



Software Testing

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Software Testing
- ▶ Objectives of Software Testing
- ▶ Verification and Validation
- ▶ Levels of Testing

Software Testing

Software testing is the process of finding errors in the developed product.

Software testing can be stated as the process of **verifying** and **validating** whether a

- software or application is **bug-free**,
- meets the **technical requirements** as guided by its design and development, and
- meets the **user requirements** effectively and efficiently by handling all the exceptional and boundary cases.

Objectives of Software Testing

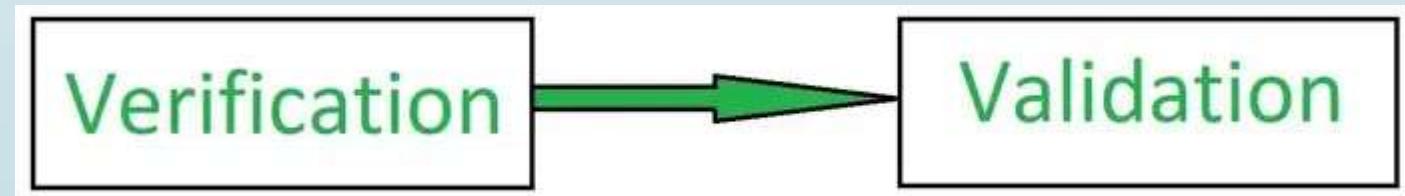
1. Identifying Defects in Software
2. Verifying Software Functionality
3. Validating User Requirements
4. Ensuring Compatibility and Integration
5. Assessing Performance and Security

Verification and Validation

Verification refers to the set of tasks that ensure that software correctly implements a specific function. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: “Are we building the product right?”

Validation: “Are we building the right product?”



Verification Techniques:

1. Static Testing
2. Walkthroughs
3. Inspections
4. Reviews
5. Peer Reviews
6. Simulation
7. Prototyping
8. Dry Runs

Static Testing

- ▶ Static testing is a software testing method that examines a program -- along with any associated documents -- but does not require the program to be executed.
- ▶ Instead of executing the code, static testing is a process of checking the code and designing documents and requirements before it's run to find errors.
- ▶ The main goal is to find flaws in the early stages of development because it is normally easier to find the sources of possible failures this way.

Formal Technical Reviews

- ▶ **Formal Technical Reviews (FTRs) or Peer Reviews** are systematic, structured, and formal processes used in software engineering to evaluate the quality and correctness of software artifacts.
- ▶ These artifacts can include requirements, **design documents, code, test plans, and other deliverables**.
- ▶ The primary goal of FTRs is to identify defects, ensure compliance with standards, and improve the quality of the software product early in the development process.

Objectives of FTRs:

- ▶ Detect and Correct Defects Early
- ▶ Ensure Conformance to Standards
- ▶ Improve Software Quality
- ▶ Facilitate Knowledge Sharing
- ▶ Verify and Validate Requirements

Types of Formal Technical Reviews:

- ▶ Inspections:

A rigorous and detailed examination of software artifacts by a team of reviewers, usually led by a trained moderator. The process involves preparation, review meetings, and follow-up.

- ▶ Walkthroughs:

A less formal review where the author of the artifact presents it to a group of peers. The focus is on gaining feedback and suggestions rather than on defect detection alone.

- ▶ Audits:

An independent review conducted to ensure compliance with standards and procedures. Audits are often performed by external parties or internal quality assurance teams.

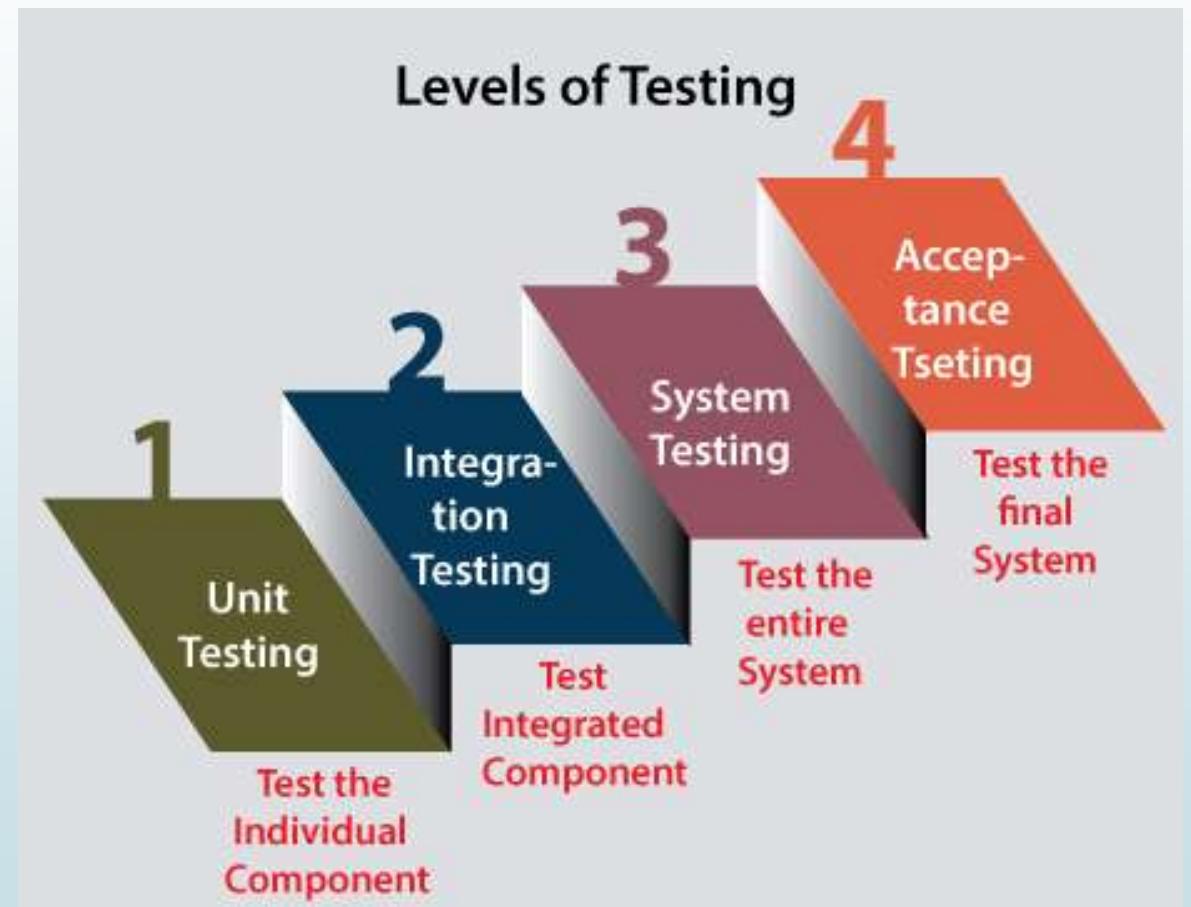
S.No.	Code Inspection	Walkthrough
1.	It is formal.	It is informal.
2.	Initiated by project team.	Initiated by author.
3.	A group of relevant persons from different departments participate in the inspection.	Usually team members of the same project take participation in the walkthrough. Author himself acts walkthrough leader.
4.	Checklist is used to find faults.	No checklist is used in the walkthrough.
6.	Formalized procedure in each step.	No formalized procedure in the steps.
7.	Inspection takes longer time as list of items in checklist is tracked to completion.	Shorter time is spent on walkthrough as there is no formal checklist used to evaluate program.
8.	Planned meeting with the fixed roles assigned to all the members involved.	Unplanned
9.	Reader reads product code. Everyone inspects it and comes up with defects.	Author reads product code and his teammate comes up with the defects or suggestions.
10.	Recorder records the defects.	Author make a note of defects and suggestions offered by teammate.
11.	Moderator has a role that moderator making sure that the discussions proceed on the productive lines.	Informal, so there is no moderator.

Validation Techniques:

1. Testing (Unit, Integration, System, Acceptance)
2. Black-Box Testing
3. White-Box Testing
4. Regression Testing
5. Alpha Testing
6. Beta Testing
7. User Acceptance Testing (UAT)
8. Stress Testing
9. Load Testing

Levels of Testing

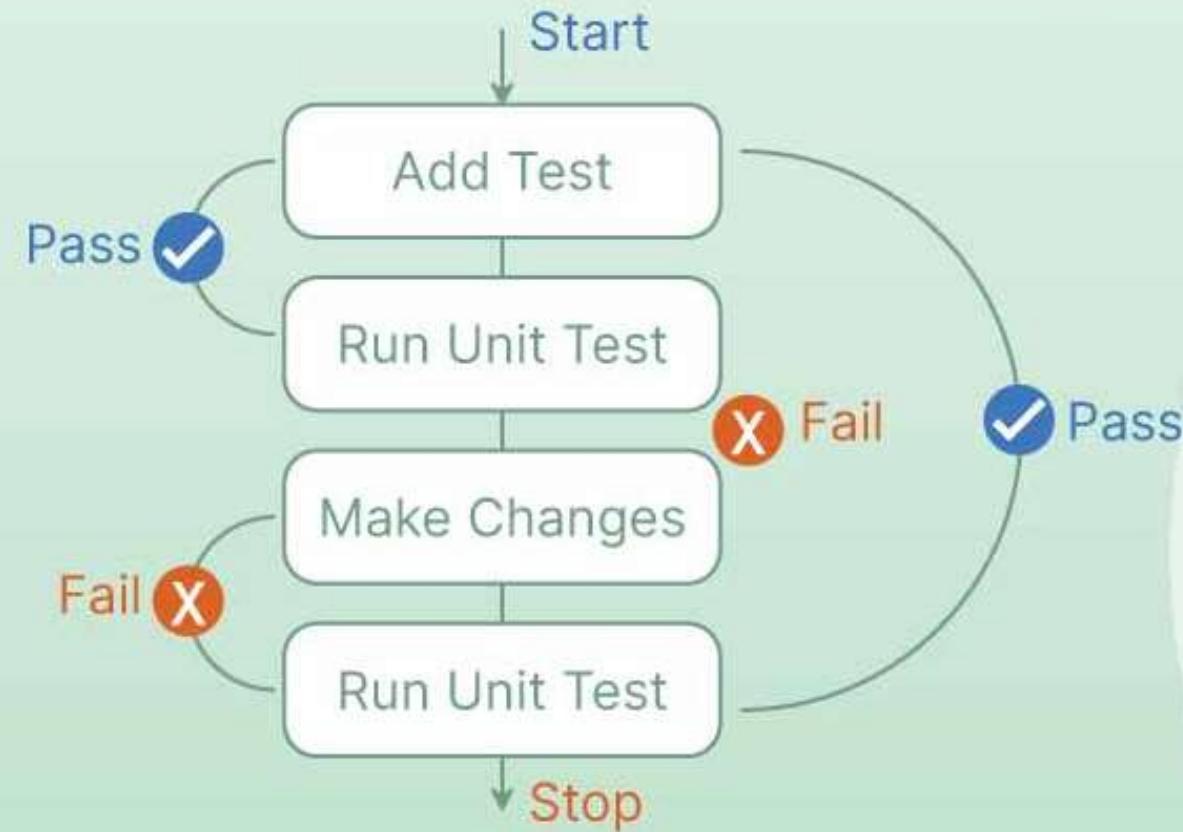
1. Unit Testing
2. Integration Testing
3. System testing
4. Acceptance Testing



1. Unit Testing

- ▶ Unit testing is a type of software testing that focuses on **individual units or components** of a software system.
- ▶ The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements.
- ▶ Developers typically perform unit testing, and it is performed early in the development process before the code is integrated and tested as a whole system.

Unit Testing In Software Engineering

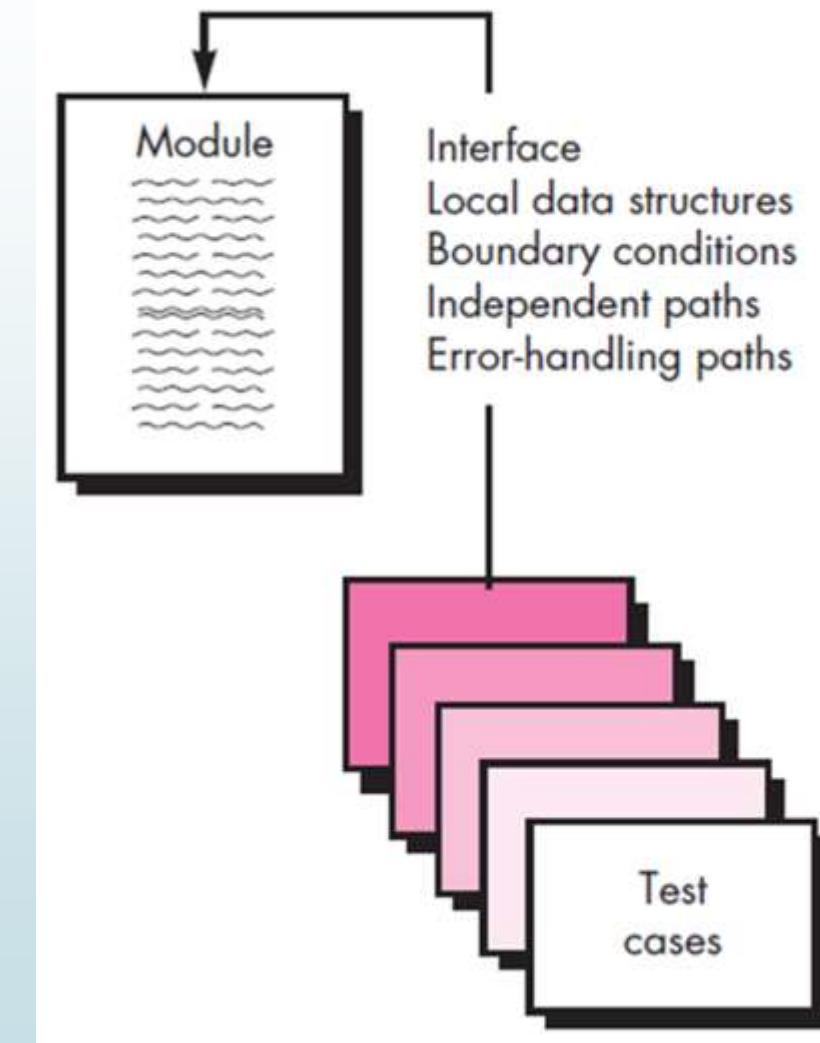


Real life example

- ▶ Suppose a car manufacturer is making a car. He will make different parts like its engine, ignition system, wheels, etc. These **different parts are the units** that he will test individually. The manufacturer will check if the ignition system and the engine are working fine or not.
- ▶ Suppose the developer is making an e-commerce website. This website has modules like a **login page, registration page, cart, payment page, dashboard**, etc. The developer will check whether these modules are working fine or not. He will check whether the user can log in from Gmail, yahoo mail, etc. And in the same way for the payment page, he will check whether the user will be able to make payment through Gray, debit card, credit card, or cash on delivery, etc.

How unit testing works?

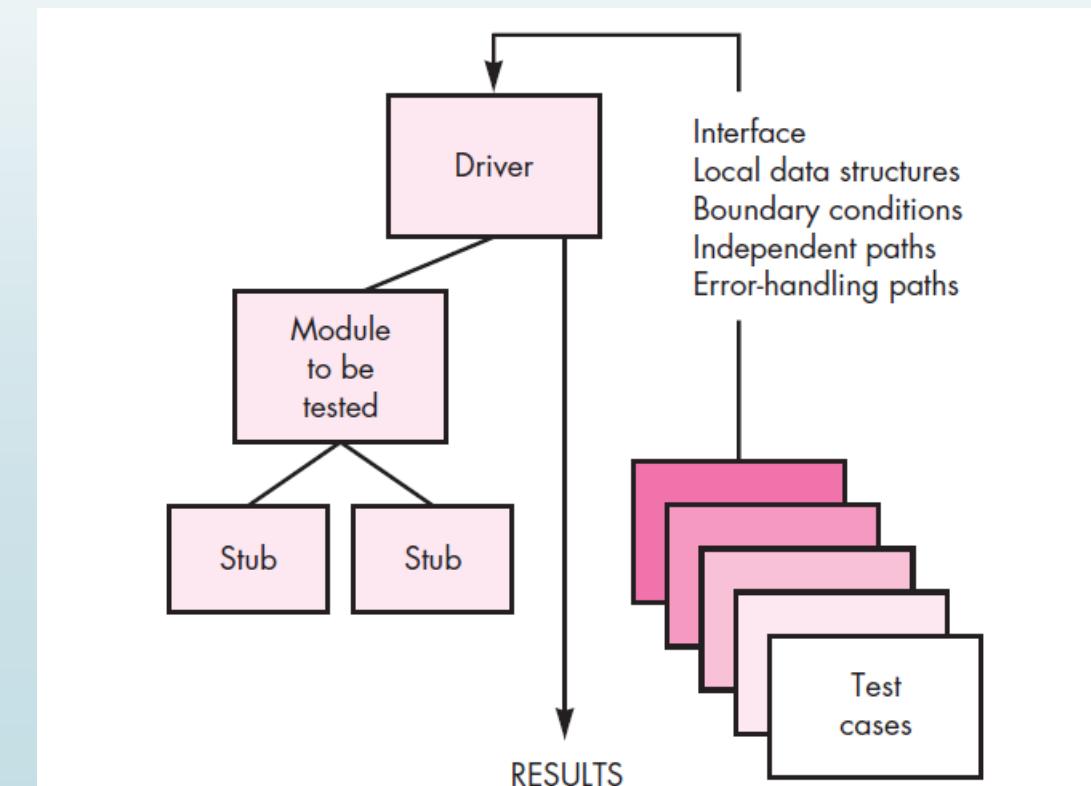
- ▶ You can first write unit tests as code.
- ▶ Then, run that test code automatically every time you make changes in the software code.
- ▶ This way, if a test fails, you can quickly isolate the area of the code that has the bug or error.



Strategies of Unit Testing:

- ▶ **Interface Testing:** Interface testing focuses on testing the interfaces of units, such as classes or modules, to ensure they conform to their specifications. This includes testing method signatures, parameter types, return types, and exception handling.
- ▶ **Local Data Structures:** Unit tests should test the behavior of local data structures used within units, such as arrays, lists, or maps. This includes testing edge cases and boundary conditions to ensure the data structure behaves as expected in all scenarios.
- ▶ **Boundary Value Conditions:** Boundary value testing involves testing input values at or near the boundaries of valid ranges. This helps identify potential off-by-one errors or other issues that might occur at boundary conditions.
- ▶ **Independent Paths:** Independent path testing aims to test all logical paths through a unit, ensuring that each path is exercised at least once. This helps uncover errors in conditional logic or branching statements.
- ▶ **Error Handling:** Error handling testing focuses on testing how a unit handles errors or exceptional conditions. This includes testing that exceptions are thrown when expected, and that error messages or recovery mechanisms are correct.

- ▶ Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.
- ▶ The unit test environment is illustrated in Figure.
- ▶ In most applications a driver is nothing more than a “main program” that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- ▶ Stubs serve to replace modules that are subordinate (invoked by) the component to be tested.

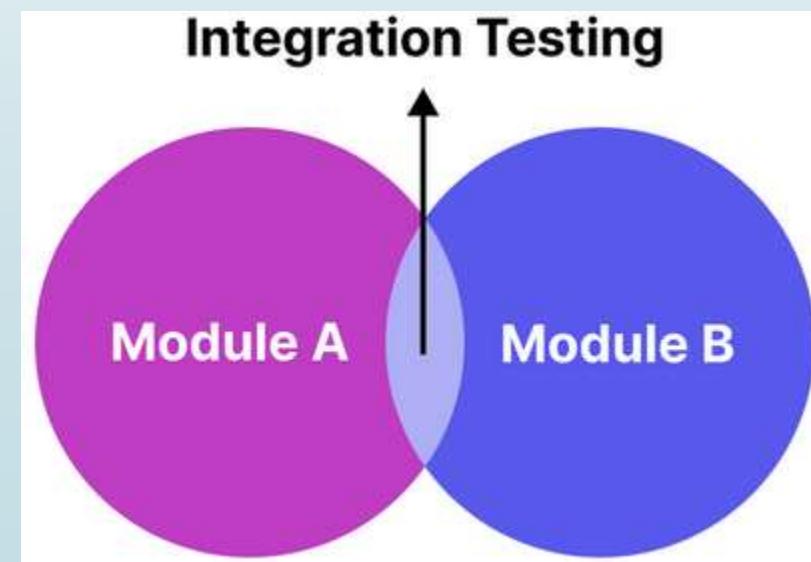


Purpose of Unit testing

1. Check code correctness.
2. Test all functions and operations.
3. Fix bugs early in the development cycle and save money.
4. To help developers understand the code base and make changes quickly.
5. Useful for code reuse.

2. Integration Testing

- ▶ Integration testing is the process of testing the interface between two software units or modules.
- ▶ The goal of integration testing is to identify any problems or bugs that arise when different components are combined and interact with each other.
- ▶ Once all the modules have been unit-tested, integration testing is performed.



