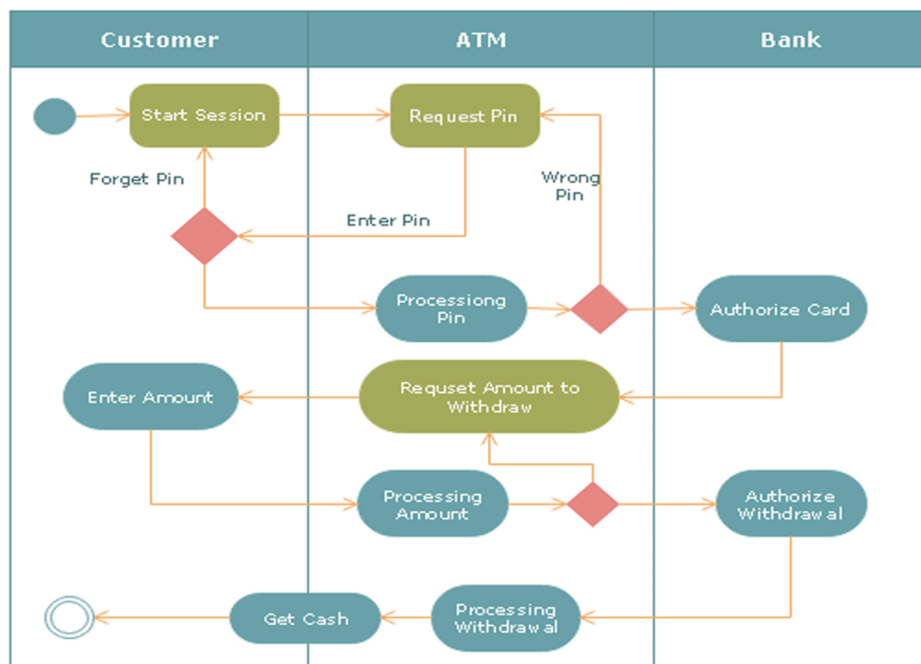
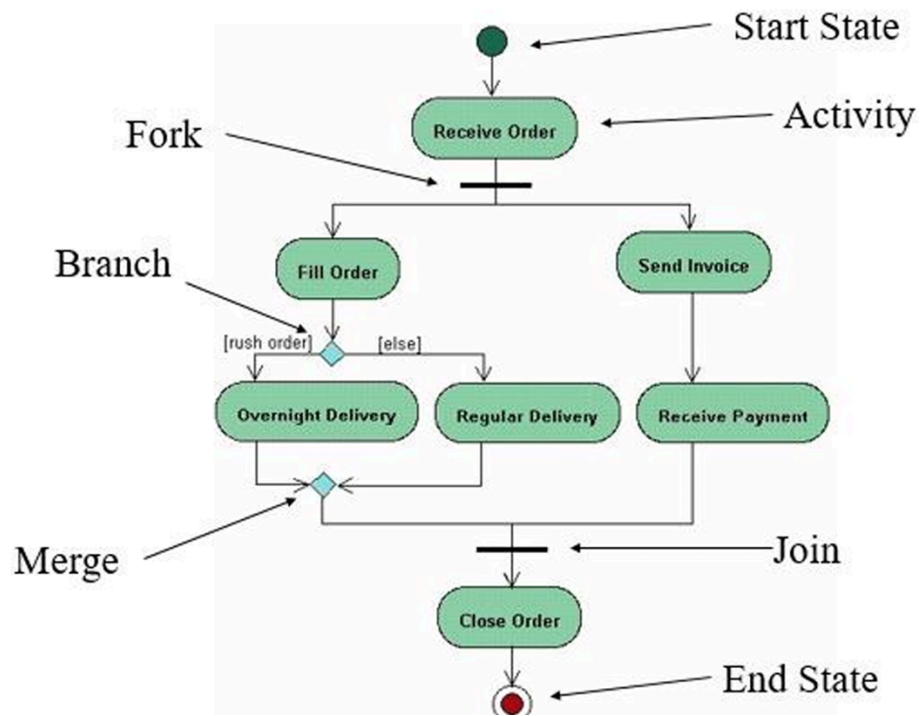
- Activity Diagrams to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case.

Purpose: To model the **flow of control** in a system, often used to describe the **steps in the execution** of a use case.

Key Points:

- Represents parallel and conditional paths.
- Focuses on flow conditions and transitions.
- Useful for modeling business processes and workflows.





2. Sequence Diagram

Definition: A sequence diagram describes the **interaction between objects** in a **sequential order**. It shows how **objects communicate with each other** through messages over time.

Purpose: To represent how different parts of the system interact in a particular scenario.

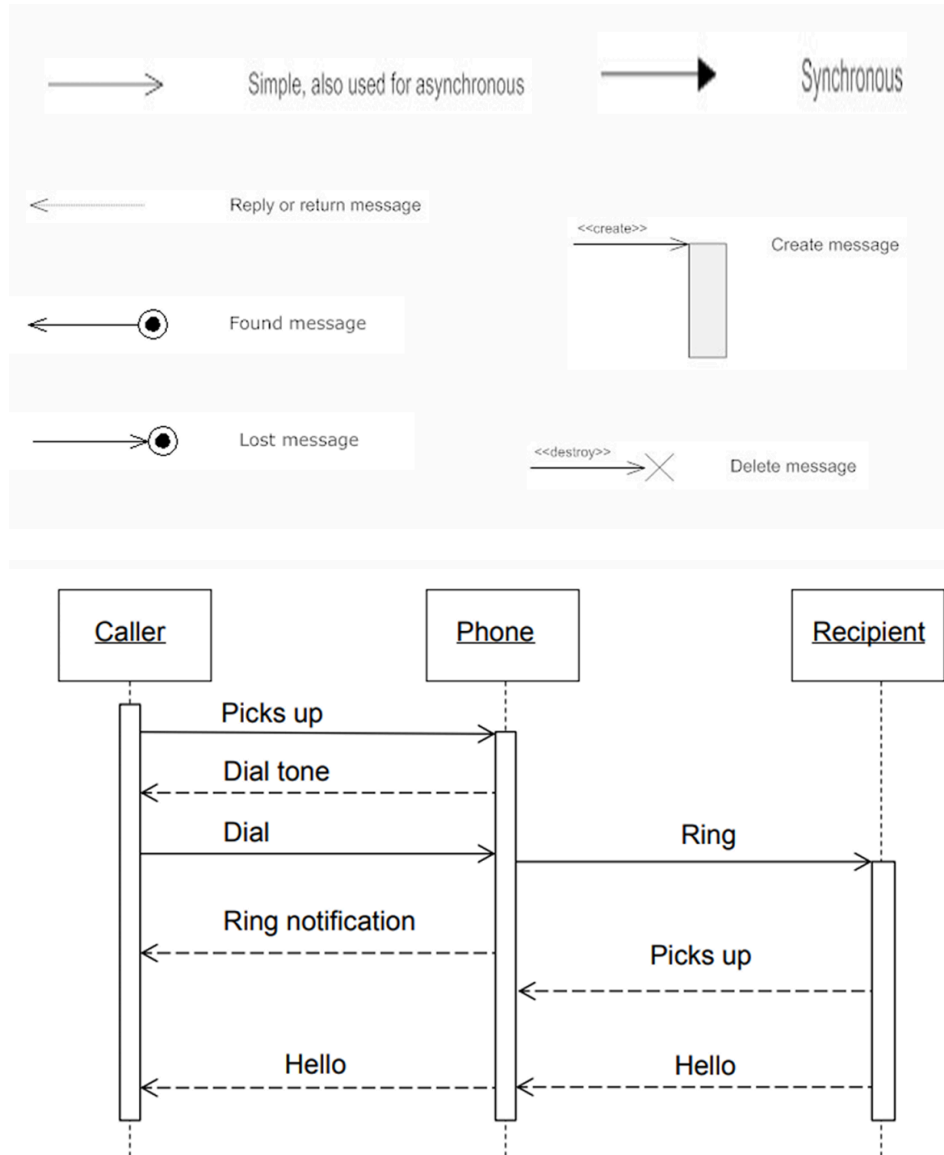
Key Components:

- **Lifelines:** Vertical dashed lines that represent an object's presence over time.
- **Messages:** Horizontal arrows that show the communication between objects.

Use:

- To describe real-time specification and usage scenarios.
- To model the logic of a complex operation or workflow.

Types of message



3. Data Flow Diagram (DFD)

Definition: A DFD is a graphical representation that illustrates the **flow of data through a system**. It depicts **how input is turned into output** through a **sequence** of transformations.

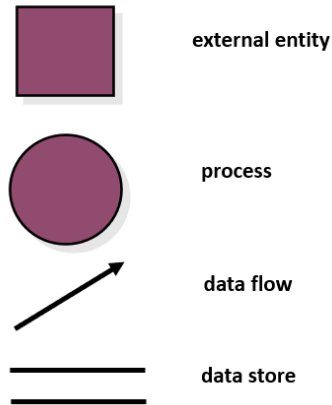
Main Parts:

- **External Entity (Actor):** Source or sink of data (represented by rectangles): Actors are the **active objects that interact with the system** by either producing data and inputting them to the system, or consuming data produced by the system. In other words, actors serve as the sources and the sinks of data.
- **Process:** Transformation of data (represented by ellipses) : Processes are the **computational activities that transform data values**. A whole system can be visualized as a high-level process. A process may be further divided into smaller components. The lowest-level process may be a simple function.
- **Data Flow:** Movement of data (represented by arrows): Data flow represents **the flow of data between two processes**. It could be between an actor and a process, or between a data store and a process. A data flow

denotes the value of a data item at some point of the computation. This value is not changed by the data flow.

- **Data Store:** Storage for data (represented by parallel lines): Data stores are the **passive objects that act as a repository of data**. Unlike actors, they cannot perform any operations. They are used to store data and retrieve the stored data. They represent a data structure, a disk file, or a table in a database.