Integration testing is carried out by 2 different methods:

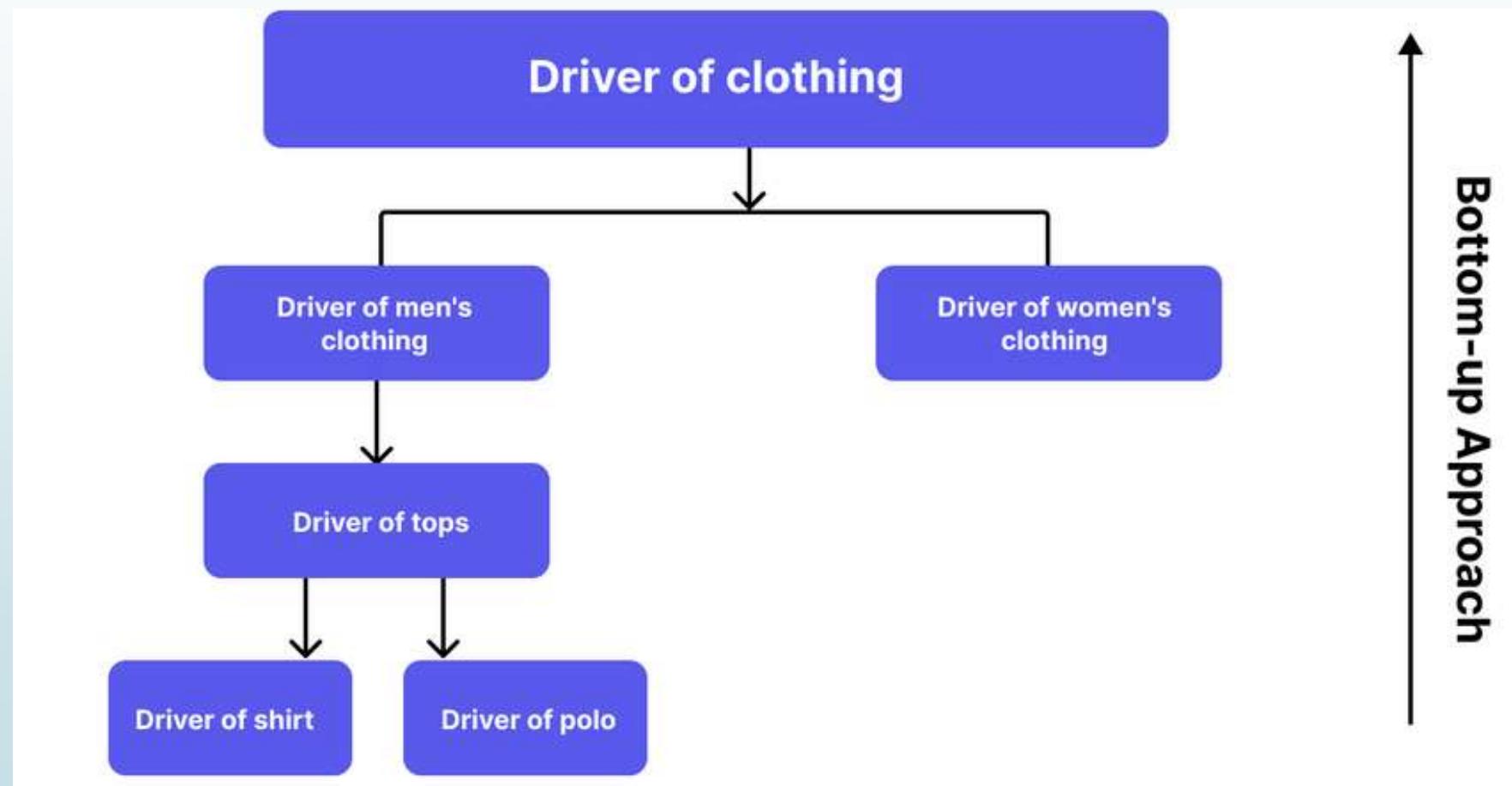
1. Bottom-up approach
2. Top-down approach

1. **Bottom-up approach**

- With the bottom-up approach, testers start with individual modules at the lowest level, then gradually move to higher-level modules, hence the term “bottom-up”.
- The rationale for this approach is that the entire system can only work reliably if these basic building blocks work reliably.

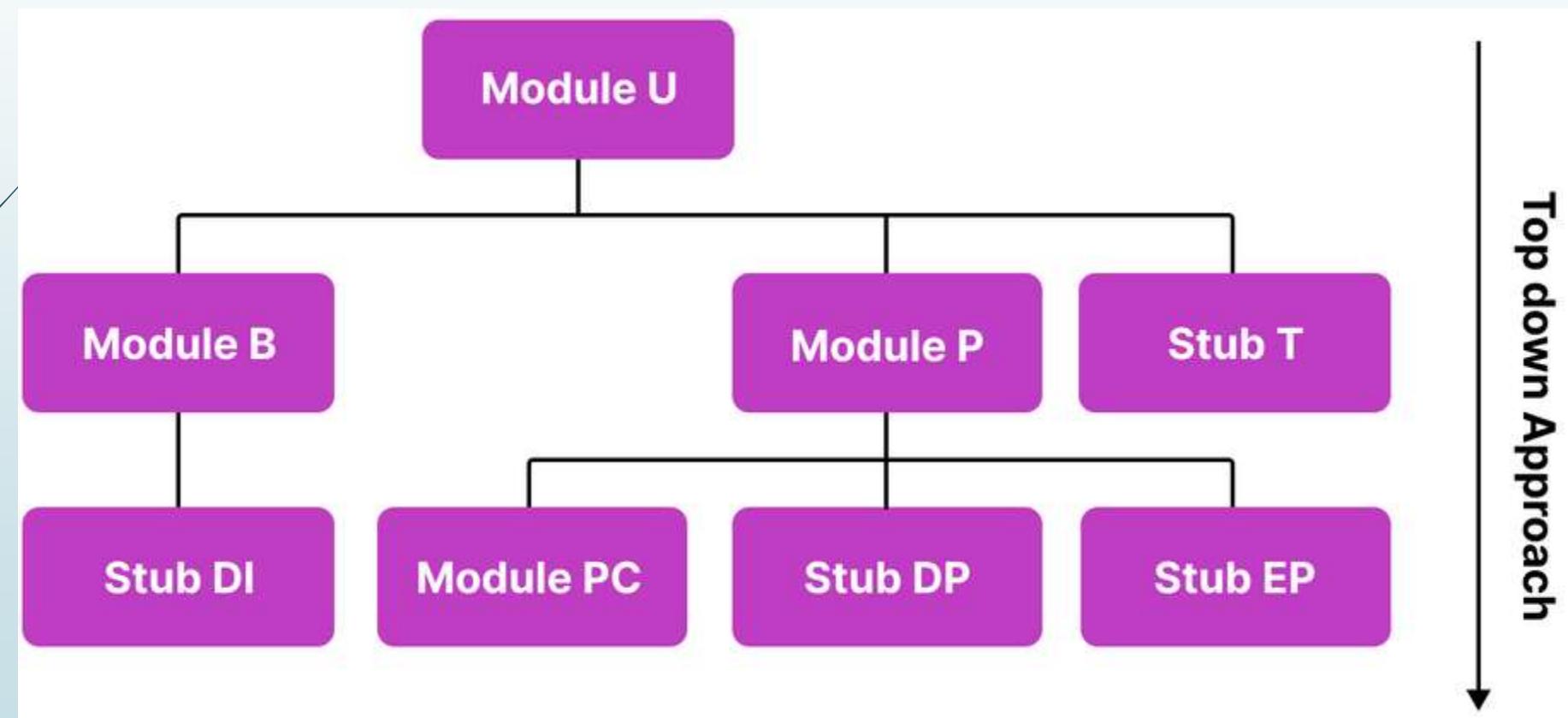
23

- A “bottom-up” approach is essentially going from the more specific and granular components to more general and comprehensive components.



2. Top-Down approach

- With the top-down approach, testers start with the highest-level modules, then gradually move to lower-level modules, hence the term “top-down”.



We should use bottom-up integration testing when:

- ▶ Complexity is primarily found in lower-level modules
- ▶ The team follows incremental development (which involves developing lower-level components first before moving to higher-level modules)
- ▶ Defect localization is a crucial aspect of the project since the granularity of the bottom-up approach provides more exact bug isolation
- ▶ Higher-level modules are still under development or likely to change frequently

Similarly, we should use top-down integration testing when:

- ▶ The critical functionalities are primarily concentrated in the higher-level components
- ▶ When it's crucial to simulate real-world scenarios and user interactions, top-down testing provides a more holistic view of the system's behavior early on.
- ▶ Lower-level modules are more well-defined, stable, and unlikely to change significantly
- ▶ Top-down testing allows for faster validation of user-facing features, enabling early prototypes for user feedback.

Note that when choosing the bottom-up testing approach, we may need to build a **stub** to substitute for high-level modules that are not yet developed, while for the top-down testing approach, we may need to build a **driver** to replace the unavailable low-level modules.

3. System Testing

- System testing is a type of software testing that evaluates the overall functionality and performance of a complete and fully integrated software solution.

Types of System Testing

1. **Performance Testing:** Performance Testing is a type of software testing that is carried out to test the speed, stability and reliability of the software product or application.
2. **Load Testing:** Load Testing is a type of software Testing which is carried out to determine the behavior of a system or software product under extreme load. Ex: Simulating 1,000 concurrent users on a website to check if it performs as expected.
3. **Stress Testing:** Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads. stress tests apply extreme load to determine the point at which your system breaks due to high traffic and heavy usage. Ex: Increasing traffic beyond the system's maximum capacity to see when it crashes.
4. **Scalability Testing:** Scalability Testing is a type of software testing which is carried out to check the performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

4. Acceptance Testing

- Acceptance testing is formal testing based on user requirements and function processing.
- It determines whether the software is conforming specified requirements and user requirements or not.

Once the application is bug-free, we handover it to the customer, no customer accept the application blindly before using it. Hence, they do one round of testing for their satisfaction, which is known as User Acceptance Testing (UAT).

Types of User Acceptance Testing (UAT).

- Alpha testing
- Beta testing

Alpha Testing

- ▶ Alpha testing is a type of acceptance testing, which is performed to identify all possible bugs/issues before releasing the product to the end-user.
- ▶ Alpha test is a preliminary software field test carried **out by a team of users** to find out the bugs that were not found previously by other tests.
- ▶ Alpha testing **needs lab environment**, and usually, the testers are an internal employee of the organization.
- ▶ This testing is called alpha because it is done early on, near the end of the software development, but before beta testing.

Beta Testing

- Beta testing of a product is implemented by "real users "of the software application in a "real environment."
- In this phase of testing, the software is released to a limited number of end-users of the product to obtain feedback on the product quality.
- It allows the real customers an opportunity to provide inputs into the design, functionality, and usability of the product.



Thank You

White Box Testing

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ White box testing
- ▶ Control structure testing
- ▶ Path testing

White box testing

- It is also called glass box testing or clear box testing or structural testing.
- The primary goal of white box testing is to focus on the **flow of inputs and outputs** through the software and strengthening the security of the software.
- The term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.
- White box testing techniques **analyze** the internal structures the **used data structures, internal design, code structure, and the working of the software** rather than just the functionality as in black box testing.

Control Structure Testing

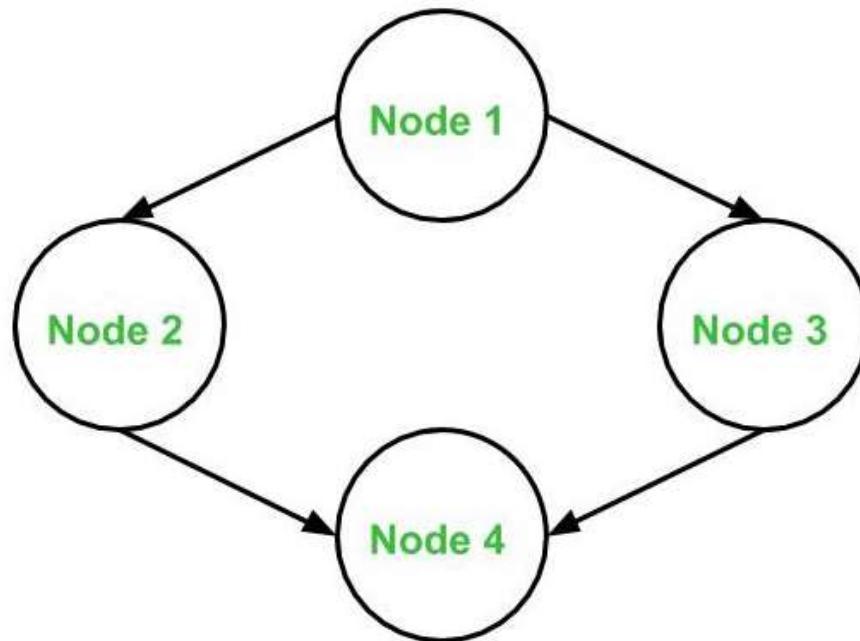
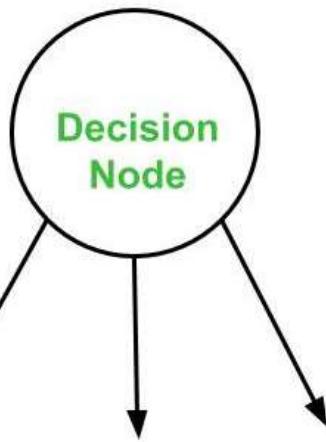
- Control structure testing is used *to increase the coverage area* by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-
 1. Condition Testing
 2. Data Flow Testing
 3. Loop Testing

5

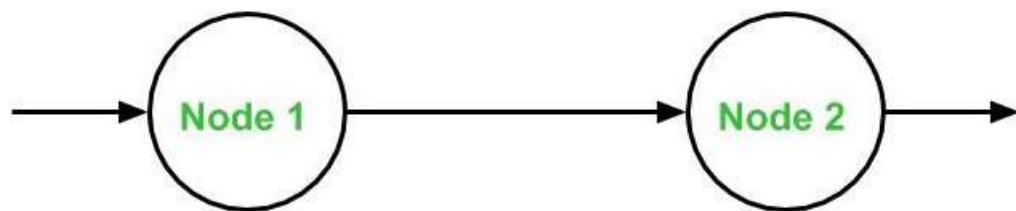
1. Condition Testing : Condition testing is a test cased design method, which ensures that the **logical condition and decision statements are free from errors**. The common types of logical conditions that are tested using condition testing are-

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

If - Then - Else



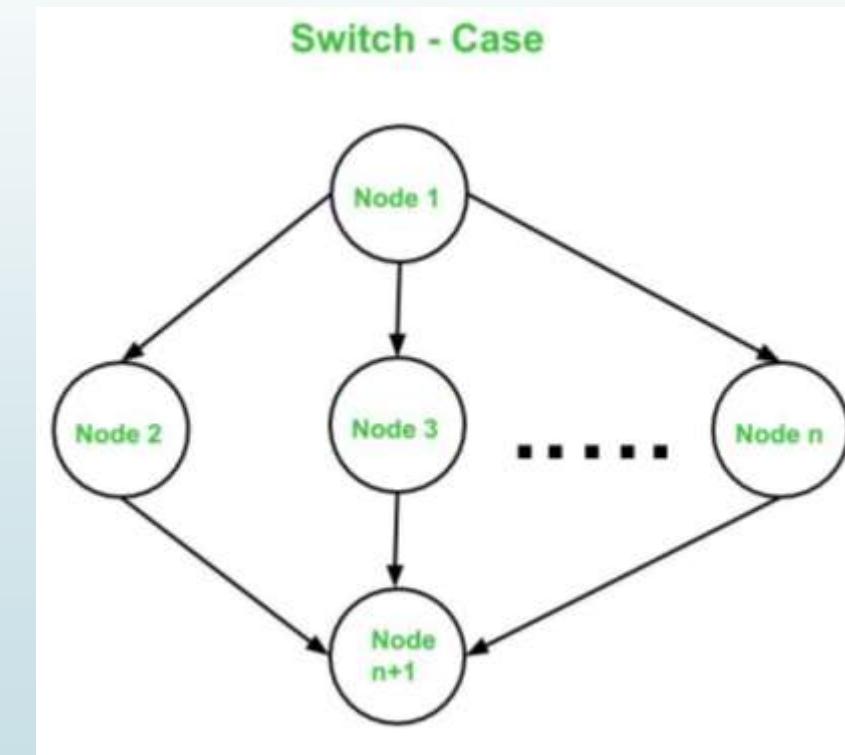
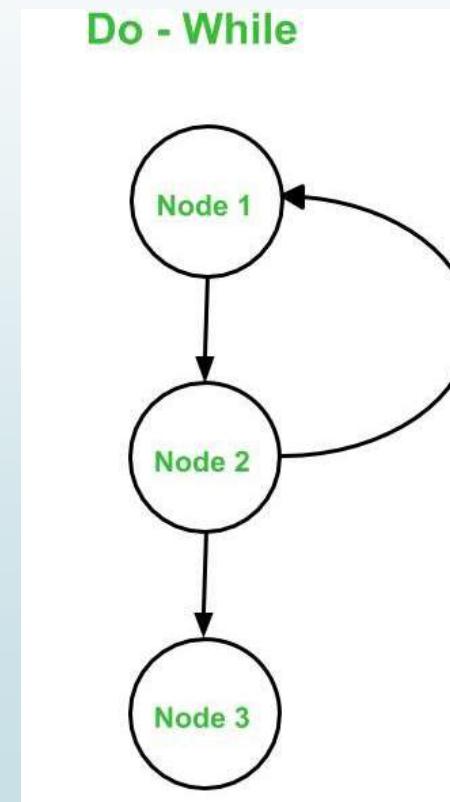
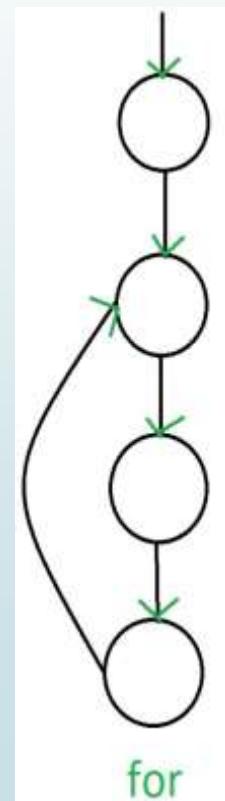
Sequence



2. Data Flow Testing : It focuses on tracking the flow of data through a program to identify anomalies, such as uninitialized variables, unused variables, and improper data modifications.

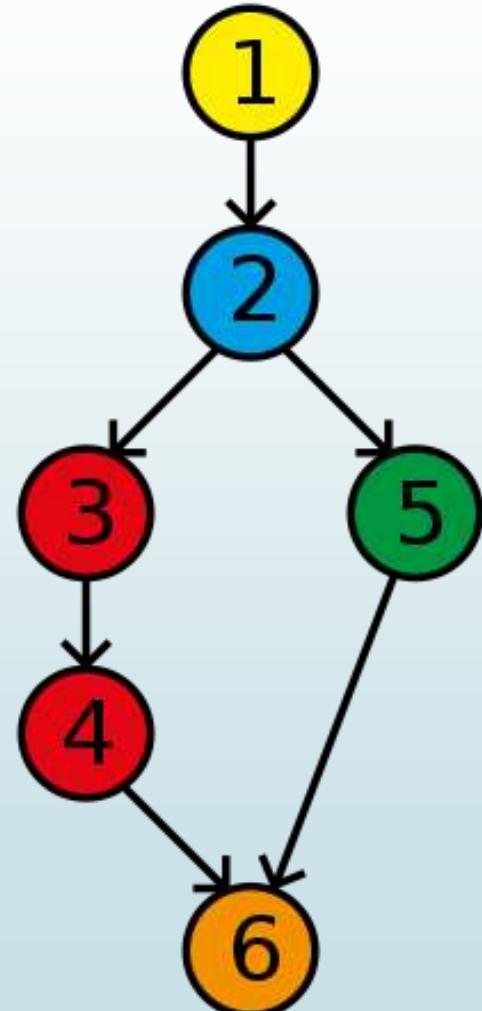
8

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



Decision-to-decision path

- ▶ A decision-to-decision path, or **DD-path**, is a path of execution (usually through a flow graph representing a program, such as a flow chart) between two decisions.
- ▶ DD-Paths are also known as segments
- ▶ So, in this flow graph, a DD path can be 1-2-5-6, or 3-4



A DD-path is a set of nodes in a program graph such that one of the following holds:

- ▶ It consists of a single node with in-degree = 0 (initial node)
- ▶ It consists of a single node with out-degree = 0 (terminal node)
- ▶ It consists of a single node with in-degree ≥ 2 or out-degree ≥ 2 (decision/merge points)
- ▶ It consists of a single node with in-degree = 1 and out-degree = 1
- ▶ It is a maximal chain of length ≥ 1 .

Basis Path testing

- ▶ Path Testing aims to uncover errors, defects, and unexpected outcomes that might occur when the application takes different paths.
- ▶ By systematically exploring all possible routes, testers can identify vulnerabilities, logic errors, and discrepancies that could undermine the reliability of the software.
- ▶ Basis Path Testing is a white-box testing technique based on the control structure of a program or a module.
- ▶ Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing.

12

To design test cases using this technique, four steps are followed :

- ▶ Construct the Control Flow Graph
- ▶ Compute the Cyclomatic Complexity of the Graph
- ▶ Identify the Independent Paths
- ▶ Design Test cases from Independent Paths

13

► **Cyclomatic Complexity** – The cyclomatic complexity $V(G)$ is said to be a measure of the logical complexity of a program.

It can be calculated using three different formulae :

1. Formula based on edges and nodes
2. Formula based on Decision Nodes
3. Formula based on number of regions

14

1. Formula based on edges and nodes :

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

Where, e is number of edges, n is number of vertices,
P is number of connected components (isolated sections of graph).

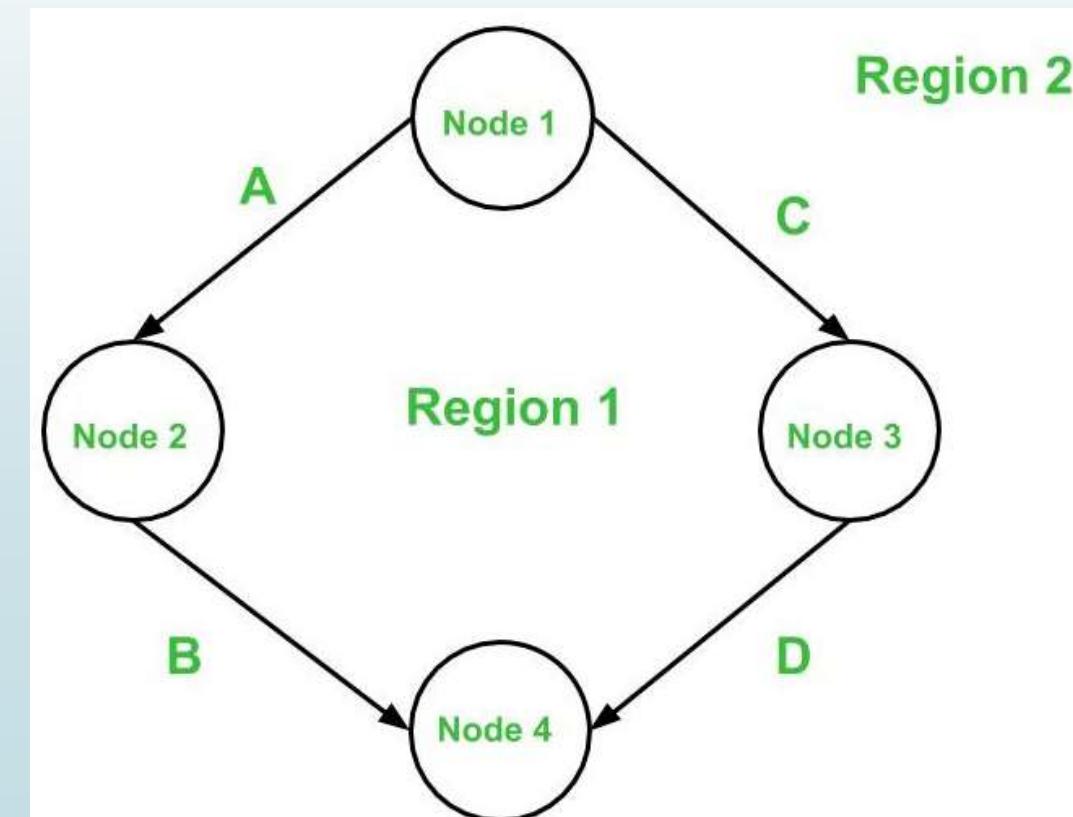
For example, consider this graph,

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

So,

Cyclomatic complexity V(G)

$$= 4 - 4 + 2 * 1 = 2$$



15

2. Formula based on Decision Nodes :

$$V(G) = D + 1$$

where, d is number of decision nodes

For example, consider this graph,

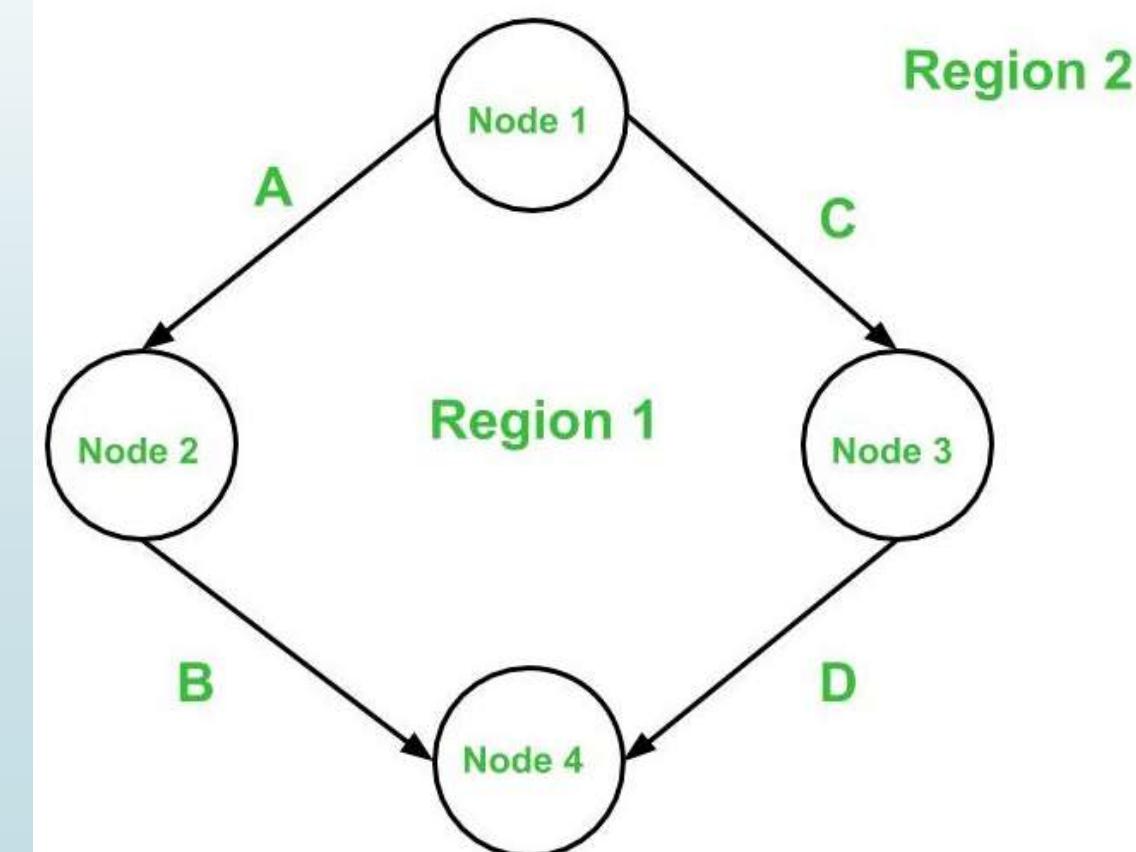
where, D = 1

So,

Cyclomatic Complexity V(G)

$$= 1 + 1$$

$$= 2$$



16

2. Formula based on Regions :

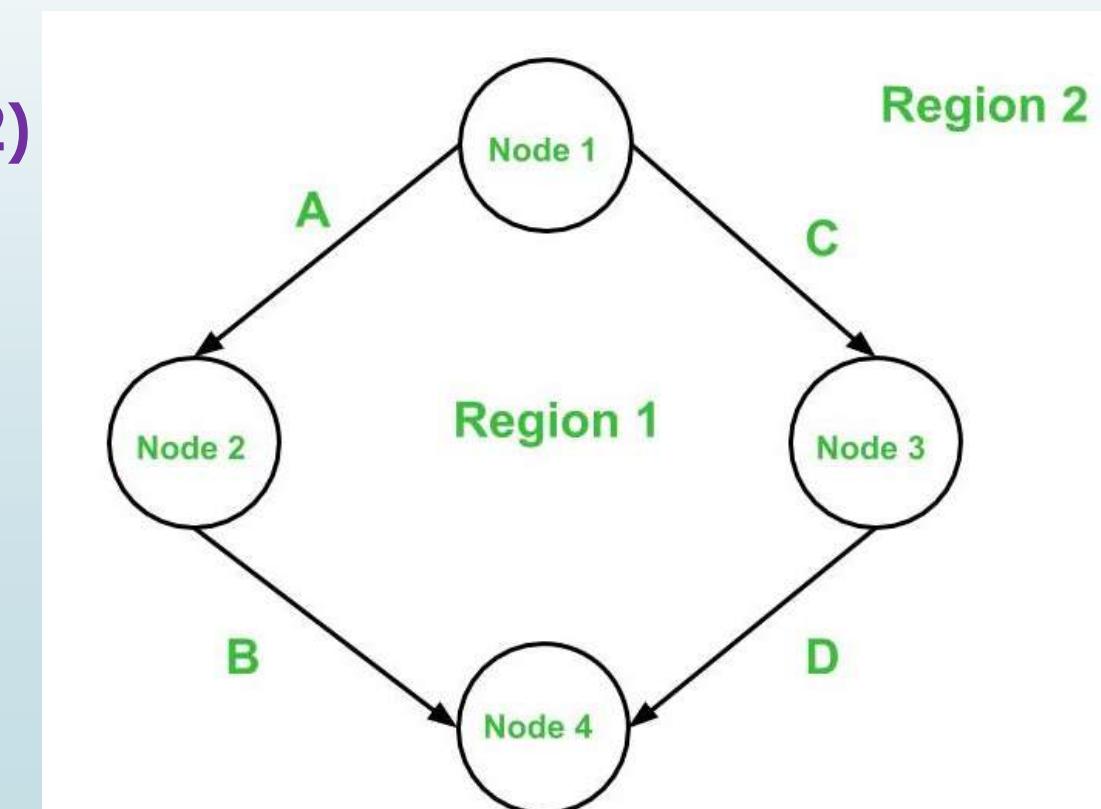
$$V(G) = \text{number of regions in the graph}$$

For example, consider first graph given above,

Cyclomatic complexity $V(G)$

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

= 2



17

Identify Independent paths:

- ▶ An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined.
- ▶ **The cyclomatic complexity gives the number of independent paths present in a flow graph.**
- ▶ Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity. So, the independent paths in above first given graph :
 - ▶ Path 1:
A -> B
 - ▶ Path 2:
C -> D

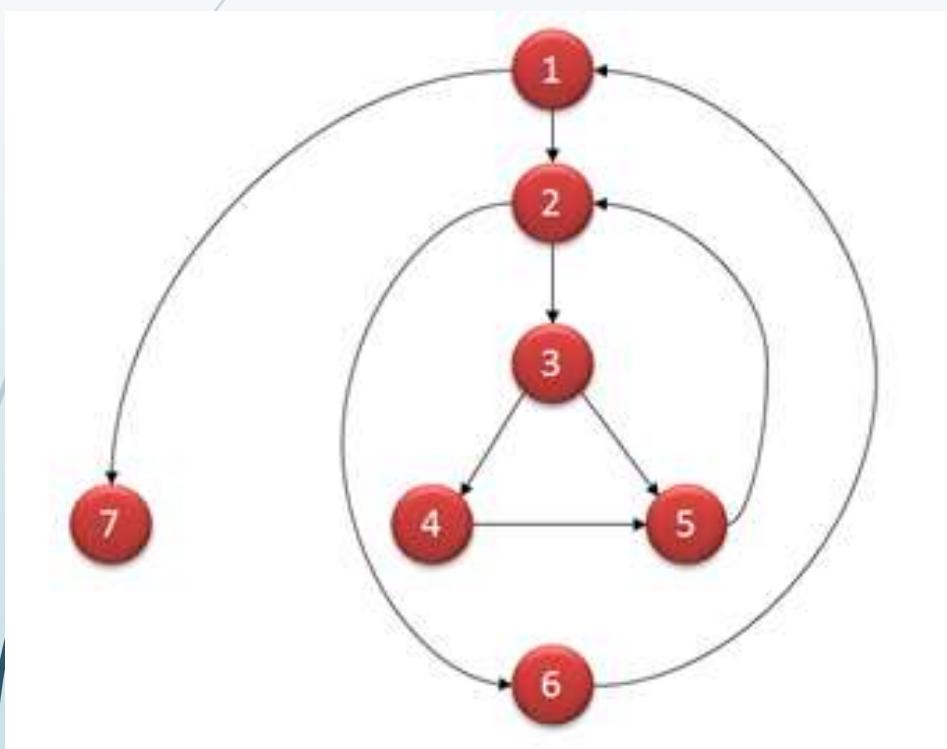
Properties of Cyclomatic complexity

Following are the properties of Cyclomatic complexity:

- ▶ $V(G)$ is the maximum number of independent paths in the graph
- ▶ $V(G) \geq 1$
- ▶ G will have one path if $V(G) = 1$

Numericals

Numerical 1



Prepared by: Ashima Tyagi (Asst. Prof. SCSE)

Cyclomatic Complexity:

- ▶ $V(G) = 9 - 7 + 2 = 4$
- ▶ $V(G) = 3 + 1 = 4$ (Condition nodes are 1,2 and 3 nodes)
- ▶ $V(G) = \text{regions} = 4$

Basis Set – A set of possible execution path of a program

- ▶ 1, 7
- ▶ 1, 2, 6, 1, 7
- ▶ 1, 2, 3, 4, 5, 2, 6, 1, 7
- ▶ 1, 2, 3, 5, 2, 6, 1, 7

Numerical 2

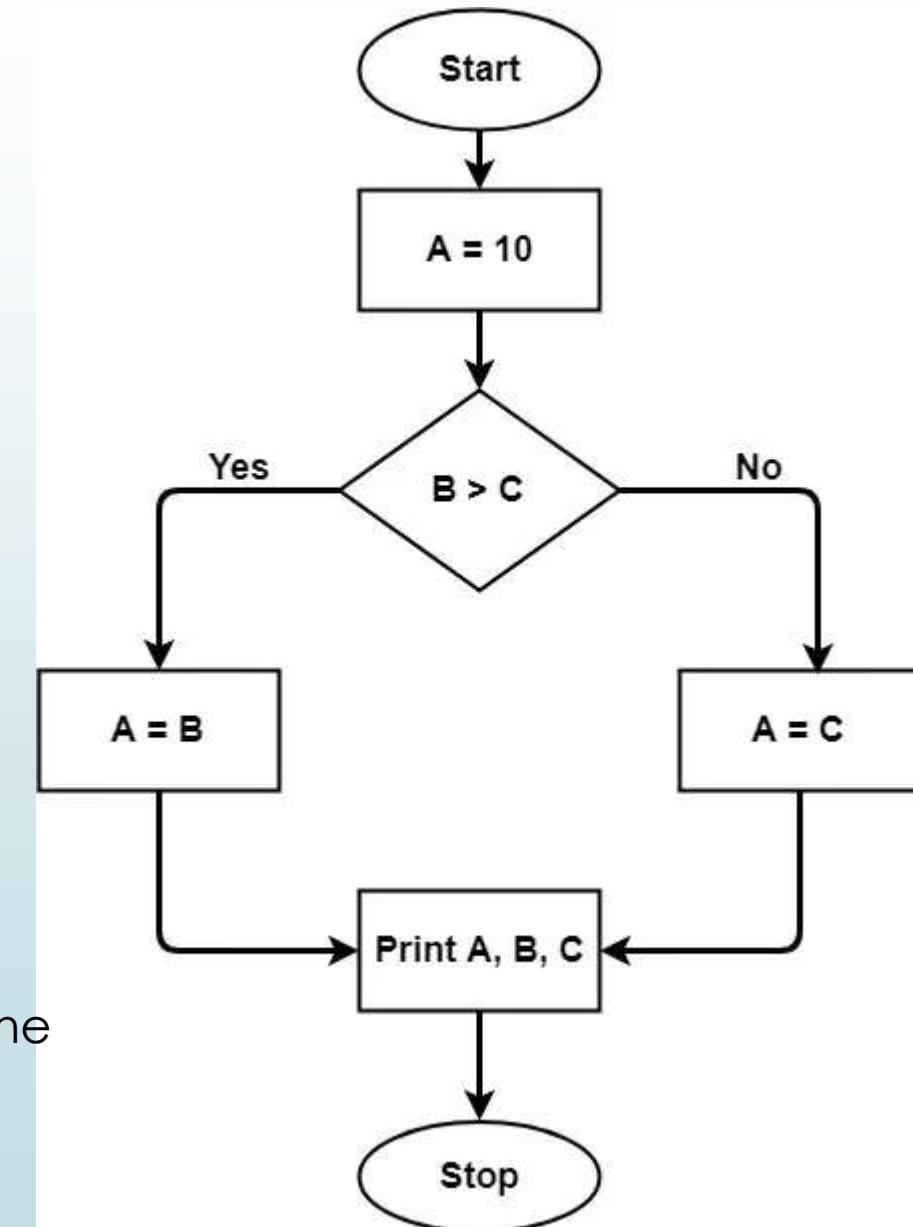
The cyclomatic complexity calculated for the above code will be from the control flow graph. The graph shows seven shapes(nodes), and seven lines(edges), hence cyclomatic complexity is $e-n+2*p = 7-7+2 = 2$.

```

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

```

Control Flow Graph of the above code



Numerical 3

Solve cyclomatic complexity of a simple program to add two numbers together:

```
int a = 1;
```

```
int b = 2;
```

```
cout<<"a + b = "<<a+b;
```

Numerical 3

22

Solve cyclomatic complexity of a simple program to add two numbers together:

```
int a = 1;
```

```
int b = 2;
```

```
cout<<"a + b = "<<a+b;
```

Solution: There is only 1 connected component(P) that is from A – B – C

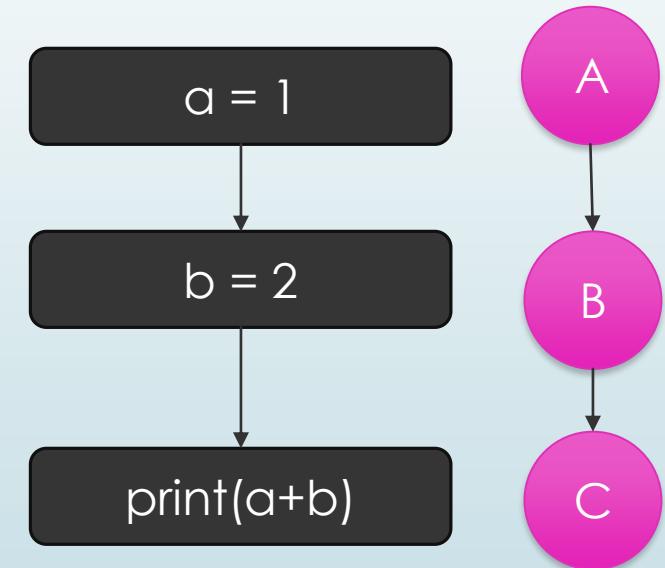
So, Cyclomatic complexity = CC = D+1 = 0 + 1

OR CC = Number of regions = 1

OR CC = E - N + 2P

$$CC = 2 - 3 + 2(1)$$

$$CC = 1$$



This shows that the graph G will have one path if $V(G) = 1$

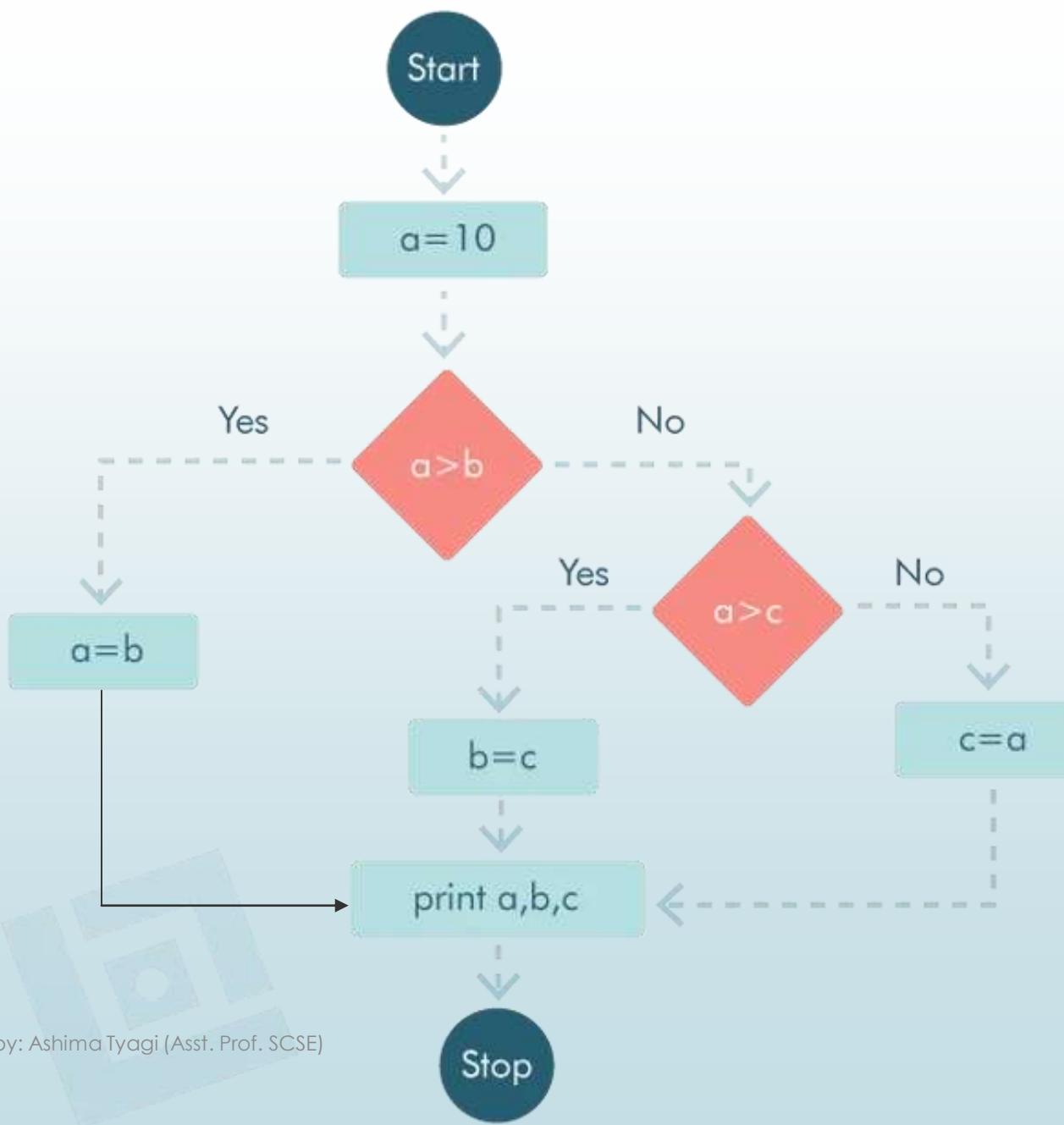
And that one path is A – B – C

Numerical 4

23

Evaluate the cyclomatic complexity of the given code: **a = 10 ;**

```
if ( a > b ) then  
    a = b ;  
else  
{  
    if ( a > c ) then  
        b = c ;  
else  
    c = a ;  
end if;  
}  
end if;  
print a  
print b  
print c
```



Using Method 1:

$$E = 10, N = 9, P = 1$$

$$\text{Hence, } V(G) = E - N + 2*P = 10 - 9 + 2 * 1$$

$$V(G) = 3$$

Using Method 2:

$$D = 2$$

$$\text{Hence, } V(G) = D + 1 = 2 + 1$$

$$V(G) = 3$$

Using Method 3:

Number of regions in the control flow graph = 3

$$\text{Hence, } V(G) = 3$$

Numerical 5

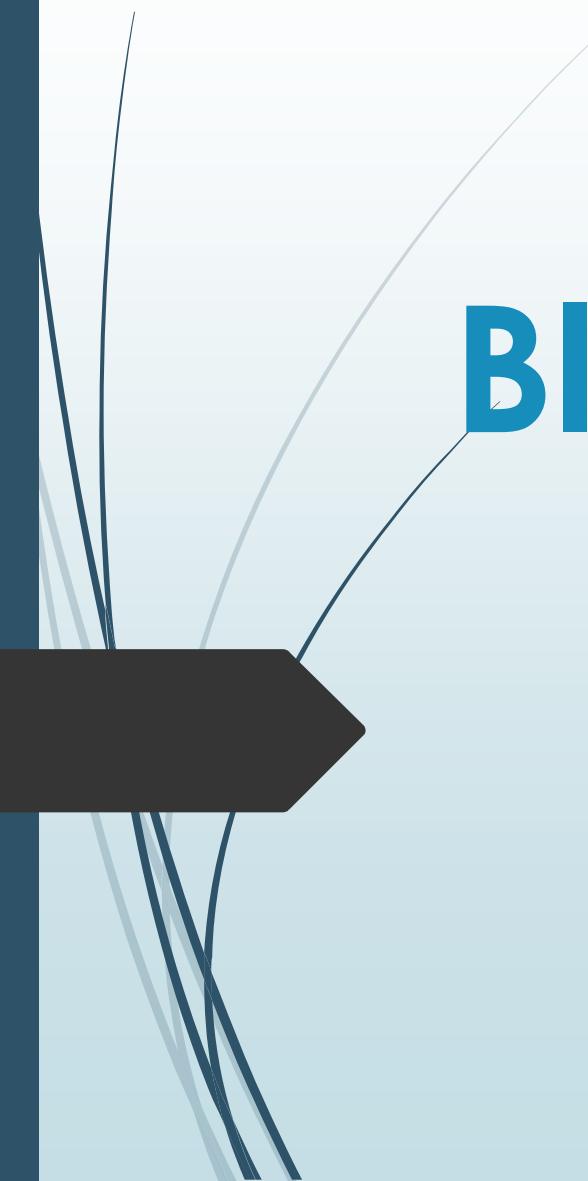
Deduct the cyclomatic complexity of A program consists of two if statements and one loop.