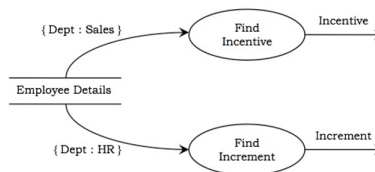
Flow Modeling Notation



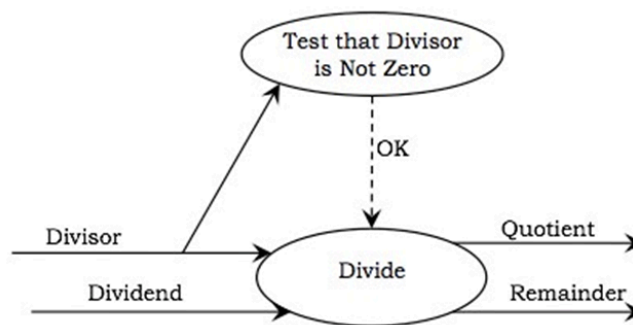
These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 7/e (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Supporting Elements:

- **Constraints:** Conditions that must hold true during data processing (represented by braces {}).



- **Control Flows:** Boolean values controlling the activation of processes (represented by dotted arrows). A process may be associated with a certain Boolean value and is evaluated only if the value is true,



4. Data Dictionary

Definition: A data dictionary is a centralized repository that **contains metadata about the data** (i.e database) in the system, including definitions, relationships, sources, usage, and formats.

- A data dictionary contains metadata i.e., data about the database.
- It plays an important role in building a database.

Purpose: To standardize definitions and facilitate consistency and communication among stakeholders.

The data dictionary in general contains information about the following:

- Names of tables and fields in the database
- Constraints on tables in the database
- Physical storage and access methods.

Advantages:

- Provides well-structured database documentation.
- Helps identify redundancy.
- Aids database administrators in management and maintenance.

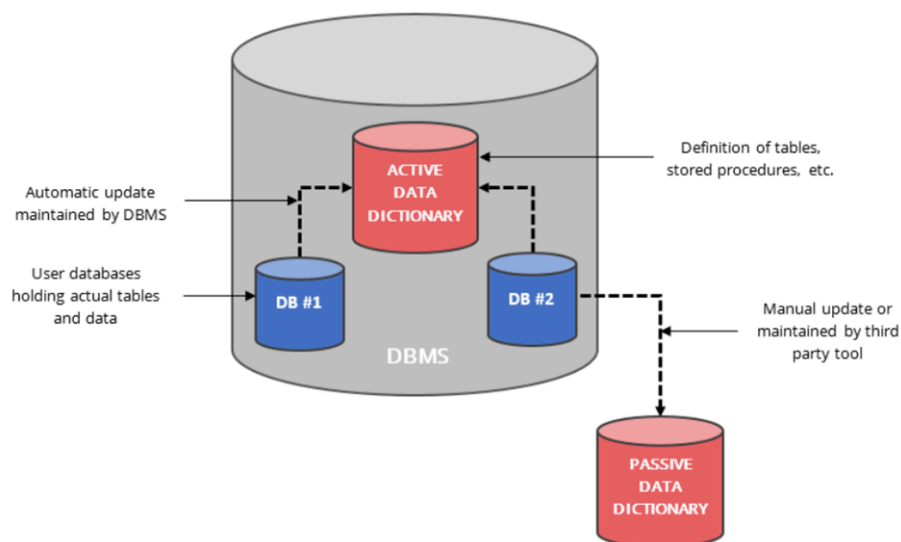
Types:

- **Active Data Dictionary:**

- An active data dictionary is a type of dictionary that is very **consistent and managed automatically by the DBMS**.
- **Automatically updated by the DBMS.**
- Changes in the database are reflected immediately.
- **No external maintenance** required.
- **Cost-efficient** and consistent.

- **Passive Data Dictionary:**

- **Maintained manually or through external tools.**
- High maintenance **cost**.
- Prone to becoming outdated.
- **Difficult to manage, less** reliable than active types.



•

Four Categories of Data Dictionary:

1. Data Flows: Collection of data elements. Data flow is a collection of data elements
2. Data Structures: Groups of elements. Usually algebraic notations
3. Data Elements: Basic unit of data with specific definition. Data elements definitions describe a data type.

4. Data Stores: Data stores are created for each different data entity being stored.

Active Data Dictionary	Passive Data Dictionary
The database management system automatically maintains the active data dictionary.	The passive data dictionary is modified whenever the structure of the database is changed.
The active data dictionary is very consistent with the structure and the definition of the database.	As the process of maintaining or modification is manual, so it is not consistent and not updated with the current structure of the database.
Another name of this dictionary is integrated data dictionary.	Another name of this dictionary is a non-integrated dictionary or a standalone dictionary.
The database management systems automatically manage this dictionary.	The users are responsible for manually managing this dictionary.
It doesn't require any separate database.	It requires a separate database for working with this dictionary.
Mostly, relational database management systems (RDBMS) contain this type of dictionary as it can be easily derived from their system catalog.	As the passive data dictionary requires a separate database, so it allows the programmers to remain independent of using a particular RDBMS.
It doesn't provide a good User Interface.	It provides you a friendly User Interface.
The information in an active data dictionary is up-to-date as it is automatically managed.	The information in a passive data dictionary is not up-to-date as it is managed manually by the users.

✓ 1. Class Model

A **class model** is a conceptual representation that describes the **structure** and **behavior** of objects within a software system. It defines:

- the objects to be manipulated,
- the attributes and operations associated with them,
- the relationships among these objects,
- and the collaborations that take place between the classes.

✓ 2. Data Modeling

Data modeling is the process of analyzing data objects without considering the processes acting upon them. It focuses on:

- independent data objects,
- their structure and characteristics,
- the relationships among them,
- and the customer's level of abstraction.

It creates a conceptual view of data that reflects how the end-users perceive it.

✓ 3. Data Object

A **data object** is a representation of any composite information required by the software. It can be:

- an external entity (e.g., user, sensor),
- a thing (e.g., report, document),
- an event (e.g., alarm),
- a role (e.g., administrator),

- a location (e.g., warehouse),
- or a structure (e.g., file or table).

| A data object only encapsulates data — it does not define operations or behaviors.

✓ 4. Attributes

Attributes are descriptors or properties that define the characteristics of a data object.

Example:

- Object: `Automobile`
- Attributes: `make`, `model`, `price`, `bodyType`, `color`

✓ 5. Relationship

A **relationship** represents the **connection between two or more data objects**. It describes how objects are associated or interact with one another.

Example:

- A person **owns** a car.
- A person is **insured to drive** a car.

✓ 6. ERD (Entity Relationship Diagram)

An **ERD** is a graphical notation used to illustrate data objects (entities), their attributes, and the relationships between them. It includes:

- Entity symbols
- Relationship connectors
- Cardinality/multiplicity indicators such as:
 - (1,1) → exactly one
 - (0,m) → zero or more

✓ 7. Class-Based Modeling

Class-based modeling represents:

- **Objects** that the system will use or manage
- **Operations** that act on those objects
- **Relationships** (including inheritance or aggregation)
- **Collaborations** between different classes

Key elements include:

- Classes, attributes, operations
- CRC models
- Collaboration diagrams
- Packages

✓ 8. Identifying Analysis Classes

To identify candidate classes from requirements:

- Perform a **grammatical parse** of scenarios (underline all nouns)
- Each noun may represent a potential class

- Determine whether the noun belongs to:
 - **Solution space** (used in implementation)
 - **Problem space** (used in description only)

✓ 9. Manifestations of Analysis Classes

Analysis classes may take the form of:

Type	Example
External Entities	Users, hardware devices
Things	Reports, signals, documents
Occurrences/Events	Transactions, triggers
Roles	Manager, customer, admin
Organizational Units	Department, team
Places	Factory floor, loading dock
Structures	Sensor, vehicle, table

✓ 10. Criteria for Potential Classes

Criterion	Description
Retained Information	The class must retain data necessary for system operation
Needed Services	It must support relevant operations (methods)
Multiple Attributes	Must have more than one meaningful attribute
Common Attributes	All instances share the same set of attributes
Common Operations	Similar operations applicable to all instances
Essential Requirements	Essential for communication between system and external entities

✓ 11. Defining Attributes

Attributes are assigned depending on the **context of the class** in the system.

Example:

- For a Sports Management System:
 - `name`, `battingAverage`, `fieldingPercentage`
- For a Pension System:
 - `salary`, `pensionOption`, `vestingStatus`

✓ 12. Defining Operations

Operations represent the **behavior of a class**. They are identified by parsing verbs from the requirements.

Four Categories:

1. **Data Manipulation** (add, update, delete)
2. **Computation** (calculate salary, total marks)
3. **Inquiry** (check status)
4. **Event Monitoring** (detect threshold breach, motion detection)

✓ 13. CRC Models (Class-Responsibility-Collaborator)

CRC (Class-Responsibility-Collaborator) modeling uses index cards to document:

- **Class name**

- **Responsibilities** (left side)
- **Collaborators** (right side)

This method helps visualize how classes interact and what responsibilities they carry.

✓ 14. Class Types

Type	Purpose
Entity Class	Represents core domain concepts (e.g., <code>Sensor</code> , <code>FloorPlan</code>)
Boundary Class	Manages user interface (e.g., forms, reports)
Controller Class	Coordinates operations, workflows, and communication between classes

✓ 15. Responsibilities

- Distribute responsibilities logically among classes
- Keep related data and behavior together
- Avoid scattering related data across multiple classes
- Responsibilities should be general and reusable
- Shared where needed across related classes

✓ 16. Collaborations

Classes fulfill their responsibilities either:

- **Independently** (using their own methods)
- **Collaboratively** (interacting with other classes)

Types of relationships:

- **Is-part-of**
- **Has-knowledge-of**
- **Depends-upon**

✓ 17. Associations and Dependencies

Term	Explanation
Association	A structural relationship (e.g., student enrolls in courses)
Dependency	A situation where one class relies on another to perform a task or supply data

✓ 18. Multiplicity

Specifies the **number of instances** that can be involved in a relationship:

- **(1,1)** → Exactly one
- **(0,m)** → Zero or many
- **(1,n)** → At least one

✓ 19. Analysis Packages

To organize a large model, related classes and use-cases are grouped into **packages**.