Numerical 6

Evaluate program cyclomatic complexity with no decision points (e.g., if, while, for, case statements).



Thank You



Black Box Testing

Ashima Tyagi

Assistant Professor

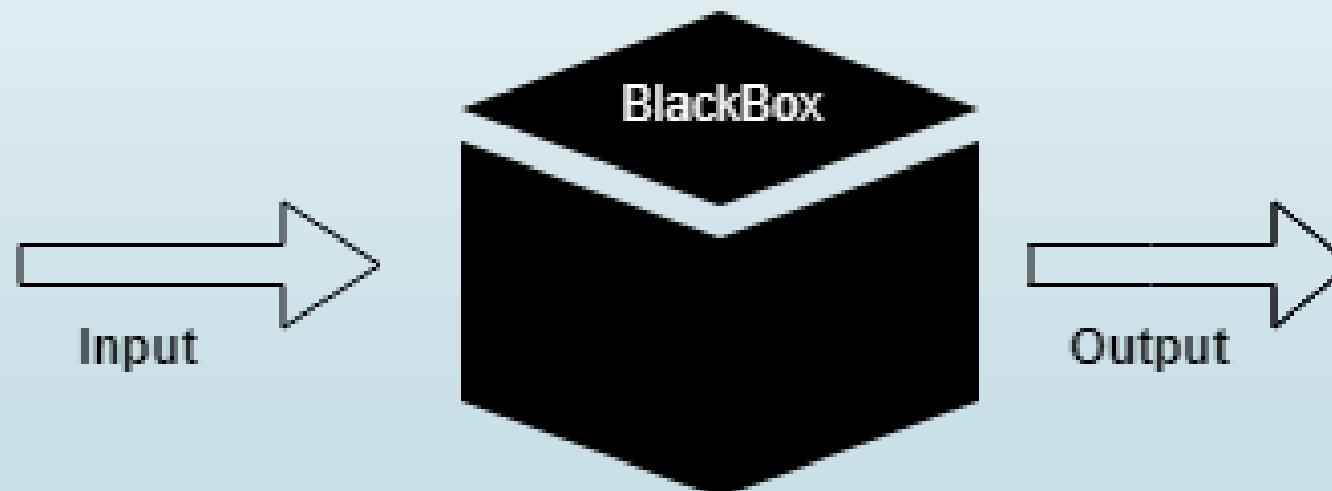
School of Computer Science & Engineering

Outline

- ▶ Black Box Testing
- ▶ Types of Black Box Testing
- ▶ Regression Testing
- ▶ Equivalence partitioning
- ▶ Boundary value analysis
- ▶ White box Vs Black Box Testing

Black box testing

- Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding.
- The primary source of black box testing is a specification of requirements that is stated by the customer.



Types of Black Box Testing

Black box testing consists of following types of testing techniques:

1. Regression Testing
2. Equivalence Partitioning
3. Boundary value analysis

1. Regression Testing

- ▶ Regression testing is a type of software testing conducted after a code update to ensure that the update introduced no new bugs.
- ▶ This is because new code may bring in new logic that conflicts with the existing code, leading to defects.
- ▶ Usually, QA teams have a series of regression test cases for important features that they will re-execute each time these code changes occur to save time and maximize test efficiency.

► **Example:** A software team develops an e-commerce website. The testing team creates 1000 test cases to check every aspect of the application. After adding new features or updates, the QA team does regression testing by creating 100 new test cases and re-executing the previous 1000. This ensures the website works as intended and maintains software quality.

When Can We Perform Regression Testing?

Regression testing is performed by testers in the following scenarios:

- ▶ When new functionality is added to the software application.
- ▶ When there is a requirement to change.
- ▶ When a defect is fixed.
- ▶ When a functional/performance issue is fixed.
- ▶ When there is a change in the environment.

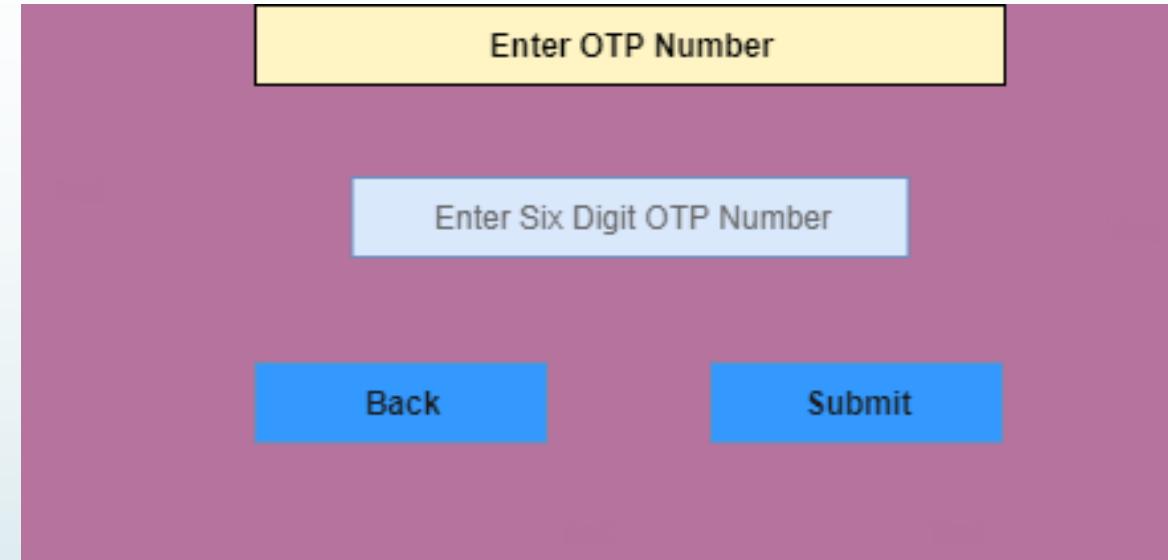
2. Equivalence partitioning

- Equivalence partitioning is a technique of software testing in which input data is divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior.
- If a condition of one partition is true, then the condition of another equal partition must also be true, and if a condition of one partition is false, then the condition of another equal partition must also be false.
- The principle of equivalence partitioning is, test cases should be designed to cover each partition at least once.
- Each value of every equal partition must exhibit the same behavior as other.
- The equivalence partitions are derived from requirements and specifications of the software. The advantage of this approach is, it helps to reduce the time of testing due to a smaller number of test cases from infinite to finite. It is applicable at all levels of the testing process.

Examples of Equivalence Partitioning technique

- ▶ Assume that there is a function of a software application that accepts a particular number of digits, not greater and less than that particular number. For example, an OTP number which contains only six digits, less or more than six digits will not be accepted, and the application will redirect the user to the error page.

10

► **OTP Number = 6 digits**

INVALID	INVALID	VALID	VALID
1 Test case	2 Test case	3 Test case	
DIGITS >=7	DIGITS<=5	DIGITS = 6	DIGITS = 6
93847262	9845	456234	451483

3. Boundary Value Analysis

- BVA is used to test boundary values because the input values near the boundary have higher chances of error.
- Whenever we do the testing by boundary value analysis, the tester focuses on, while entering boundary value whether the software is producing correct output or not.
- Boundary values are those that contain the upper and lower limit of a variable. Assume that, age is a variable of any function, and its minimum value is 18 and the maximum value is 30, both 18 and 30 will be considered as boundary values.
- The basic assumption of boundary value analysis is, the test cases that are created using boundary values are most likely to cause an error.

12

There is 18 and 30 are the boundary values that's why tester pays more attention to these values, but this doesn't mean that the middle values like 19, 20, 21, 27, 29 are ignored. Test cases are developed for each and every value of the range.

Name	Enter Your Name
Age	Between 18 to 30
Adhar	Number of 12 Digits
Address	Enter Your Address

Testing of boundary values is done by making valid and invalid partitions. Invalid partitions are tested because testing of output in adverse condition is also essential.

13

Example:

- Imagine, there is a function that accepts a number between 18 to 30, where 18 is the minimum and 30 is the maximum value of valid partition, the other values of this partition are 19, 20, 21, 22, 23, 24, 25, 26, 27, 28 and 29. The invalid partition consists of the numbers which are less than 18 such as 12, 14, 15, 16 and 17, and more than 30 such as 31, 32, 34, 36 and 40. Tester develops test cases for both valid and invalid partitions to capture the behavior of the system on different input conditions.

14

12 14 15 16 17 **18** 20 22 24 25 26 28 **30** 31 32 34 36 38 40

-----|-----|-----
Invalid Partition

Valid Partition

Invalid Partition

Invalid test cases	Valid test cases	Invalid test cases
11, 13, 14, 15, 16, 17	18, 19, 24, 27, 28, 30	31, 32, 36, 37, 38, 39

- ▶ The software system will be passed in the test if it accepts a valid number and gives the desired output, if it is not, then it is unsuccessful.
- ▶ In another scenario, the software system should not accept invalid numbers, and if the entered number is invalid, then it should display error message.
- ▶ If the software which is under test, follows all the testing guidelines and specifications then it is sent to the releasing team otherwise to the development team to fix the defects.

White box Vs Black Box Testing

S.no	On the basis of	Black Box testing	White Box testing
1.	Basic	<ul style="list-style-type: none">It is a software testing technique that examines the functionality of software without knowing its internal structure or coding.	In white-box testing, the internal structure of the software is known to the tester.
2.	Also known as	<ul style="list-style-type: none">Black Box Testing is also known as functional testing, data-driven testing, and closed-box testing.	<ul style="list-style-type: none">It is also known as structural testing, clear box testing, code-based testing, and transparent testing.
3.	Programming knowledge	<ul style="list-style-type: none">In black-box testing, there is less programming knowledge required.	<ul style="list-style-type: none">In white-box testing, there is a requirement of programming knowledge.
4.	Algorithm testing	<ul style="list-style-type: none">It is not well suitable for algorithm testing.	<ul style="list-style-type: none">It is well suitable and recommended for algorithm testing.
5.	Usage	<ul style="list-style-type: none">It is done at higher levels of testing that are system testing and acceptance testing.	<ul style="list-style-type: none">It is done at lower levels of testing that are unit testing and integration testing.
6.	Automation	<ul style="list-style-type: none">It is hard to automate black-box testing due to the dependency of testers and programmers on each other.	<ul style="list-style-type: none">It is easy to automate the white box testing.
7.	Tested by	<ul style="list-style-type: none">It is mainly performed by the software testers.	<ul style="list-style-type: none">It is mainly performed by developers.
8.	Time-consuming	<ul style="list-style-type: none">It is less time-consuming. In Black box testing, time consumption depends upon the availability of the functional specifications.	<ul style="list-style-type: none">It is more time-consuming. It takes a long time to design test cases due to lengthy code.
9.	Defect detection	<ul style="list-style-type: none">In black-box testing, defects are identified once the code is ready.	<ul style="list-style-type: none">Whereas, in white box testing, there is a possibility of early detection of defects.
10.	Errors	<ul style="list-style-type: none">It does not find the errors related to the code.	<ul style="list-style-type: none">In white-box testing, there is the detection of



Thank You



Software Measurement and Metrics

Ashima Tyagi

Assistant Professor

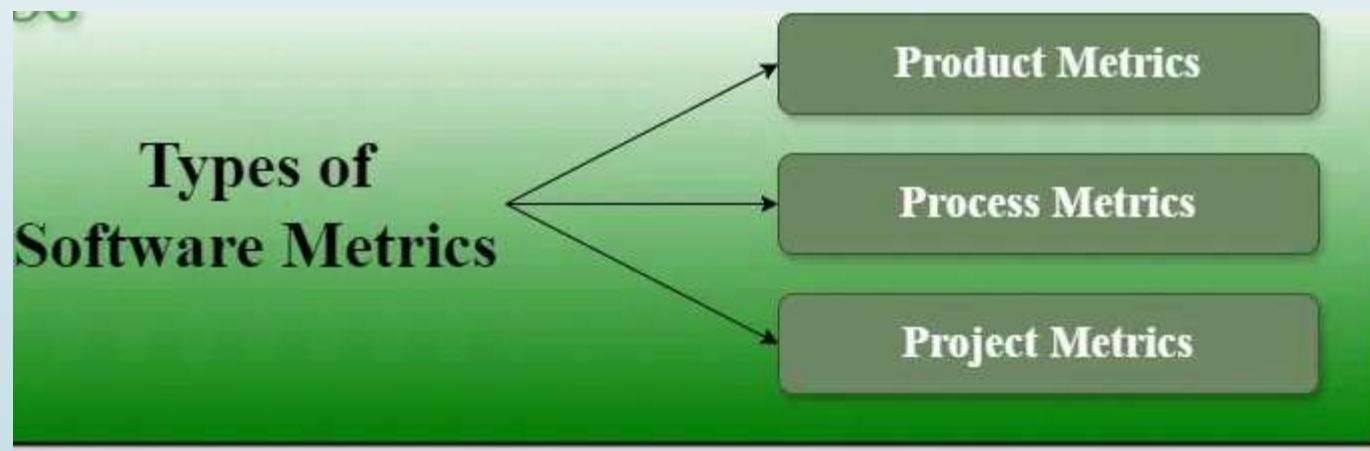
School of Computer Science & Engineering

Outline

- ▶ Software Measurement and Metrics
- ▶ Function Point (FP) Based Measures
- ▶ Practice questions
- ▶ Estimation of Various Parameters such as Cost, Efforts, Schedule/Duration
- ▶ Constructive Cost Models (COCOMO),
- ▶ Software Testing Tools

Software Metrics

- A software metric is a measure of software characteristics which are measurable or countable.
- Software metrics are valuable for many reasons, including measuring software performance, planning work items, measuring productivity, size, cost and many other uses.



Product Metrics

- Measure the **quality of the software product** (performance, reliability, maintainability).
- Used for **evaluating the software itself.**

◆ Examples:

- Cyclomatic Complexity → Measures code complexity.
- Response Time → Measures system performance.
- Reliability Metrics → Failure rate, bug count, etc.
- Defect Density

Process Metrics

- Measure the **efficiency and effectiveness** of the software development process.
- Used for **process improvement and defect prevention**.

◆ Examples:

- Defect Removal Efficiency (DRE) → Measures how well defects are removed before release.
- Mean Time to Failure (MTTF) → Average time before software fails.
- Review Efficiency → Measures the effectiveness of code reviews.

Project Metrics

- Measure **project management factors** like cost, schedule, and productivity.
- Used for **tracking project progress** and **resource allocation**.

◆ Examples:

- Cost Variance (CV) → Difference between planned and actual cost.
- Schedule Variance (SV) → Difference between planned and actual schedule.
- Effort Estimation → Measures how much effort is needed to complete a project.

Size Metrics

- Size metrics measure the **size of software** in terms of
 1. **lines of code (LOC)** or
 2. **functional units.**

I. Lines of Code (LOC)

- Counts the **total lines of source code** in a program.
- Used for **effort estimation, cost prediction, and productivity analysis**.

◆ Example:

Project	LOC	Developer Effort (Person-Months)
A	10,000	4
B	25,000	10

✗ Limitations of LOC:

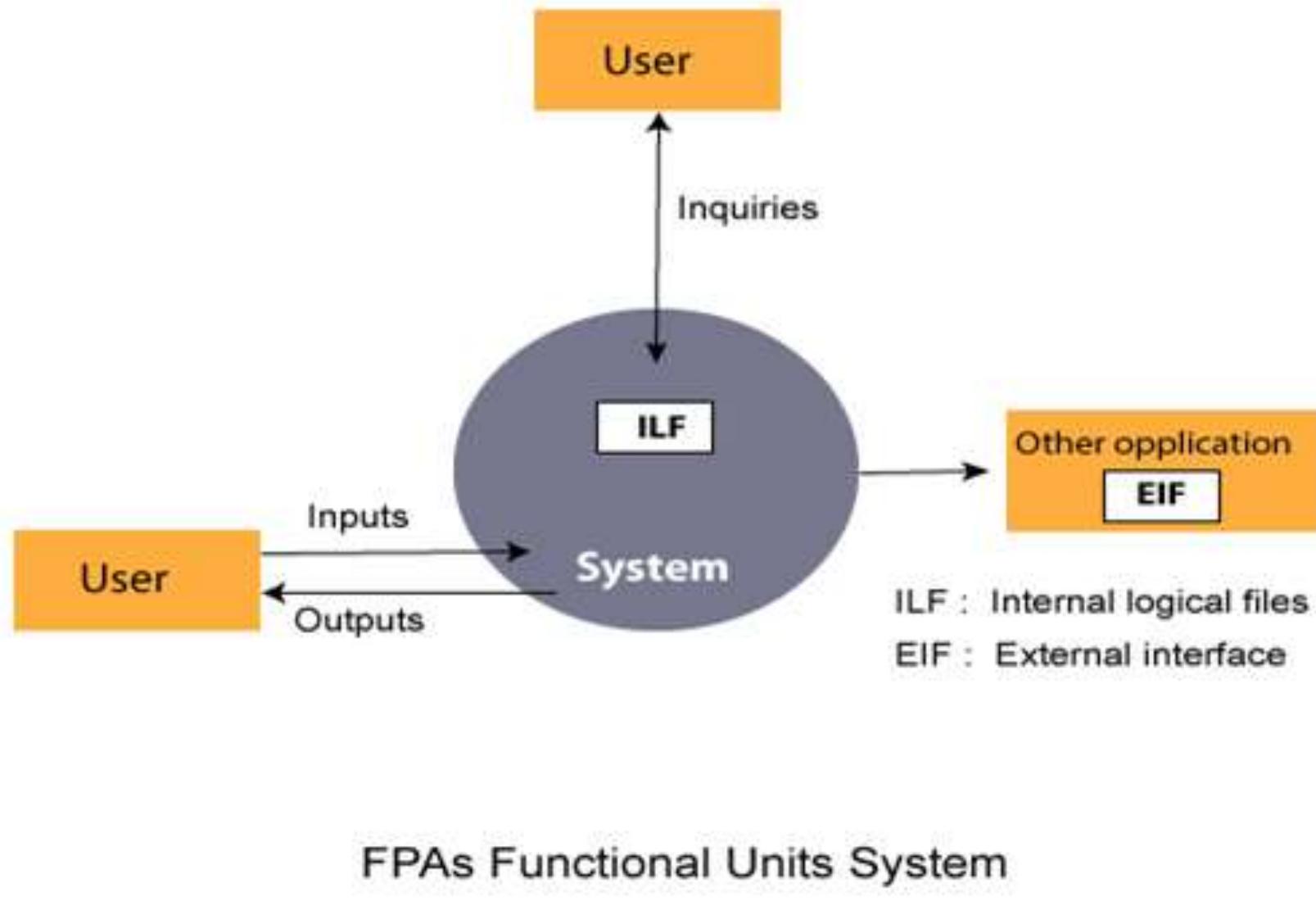
- Does not measure **code quality**.
- Different programming languages have **different LOC counts** for the same functionality.

Functional Point (FP) Analysis

- The basic and primary purpose of the functional point analysis is to measure and provide the software application functional size to the client, customer, and the stakeholder on their request.
- Further, it is used to measure the software project development along with its maintenance, consistently throughout the project irrespective of the tools and the technologies.

Types of FP Attributes

Measurements Parameters	Examples
1. Number of External Inputs(EI)	Input screen and tables
2. Number of External Output (EO)	Output screens and reports
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories
5. Number of external interfaces (EIF)	Shared databases and shared routines



Weight of FP attributes

Measurement Parameter	Low	Average	High
1. Number of external inputs (EI)	3	4	6
2. Number of external outputs (EO)	4	5	7
3. Number of external inquiries (EQ)	3	4	6
4. Number of internal files (ILF)	7	10	15
5. Number of external interfaces (EIF)	5	7	10

Function Point(FP)= UFP(Unadjusted FP) x CAF

- ▶ $F = 14 * \text{scale}$
- ▶ Scale varies from 0 to 5 according to character of Complexity Adjustment Factor (CAF). Below table shows scale:

0 - No Influence
1 - Incidental
2 - Moderate
3 - Average
4 - Significant
5 - Essential
- ▶ Calculate Complexity Adjustment Factor (CAF).

$$\text{CAF} = 0.65 + (0.01 * F)$$

Practice questions

1. Given the following values, compute function point when all complexity adjustment factor (CAF) and weighting factors are average.

- ▶ User Input = 50
- ▶ User Output = 40
- ▶ User Inquiries = 35
- ▶ User Files = 6
- ▶ External Interface = 4

Step-1: As complexity adjustment factor is average (given in question), hence, scale = 3.

- $F = 14 * 3 = 42$

Step-2:

- $CAF = 0.65 + (0.01 * 42) = 1.07$

Step-3: As weighting factors are also average (given in question) hence we will multiply each individual function point to corresponding values in TABLE.

- $UFP = (50*4) + (40*5) + (35*4) + (6*10) + (4*7) = 628$

Step-4:

- Function Point = $628 * 1.07 = 671.96$

Function Units	Low	Avg	High
EI	3	4	6
EO	4	5	7
EQ	3	4	6
ILF	7	10	15
EIF	5	7	10

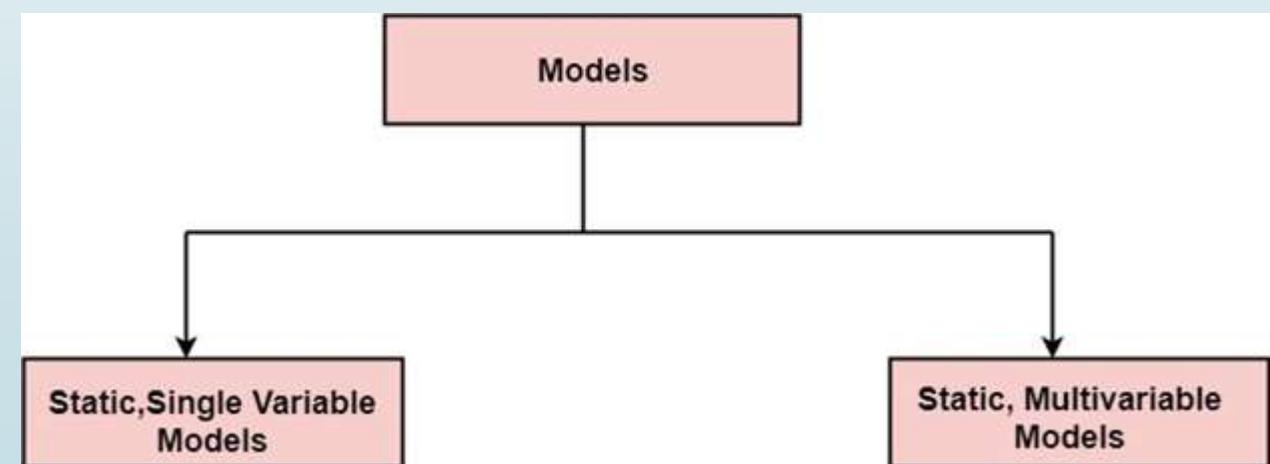
2. Your company is developing a new software application, and you have been tasked with estimating the size of the project using the Function Point (FP) analysis. After a preliminary analysis, you have gathered the following information:

- ▶ Number of External Inputs (EI): 50, each of average complexity.
- ▶ Number of External Outputs (EO): 40, each of low complexity.
- ▶ Number of External Inquiries (EQ): 30, each of high complexity.
- ▶ Number of Internal Logical Files (ILF): 20, each of average complexity.
- ▶ Number of External Interface Files (EIF): 10, each of high complexity.
- ▶ Using the standard complexity weighting values:
 - ▶ For EI, low=3, average=4, high=6.
 - ▶ For EO, low=4, average=5, high=7.
 - ▶ For EQ, low=3, average=4, high=6.
 - ▶ For ILF, low=7, average=10, high=15.
 - ▶ For EIF, low=5, average=7, high=10.

Evaluate the total function points (FP) for the project.

Cost Estimation Models

- A model may be static or dynamic.
- In a static model, **a single variable** is taken as a key element for calculating cost and time.
- In a dynamic model, all variables are interdependent, and there is no basic variable.



The Methods using this model utilize an equation to get the desired values such as cost, time, effort, etc. And these all depend on the same variable used as a predictor like, size.

This model is used to estimate the effort, cost and development time for a software project which depends on a single variable. The relationship is given by:

$$\text{Cost (C)} = a * (\text{LOC})^b$$

$$\text{Effort (E)} = a * (\text{LOC})^b \text{ PM}$$

$$\text{Development Time (DT)} = a * (\text{LOC})^b \text{ Months}$$

Where LOC = Number of Lines of Code.

Below is an example of the most common equation:

$$C = aL^b$$

Where C is cost, L is size and a, b are constants.

We have an example of the static single-variable mode. e. i.e. **SEL model** which is used for estimating software production. The equation of this model is given below:

$$E = 1.4L^{0.93}$$

$$DOC = 30.4L^{0.90}$$

$$D = 4.6L^{0.26}$$

Where E is in Person-months, DOC i.e, documentation is in the number of pages and D is duration which is months.

Question 1:

A software project has an estimated size of 12,000 lines of code (LOC). The constants $a=1.8$ and $b=1.15$ are given. Estimate the cost of the project.

Solution:

We are given:

- $a=1.8$
- $b=1.15$
- $L=12,000$ LOC

Using the formula the cost of the project is **\$38,826.94**.

Question 2:

A software project is estimated to have 8,000 lines of code (LOC). The cost constant $a=2.2$ and exponent $b=1.3$ are given. Calculate the cost of the project.

Solution:

We are given:

- $a=2.2$
- $b=1.3$
- $L=8,000$ LOC

Using the formula the estimated cost of the project is **\$30,633.74**.

CCOMO Model

- ▶ **COCOMO (Constructive Cost Estimation Model)** is one of the most generally used software estimation models in the world.
- ▶ COCOMO predicts the efforts and schedule of a software product based on the size of the software.

Advantages :

- ▶ It works on historical data and provides more accurate details.
- ▶ Easy to implement with various factors. One can easily understand how it works.
- ▶ Easy to estimate the total cost of the project.
- ▶ The drivers are very helpful to understand the impact of the different factors that affect project crises.

Disadvantages :

- ▶ It ignores the hardware issues as well as the personal turnover level.
- ▶ It ignores all the documentation and requirements.
- ▶ It mostly depends on time factors.
- ▶ It limits the accuracy of software costs.
- ▶ It oversimplifies the impact of safety or security aspects.
- ▶ It also ignores customer skills, cooperation, and knowledge.

Basic COCOMO Model: The basic COCOMO model provides an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model:

$$\text{Effort } E = a(\text{KLOC})^b \text{ PM}$$

$$\text{Time} = c(\text{Effort})^d \text{ Months}$$

$$\text{Person required} = \text{Effort} / \text{time}$$

Where

- ▶ KLOC is the estimated size of the software product indicate in Kilo Lines of Code,
- ▶ a and b are constants for each group of software products,
- ▶ time is the estimated time to develop the software, expressed in months,
- ▶ Effort is the total effort required to develop the software product, expressed in person months (PMs).

- The above formula is used for the cost estimation of the basic COCOMO model and also is used in the subsequent models. The constant values a, b, c, and d for the Basic Model for the different categories of the system:

Software Projects	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Practice questions

1. Apply the COCOMO model to estimate the effort required for a project that is classified as 'organic' and consists of 32,000 lines of code (LOC). Use the basic COCOMO model formula. For an organic project, the typical values of A and B are 2.4 and 1.05

Solution: Given Data

$$\text{LOC} = 32,000$$

$$A = 2.4$$

$$B = 1.05$$

Convert LOC to KLOC

$$KLOC = 32,000 / 1,000 = 32$$

Calculate Effort

$$E = 2.4 \times (32) ^{1.05}$$

$$E \approx 91.33 \text{ Person-Months (PM)}$$

Estimated Effort: The estimated effort required for the project is approximately 91.33 person-months.

2. A software development project is estimated to require 1,000 hours of development time. The hourly rate for developers is \$50. Estimate the cost for the development of the software?

► Total Cost=Development Time×Hourly Rate

► Given:

- Development Time = 1,000 hours
- Hourly Rate = \$50/hour

► **Calculation**

$$\text{Total Cost}=1,000 \text{ hours} \times 50 \text{ dollars/hour}$$

$$\text{Total Cost}=50,000 \text{ dollars}$$

Estimated Cost

► The estimated cost for the development of the software is **\$50,000**.

3. A software project is estimated to require 500 person-days of effort. If the team consists of 5 developers, solve how many days will it take to complete the project?

- ▶ Number of Days=Number of Developers/Total Person-Days
- ▶ Given:
 - Total Person-Days = 500
 - Number of Developers = 5
- ▶ **Calculation**

$$\text{Number of Days}=500 \text{ person-days} / 5 \text{ developers}$$

$$\text{Number of Days}=100 \text{ days}$$

Result

- ▶ It will take 100 days to complete the project with a team of 5 developers.

4. A software project has a fixed budget of \$150,000. The hourly rate for developers is \$75, and the estimated development time is 2,000 hours. Predict whether the project will stay within budget?

Calculation

- **Total Cost of Development:** Total Cost=Hourly Rate × Estimated Development Time

Given:

- Hourly Rate = \$75
- Estimated Development Time = 2,000 hours

$$\text{Total Cost} = 75 \text{ dollars/hour} \times 2,000 \text{ hours}$$

$$\text{Total Cost} = 150,000 \text{ dollars}$$

- **Comparison**

- **Fixed Budget:** \$150,000
- **Estimated Total Cost:** \$150,000

Result

- The estimated total cost of development is equal to the fixed budget of \$150,000. Therefore, the project is predicted to stay within budget if the estimated development time and hourly rate remain accurate.

5. **Estimating Effort for a Semi-Detached Project** A semi-detached software project has 50,000 LOC. Use: $A=3.0$ $B=1.12$. Find the Effort (in PM).
6. **Estimating Effort and Time for an Embedded Project** A project is classified as Embedded and has 100,000 LOC. Use: $A=3.6$ $B=1.20$. Also, use the Duration (Time in Months) formula:

$$\text{Time} = C \times (\text{Effort})^D$$

where for Embedded projects:

$$C=2.5$$

$$D=0.32$$

7. **Finding LOC from Given Effort** A project classified as Organic has an estimated Effort of 50 PM. Using: $A=2.4$ $B=1.05$ Find the estimated KLOC.

Software Testing Tools

Software testing tools help automate, manage, and execute tests efficiently.

Types of Testing Tools

Software testing is of two types, static testing, and dynamic testing.

1. Static Test Tools
2. Dynamic Test Tools

- 1. Static Test Tools:** Static test tools are used to work on the static testing processes. In the testing through these tools, the typical approach is taken. These tools do not test the real execution of the software. Certain input and output are not required in these tools. Static test tools consist of the following:
 - **Flow analyzers:** Flow analyzers provides flexibility in the data flow from input to output.
 - **Path Tests:** It finds the not used code and code with inconsistency in the software.
 - **Coverage Analyzers:** All rationale paths in the software are assured by the coverage analyzers.
 - **Interface Analyzers:** They check out the consequences of passing variables and data in the modules.

- 2. Dynamic Test Tools:** Dynamic testing process is performed by the dynamic test tools. These tools test the software with existing or current data. Dynamic test tools comprise the following:
- **Test driver:** The test driver provides the input data to a module-under-test (MUT).
 - **Test Beds:** It displays source code along with the program under execution at the same time.
 - **Emulators:** Emulators provide the response facilities which are used to imitate parts of the system not yet developed.
 - **Mutation Analyzers:** They are used for testing the fault tolerance of the system by knowingly providing the errors in the code of the software.

Some popular testing tools:

- 1. Jira**
- 2. Selenium**
- 3. Junit**

1 Jira (Test Management & Issue Tracking)

✓ What is Jira?

Jira is a test management and bug tracking tool widely used in Agile and DevOps environments.

✓ Key Features:

- Bug Tracking & Issue Management – Track software defects and improvements.
- Test Case Management (via plugins like Xray, Zephyr) – Manage test cases and execution.
- Agile Board Support – Scrum/Kanban support for sprint planning.
- Custom Workflows – Define test cycles, status, and issue resolution steps.

✓ Usage in Software Testing:

- Documenting test cases, linking them to user stories.
- Logging bugs and assigning them to developers.
- Managing test cycles and reporting defects.

2 □ Selenium (Automation Testing)

✓ What is Selenium?

Selenium is a popular automation testing tool for web applications.

✓ Components of Selenium:

- Selenium WebDriver – Automates web browser actions.
- Selenium Grid – Runs tests on multiple machines/browsers in parallel.
- Selenium IDE – A record-and-playback tool for simple tests.

✓ Use Cases:

- Automating UI testing for web applications.
- Performing cross-browser testing (Chrome, Firefox, Edge, etc.).
- Running data-driven tests using frameworks like TestNG or JUnit.

3 JUnit (Unit Testing in Java)

✓ What is JUnit?

JUnit is a **Java testing framework** used for unit testing. It ensures individual components of the software function correctly.

✓ Key Features:

- Annotations like @Test, @BeforeEach, @AfterEach.
- Assertions for expected vs. actual results.
- Parameterized Tests for multiple test cases.
- Test Suites to run multiple tests together.



Thank You



Software Quality

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Software Quality
- ▶ Software Quality Attributes
- ▶ McCall quality model
- ▶ Software Quality Assurance
- ▶ SQA Plans
- ▶ Software Quality Frameworks
- ▶ SEI-CMM Model

Software Quality

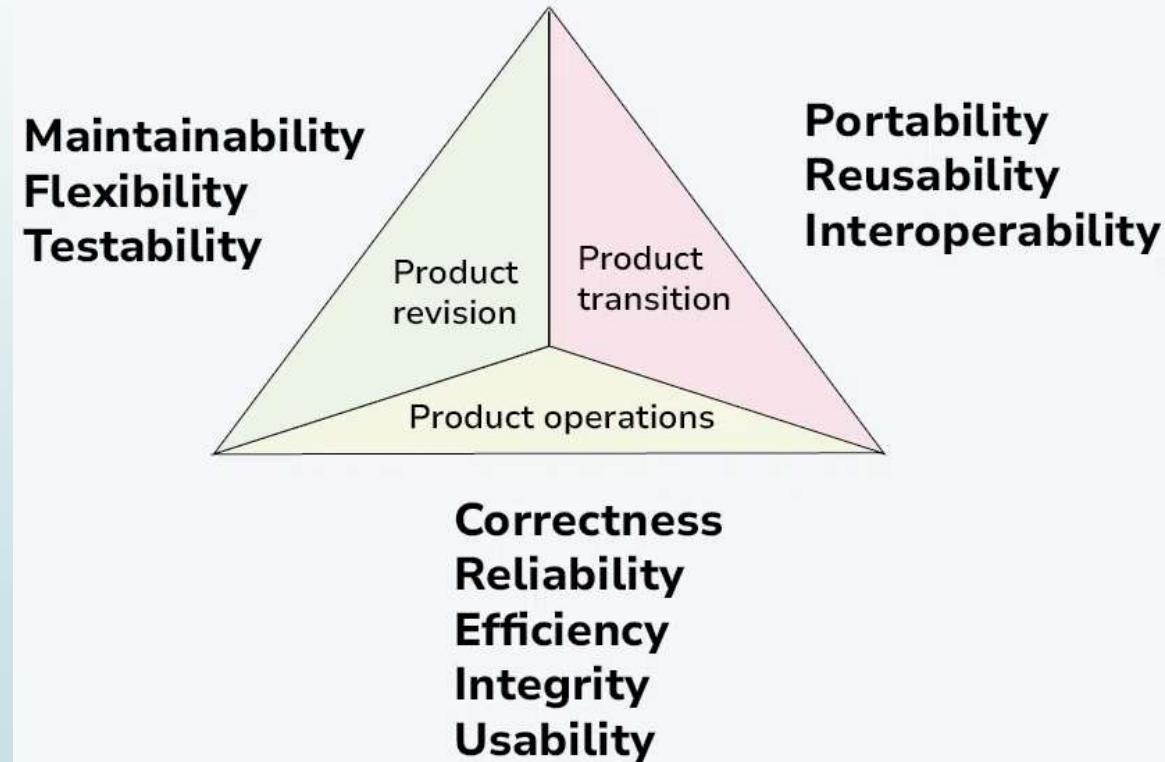
- ▶ Software quality refers to the degree to which a software product meets established requirements and user expectations.
- ▶ It encompasses various aspects, including functionality, reliability, usability, efficiency, maintainability, and portability.
- ▶ There is a misconception that if the software application is bug-free then the quality of the software is high. However, a bug-free product is just one of the Software Quality Attributes.

McCall quality model

McCall's Software Quality Model is incorporated with many attributes, termed software factors, which influence software.

- This model classifies all software requirements into 11 software quality factors. The 11 factors are organized into three product quality factors: **Product Operation**, **Product Revision**, and **Product Transition**.

McCall's Quality Model Triangle



Product Operation

Product Operation includes five software quality factors, which are related to the requirements that directly affect the operation of the software such as operational performance, convenience, ease of usage, and correctness. These factors help in providing a better user experience.

- ▶ Correctness
- ▶ Efficiency
- ▶ Integrity
- ▶ Reliability
- ▶ Usability

6

1. **Correctness:** Correctness refers to the ability to behave or function **as per software requirement specifications**. This may include navigations, calculations, form submissions, etc. Consider an example of sign-up. Your application should navigate to the terms and conditions page after signing up as per the requirement specification. However, it is landing on the home page without showing the terms and conditions page.
2. **Efficiency:** The software **should use system resources efficiently**, such as memory and processing power, to achieve its objectives. For example, if you open one of the video editor applications on your desktop, as soon as you open your system freezes, and all the other open application start to behave in an unintended way. This is a bad application design and shows the poor efficiency of the software.

3. **Integrity:** The extent to which the software can control an unauthorized person from accessing the data or software.
4. **Reliability:** Reliability is the ability of software applications to behave as **expected and function under the maximum possible load**. Let's take an example of an e-commerce website. The website went down as one of the server nodes crashed. In an ideal situation, if the reliability attribute is implemented then the system should automatically failover to the other server node.
5. **Usability:** The software should be **user-friendly, intuitive, and easy to use** for its intended users. let's consider an e-commerce page – user has purchased an item and wants to return the item. Good usability makes the return option available on the orders page. In some cases, the return option may appear on to contact us page – in this situation user easily gets confused and faces difficulty to find the return option.

Product Revision

Product Revision includes three software quality factors, which are required for testing and maintenance of the software. They provide ease of maintenance, flexibility, and testing efforts to support the software to be functional according to the needs and requirements of the user in the future.

- Maintainability
- Flexibility
- Testability

1. **Maintainability:** The software should be **easy to maintain, update, and modify** as needed to meet changing requirements or fix issues.
3. **Flexibility:** Flexibility refers to how quickly your application can adapt to future and current technology demands. For example, You are using a third-party library for styling your application. Due to some reasons, the third-party library declares the end of development. Now the question arises of how quickly your application can switch to another library. If it takes longer, then it might cost your business.
3. **Testability:** Testability is how easily QA members can test the software and log a defect and how easy it is to automate the software applications.

Product Transition

Product Transition includes three software quality factors, that allow the software to adapt to the change of environments in the new platform or technology from the previous.

- Portability
- Re-usability
- Interoperability

1. **Portability:** The software should be able to run on different platforms or environments without major modifications. An example of a portability issue – you have designed a web application that works perfectly fine on Android devices but when it is ported to iPhone devices (iOS), it fails to render.
2. **Reusability:** It is the degree to which software components can be reused in another application or the same application. Reusable software components reduce the development cost and effort. For example, your organization is developing two different applications where both the application needs sign-in and sign-up forms. If you don't have a reusable components strategy then you need to develop the same thing two times.
3. **Interoperability:** Interoperability refers to the ability to communicate or exchange data between different systems. Which can be operating systems, databases, or protocols. The interoperability problem arises due to the legacy code base, poorly architected application, and poor code quality. For example, Your application needs to communicate with the payment gateway but you are facing challenges to integrate with the payment gateway due to various standards mismatch. If you had taken all the security, data, and standard API design approach this wouldn't have happened.

Software Quality Assurance (SQA)

Software Quality Assurance (SQA) is simply a way to assure quality in the software. It is **the set of activities which ensure processes, procedures as well as standards are suitable for the project and implemented correctly.**

Software quality assurance focuses on:

- ▶ software's portability
- ▶ software's usability
- ▶ software's reusability
- ▶ software's correctness
- ▶ software's maintainability
- ▶ software's error control

13

Elements Of Software Quality Assurance:

1. **Standards:** The IEEE, ISO, and other standards organizations have produced a broad array of software engineering standards and related documents. The job of SQA is to ensure that standards that have been adopted are followed, and all work products conform to them.
2. **Reviews and audits:** Technical reviews are a quality control activity performed by software engineers for software engineers. Their intent is to uncover errors. Audits are a type of review performed by SQA personnel (people employed in an organization) with the intent of ensuring that quality guidelines are being followed for software engineering work.
3. **Testing:** Software testing is a quality control function that has one primary goal—to find errors. The job of SQA is to ensure that testing is properly planned and efficiently conducted for primary goal of software.
4. **Error/defect collection and analysis:** SQA collects and analyzes error and defect data to better understand how errors are introduced and what software engineering activities are best suited to eliminating them.
5. **Change management:** SQA ensures that adequate change management practices have been instituted.
6. **Education:** Every software organization wants to improve its software engineering practices. A key contributor to improvement is education of software engineers, their managers, and other stakeholders. The SQA organization takes the lead in software process improvement which is key proponent and sponsor of educational programs.
7. **Security management:** SQA ensures that appropriate process and technology are used to achieve software security.
8. **Safety:** SQA may be responsible for assessing the impact of software failure and for initiating those steps required to reduce risk.
9. **Risk management:** The SQA organization ensures that risk management activities are properly conducted and that risk-related contingency plans have been established.

What is verification and validation in quality assurance?

Validation is the process of checking whether the specification captures the customer's requirements, while verification is the process of checking that the software meets specifications.

SQA Plans

A Quality Assurance Plan (QAP) **is a document or set of documents** that outlines the systematic processes, procedures, and standards for ensuring the quality of a product or service.

Importance of Software Quality Assurance Plan

1. Quality Standards and Guidelines
2. Risk management
3. Standardization and Consistency
4. Customer Satisfaction
5. Resource optimization
6. Early Issue Detection

16

Steps to Develop Software Quality Assurance Plan:

1. Define Project Objectives and Scope
2. Identify Quality Standards and Criteria
3. Identify Stakeholders
4. Define Roles and Responsibilities
5. Conduct a Risk Assessment
6. Establish Quality Assurance Activities
7. Develop Testing and Inspection Procedures
8. Document Processes and Procedures
9. Establish Documentation and Reporting Mechanisms
10. Allocate Resources and Training
11. Define Change Control Processes
12. Review and Approval
13. Monitoring and Continuous Improvement
14. Communication and Training: Communicate the Quality Assurance Plan to all relevant stakeholders and provide training as necessary. Ensure that everyone involved understands their roles and responsibilities in maintaining and improving quality.