Visibility symbols:

- `+` → Public (accessible everywhere)
- `-` → Private (not accessible outside the package)

- # → Protected (accessible only to related packages)

Project Planning

Project Planning is an organized process from requirements gathering to testing and support.

Software Project Manager

Managing People

- Act as a Project Leader
- Contact with stakeholders
- Managing human resources

Managing Projects

- Defining and setting up project scope
- Managing project management activities
- Monitoring progress and performance
- Risk analysis
- Take necessary step to avoid or come out of problems

Project Planning Process

1. Identify Stakeholders Need: meet the expectations of stakeholders
2. Identify Project Objectives: specific, measurable, achievable objectives
3. Deliverable and due dates: fixed date and time that an objective is due, deliverables = products, service or result
4. Project Schedules: project start and end date
5. Provide Roles and Responsibilities: effective communication, who is involved and their task, understand expected objective
6. Identify Project budget: anticipating budget cost, monitoring budget
7. Identify Communication Plan: effective communicate with client, team and others
8. Provide tracking and management: deliver project on time and organize task, track productivity and growth of project

Resource used in Project Management

1. Human
2. Reusable Components
3. Hardware and Software tools

Project Scope Management

Scope = set of deliverables or features of a project

Scope management = creates boundaries of the project by clearly defining what would be done and what not

Steps

1. Plan Scope Managements : documentarian and guidelines of project scope, product scope, project life, how to define, validate and control.
2. Collect Requirements: collect requirements from all stakeholders
3. Defining Scope: identifying project objectives, goals, tasks, budget, resources, schedule, expectation
4. Create WBS: Work Breakdown Structure = subdividing project deliverables into smaller units, break down into phases, including priority task
5. Validate Scope: focuses on mainly customer acceptance, customer gives feedback
6. Control Scope: monitoring the status of the project and managing changes

WBS : Work Breakdown Structure

A top-down hierarchical decomposition of the total project scope into manageable sections.

Working of WBS Steps

1. Project managers decide project name at top
2. Project managers identifies the main deliverables of the project
3. These main deliverables are broke down into smaller higher-level tasks
4. Process is done recursively to produce much smaller independent task
5. Choose task owner. they need to get the job down

Components

1. WBS Dictionary: Document that defines the various wbs element
2. WBS levels: determines the hierarchy level of a wbs elemet
3. Task : main deliverable tasks
4. Sub tasks: devided tasks
5. Control account : group work packages and measure their status
6. project deliverables: desired outcome of project tasks and work package

Project Scheduling

| Definition: The process of converting a project plan into an operating timetable.

Project scheduling is responsible activity of project manager.

Process:

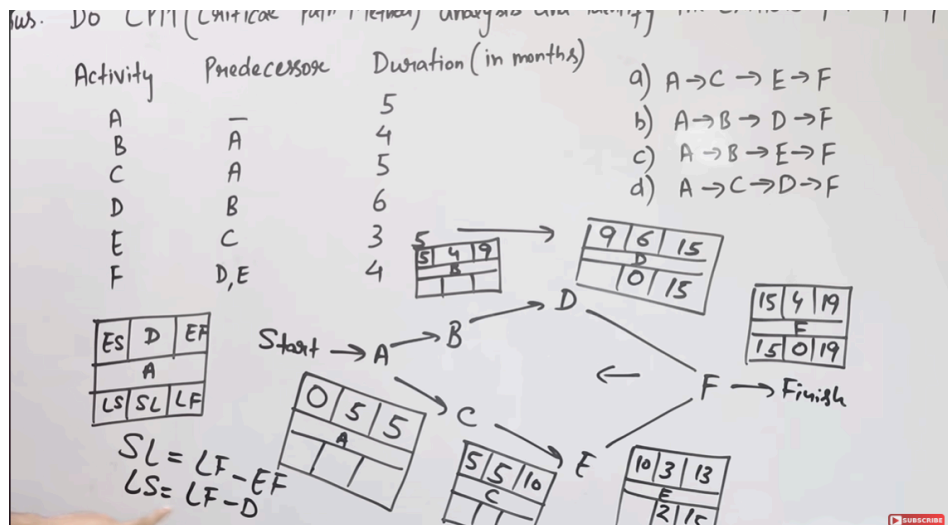
1. Identify all the functins required
2. \Break down large function into smaller activites , wbs
3. Determine the dependency among various activities
4. Allocate resources to activities
5. Assign people to conduct different activities
6. Plan the beginning and ending dates for different activities
7. Create activity network and bar or gratt chart

Techniques:

1. CPM :

The Critical Path Method (CPM), also known as Critical Path Analysis (CPA), is **a project management technique that identifies the longest sequence of tasks (the critical path) that must be completed on time to finish the**

project, highlighting tasks that directly impact the project's timeline.

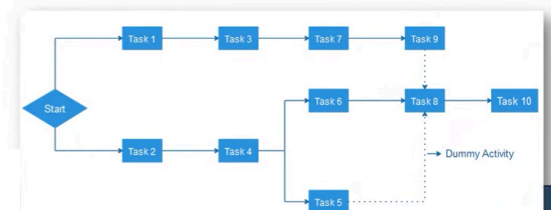


2. Program Evaluation and Review Technique (PERT):

- It is a way to schedule flow of tasks in a project and estimate total time taken to complete it.
- PERT charts offer a visual representation of the major activities (and dependencies) in a project

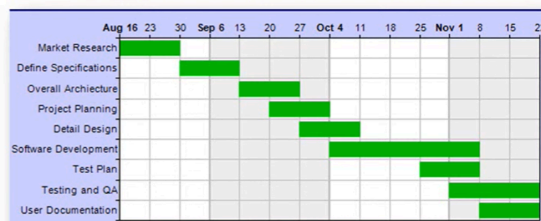
It calculate:

- **Optimistic time (O):** Quickest time you can complete a project
- **Pessimistic time (P):** Longest time it'll take to complete your project
- **Most likely time (M):** How long it'll take to finish your project if there are no problems.
- $(O + 4M + P)/6$



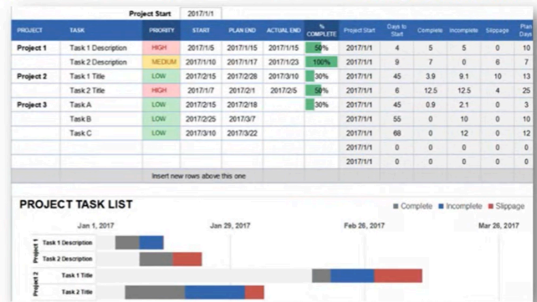
3. Gantt Chart:

- A Gantt chart is a type of bar graph that project managers use for planning and scheduling in complex project.
- It represent each task horizontally on a bar chart, which shows the start and end dates & they frequently include deadlines & dependencies of tasks.
- It easier to visualize the progress of a project and see how different tasks interact with one another.



4. Task List:

- One of the simplest project scheduling techniques is the creation of a task list.
- Create task list using a word processor or spreadsheet software.
- It create a list of tasks and include important information like the task manager, start date, deadline & completion status.

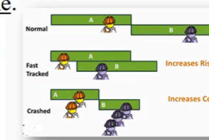


5. Fast Tracking :

- In Fast Tracking, Project is being implemented by either simultaneously executing many tasks or by overlapping many tasks to each other.
- **Example:** In software development project, designing and development can be taken up in parallel. Once design of essential features is ready and approved, development team can work on it. Meanwhile, the designing team will work on the remaining elements and functions.

6. Crashing:

- Crashing deals with involving more resources to finish the project on time.
- **Example:** Add more developer in project, Paying overtime to employee,
- Crashing can only be applied when it fits your project budget.



Lecture 10: Risk Management in Software Projects

Introduction

Risk management is a critical component of software project planning, aimed at identifying, analyzing, and controlling risks to ensure project success. A **risk** is an uncertain future event with a probability of occurrence and potential for loss, impacting time, budget, performance, or project outcomes.

Sources of Risk

Risks in software projects arise from various sources, including:

- **Misunderstanding Customer Requirements:** Incorrect or unclear interpretation of client needs.
- **Uncontrolled Requirement Changes:** Frequent or poorly managed changes to requirements.
- **Unrealistic Promises:** Overcommitting to clients beyond project capabilities.
- **Misjudging New Methodologies:** Overestimating the benefits or underestimating the challenges of new tools or processes.
- **Poor Software Design:** Underestimating the robustness or extensibility of the design.
- **Team Effectiveness Miscalculation:** Overestimating team collaboration or productivity.
- **Inaccurate Budget Estimation:** Underestimating costs or resources needed.

Risk Identification

Risk identification involves detecting potential risks early to minimize their impact. Techniques include:

- **Brainstorming:** Collaborative sessions to identify potential risks.

- **SWOT Analysis:** Evaluating strengths, weaknesses, opportunities, and threats.
- **Causal Mapping:** Mapping cause-and-effect relationships to uncover risks.
- **Flowcharting:** Visualizing processes to identify risk points.

Types of Risks

1. **Technology Risks:** Issues with hardware or software used in development.
2. **People Risks:** Challenges related to team members, such as turnover or skill gaps.
3. **Organizational Risks:** Problems stemming from the organizational environment.
4. **Tools Risks:** Issues with development tools or support software.
5. **Requirement Risks:** Risks from changing or mismanaged customer requirements.
6. **Estimation Risks:** Errors in estimating resources, time, or costs.

Risk Analysis and Prioritization

Risk analysis assesses identified risks by:

1. **Identifying Problems:** Determining the root causes of risks.
2. **Estimating Probability:** Calculating the likelihood of occurrence.
3. **Assessing Impact:** Evaluating the potential effect on the project.

Probability Categories

- **Very Low (0–10%):** Tolerable risk with no significant harm.
- **Low (10–25%):** Minor effect on the project.
- **Moderate (25–50%):** Impacts project timeline.
- **High (50–75%):** Affects timeline and budget.
- **Very High (>75%):** Severe impact on output, time, budget, and performance.

Risk Control

Risk control involves planning, monitoring, and resolving risks to achieve desired project outcomes.

1. Risk Planning

Risk planning develops strategies to mitigate significant risks. Methods include:

- **Avoid the Risk:** Modify requirements, reduce scope, or offer incentives to retain staff.
- **Transfer the Risk:** Outsource risky components or purchase insurance.
- **Reduce the Risk:** Plan for potential losses, such as recruiting additional staff to cover turnover.

2. Risk Monitoring

Risk monitoring is an ongoing process to track project progress and evaluate risks:

- Continuously assess assumptions about risks.
- Identify changes in risk probability or impact.
- Take corrective actions as needed to keep risks under control.

3. Risk Resolution

Risk resolution ensures risks are managed within acceptable levels:

- Depends on accurate risk identification, analysis, and planning.
- Requires prompt and effective responses to emerging issues.
- Keeps the project on track by addressing risks as they arise.

RMMM Plan

The **Risk Mitigation, Monitoring, and Management (RMMM)** Plan is a structured approach integrated into the overall project plan. It documents risks using a **Risk Information Sheet (RIS)**, which includes:

- Risk ID, date, probability, impact, description, avoidance strategies, monitoring actions, management plan, and current status.
- Managed via a database for easy creation, prioritization, searching, and analysis.

Example: Late Project Delivery

Risk: Project delivery exceeds the deadline.

1. Mitigation:

- Estimate development time as 20 days but quote 30 days to the client for buffer.
- Implement precautionary measures before development starts.

2. Monitoring:

- Create a project schedule with clear start and end dates.
- Track progress within the 20–30-day window.

3. Management:

- If the deadline is missed, negotiate with the client for extra time or offer additional features to maintain satisfaction.

Risk Mitigation

Risk mitigation is a proactive approach to avoid risks before they occur. Steps include:

1. Communicate with staff to identify potential risks.
2. Eliminate causes of risks (e.g., unclear requirements).
3. Develop policies to ensure project continuity.
4. Regularly review and control project documents.
5. Conduct timely reviews to accelerate progress.

Risk Management

Risk management is a reactive approach applied after risks materialize:

- Assumes mitigation efforts failed, and the risk has occurred.
- Handled by the project manager to resolve issues.
- Effective mitigation simplifies management (e.g., sufficient staff, clear documentation, and shared knowledge ease onboarding of new team members).

Example Scenario

Risk: High staff turnover.

- **Mitigation Success:** Sufficient additional staff, comprehensive documentation, and shared knowledge ensure new employees can quickly adapt.
- **Management:** Project manager leverages these resources to maintain development continuity.

Drawbacks of RMMM

While effective, the RMMM approach has limitations:

- **Additional Costs:** Implementing RMMM increases project expenses.
- **Time-Intensive:** Requires significant time for planning and execution.
- **Complex for Large Projects:** RMMM can become a project in itself.
- **No Guarantee:** Risks may still emerge post-delivery, and RMMM does not ensure a risk-free project.

Conclusion

Effective risk management in software projects involves identifying, analyzing, and controlling risks through proactive mitigation and reactive management. The RMMM Plan provides a structured framework to document and address risks, but it requires careful planning and resources. By understanding risk sources, types, and control strategies, project managers can enhance the likelihood of successful project outcomes.

Lecture 11 : Design Concepts

Introduction

Software design is the process of transforming user requirements into a form suitable for coding and implementation. As the first step in the Software Development Life Cycle (SDLC), it shifts focus from the problem domain to the solution domain, specifying how to fulfill requirements outlined in the Software Requirements Specification (SRS). Software design is typically performed by software design engineers or UI/UX designers.

Mitch Kapor's Software Design Manifesto

Mitch Kapor, creator of Lotus 1-2-3, emphasized three qualities of good software design:

- **Firmness:** The software should be free of bugs that impair functionality.
- **Commodity:** The software should meet its intended purpose.
- **Delight:** The user experience should be pleasurable.

Objectives of Software Design

A well-designed software system should achieve:

- **Correctness:** Accurately meet the specified requirements.
- **Completeness:** Include all necessary components, such as data structures, modules, and interfaces.
- **Efficiency:** Optimize resource usage.
- **Flexibility:** Adapt to changing needs.
- **Consistency:** Maintain uniformity across the design.
- **Maintainability:** Be simple enough for other designers to maintain.

Software Quality Attributes (FURPS)

The FURPS model defines key quality attributes for software design:

- **Functionality:** Evaluates the feature set, capabilities, generality of functions, and system security.
- **Usability:** Considers human factors, aesthetics, consistency, and documentation.
- **Reliability:** Measures failure frequency, output accuracy, mean-time-to-failure (MTTF), recovery ability, and predictability.
- **Performance:** Assesses processing speed, response time, resource consumption, throughput, and efficiency.
- **Supportability:** Encompasses extensibility, adaptability, serviceability, testability, compatibility, configurability, ease of installation, and problem localization (collectively contributing to maintainability).

Software Quality Guidelines

To ensure high-quality design, the following guidelines should be followed:

- Use recognizable **architectural styles or patterns** and components with good design characteristics.
- Enable **evolutionary implementation**, allowing incremental development (though smaller systems may use linear design).

- Ensure **modularity** by logically partitioning software into elements or subsystems.
- Provide distinct representations of **data, architecture, interfaces, and components**.
- Select **data structures** based on recognizable patterns suitable for implementation.
- Design **components** with independent functional characteristics.
- Create **interfaces** that reduce complexity between components and external systems.
- Derive the design using a **repeatable method** driven by requirements analysis.
- Represent the design with **notation** that clearly communicates its meaning.

Software Design Process

The software design process translates the analysis model (requirements) into a design model (solution) and consists of three levels:

1. Interface Design:

- Focuses on interactions between the system and users/devices.
- Uses scenario-based and behavioral diagrams (e.g., use case diagrams).
- Does not address internal structure.

2. Architectural Design:

- Defines major system components, their responsibilities, properties, interfaces, and interactions.
- Uses class-based and flow-based diagrams (e.g., class diagrams, data flow diagrams).

3. Detailed Design:

- Specifies internal elements of major components, including properties, relationships, processing, algorithms, and data structures.

The design must:

- Implement all explicit and implicit requirements from the analysis model.
- Be a readable, understandable guide for coders, testers, and support teams.
- Provide a complete picture of the software, addressing data, functional, and behavioral domains from an implementation perspective.

Fundamental Design Concepts

Software design concepts provide the principles and logic behind creating effective software designs. These concepts form a supporting structure for development.

1. Abstraction

Abstraction hides unnecessary implementation details, showing only essential information to users.

- **Procedural Abstraction:** Divides subprograms into hidden and visible groups of functionalities (e.g., a function like `open()` hides the details of the enter algorithm).
- **Data Abstraction:** Represents data objects while hiding manipulation details (e.g., a stack's `Push()`, `Pop()`, `Top()`, and `Empty()` methods hide internal data structure implementation).
- **Example:** A door object might expose attributes like manufacturer, model, type, and swing direction while hiding internal data structures.

2. Architecture

Architecture defines the overall structure of program modules and their interactions, providing conceptual integrity.

- **Structural Properties:** Defines components (e.g., modules, objects) and their interactions (e.g., method invocations).
- **Extra-Functional Properties:** Addresses performance, capacity, reliability, security, and adaptability requirements.
- **Families of Systems:** Leverages reusable architectural patterns for similar systems.

- **Reference:** Shaw and Garlan [SHA95a].

3. Design Patterns

Design patterns are reusable solutions to common design problems.

- **Pattern Template:**
 - **Name:** A concise, expressive name.
 - **Intent:** Describes the pattern's purpose.
 - **Also-Known-As:** Lists synonyms.
 - **Motivation:** Provides a problem example.
 - **Applicability:** Specifies relevant design situations.
 - **Structure:** Describes required classes.
 - **Participants:** Outlines class responsibilities.
 - **Collaborations:** Details participant interactions.
 - **Consequences:** Discusses trade-offs and design forces.
 - **Related Patterns:** References related patterns.

4. Separation of Concerns

Separation of concerns divides complex problems into smaller, independently solvable pieces.

- Each concern represents a feature or behavior from the requirements model.
- Reduces effort and time by making problems more manageable.

5. Modularity

Modularity divides a system into smaller, manageable parts (modules) to reduce complexity.

- Modules are integrated to meet software requirements.
- **Benefits:** Makes software intellectually manageable, reduces development costs, and improves understanding [Mye78].
- **Contrast:** Monolithic software (single module) is difficult to understand due to numerous control paths, variables, and complexity.

6. Information Hiding

Information hiding ensures modules only share necessary information, hiding internal data structures and algorithms.

- **Benefits:**
 - Reduces side effects.
 - Limits the global impact of local design decisions.
 - Encourages controlled interfaces.
 - Discourages global data usage.
 - Promotes encapsulation, enhancing design quality.

7. Functional Independence

Functional independence is achieved through modules with single-minded functions and minimal interaction (low coupling, high cohesion).

- **Cohesion:** The degree to which a module performs a single task with little interaction with others (high cohesion is desirable).
- **Coupling:** The degree of interdependence between modules (low coupling is desirable).
- **Benefits of High Cohesion and Low Coupling:**
 - **Readability:** Modules are easier to understand.