Software Quality Framework

17

A software quality framework is a **structured approach or methodology used to ensure that software products meet defined quality standards**. It provides a systematic way to manage and improve the quality of software throughout its lifecycle. A software quality framework typically includes the following components:

1. **Quality Planning:** Defining quality objectives, standards, and metrics for the software project.
2. **Quality Assurance:** Processes and activities to ensure that quality standards are being followed throughout the software development lifecycle. This may include reviews, audits, and process improvements.
3. **Quality Control:** Processes and activities to monitor and verify that the software meets the defined quality standards. This may include testing, inspections, and defect tracking.
4. **Quality Improvement:** Continuous improvement activities to enhance the quality of the software and the development process. This may include root cause analysis, process optimization, and lessons learned.
5. **Quality Metrics:** Measurement and analysis of quality-related data to track progress, identify trends, and make informed decisions about quality improvement efforts.
6. **Quality Management:** Overall management of quality within the organization, including leadership, commitment, and support for quality initiatives.

Some common software quality frameworks include **the ISO 9000 series, the Capability Maturity Model (CMM), and the Six Sigma approach**.

ISO Model

The ISO (International Organization for Standardization) has provided internationally recognized models for evaluating software quality.

The most commonly referenced are:

- ISO/IEC 9126 (older, foundational model)
- ISO/IEC 25010 (updated and extended version)

ISO/IEC 9126 Quality Model

It classifies software quality into **6 major characteristics**, each with sub-characteristics:

- 1 Functionality – Does the software meet its purpose?
- 2 Reliability – Can it perform under stated conditions?
- 3 Usability – Is it easy to use?
- 4 Efficiency – Is it resource-efficient?
- 5 Maintainability – How easy is it to fix or improve?
- 6 Portability – Can it run in different environments?

⌚ ISO/IEC 25010 (Updated Version)

Introduces 8 quality characteristics:

ISO 25010 Characteristic

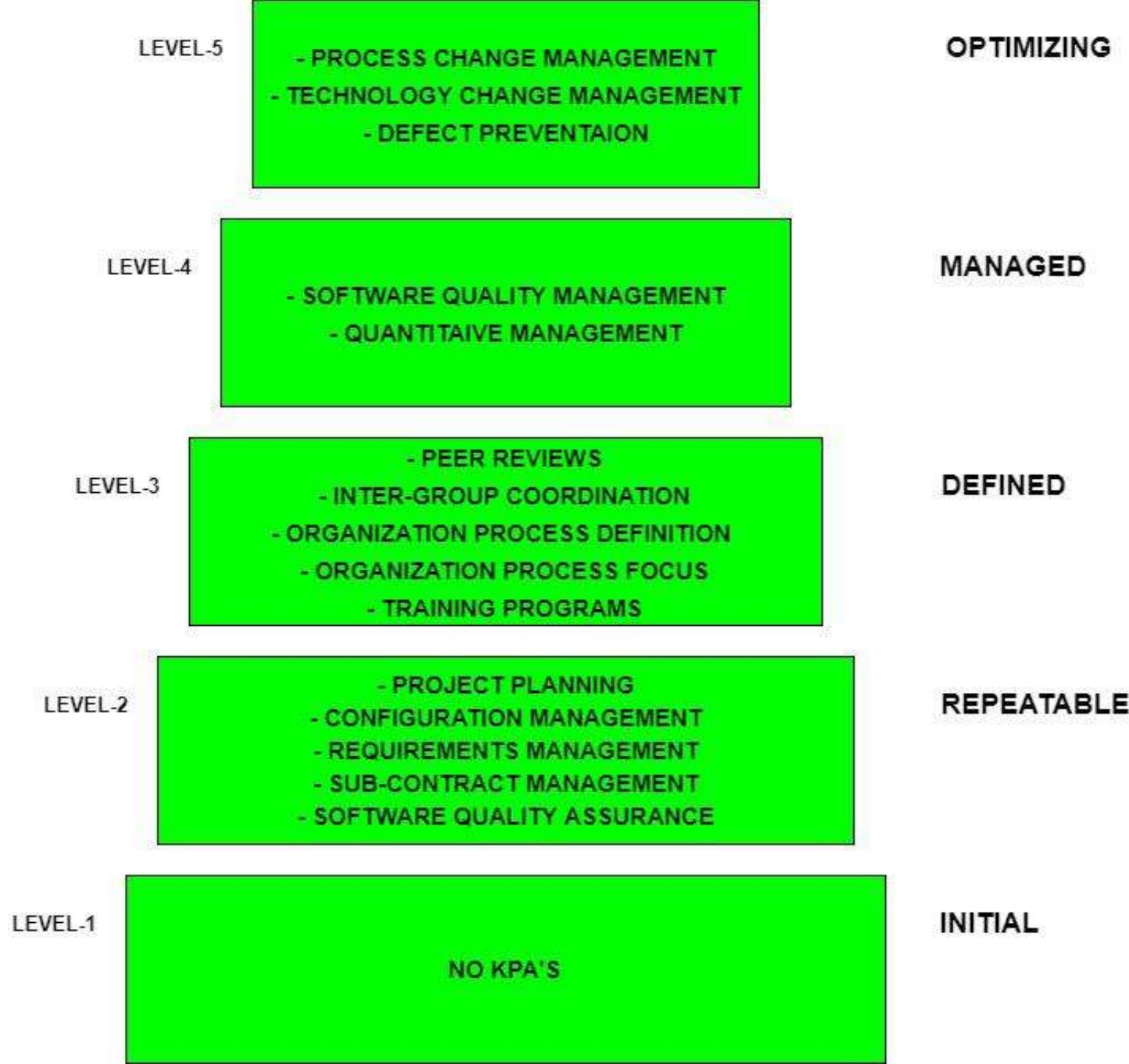
- 1. Functional Suitability**
- 2. Performance Efficiency**
- 3. Compatibility**
- 4. Usability**
- 5. Reliability**
- 6. Security**
- 7. Maintainability**
- 8. Portability**

Description

- Accuracy and completeness of functions
- Response time, resource usage
- Co-existence and interoperability
- Ease of learning and using the system
- Stability and fault tolerance
- Confidentiality, integrity, authentication
- Ease of analysis, change, testing
- Ease of transfer between environments

Capability Maturity Model (CMM)

- ▶ The Capability Maturity Model (CMM) is a tool used to improve and refine software development processes.
- ▶ It provides a structured way for organizations to assess their current practices and identify areas for improvement.
- ▶ CMM consists of five maturity levels: initial, repeatable, defined, managed, and optimizing.
- ▶ By following the CMM, organizations can systematically improve their software development processes, leading to higher-quality products and more efficient project management.
- ▶ Each level of maturity shows a process capability level.
- ▶ All the levels except level 1 are further described by Key Process Areas (KPA).



Key Process Areas (KPA)

Each of these KPA (Key Process Areas) defines the basic requirements that should be met by a software process to satisfy the KPA and achieve that level of maturity.

Levels of Capability Maturity Model (CMM) are as following below.

1. Level One : Initial – Work is performed informally.

A software development organization at this level is characterized by AD HOC activities (organization is not planned in advance.).

2. Level Two : Repeatable – Work is planned and tracked.

This level of software development organization has a basic and consistent project management processes to **TRACK COST, SCHEDULE, AND FUNCTIONALITY**. The process is in place to repeat the earlier successes on projects with similar applications.

3. Level Three : Defined – Work is well defined.

At this level the software process for both management and engineering activities are **DEFINED AND DOCUMENTED**.

4. Level Four : Managed – Work is quantitatively controlled.

Software Quality management – Management can effectively control the software development effort using **precise measurements**. At this level, organization set a quantitative quality goal for both software process and software maintenance.

Quantitative Process Management – At this maturity level, The performance of processes is controlled using statistical and other **quantitative techniques**, and is quantitatively predictable.

5. Level Five : Optimizing – Work is Based Upon Continuous Improvement.

The key characteristic of this level is focusing on **CONTINUOUSLY IMPROVING PROCESS performance**.

Key features are:

Process change management

Technology change management

Defect prevention

Tools and Techniques for Quality Control

- Quality Control in software engineering ensures that the product meets the required quality standards.
- It focuses on detecting defects in the product.
- Below are the most commonly used tools and techniques for QC, often referred to as part of **the Seven Basic Quality Tools** (from manufacturing, also adapted in software):



1. Checksheet

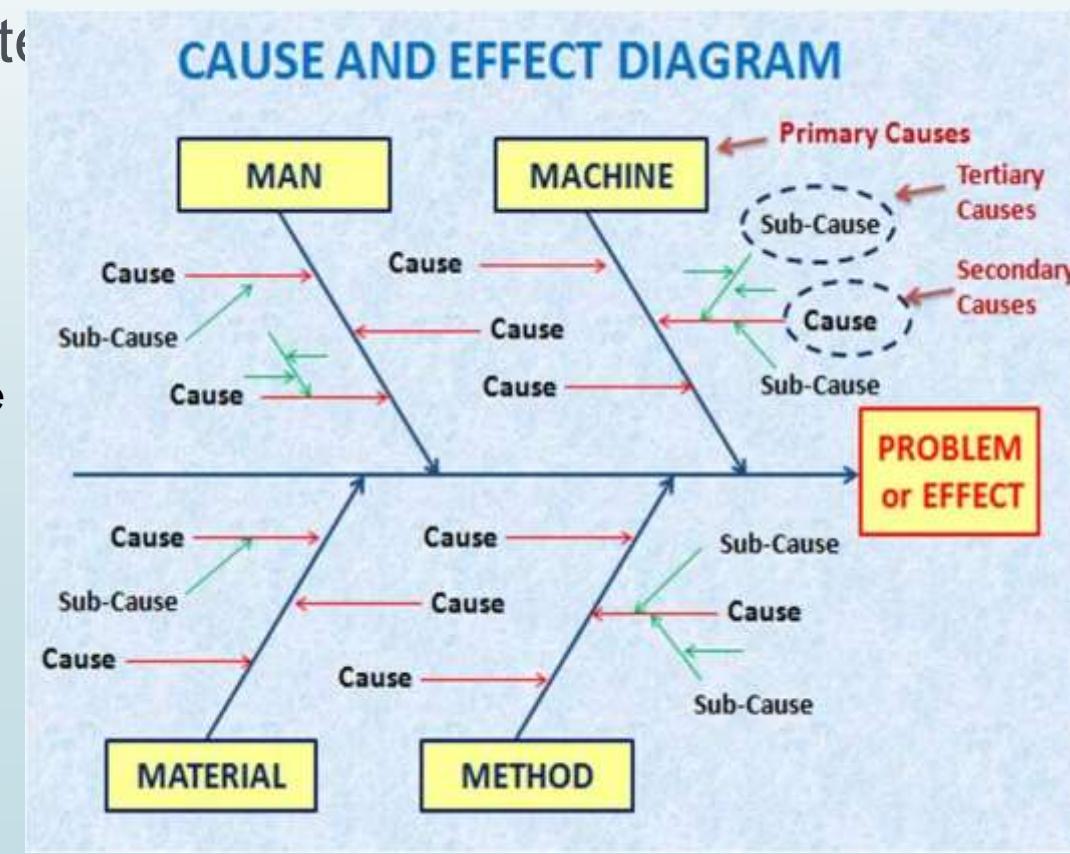
The **check sheet** is used for collecting, recording, and analyzing the data. Data collection is an important activity in the problem-solving process as it provides a basis for further action. Data may be numerical, observations and opinions, etc.

	Monday	Tuesday	Wednesday	Thursday	Friday	Total
Wrong item in the pick location						17
Item was picked incorrectly						19
Two customers received each other's items						4
Product was damaged						7
Total	13	8	8	12	6	47

2. Fishbone Diagram

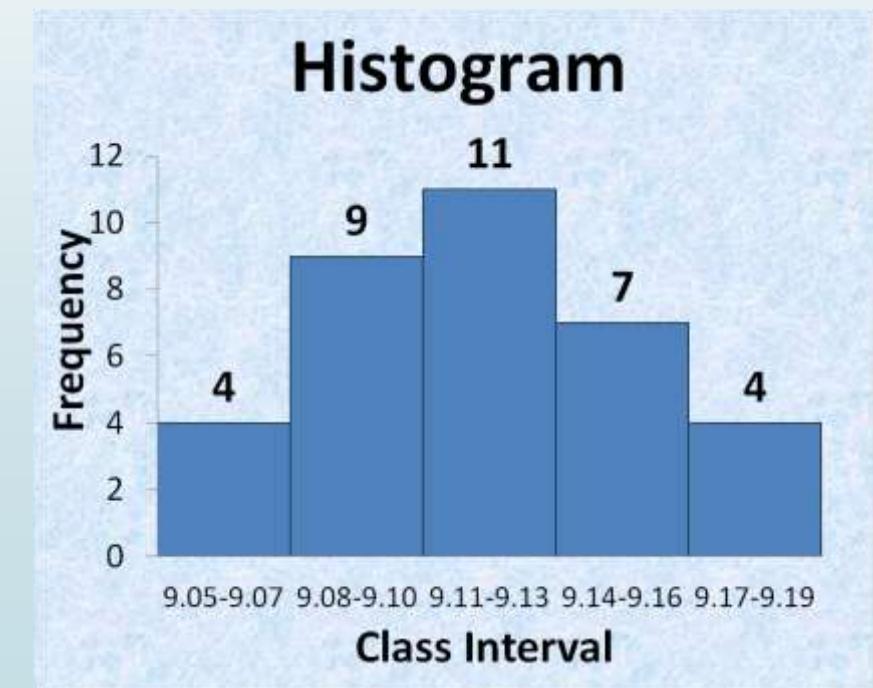
Fishbone diagram is also called as Cause and Effect diagram and Ishikawa diagram. It helps to Identify all possible potential causes and select the real/best potential cause which contributes to the problem/effect. The brainstorming technique is used for potential causes.

In a brainstorming session, all 4M or 6M factors are taken into consideration to identify the potential causes. 4M or 6M factors are – Man, Machine, Method, Material, Measurement, and Mother nature also called Environment.



3. Histogram

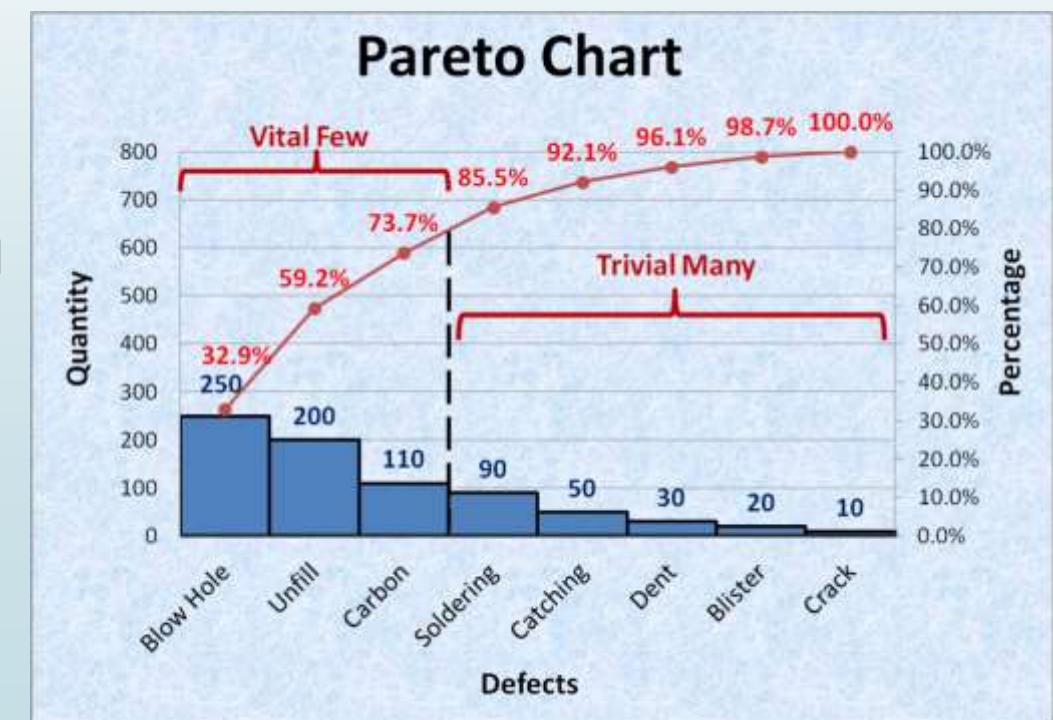
A Histogram is a **pictorial representation of a set of data**, and the most commonly used bar graph for showing frequency distributions of data/values. Histogram frequency distribution chart is widely used in Six Sigma problem solving process.



4. Pareto Chart

The Pareto chart helps to Narrow the problem area or prioritize the significant problems for corrective measures. The pareto principle is based on the 80-20 rule. It means **that 80 percent of the problems/failures are caused by 20 percent of the few major causes/factors** which are often referred to as **Vital Few**.

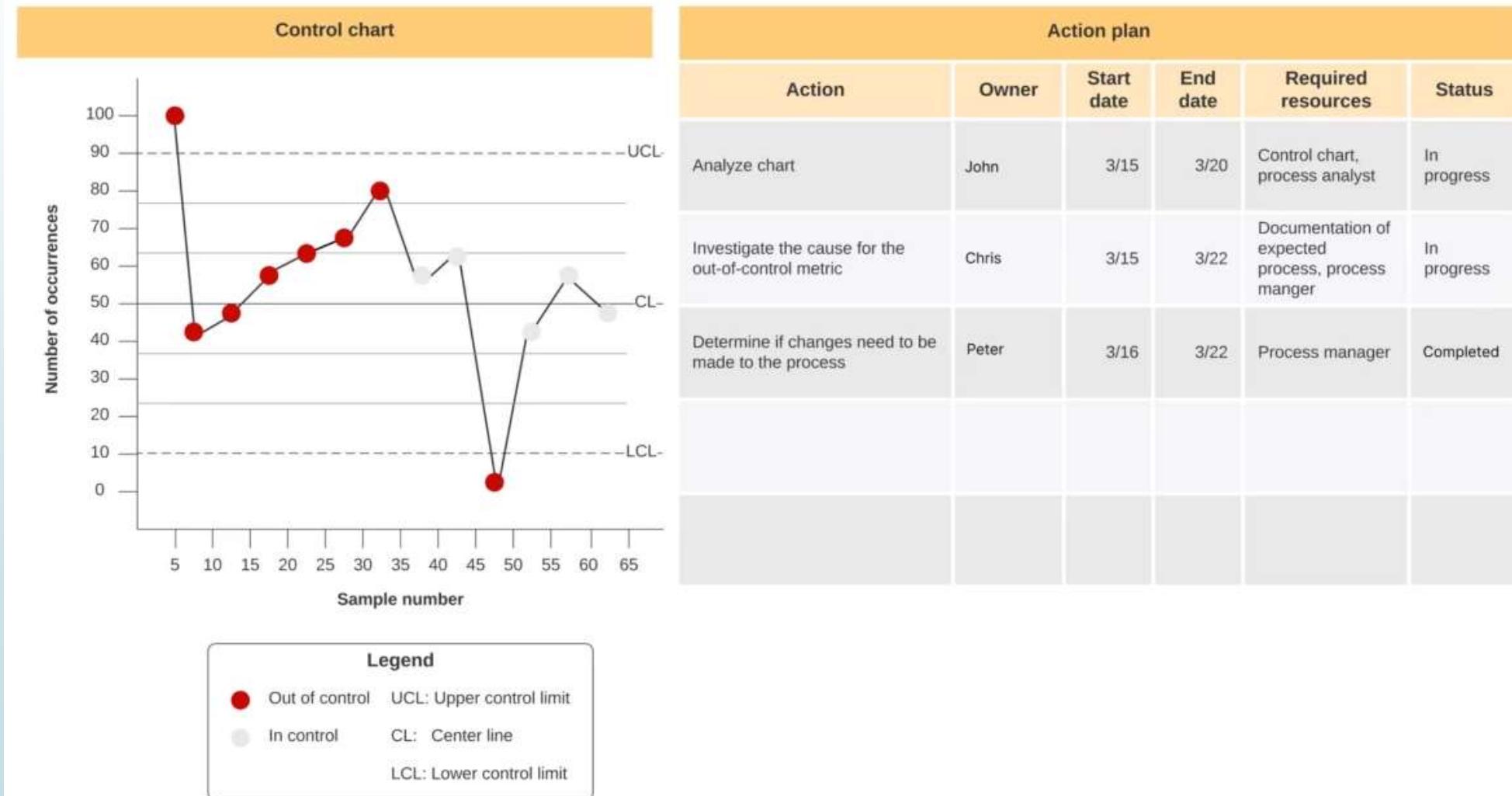
And the remaining 20 percent of the problems are caused by 80 percent of many minor causes which are referred to as **Trivial Many**. Hence, it gives us information about Vital few from Trivial many.