- **Maintainability:** Changes in one module have minimal impact on others.
- **Modularity:** Simplifies module development.
- **Scalability:** Facilitates adding or removing modules.
- **Testability:** Simplifies testing and debugging.
- **Reusability:** Modules can be reused in other systems.
- **Reliability:** Improves overall system quality.

8. Refinement

Refinement is a top-down approach that elaborates procedural details hierarchically until programming language statements are reached.

- **Example:** The task “open door” is refined into steps like “walk to door,” “reach for knob,” “turn knob clockwise,” and so on, until all details are specified.

9. Aspects

Aspects represent cross-cutting concerns—requirements that affect multiple parts of the design.

- **Definition:** Requirement A cross-cuts requirement B if B cannot be satisfied without considering A [Ros04].
- **Example:** In the SafeHomeAssured.com WebApp:
 - Requirement A: Access camera surveillance via the internet.
 - Requirement B: Validate registered users before access.
 - B cross-cuts A because user validation (B) must be implemented across all functions, including camera access (A).

10. Refactoring

Refactoring improves a software system's internal structure without altering its external behavior [FOW99].

- **Purpose:** Eliminates redundancy, unused elements, inefficient algorithms, or inappropriate data structures to enhance design quality.

Object-Oriented (OO) Design Concepts

OO design leverages principles to create flexible, reusable designs:

- **Design Classes:**
 - **Entity Classes:** Refined from analysis classes to represent data and behavior.
 - **Boundary Classes:** Manage user interfaces (e.g., screens, reports) and represent entity objects to users.
 - **Controller Classes:** Handle creation/update of entity objects, instantiation of boundary objects, complex communication, and data validation.
- **Inheritance:** Subclasses inherit all responsibilities of their superclass.
- **Messages:** Stimulate behavior in receiving objects.
- **Polymorphism:** Allows objects to be treated as instances of their parent class, reducing effort to extend designs.

Design Model Elements

The design model transforms the analysis model into a detailed implementation plan, encompassing:

1. Data Elements:

- Data structures derived from the data model.
- Database architecture for persistent storage.

2. Architectural Elements:

- Derived from the application domain, analysis classes, and architectural patterns/styles [Sha96].
- Define relationships, collaborations, and behaviors for design realizations.

3. Interface Elements:

- User interfaces (UI).
- External interfaces to other systems, devices, or networks.
- Internal interfaces between design components.

4. Component Elements:

- Detailed specifications of software components, including algorithms and processing logic.

5. Deployment Elements:

- Define how software components are deployed across hardware or network environments.

Conclusion

Software design bridges user requirements and implementation, ensuring correctness, efficiency, and maintainability. By adhering to quality guidelines, leveraging fundamental concepts (e.g., abstraction, modularity, functional independence), and applying OO principles, designers create robust, scalable systems. The design process—interface, architectural, and detailed design—transforms analysis models into actionable plans, supported by a comprehensive design model.

Reference

Pressman, R. S. (2010). *Software Engineering: A Practitioner's Approach, 7th Edition*. McGraw-Hill.

Lecture 11: Software Project Management and Scheduling

Introduction

Software project management involves planning, organizing, and controlling resources to deliver a software product that meets customer requirements within time and budget constraints. Effective project management focuses on scheduling, cost estimation, and risk management to ensure project success. This guide covers key concepts, techniques, and models for managing and scheduling software projects, optimized for exam preparation.

Project Management Spectrum

Effective software project management revolves around four key components, known as the **Four P's**:

1. People

- **Importance:** Human resources are the most critical factor in project success.
- **Selection:** Choose individuals with the right skills and talents.
- **Roles and Responsibilities:**
 - **Senior Manager:** Defines business issues and influences the project.
 - **Project Manager:** Plans, motivates, organizes, and controls project activities; possesses problem-solving and team management skills.
 - **Software Engineer:** Delivers technical expertise.
 - **Customer:** Specifies requirements.
 - **End Users:** Interact with the final software product.

2. Product

- **Definition:** The software product is the ultimate deliverable of the project.
- **Planning Requirements:**
 - Establish objectives and scope.

- Identify alternative solutions.
- Define technical and management constraints.
- **Importance:** Accurate product definition enables realistic cost estimation, risk identification, and scheduling.

3. Process

- **Definition:** A methodology that outlines steps to complete the project as per requirements.
- **Importance:** A clear process ensures team members know their tasks and timelines, increasing the likelihood of meeting project goals.
- **Phases:**
 - Documentation
 - Designing
 - Implementation
 - Software Configuration Management
 - Deployment
 - Interaction

4. Project

- **Definition:** Encompasses requirement analysis, development, delivery, maintenance, and updates.
- **Project Manager's Role:**
 - Guides the team to achieve objectives.
 - Resolves issues, monitors costs, and ensures adherence to deadlines.
 - Manages activities to prevent project failure.

Boehm's W5HH Principle

Barry Boehm's W5HH (Why, What, When, Who, Where, How, How Much) principle provides a framework for efficient project management by answering key questions:

1. **Why is the system being developed?**
 - Identifies business reasons and problem statements to justify the project's cost and time.
2. **What activities are needed?**
 - Defines key tasks required by the customer to establish a project schedule.
3. **When will it be completed?**
 - Specifies start and end dates for tasks to meet project goals.
4. **Who is responsible for each activity?**
 - Assigns roles and responsibilities to team members.
5. **Where are they organizationally located?**
 - Clarifies that responsibilities may extend beyond the software team to customers, users, and stakeholders.
6. **How will the job be done technically and managerially?**
 - Defines technical and management strategies once the product scope is established.
7. **How much of each resource is needed?**
 - Estimates resources required to complete the project within budget and requirements.

Software Measurements and Metrics

Software measurements and metrics quantify attributes of the software product or process to aid decision-making and project success.

Software Measurements

- **Definition:** Indicators of size, quantity, or dimension of a product or process attribute.
- **Categories:**
 - **Direct Measures:** Cost, effort, lines of code (LOC), execution speed, error count.
 - **Indirect Measures:** Functionality, quality, complexity, reliability, maintainability.

Software Metrics

- **Definition:** Formulas or measures for software process and product aspects (e.g., performance, productivity).
- **Categories:**
 - **Product Metrics:** Measure size, complexity, quality, and reliability.
 - **Process Metrics:** Measure fault rates, testing defect patterns, and operation times.
 - **Project Metrics:** Measure developer count, cost, scheduling, and productivity.

Principles of Software Measurement

1. **Formulation:** Derive appropriate measures and metrics.
2. **Collection:** Gather data to compute metrics.
3. **Analysis:** Apply mathematical tools to compute metrics.
4. **Interpretation:** Evaluate metrics to gain insights into quality.
5. **Feedback:** Provide recommendations to the software team based on metric analysis.

Size Metrics

Size metrics estimate the software's size, critical for cost and effort estimation.

1. Lines of Code (LOC)

- **Definition:** Counts executable code lines, excluding comments and blank lines.
- **Use:** Estimates program size and compares programmer productivity.
- **Example:**

```
//Import header file
#include <iostream>
int main() {
    int num = 10;
    //Logic of even number
    if (num % 2 == 0) {
        cout << "It is even number";
    }
    return 0;
}
```

- Total LOC = 9 (executable lines only).
- **Advantages:**
 - Widely used for cost estimation.
 - Easy to estimate efforts.
- **Disadvantages:**
 - Cannot measure specification size.
 - Poor design may inflate LOC.
 - Language-dependent.
 - Difficult for users to understand.

- **Example Table:**

Project	LOC	Cost (SR)	Efforts (Persons/Month)	Documents	Errors
A	10,000	110	18	365	39
B	12,000	115	20	370	45
C	15,400	130	25	400	32

2. Function Points (FP)

- **Definition:** Measures functionality by counting functions in the application.
- **Attributes:**
 - External Inputs (EI): Input screens, tables.
 - External Outputs (EO): Output screens, reports.
 - External Inquiries (EQ): Prompts, interrupts.
 - Internal Logical Files (ILF): Databases, directories.
 - External Interface Files (EIF): Shared databases, routines.
- **Calculation:**
 - Count each attribute, assign weights, and compute Unadjusted Function Count (UFC).
 - Apply Complexity Adjustment Factor (CAF) to get FP.
- **Example:**

Information Domain	Optimistic	Likely	Pessimistic	Est. Count	Weight	FP Count
# of Inputs	22	26	30	26	4	104
# of Outputs	16	18	20	18	5	90
# of Inquiries	16	21	26	21	4	84
# of Files	4	5	6	5	10	50
# of External Interfaces	1	2	3	2	7	14
UFC						342
CAF						1.17
FP						400

- **Advantages:**
 - Requires detailed specifications.
 - Not restricted to code.
 - Language-independent.
- **Disadvantages:**
 - Ignores quality issues.
 - Subjective counting relies on estimation.

Software Project Estimation

Software project estimation predicts the time, effort, and cost required to complete a project, critical for preventing project failure.

Responsible Persons

- Software Manager
- Cognizant Engineers
- Software Estimators

Factors Affecting Estimation

1. **Cost:** Ensure sufficient funds to avoid project failure.
2. **Time:** Estimate overall duration and task timings to manage client expectations.
3. **Size and Scope:** Identify all tasks to ensure adequate materials and expertise.
4. **Risk:** Predict potential events and their severity to create risk management plans.
5. **Resources:** Ensure availability of tools, people, hardware, and software.

Steps of Project Estimation

1. **Estimate Project Size:**
 - Use LOC or FP, based on customer requirements, SRS, and system design documents.
 - Methods: Estimation by analogy (past projects) or dividing the system into subsystems.
2. **Estimate Efforts:**
 - Calculate person-hours or person-months for activities (design, coding, testing, documentation).
 - Methods:
 - Use historical organizational data for similar projects.
 - Apply algorithmic models (e.g., COCOMO) for unique projects.
3. **Estimate Project Schedule:**
 - Define the Work Breakdown Structure (WBS) to assign tasks, start/end dates, and team members.
 - Convert efforts to calendar months using: **Schedule (months) = 3.0 * (man-months)^(1/3)** (The constant 3.0 varies by organization).
4. **Estimate Project Cost:**
 - Include labor costs (effort hours * labor rate) and other expenses (hardware, software, travel, training, office space).
 - Use specific labor rates for different roles for accuracy.

Decomposition Techniques

Decomposition techniques break down the project into manageable parts for accurate estimation.

1. Software Sizing

- **Challenge:** Accurately estimate the product size.
- **Factors for Accuracy:**
 - Correct size estimation.
 - Translation of size into effort, time, and cost.
 - Reflection of team abilities in the plan.
 - Stability of requirements and environment.
- **Approaches:**
 - **Fuzzy Logic Sizing:** Uses application type and historical data.
 - **Function Point Sizing:** Estimates based on information domain characteristics.
 - **Standard Component Sizing:** Estimates size of subsystems, modules, or screens using historical data.
 - **Change Sizing:** Estimates modifications to existing software (reuse, add, change, delete code).

2. Problem-Based Estimation

- **Method:** Uses LOC or FP to size software elements and project costs/efforts.
- **Steps:**

- Size each element using LOC or FP.
- Use historical data to project costs and efforts.
- Compute expected value: $S = (S_{opt} + 4S_m + S_{pess}) / 6$ (S = size, S_{opt} = optimistic, S_m = most likely, S_{pess} = pessimistic).

• **Example:**

Function	Estimated LOC
User Interface (UICF)	2,300
2D Geometric Analysis	5,300
3D Geometric Analysis	6,800
Database Management	3,350
Graphics Display	4,950
Peripheral Control	2,100
Design Analysis	8,400
Total	33,200

3. Process-Based Estimation

- **Method:** Decomposes the process into tasks and estimates effort for each.
- **Steps:**
 - Identify software functions and process activities.
 - Estimate effort (person-months) for each activity per function.
 - Apply labor rates to calculate costs (senior staff may have higher rates).
- **Importance:** Ensures detailed task-level estimation for accuracy.

Software Cost Estimation

Software cost estimation predicts the approximate cost before development begins, considering multiple factors.

Factors for Project Budgeting

- Specification and scope
- Location
- Duration
- Team efforts
- Resources

Tools and Techniques

1. **Expert Judgment:**

- Leverages experienced professionals' insights for similar projects.

2. **Analogous Estimation:**

- Uses historical data from similar projects (scope, budget, size).
- Cost-effective but less accurate, used in early phases.

3. **Parametric Estimation:**

- Uses statistical models to estimate man-hours based on past project data.

4. **Bottom-Up Estimation:**

- Divides project into work packages (WBS), estimates each, and sums for total cost.
- Time-consuming but highly accurate.

5. **Three-Point Estimation (PERT):**

- Uses optimistic, most likely, and pessimistic estimates to handle uncertainties.

6. Reserve Analysis:

- Allocates reserve budget for unforeseen events, approved by sponsors.

7. Cost of Quality:

- Includes costs to prevent and address failures during and after the project.

8. Vendor Bid Analysis:

- Compares multiple vendor bids to estimate project cost.

Typical Problems

1. **Complexity:** Large projects are hard to estimate accurately, especially early on.
2. **Inexperience:** Estimators may lack sufficient experience.
3. **Bias:** Tendency to underestimate, especially by senior professionals.
4. **Oversights:** Forgetting integration and testing costs in large projects.
5. **Management Pressure:** Demanding precise estimates for bids or funding.

COCOMO Model

The **Constructive Cost Model (COCOMO)**, developed by Barry Boehm in 1981, estimates effort, cost, and schedule based on software size (KLOC).

Software Project Types

1. Organic:

- Small, simple projects (2–50 KLOC).
- Small, experienced team; well-understood problem.
- Examples: Simple inventory or data processing systems.

2. Semidetached:

- Medium-sized projects (50–300 KLOC) with mixed requirements.
- Mixed team experience; some known/unknown modules.
- Examples: Database management, complex inventory systems.

3. Embedded:

- Large, complex projects (>300 KLOC) with fixed requirements.
- Large team, often less experienced; high complexity.
- Examples: ATMs, air traffic control, banking software.

Types of COCOMO Models

1. Basic COCOMO:

- Static model for quick, rough estimates based on KLOC.
- **Formulas:**
 - Effort (E) = $a * (KLOC)^b$ Man-Months (MM)
 - Scheduled Time (D) = $c * (E)^d$ Months
- **Constants:**

Project Type	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

- **Example** (Semidetached, 300 KLOC):
 - $E = 3.0 * (300)^{1.12} = 1784.42 \text{ MM}$
 - $D = 2.5 * (1784.42)^{0.35} = 34.35 \text{ Months}$
 - $\text{Persons} = E / D = 1784.42 / 34.35 \approx 52$

2. Intermediate COCOMO:

- Enhances accuracy by including cost drivers (product, hardware, resource, project parameters).
- **Formulas:**
 - $\text{Effort (E)} = a * (\text{KLOC})^b * \text{EAF MM}$
 - $\text{Scheduled Time (D)} = c * (E)^d \text{ Months}$
- **Effort Adjustment Factor (EAF):** Product of cost driver values (ideal = 1).
- **Example** (Semidetached, 300 KLOC, very high application experience = 0.82, very low programming experience = 1.14):
 - $\text{EAF} = 0.82 * 1.14 = 0.9348$
 - $E = 3.0 * (300)^{1.12} * 0.9348 = 1668.07 \text{ MM}$
 - $D = 2.5 * (1668.07)^{0.35} = 33.55 \text{ Months}$

3. Detailed COCOMO:

- Applies Basic/Intermediate COCOMO to each software engineering phase (planning, system design, detailed design, coding/testing, integration, cost model).
- Divides software into modules, estimates effort per module, and sums for total effort.
- **Example** (Distributed MIS System):
 - Database (Semidetached)
 - GUI (Organic)
 - Communication (Embedded)
 - Estimate costs separately and sum for total cost.

Advantages of COCOMO

- Systematic estimation at different development stages.
- Identifies key cost and effort factors.
- Leverages historical project data.
- Easy to implement with various factors.

Disadvantages of COCOMO

- Ignores requirements, customer skills, and hardware issues.
- Limits accuracy due to assumptions and averages.
- Heavily time-dependent.
- Assumes size (KLOC) is the primary cost driver, which may not always apply.

Conclusion

Software project management and scheduling require careful planning of people, product, process, and project activities. Boehm's W5HH principle guides objective setting, while measurements and metrics (LOC, FP) quantify progress. Estimation techniques (decomposition, COCOMO) predict time, effort, and cost, addressing risks and resource needs. Understanding these concepts ensures effective project execution and is critical for exam success.

Lecture 12 : Software Testing

Introduction

Software testing is a critical process in software development to ensure a product meets customer requirements, is defect-free, and performs reliably. It identifies errors, gaps, or missing requirements by evaluating attributes like reliability, scalability, portability, reusability, and usability. This guide covers software testing principles, test cases, white box and black box testing, unit testing, integration testing, and comparisons, optimized for exam preparation.

Software Testing Overview

- **Definition:** A method to verify that a software product matches expected requirements and is free of defects.
- **Purpose:**
 - Identify errors, gaps, or missing requirements.
 - Ensure the product meets customer needs and performs reliably.
 - Prevent failures that could lead to dangerous situations.
- **Importance:** Mandatory to avoid deploying faulty software to end users.
- **Process:**
 - Conducted at every phase of the Software Development Life Cycle (SDLC).
 - Performed by software testers, developers, project managers, and end users.

Principles of Software Testing

The following principles guide effective software testing:

1. **Testing Shows the Presence of Defects:**
 - Aims to identify defects that could cause product failure.
 - Cannot guarantee 100% error-free software but reduces undiscovered defects.
 - Requires well-designed test cases to maximize defect detection.
2. **Exhaustive Testing is Impossible:**
 - Testing all modules and features exhaustively is impractical.
 - Use risk analysis and prioritize critical modules to deliver on schedule.
3. **Early Testing:**
 - Involve testers from the requirement gathering phase to understand the product deeply.
 - Early defect detection saves time and cost compared to late-stage fixes.
4. **Defect Clustering:**
 - Most defects (80%) are found in a small portion (20%) of code (Pareto's 80-20 Rule).
 - Focusing on high-defect areas may miss bugs in other modules.
5. **Pesticide Paradox:**
 - Repeated use of the same test cases fails to uncover new defects.
 - Regularly update test cases to cover different software parts and find new bugs.
6. **Testing is Context-Dependent:**
 - Testing varies by project type (e.g., e-commerce, banking, commercial websites).
 - Different products require tailored test cases based on their features and requirements.
7. **Absence of Errors Fallacy:**
 - A bug-free application may still be unusable if it fails to meet user needs.

- Testing must ensure both defect-free code and alignment with client requirements.

Test Cases

- **Definition:** A set of actions or conditions to compare expected and actual results, verifying software functionality against customer requirements.
- **Components:**
 - **Test Scenario ID:** Identifies the test scenario (e.g., Login-1).
 - **Test Case ID:** Unique identifier for the test case (e.g., Login-1A).
 - **Description:** Details the test purpose (e.g., positive login test).
 - **Priority:** Importance level (e.g., High).
 - **Pre-Requisite:** Conditions required (e.g., valid user account).
 - **Post-Requisite:** Actions after testing (e.g., none).
 - **Execution Steps:** Actions, inputs, expected outputs, actual outputs, browser, and result.
- **Example (Login Test Case):**

S.No	Action	Inputs	Expected Output	Actual Output	Browser	Result
1	Launch application	https://www.facebook.com/	Facebook home	Facebook home	IE-11	Pass
2	Enter email & password, hit login	Email: test@xyz.com , Password	Login success	Login success	IE-11	Pass

White Box Testing

- **Definition:** Testing that analyzes the **internal code structure, design, and functionality, requiring knowledge of** the software's programming.
- **Also Known As:** Clear box, open box, transparent box, code-based, or glass box testing.
- **Performed By:** Software developers.
- **Testing Levels:** Unit testing and integration testing.
- **Tools:** EclEmma, NUnit, PyUnit, HtmlUnit, CppUnit.
- **Verification Areas:**
 - Internal code security.
 - Poorly structured code paths.
 - Input flow through code.
 - Loops, decision conditions, and statements.
 - Individual functions and modules.
 - Expected outputs.