5. Control Chart

30

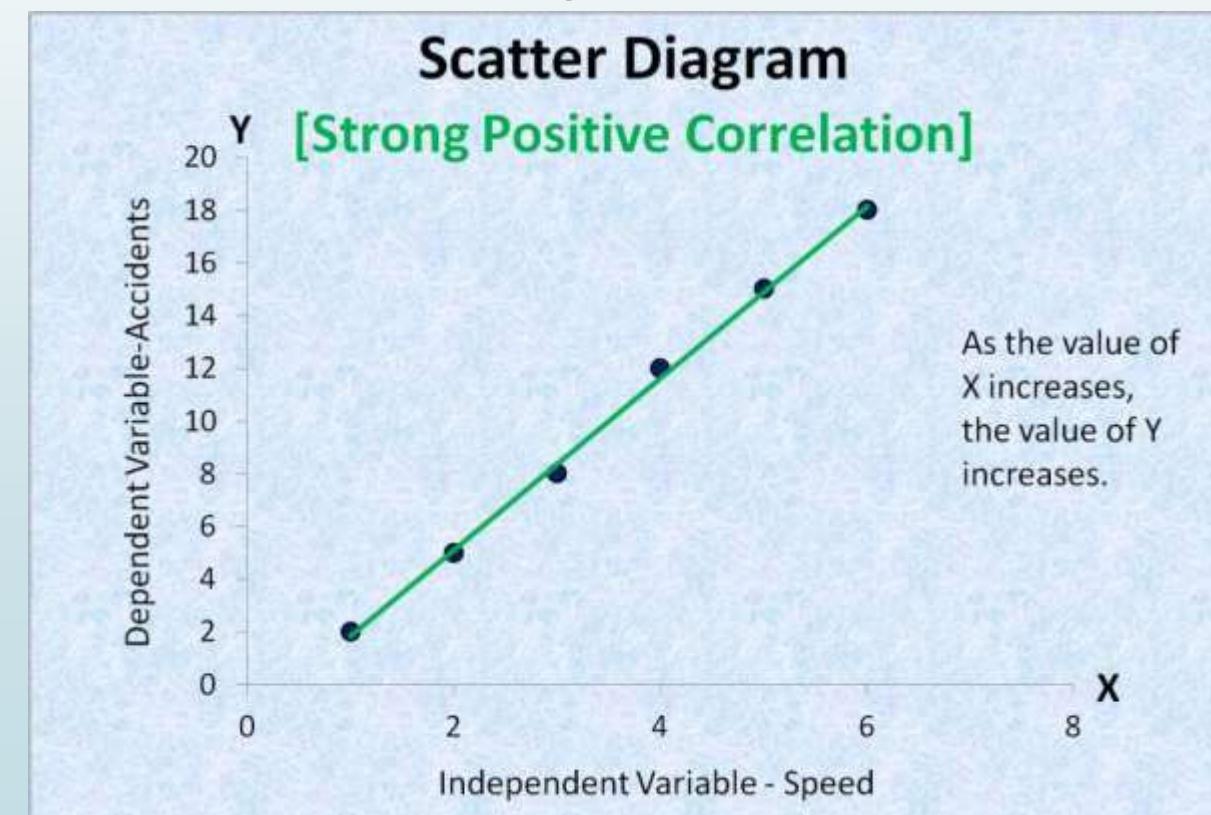
Control charts, also known as Shewhart charts, are powerful statistical tools used for monitoring and analyzing process performance over time.



6. Scatter Diagram

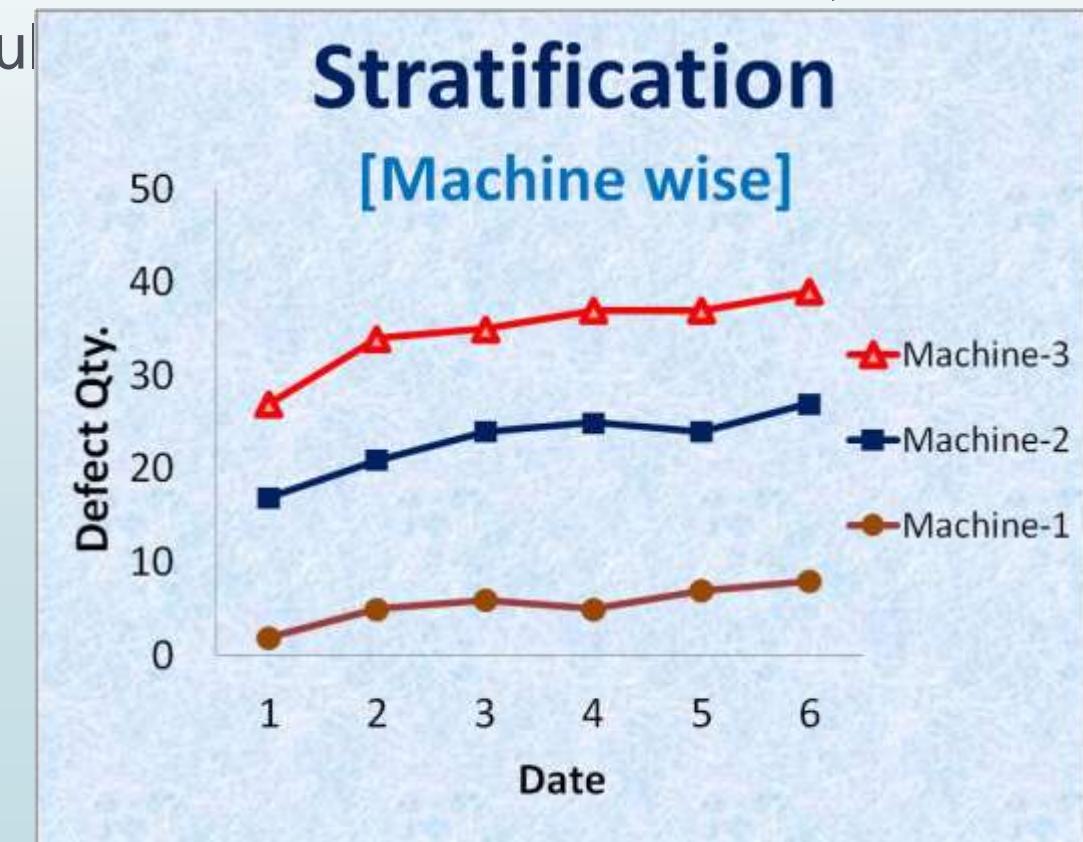
A Scatter diagram is also known as Correlation Chart, Scatter Plot, and Scatter Graph. A Scatter graph is used to find out the **relationship between two variables**. In other words, it shows the relationship between two sets of numerical data. Scatter graph shows a Positive or Negative correlation between two variables.

Independent variable data and dependent Variable data are customarily plotted along the horizontal X-axis and Vertical Y-axis respectively. Independent variable is also called controlled parameters.



7. Stratification Diagram

A technique used to analyze and divide a universe of data into homogeneous groups is called -Strata. Stratification tools are used when the data come from different sources or conditions, such as data collected from different shifts, machines, people, days, suppliers and popu



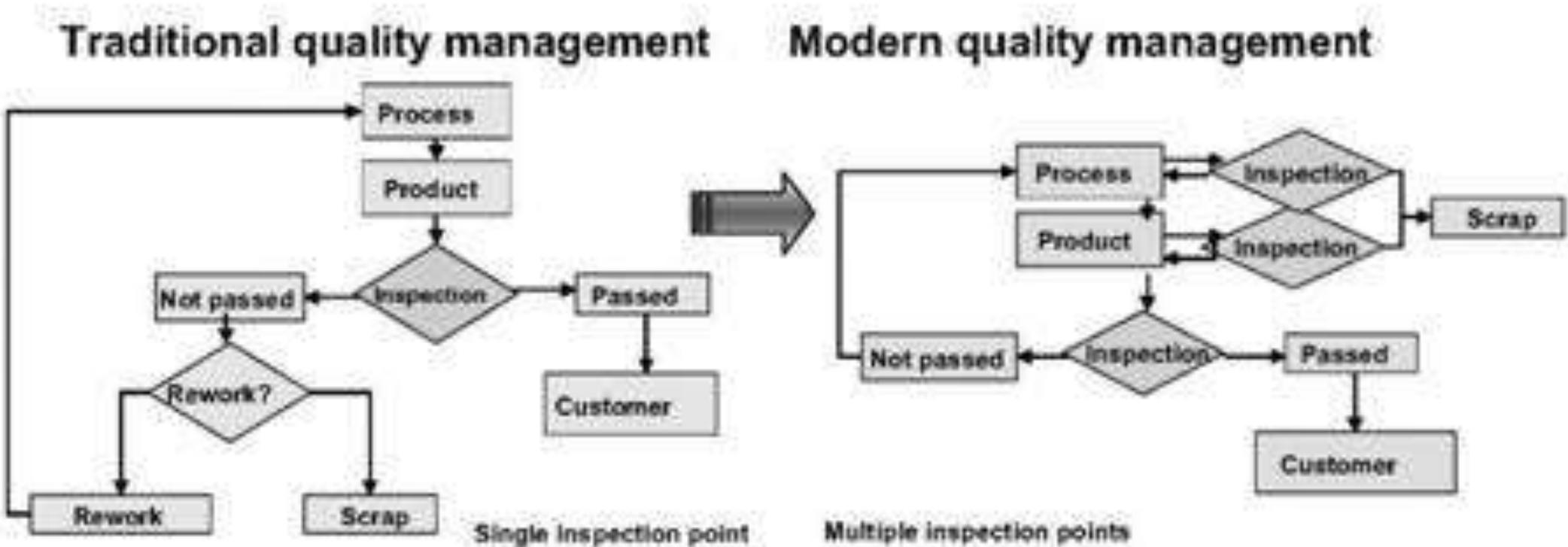
Example:

An Indian software company like **Wipro** or **Infosys** might use these tools for quality improvement. A **Check Sheet** could track types of bugs found during testing. A **Pareto Chart** could identify the modules causing the most bugs. A **Cause-and-Effect Diagram** could explore reasons for project delays. A **Histogram** might analyze the distribution of task completion times. A **Scatter Diagram** could check if developer experience correlates with bug rates. A **Flow Chart** documents the software development lifecycle. **Control Charts** might monitor metrics like code complexity or defect density over time.

Modern Quality Management

Modern Quality Management (MQM) refers to contemporary practices, tools, and principles that ensure high-quality software while emphasizing customer satisfaction, continuous improvement, and collaborative development.

Changes in management concepts corrections to be made before additional work is done. Scrap and rework cost is significantly reduced in MQM.



□ Key Principles of Modern Quality Management

1 □ Customer-Centric Approach

- Quality is defined by how well the product **meets customer needs** and expectations.
- Involves **user feedback, continuous delivery, and fast iterations.**

2 □ Total Quality Management (TQM)

- **Everyone in the organization is responsible for quality.**
- Focus on **process improvement, employee involvement, and long-term success.**

3 □ Continuous Improvement (Kaizen)

- Regular and incremental improvements in processes, code, and performance.
- Encourages **experimentation, learning, and feedback loops.**

4 Risk Management and Prevention

- Focus on **defect prevention** rather than detection.
- Emphasizes **early testing**, code reviews, static analysis, etc.

5 Process Metrics & Automation

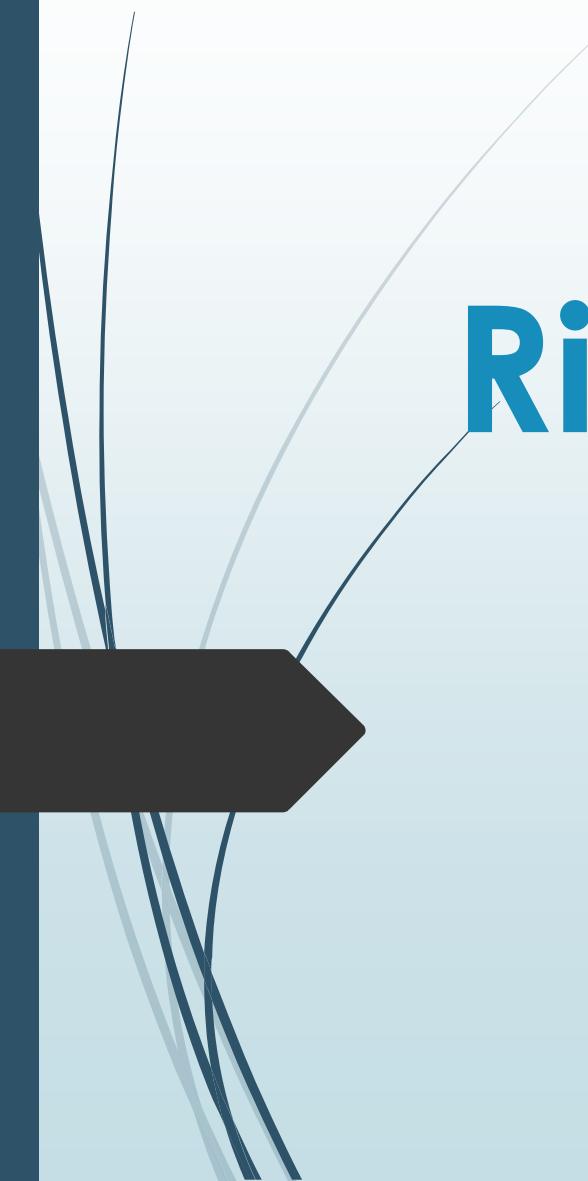
- Using **automated tools** (e.g., CI/CD pipelines, automated testing).
- Collecting and analyzing metrics like **code coverage**, **defect density**, and **deployment frequency**.

vs Traditional vs Modern Quality Management

Feature/Aspect	<input type="checkbox"/> Traditional Quality Management	 Modern Quality Management
Focus	Product-centric	Customer-centric
Quality Control	After development (post-mortem)	Continuous, during development
Processes	Waterfall	Agile, DevOps, Lean
Responsibility	QA Team only	Everyone (developers, testers, ops)
Feedback	Late	Fast, continuous
Tools	Manual testing, documentation	Automation, CI/CD, analytics
Documentation	Heavy	Lightweight, just enough
Improvement	Periodic	Continuous (Kaizen)
Standards	ISO 9001, CMM	ISO 25010, CMMI, Six Sigma
Customer Involvement	Limited	High (feedback loops, demos)



Thank You



Risk Management

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Risk Management
- ▶ Types
- ▶ Process
- ▶ Risk Analysis and Assessment
- ▶ Risk Strategies
- ▶ Risk Monitoring and Control
- ▶ Risk Response and Evaluation

Risk Management

A risk is a probable problem; it might happen, or it might not. There are main two characteristics of risk.

- **Uncertainty:** the risk may or may not happen which means there are no 100% risks.
- **Loss:** If the risk occurs in reality, undesirable results or losses will occur.

What is Risk Management?

Risk Management is a systematic process of recognizing, evaluating, and handling threats or risks that have an effect on the finances, capital, and overall operations of an organization.

These risks can come from different areas, such as financial instability, legal issues, errors in strategic planning, accidents, and natural disasters.

The main goal of risk management is to predict possible risks and find solutions to deal with them successfully.

IMPORTANCE

Risk management is important because it helps organizations to prepare for **unexpected circumstances that can vary from small issues to major crises.**

By actively understanding, evaluating, and planning for potential risks, organizations can protect their financial health, continued operation, and overall survival.

6

Let's Understand why risk management important with an example.

Suppose In a software development project, one of the key developers unexpectedly falls ill and is unable to contribute to the product for an extended period.

One of the solution that organization may have , The team uses collaborative tools and procedures, such as shared work boards or project management software, to make sure that each member of the team is aware of all tasks and responsibilities, including those of their teammates.

An organization must focus on providing resources to minimize the negative effects of possible events and maximize positive results in order to reduce risk effectively.

Organizations can more effectively identify, assess, and mitigate major risks by implementing a consistent, systematic, and integrated approach to risk management.

Types of Risk

There are mainly 3 classes of risks that may affect a computer code project:

1. Project Risks
2. Technical Risks
3. Business Risks

1. Project Risks:

Project risks concern various sorts of monetary funds, schedules, personnel, resources, and customer-related issues. A vital project risk is schedule slippage. Since computer code is intangible, it's tough to observe and manage a computer code project. It's tough to manage one thing that can not be seen. For any producing project, like producing cars, the project manager will see the merchandise taking form.

For example, see that the engine is fitted, at the moment the area of the door unit is fitted, the automotive is being painted, etc. so he will simply assess the progress of the work and manage it. The physical property of the merchandise being developed is a vital reason why several computer codes come to suffer from the danger of schedule slippage.

2. Technical Risks:

Technical risks concern potential style, implementation, interfacing, testing, and maintenance issues. Technical risks conjointly embody ambiguous specifications, incomplete specifications, dynamic specifications, technical uncertainty, and technical degeneration. Most technical risks occur thanks to the event team's lean information concerning the project.

3. Business Risks:

This type of risk embodies the risks of building a superb product that nobody needs, losing monetary funds or personal commitments, etc.

Classification of Risk in a project

Example: Let us consider a satellite-based mobile communication project. The project manager can identify many risks in this project. Let us classify them appropriately.

- What if the project cost escalates and overshoots what was estimated? – **Project Risk**
- What if the mobile phones that are developed become too bulky to conveniently carry? **Business Risk**
- What if call hand-off between satellites becomes too difficult to implement? **Technical Risk**

Risk Management Process

Risk management is a sequence of steps that help a software team to understand, analyze, and manage uncertainty.

Risk management process consists of

- Risks Identification.
- Risk Assessment.
- Risks Planning.
- Risk Monitoring

RISK MANAGEMENT PROCESS



Risk Identification

Risk identification refers to the systematic process of recognizing and evaluating potential threats or hazards that could negatively impact an organization, its operations, or its workforce. This involves identifying various types of risks, ranging from IT security threats like viruses and phishing attacks to unforeseen events such as equipment failures and extreme weather conditions.

Risk analysis

Risk analysis is the process of evaluating and understanding the potential impact and likelihood of identified risks on an organization. It helps determine how serious a risk is and how to best manage or mitigate it. Risk Analysis involves evaluating each risk's probability and potential consequences to prioritize and manage them effectively.

Risk Planning

Risk planning involves developing strategies and actions to manage and mitigate identified risks effectively. It outlines how to respond to potential risks, including prevention, mitigation, and contingency measures, to protect the organization's objectives and assets.

Risk Monitoring

Risk monitoring involves continuously tracking and overseeing identified risks to assess their status, changes, and effectiveness of mitigation strategies. It ensures that risks are regularly reviewed and managed to maintain alignment with organizational objectives and adapt to new developments or challenges.

Risk Assessment

- ▶ Risk assessment is a systematic process of evaluating potential risks that may be involved in a projected activity, undertaking, or business decision.
- ▶ It aims to identify, analyze, and evaluate the likelihood and impact of risks to determine appropriate measures to mitigate or manage them effectively.
- ▶ A broader process that includes **risk analysis** and adds **risk evaluation**.
- ▶ **Risk analysis:** Think of it as the "What could go wrong?" phase.
- ▶ **Risk assessment:** Think of it as "Should we be worried, and what should we do about it?"

Objective of Risk Assessment

The objective of Risk Assessment is to rank the risks in terms of their harm inflicting potential. For risk assessment, initial every risk ought to be rated in 2 ways:

- ▶ The chance of a risk coming back true (denoted as r).
- ▶ The consequence of the issues related to that risk (denoted as s).

Based on these 2 factors, the priority of every risk is computed:

$$p=r*s$$

Where p is the priority with which the danger should be handled, r is the likelihood of the danger changing into true, and s is the severity of harm caused by the danger changing into true. If all known risks are prioritized, then the foremost probably and damaging risks are handled initial and a lot of comprehensive risk abatement procedures are designed for these risks.

Risk Assessment Steps

The risk assessment process typically involves several key steps to ensure that risks are properly identified, evaluated, and managed:

- **Identify Risks:** The first step is to identify potential risks that could affect the project or organization. This involves gathering information from various sources, such as stakeholder input, historical data, and expert opinions, to recognize possible threats or vulnerabilities.
- **Analyze Risks:** Once risks are identified, they are analyzed to determine their potential impact and likelihood. This step involves evaluating how these risks could affect objectives and what the consequences might be. The analysis helps in understanding the severity and urgency of each risk.
- **Evaluate Risks:** In this step, risks are prioritized based on their potential impact and likelihood. This evaluation helps to determine which risks are the most significant and need immediate attention. It often involves comparing risks to established criteria or benchmarks.
- **Mitigate Risks:** After evaluating risks, strategies are developed to manage or mitigate them. This can include avoiding the risk, reducing its impact, transferring it to another party, or accepting it if it is within acceptable limits.
- **Monitor and Review:** The final step involves continuously monitoring risks and reviewing the effectiveness of the mitigation strategies. This ensures that risk management efforts remain relevant and effective as conditions change.

How to Use a Risk Assessment Matrix

A risk assessment matrix is a tool used to evaluate and prioritize risks based on their likelihood and impact. It is typically represented as a grid, where the x-axis represents the probability of a risk occurring, and the y-axis represents the potential impact of the risk.

How to Use It:

- **Plot Risks:** Identify and plot each risk on the matrix according to its likelihood and impact.
- **Categorize Risks:** Risks are categorized into different levels, such as low, medium, or high, based on their position on the matrix.
- **Prioritize Actions:** The matrix helps prioritize which risks require immediate attention and which can be monitored over time. High likelihood and high impact risks are addressed first, while low likelihood and low impact risks may require less urgent responses.

Risk Analysis

17

- Software risk analysis in Software Development involves identifying which application risks should be tested first.
- Risk is the possible loss or harm that an organization might face.
- Risk can include issues like project management, technical challenges, resource constraints, changes in requirements, and more
- **Finding every possible risk and estimating are the two goals of risk analysis.**
- Think about the potential consequences of testing your software and how it could impact your software when creating a test plan.
- Risk detection during the production phase might be costly.
- **Therefore, risk analysis in testing is the best way to figure out what goes wrong before going into production.**

Types of risk analysis

In risk management, you can perform two types of risk analysis:

Qualitative Risk Analysis

Quantitative Risk Analysis

1. Qualitative Risk Analysis:

- ▶ In qualitative risk analysis, you will examine risks for severity and likelihood of occurrence during the project.
- ▶ Afterward, you will find the risk ratings using **the risk assessment matrix**. The risk rating can be high, serious, medium, and low.
- ▶ The risk assessment matrix is a great communication tool to communicate the risk information to stakeholders to decide on the next action.

20

As shown below, you will order the risks on a risk assessment matrix.

Risk Assessment Matrix				
Severity				
	Catastrophic - 4	Critical - 3	Marginal - 2	Negligible - 1
Frequent - 4	High (16)	High (12)	Serious (8)	Medium (4)
Probable - 3	High (12)	Serious (9)	Serious (6)	Medium (3)
Remote - 2	Serious (8)	Serious (6)	Medium (4)	Low (2)
Improbable - 1	Medium (4)	Medium (3)	Low (2)	Low (1)

To provide a better understanding, let's take a look at the below **risk register** that is created during identify risks process. This risk register includes both positive and negative risks.

As shown in the table above, in qualitative risk analysis, probability and impact values are determined for each risk and the risk score is calculated by multiplying them. Depending on the risk scores, you prioritize each risk and establish risk response strategies for both positive and negative risks if the score exceeds risk tolerance limits.