Techniques

1. **Path Coverage/Testing:**
 - Tests all possible paths from entry to exit based on the program's control flow.
 - Example: Functions $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow H \rightarrow I$.
2. **Loop Testing:**
 - Verifies simple, nested, and concatenated loops (e.g., while, for, do-while).
 - Checks loop conditions and termination.
 - Example: `while (condition) { statement(s); }`.

3. Branch Coverage/Condition Testing:

- Tests logical conditions (true/false) for if and else branches.
- Ensures all decision points are covered.

4. Statement Coverage:

- Executes every code statement at least once to verify functionality.
- Example:

```
Prints(int a, int b) {  
    int result = a + b;  
    if (result > 0)  
        Print("Positive", result);  
    else  
        Print("Negative", result);  
}
```

- Tests ensure lines 1–6 are executed.

Advantages

- **Optimizes code** by identifying hidden errors.
- Easily **automated test cases**.
- **Early error detection** improves code quality.
- Covers most **code paths**.

Disadvantages

- **Complex, expensive, and time-consuming**.
- Requires **skilled programmers**.
- Cannot **detect missing functionalities**.
- Code redesign **requires rewriting test cases**.

Black Box Testing

- **Definition:** Testing functionalities **without knowledge of internal** code structure, **focusing on inputs and outputs**.
- **Also Known As:** Behavioral, functional, or closed box testing.
- **Performed By:** Software testers.
- **Testing Levels:** System and acceptance testing.
- **Tools:** QTP, Selenium, LoadRunner, JMeter.
- **Types:**
 - **Functional Testing:** Tests features and functionalities.
 - **Non-Functional Testing:** Tests performance, usability, scalability, etc.
 - **Regression Testing:** Ensures new changes do not affect existing functionalities.

Techniques

1. Equivalence Partitioning:

- Divides input values into classes with similar outcomes.
- Tests one value per class to reduce test cases.
- Example: Age (18–60) → Invalid: ≤ 17 , Valid: 18–60, Invalid: ≥ 61 .

2. Boundary Value Analysis (BVA):

- Tests boundary values (upper/lower limits) of input ranges.

- Example: Age (18–60) → Test: 17 (invalid), 18, 19, 59, 60 (valid), 61 (invalid).

3. Decision Table Testing:

- Captures input combinations and system behavior in a table.
- Example (Gmail Login):

Email (C1)	Password (C2)	Expected Result
True	True	Account Page
True	False	Incorrect password
False	True	Incorrect email
False	False	Incorrect email

4. Error Guessing:

- Uses tester experience to identify problematic areas.
- Examples: Divide by zero, null values, empty submit, invalid file uploads.

5. State Transition Testing:

- Tests behavior for different inputs to the same function.
- Example: Limited login attempts (e.g., lock account after 3 failed tries).

6. All Pairs Testing:

- Tests discrete combinations of inputs (e.g., checkboxes, radio buttons).
- Reduces test cases for combinatorial inputs.

Advantages

- No programming knowledge required.
- **Efficient for large systems.**
- Tests from the **user's perspective**.
- **Identifies** specification **ambiguities**.

Disadvantages

- **Difficult to design test cases** without code knowledge.
- Cannot detect control **structure errors**.
- Exhaustive input testing is **time-consuming**.

Black Box vs. White Box Testing

Black Box testing	White Box testing
① The internal structure and working of the software is unknown to the tester	① known to the tester
② known as closed-box testing, Functional testing, Behavioral testing	② Open-box testing, code based testing, Structural testing
③ performed by software testers	③ developers
④ required less programming knowledge	④ complete programming knowledge
⑤ Less time consuming	⑤ more
⑥ used for fulfill customer need	⑥ check code quality
⑦ done by higher levels of testing	⑦ Lower levels
⑧ Example: searching something on Google but with no intention	⑧ By input to check & verify code.

Aspect	Black Box Testing	White Box Testing
Knowledge	No internal structure knowledge; focuses on input/output.	Knows internal structure and code.
Also Known As	Functional, data-driven, closed-box, behavioral.	Structural, glass box, code-based, transparent.
Programming Knowledge	Minimal required.	Complete knowledge required.
Testing Levels	System, acceptance testing.	Unit, integration testing.
Performed By	Software testers.	Software developers.
Time	Less time-consuming.	More time-consuming.
Basis	External expectations.	Internal code workings.
Testing Approach	Trial and error; tests data domains.	Tests internal boundaries and code paths.
Example	Search on Google.	Verify loops with input keywords.

Unit Testing

- **Definition:** The first level of testing, where individual units (functions, methods, modules, or objects) are tested in isolation.
- **Also Known As:** Component testing.
- **Performed By:** Software developers.
- **Testing Technique:** White box testing.
- **SDLC Phase:** Coding phase.
- **Tools:** JUnit, NUnit, PHPUnit, EMMA, JMockit.

Purpose

- Verify code correctness and enable quick changes.
- Test every function and procedure.
- Fix bugs early to save costs.

- Aid documentation and code reuse.
- Improve software efficiency.

Unit Testing in Object-Oriented Context

- Tests packages, classes, methods, subclasses, and attributes (public, private, protected).
- Example: Test individual modules (e.g., login, search, payment) in a project.

Advantages

- Modular testing without waiting for other components.
- Focuses on unit functionality.
- Early issue detection improves quality.
- Enhances development efficiency.

Disadvantages

- Time-consuming to create and maintain test cases.
- Limited to individual units, not interactions.
- Requires ongoing maintenance with code changes.

Integration Testing

- **Definition:** The second level of testing, where integrated modules are tested as a group to verify communication and functionality.
- **Also Known As:** Thread testing, string testing.
- **Performed By:** Developers and testers.
- **Goal:** Ensure correctness and interaction among modules.
- **Tools:** Selenium, PyTest, JUnit, Jasmine, Steam, Mockito.

Purpose

- Verify differing programming logic across modules.
- Check database interactions.
- Address untested requirement changes.
- Detect module incompatibilities.
- Ensure hardware-software compatibility.

Example

- **Gmail Application:**
 - User 1: Logs in, composes, and sends mail to User 2; saves to Draft or Sent Items.
 - User 2: Logs in, checks Inbox, verifies mail receipt, replies if needed, logs out.

Types

1. Incremental Integration Testing:

- Modules are integrated and tested one by one in ascending order.
- Tests data flow and function correctness.
- **Subtypes:**
 - **Top-Down:**
 - Starts with top-level modules, moving downward.
 - Uses **stubs** (dummy programs) for missing lower modules.

- Prioritizes critical modules for early flaw detection.
- **Bottom-Up:**
 - Starts with lowest modules, moving upward.
 - Uses **drivers** (dummy programs) for missing higher modules.
 - Allows simultaneous testing of subsystems.
- **Example** (Flipkart Application):
 - Flow: Login → Home → Search → Add to Cart → Payment → Logout.

2. Non-Incremental Integration Testing (Big Bang Testing):

- Integrates and tests all modules at once after individual testing.
- Suitable for smaller systems.
- Difficult to pinpoint errors due to lack of parent-child hierarchy.

Incremental vs. Non-Incremental Testing

Aspect	Incremental Testing	Non-Incremental Testing
Integration Approach	Tests modules gradually.	Tests all modules at once.
Planning	Requires step-by-step planning.	Simpler, one-time planning.
Resource Efficiency	Uses more resources (separate tests).	Uses fewer resources (single test).
Issue Detection	Early detection with progressive testing.	Late detection due to bulk testing.
Complexity	Manages smaller pieces, less complex.	More complex due to testing everything.

Conclusion

Software testing ensures a product is reliable, functional, and meets user needs. Key principles guide defect detection, while test cases verify functionality. White box testing examines code structure, black box testing focuses on functionality, unit testing isolates components, and integration testing verifies module interactions. Understanding these concepts, techniques, and examples is critical for exam success and effective software development.

Testing and Quality Assurance Exam Notes

1. System Testing

- **Definition:** Validates the fully integrated software product to ensure it meets end-to-end specifications.
- **Type:** Black-box testing by Quality Assurance (QA) team during testing phase.
- **Focus:** Functionality, accuracy, quality, expected output, overall behavior (not internal workings).
- **Tools:** Selenium, LoadRunner, JMeter, Microsoft Test Manager, SoapUI.
- **Tool Choice Factors:** Technology, project size, budget, testing requirements.
- **Example:** Testing an e-commerce app's checkout process for correct functionality.
- **Key Point:** Ensures the complete system works as intended.

2. Importance of System Testing

- **Benefits:**
 - **Improved Quality:** Works across platforms/environments.
 - **Error Reduction:** Exposes errors missed in unit/integration testing.
 - **Cost Savings:** Reduces unexpected costs/delays.
 - **Security:** Identifies vulnerabilities to protect user data.
 - **Customer Satisfaction:** Enhances user experience, builds confidence.

- **Performance:** Tracks memory, CPU usage, and system behavior.
- **Example:** System testing ensures a banking app is secure and performs well.
- **Key Point:** Critical for quality, security, and cost-effective delivery.

3. Types of Software Testing

- **Performance Testing:** Measures speed, load time, stability, reliability, response times.
 - **Example:** Testing app response under heavy user load.
- **Load Testing:** Evaluates performance under real-life extreme loads (e.g., throughput, user count).
 - **Example:** Simulating 10,000 users on a website.
- **Usability Testing:** Assesses ease of use, user error rates, task success, and satisfaction.
 - **Example:** Checking if users can navigate an app intuitively.
- **Regression Testing:** Ensures new changes don't introduce defects or reintroduce old bugs.
 - **Example:** Retesting after a software update.
- **Migration Testing:** Verifies system works after infrastructure changes.
 - **Example:** Moving an app to a new server without issues.
- **Functional Testing:** Identifies missing functions to improve system quality.
 - **Example:** Checking if all required features (e.g., login) work.
- **Recovery Testing:** Tests system recovery from errors/crashes.
 - **Example:** Ensuring an app restarts after a crash.
- **Stress Testing:** Tests robustness under extreme loads.
 - **Example:** Testing app behavior with overloaded servers.
- **Software & Hardware Testing:** Checks compatibility between software and hardware.
 - **Example:** Ensuring an app runs on specific devices/OS.
- **Key Point:** Each type targets specific system aspects for comprehensive testing.

4. System Testing Example

- **Test Cases:**
 - **Functionality:** Input field accepts up to 20 characters (Expected: All characters valid).
 - **Security:** Password rules enforced (Expected: Valid passwords accepted).
 - **Usability:** Links work correctly (Expected: Links navigate to correct pages).
- **Example:** Testing a login page for character limits, password security, and link functionality.
- **Key Point:** Test cases verify specific system behaviors.

5. Acceptance Testing

- **Definition:** Final testing before release, performed by end-users/clients.
- **Type:** Black-box testing (also called User Acceptance Testing, Functional Acceptance Testing, Red Box Testing).
- **Environment:** User/live environment or real-time scenarios.
- **Purpose:** Ensures software meets user expectations and requirements.
- **Tools:** Fitness Tools, Watir.
- **Example:** Client testing an app's payment feature in a live-like setup.
- **Key Point:** Validates software for real-world use.

6. Importance of Acceptance Testing

- **Benefits:**
 - Identifies bugs missed during development.
 - Confirms product meets client/user expectations.
 - Builds client confidence through direct involvement.
 - Ensures bug-free delivery.
 - Satisfies SRS functionalities.
- **Example:** Ensuring a shopping app meets client needs before launch.
- **Key Point:** Critical for client satisfaction and final validation.

7.(types of acceptance testing)

- **User Acceptance Testing (UAT):**
 - Performed from end-user perspective.
 - Checks if software meets user requirements in a production-like environment.
 - Focuses on functionality, not bugs.
 - **Example:** Users testing a mobile app's navigation.
- **Business Acceptance Testing (BAT):**
 - Verifies software meets business requirements and operational needs.
 - Focuses on business risks and financial factors.
 - **Example:** Ensuring an app supports business goals in a dynamic market.
- **Regulations Acceptance Testing (RAT):**
 - Ensures compliance with regional rules/regulations.
 - Non-compliance holds the product owner accountable.
 - **Example:** Checking if an app adheres to GDPR regulations.
- **Key Point:** Each type ensures specific acceptance criteria are met.

8. Alpha vs. Beta Testing

- **Alpha Testing:**
 - By internal testers (skilled employees).
 - Uses white-box and black-box techniques.
 - Focuses on bugs/errors, not in-depth reliability/security.
 - Long execution cycles, done near development end.
 - **Example:** Internal team testing app before beta.
- **Beta Testing:**
 - By clients/end-users in real-time environments.
 - Black-box testing only.
 - Checks reliability, security, robustness.
 - Short cycles (few weeks), final test before release.
 - **Example:** Users testing a game app for real-world performance.
- **Key Point:** Alpha improves quality pre-beta; beta ensures readiness for users.
-

Alpha testing	Beta testing
① It is performed by highly skilled testers.	① Performed by client.
② It involves both white box & black box testing.	② only black box testing.
③ execution life cycle is long.	③ only few weeks.
④ focus on finding bugs.	④ reliability security.
⑤ Critical issues identified by developer.	⑤ feedback collected from users.
⑥ check product quality before beta testing.	⑥ before real time use.
⑦ last testing in development.	⑦ last testing before release.

9. Verification vs. Validation

- **Verification:**

- "Are we building the software right?" (Static testing).
- Checks documents, design, code by QA/developers.
- Methods: Inspections, reviews, walkthroughs.
- Finds bugs early, no code execution.
- **Example:** Reviewing code for correct syntax.

- **Validation:**

- "Are we building the right software?" (Dynamic testing).
- Tests actual product by testing team.
- Includes functional, system, integration, UAT.
- Involves code execution, catches missed bugs.
- **Example:** Testing a chat app's features (chatting, calls, sharing).

- **Key Point:** Verification ensures correctness; validation ensures user needs are met.

10. Defect/Bug

- **Definition:** Errors causing abnormal software behavior due to design/coding issues.
- **Variation:** Difference between actual and expected results.
- **Defect Life Cycle:** States a defect goes through for systematic fixing.
- **Performed By:** Developers and testers.
- **Tools:** JIRA, Trac, Redmine.
- **Example:** A login failure due to incorrect code is a defect.
- **Key Point:** Defects disrupt functionality; life cycle ensures efficient fixes.

11. Defect Life Cycle

- **Stages:**

- **New:** Tester identifies defect, sends document to developers.
- **Assigned:** Defect assigned to developer team.
- **Open:** Developers fix or mark as Duplicate/Rejected/Deferred.
- **Fixed:** Developer corrects code to remove defect.
- **Retest:** Testers verify if defect is fixed.
- **Reopened:** If defect persists, cycle restarts.
- **Verified:** Tester confirms defect is fixed.
- **Closed:** Issue closed after verification.
- **Example:** A bug in payment processing is identified, fixed, and verified.
- **Key Point:** Systematic process to track and resolve defects.

12. Testing vs. Debugging

- **Testing:**
 - Finds bugs/errors, done by testers.
 - Manual or automated, based on testing levels (unit, integration, system).
 - No programming knowledge required.
 - Part of SDLC, post-coding.
 - **Example:** Testing a form for input errors.
- **Debugging:**
 - Fixes bugs found during testing, done by developers.
 - Always manual, requires programming knowledge.
 - Subset of testing, starts with test case execution.
 - **Example:** Fixing code causing form input errors.
- **Key Point:** Testing identifies issues; debugging resolves them.

13. Software Quality

- **Definition:** Software's ability to function as per user requirements and SRS.
- **Aspects:**
 - **Good Design:** Attractive visualization.
 - **Durability:** Long-term functionality.
 - **Consistency:** Works across platforms/devices.
 - **Maintainability:** Easy bug fixes/feature additions.
 - **Value for Money:** Worth the investment.
- **Example:** A durable, user-friendly app justifies its cost.
- **Key Point:** Ensures functionality and user satisfaction.

14. Software Quality Dimensions

- **Dimensions:**
 - **Maintainability:** Ease of modifying (features, bugs).
 - **Portability:** Transferable across locations.
 - **Functionality:** Performs specified functions.
 - **Performance:** Speed under load.
 - **Compatibility:** Works across devices/OS/browsers.

- **Usability:** Ease of use.
- **Reliability:** Error-free under stated conditions.
- **Security:** Protects against unauthorized access.
- **Example:** A secure, portable app works on multiple devices.
- **Key Point:** Dimensions define quality standards.

15. Factors Affecting Software Quality

- **Product Operation:**
 - Correctness, Reliability, Efficiency, Integrity, Usability.
- **Product Revision:**
 - Maintainability, Flexibility, Testability.
- **Product Transition:**
 - Portability, Reusability, Interoperability.
- **Example:** Usability ensures intuitive UI; portability supports multiple platforms.
- **Key Point:** Factors ensure operational, revisable, and transferable quality.

16. Software Quality Metrics

- **Customer Problem Metrics:**
 - Measures customer-reported issues.
 - Formula: $PUM = \text{Total problems} \div \text{Total license months}$.
- **Customer Satisfaction Metrics:**
 - Rates satisfaction (Very Satisfied to Very Dissatisfied).
- **Software Maintenance Metrics:**
 - Tracks defects post-release in customer environment.
- **Example:** PUM calculates issues per license month for an app.
- **Key Point:** Metrics quantify quality and satisfaction.

17. Software Quality Management (SQM)

- **Definition:** Process ensuring software meets national/international standards (e.g., ANSI, IEEE, ISO).
- **Needs:**
 - Delivers high-quality products on time.
 - Builds stakeholder trust.
 - Ensures customer satisfaction.
- **Example:** SQM ensures an app meets ISO standards for quality.
- **Key Point:** Maintains high standards for delivery and trust.

18. How to Achieve Software Quality

- **Quality Assurance (QA):**
 - Ensures system meets requirements/expectations.
 - Defines standards/methodologies for development.
 - Covers correctness, efficiency, flexibility, etc.
- **Quality Control (QC):**
 - Ensures quality parameters are met.

- Focuses on timely, cost-accurate delivery.
- **Quality Planning:**
 - Selects/modifies standards for a project-specific quality plan.
- **Example:** QA defines app standards; QC verifies functionality.
- **Key Point:** QA, QC, and planning ensure quality delivery.

19. Quality Assurance vs. Quality Control

- **Quality Assurance (QA):**
 - Proactive, process-oriented, prevents defects.
 - Defines standards, involves full SDLC.
 - Done by all team members, no code execution.
 - **Example:** Setting coding standards for an app.
- **Quality Control (QC):**
 - Reactive, product-oriented, identifies/fixes defects.
 - Verifies standards, involves testing life cycle.
 - Done by testing team, involves code execution.
 - **Example:** Testing app for defects post-development.
- **Key Point:** QA prevents issues; QC ensures product quality.