Risk ID	Risk Description	Probability	Impact	Risk Score
01	Insufficient Amount of Coders	2	4	8
02	Delayed Delivery	2	2	4
03	Need for Advanced Testing to Eliminate Defects	3	5	15
04	Reuse of Existing Testing System	4	5	20

2. Quantitative Risk Analysis:

- ▶ After performing a qualitative risk analysis, the quantitative risk analysis process is carried out. Organizations utilize this procedure for significant and complicated initiatives since it demands resources and time.
- ▶ Calculating risk using objective data is known as quantitative risk analysis.
- ▶ During this process, you will use verifiable data to examine the potential impact of risks on your project objectives, such as schedule baseline, cost baseline, scope baseline, etc.

In the below table, we will perform quantitative risk analysis by assigning monetary values for the impact of each risk and percentages for their probability of occurrence.

Risk ID	Risk Description	Probability	Impact	Expected Monetary Value (EMV)
01	Insufficient Amount of Coders	40%	\$30.000	\$12.000
02	Delayed Delivery	40%	\$40.000	\$16.000
03	Need of Advanced Testing to Eliminate Defects	60%	\$25.000	\$15.000
04	Reuse of Existing Testing System	80%	\$45.000	\$36.000

Qualitative Risk Analysis	Quantitative Risk Analysis
Risk level	Project level
Easy to perform	Time-consuming
Does not require a software tool	Software tools facilitate the process
Subjective evaluation of probability and impact	Objective/numeric evaluation of probability and impact
Should be done first	Should be done after performing qualitative analysis
Analyzes the effects of individual project risks	Analyzes the combined effects of risks as a whole to specify an overall project risk.
Useful in all-size projects	Useful especially in large and sophisticated projects
Risk scale and scores are qualitative	Risk scale and scores are quantitative -often specified in monetary and duration terms
Not Applicable	Determines the amount of Contingency Reserve
Provides quick information	Provides detailed information

Risk Strategies

25

- Risk strategies define how you will handle a risk once it is identified and assessed. They are part of the Risk Response Planning phase in Risk Management.

Risk Management Strategies:

- Risk Avoidance
- Risk Reduction
- Risk Sharing
- Risk Transfer
- Contingency Planning

1. Risk Avoidance

- ▶ Risk Avoidance is one of the simplest risk management strategies.
- ▶ It involves completely avoiding activities that could lead to risk.
- ▶ For example, a business might decide not to enter a highly volatile foreign market to avoid the risk of financial loss.
- ▶ The main advantage of risk avoidance is that it eliminates the possibility of facing negative outcomes from the avoided activities.
- ▶ However, this strategy also means missing out on potential opportunities that taking the risk might have provided.

When to use it?

The best time to use risk avoidance is when the potential risks far outweigh the expected benefits.

It's particularly useful when the consequences of a risk could be catastrophic to the business or project.

Although risk avoidance can seem like a conservative approach, it's a prudent choice when dealing with risks that can have severe legal, financial, or health-related repercussions.

27

Pros:

Completely eliminates specific risks.

Reduces uncertainties and potential costs.

Cons:

Limits opportunities for growth or profit.

Can lead to stagnation if overused.

Example:

A tech company might avoid storing sensitive user data to circumvent the risks associated with data breaches and the resultant legal consequences.

2. Risk Reduction



Risk Reduction Strategy

What is it?

Risk Reduction aims to minimize the potential impact or likelihood of a risk.

This might involve installing safety systems to reduce workplace accidents or improving software security to fend off cyber attacks.

When to use it?

This strategy is ideal when risks cannot be avoided but can be controlled to a manageable level. Risk reduction is essential in industries like manufacturing, where potential hazards are prevalent.



Pros:

- Decreases the severity and/or likelihood of risks.
- Allows for safer pursuit of potentially risky activities.

Cons:

- May involve significant upfront or ongoing costs.
- Does not eliminate risks completely.

Example: A construction company using high-quality materials and strict safety protocols to reduce the risk of structural failures and worker injuries.

3. Risk Sharing

- ▶ Risk Sharing is a collaborative tactic among risk management strategies where risk is distributed among multiple parties.
- ▶ By sharing risks, for instance through partnerships or outsourcing, companies can leverage external expertise and reduce their own exposure to potential losses.
- ▶ This is common in projects that are too large or complex for a single entity to manage alone.

When to use it?

- ▶ The strategy is particularly effective when engaging in new or innovative projects where the risks are high and uncertain. It is a smart choice for businesses that need to mitigate potential financial losses or when specialized knowledge is required to manage specific risks.

30

Pros:

- Spreads and mitigates risks across parties.
- Leverages collective resources and expertise.

Cons:

- Can lead to dependencies on other parties.
- May result in reduced control over project outcomes.

Example:

A small tech firm partnering with a larger company to co-develop a new software product, sharing the financial risks involved in its development and market introduction.

4. Risk Transfer



What is it?
Risk Transfer involves moving potential risks to a third party, usually through insurance or contractual agreements.

When to use it?
Risk transfer is particularly useful in situations where the potential financial impact of a risk could be devastating. It's commonly employed in industries prone to large-scale risks, such as construction, healthcare, and information technology.

Pros:

- Reduces the financial burden on the company by offloading risks.
- Provides peace of mind and financial stability in the event of major incidents.

Cons:

- Can be costly in terms of insurance premiums or contract costs.
- Sometimes leads to a false sense of security, potentially causing neglect of other risk management practices.

intellspot.com

Example:

A tech startup might use a cloud service provider to host data, thereby transferring the risk of data center management and potential downtime to the provider.

5. Contingency Planning

- ▶ Contingency Planning involves preparing alternative action plans that can be implemented if initial plans fail due to unforeseen risks.
- ▶ This strategy is about readiness and flexibility, ensuring that a business can continue operations under various scenarios.
- ▶ It often includes developing detailed recovery plans, backup operations, and maintaining reserves of essential resources.



When to use it?

Contingency planning is crucial for businesses in environments where sudden changes can occur, such as natural disasters, economic instability, or technology failures. It allows organizations to react quickly and effectively, minimizing downtime and operational losses.

Pros:

- Enhances organizational resilience against unexpected events.
- Helps maintain business continuity during crises.

Cons:

- Requires significant time and resources to develop effective contingency plans.
- May not be used if no crisis occurs, leading some to view resources spent on contingency planning as non-productive.

Example:

- A bank might have contingency plans for IT system failures that include manual transaction processing methods and secondary data centers to ensure continuous customer service.

Risk Monitoring and Control

Risk monitoring and control begins at the start of projects when all potential and known risks are identified, and then just as importantly, continues throughout a project as those initial risks are continuously tracked while new risks are also identified as work continues, changes and progress.

Activity	Description
Reviewing risk triggers	Check for early signs that a risk may occur.
Tracking residual risks	Monitor risks that remain after mitigation.
Reassessing risks	Regularly re-analyze and re-prioritize risks.
Identifying new risks	Watch for emerging risks throughout the project.
Evaluating risk response	Assess whether mitigation or contingency plans are working.
Updating the Risk Register	Log changes in status, ownership, or strategies.
Communicating risk status	Share updates with stakeholders and project team.

Tools for Monitoring and Controlling Project Risks

- **Risk Assessment:** It involves checking for the past identified risks and new risks, and generating risk responses.
- **Risk Audits:** The effectiveness of risk responses applied should be documented. These documents are needed in scheduled meetings to discuss the project's progress.
- **Variance and Trend Analysis:** The performance of the project in terms of the risks incurred, responses taken instead of that, the progress of the project, etc. should be monitored. This analysis helps in predicting the output of the project in the future.
- **Technical Performance Measurement:** Technical aspects of the project should be measured by testing the achievement of the project till that time. It determines the technical success of the project.
- **Reserve Analysis:** Reserves of the risk should be analyzed for deciding on their adequacy in the complete project tenure.
- **Status Meetings:** Regular team meetings should be conducted for all the issues and achievements of the project.

Outputs for Monitoring and Controlling Project Risks

- **Work Performance Information:** It includes a record of risks occurring in the project throughout the term and whether a response is needed for the same or they can be ignored. It also includes schedule progress, deliverables status, costing, the requirement of resources, etc.
- **Organizational Process Assets Updates:** They involve templates for risk planning, risk breakdown structure, and lessons learned from previous issues and errors.
- **Change Requests:** They involve modifications in the project plan as per the need, such as adding the extra budget, extending the schedule, including more resources, etc. for dealing with emergency purposes.

Risk Response and Evaluation

Risk response planning is the final step in the risk management process. In this process, risk responses are developed. The risk response can be drawn based on three categories of risks, viz. controllable known risks, uncontrollable known risks, and unknown risks. While risk responses can be developed for known risks, no response can be developed for unknown risks.

Now let us study how a risk response is created for controllable and uncontrollable known risks.

- Controllable Known Risks
- Uncontrollable Known Risks

Controllable Known Risks

Risk response for controllable known risks involves risk prevention. Risk prevention can be achieved in two ways, which are:

Risk Avoidance

As the name suggests, the risk avoidance approach is all about eliminating risks and not taking them at all. The simplest way of avoiding a risky project is not to take that project.

Risk Mitigation

It involves taking measures for reducing the probability of the occurrence of a risk and its impact on the overall project. It is not meant to eliminate the risk, but to reduce the risk exposure. Providing safety training and safety gear to construction workers on a site is for mitigating the risk of injury or accident. It does not eliminate the risk, but it certainly reduces the probability or impact of risk. Similarly, some risks are not mitigated within a project but are transferred to others.

Uncontrollable Known Risks

Risk response for uncontrollable known risks involves reactive response after a risk event has occurred. It may be done through contingency planning, where a contingency budget is set aside to take care of the risk event.

- For example, assume that 2% of the inventory will get damaged, destroyed, or stolen and will have to be purchased again. However, the budget for the additional 2% inventory can be set aside as a contingency budget if, in case, the inventory damage exceeds the quantity already factored in.
- The other possibility, concerning dealing with effects, is that you may simply “accept” the risk if it occurs and not respond. It can be for two extreme reasons:
 - If the loss in case of a risk event is too trivial, you may not want to respond to such a risk since planning the response for it might be costlier.
 - If the loss in case of a risk event is massive and beyond control, the only option is to “accept” the risk.

41

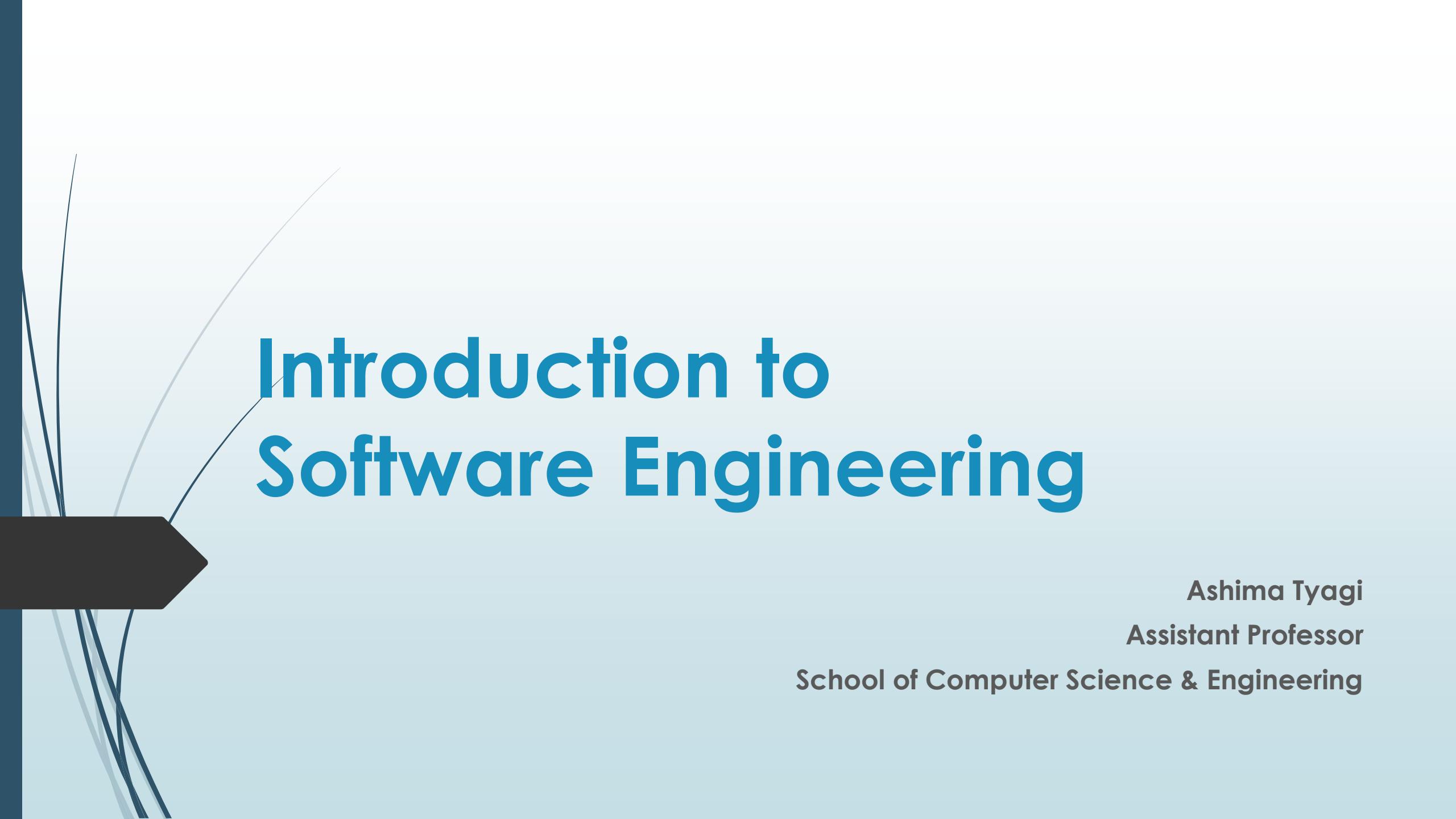
Several risk response strategies are being developed in the planning process and the one best suited for the risk applied at that time to deal with risks.

The risk response actions are based on the 4Ts:

- **Terminate:** Risk termination is done by changing the project plan to terminate the impact of risk on the project. If the termination of the risk is not possible, it may be avoided by taking pre-emptive actions.
- **Treat:** Treatment involves reducing the probability of risk by taking actions in advance at an early stage. It may require extra resources, an extra schedule, and modification in a plan to minimize risks.
- **Tolerate:** Tolerance doesn't ask for any changes in the project plan. Risks are borne as no strategies can be applied to that.
- **Transfer:** A third party is hired to deal with the risk and therefore, the project risk is transferred to them



Thank You



Introduction to Software Engineering

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Software & Engineering
- ▶ Software Engineering
- ▶ Software components
- ▶ Software characteristics
- ▶ Software crisis
- ▶ Software Attributes

Software & Engineering??

- Software is a program or set of programs containing instructions that provide desired functionality.
- Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems.

Software Engineering

Software Engineering is the process of designing, developing, testing, and maintaining software. It is a systematic and disciplined approach to software development that aims to create high-quality, reliable, and maintainable software.

Software Components

1. Program –

- A computer program is a list of instructions that tell a computer what to do.
- In software engineering, a program is a set of instructions written in a programming language to perform a specific task or solve a particular problem.
- It consists of algorithms, data structures, and control flow mechanisms, organized systematically to achieve the desired functionality within a computer system.

2. Documentation –

- Source information about the product contained in design documents, detailed code comments, etc.
- Software documentation in software engineering encompasses written descriptions of a software system's architecture, design, requirements, and functionality.
- It includes manuals, user guides, technical specifications, API documentation, and code comments.
- Documentation aids in understanding, using, maintaining, and evolving software systems throughout their lifecycle.

3. Operating Procedures –

- Set of step-by-step instructions compiled by an organization to help workers carry out complex routine operations.
- Operating procedures in software engineering are *documented guidelines outlining the steps and protocols for deploying, maintaining, and managing software systems.*
- They ensure consistency, reliability, and efficiency in software operations, covering tasks like installation, configuration, troubleshooting, and regular maintenance activities.

Software Characteristics

1. **Software is developed or engineered; it is not manufactured in the classical sense:**

Although some similarities exist between software development and hardware manufacturing, few activities are fundamentally different.

In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems than software.

2. **The software doesn't "wear out.":**

Hardware components suffer from the growing effects of many other environmental factors. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out.

When a hardware component wears out, it is replaced by a spare part.

Software Characteristics

There are no software spare parts.

Every software failure indicates an error in design or in the process through which the design was translated into machine-executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance. However, the implication is clear—the software doesn't wear out. But it does deteriorate.

3. The software continues to be custom-built:

A software part should be planned and carried out with the goal that it tends to be reused in various projects.

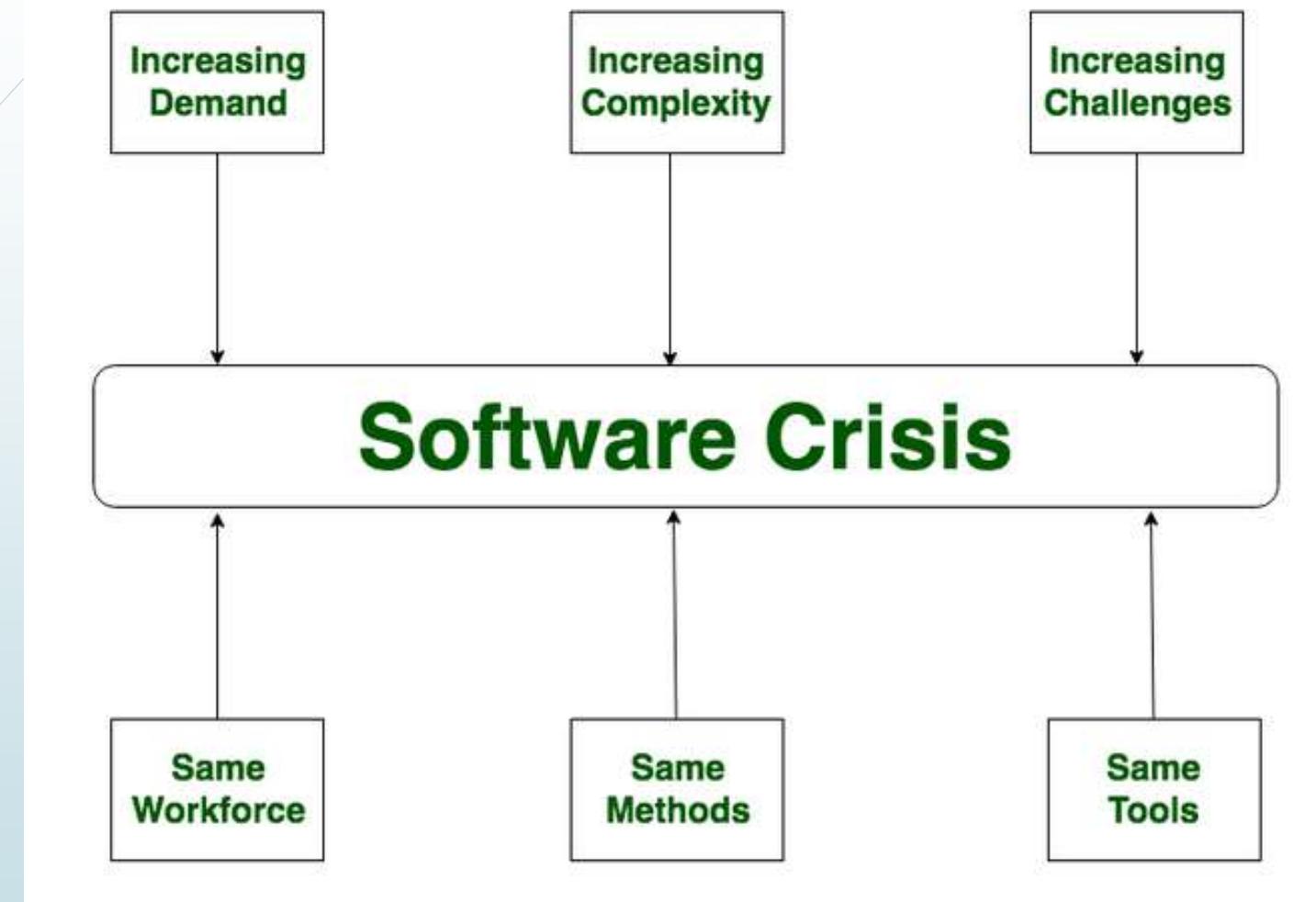
Current reusable segments encapsulate the two pieces of information and the preparation that is applied to the information, empowering the programmer to make new applications from reusable parts.

In the hardware world, component reuse is a natural part of the engineering process.

Software Crisis

Software Crisis is a term used in computer science for the difficulty of writing useful and efficient computer programs in the required time. The software crisis was due to using the same workforce, same methods, and same tools even though rapidly increasing software demand, the complexity of software, and software challenges. With the increase in software complexity, many software problems arise because existing methods were insufficient.

If we use the same workforce, same methods, and same tools after the fast increase in software demand, software complexity, and software challenges, then there arise some issues like software budget problems, software efficiency problems, software quality problems, software management, and delivery problems, etc. This condition is called a Software Crisis.



Causes of Software Crisis:

- The cost of owning and maintaining software was as expensive as developing the software.
- At that time Projects were running overtime.
- At that time Software was very inefficient.
- The quality of the software was low quality.
- Software often did not meet user requirements.
- The average software project overshoots its schedule by half.
- At that time Software was never delivered.
- Non-optimal resource utilization.
- Challenging to alter, debug, and enhance.
- The software complexity is harder to change.

Factors Contributing to Software Crisis:

- Poor project management.
- Lack of adequate training in software engineering.
- Less skilled project members.
- Low productivity improvements.

Solution of Software Crisis:

There is no single solution to the crisis. One possible solution to a software crisis is Software Engineering because software engineering is a systematic, disciplined, and quantifiable approach. For preventing software crises, there are some guidelines:

- Reduction in software over budget.
- The quality of the software must be high.
- Less time is needed for a software project.
- Experienced and skilled people working on the software project.
- Software must be delivered.
- Software must meet user requirements.

MAIN ATTRIBUTES OF SOFTWARE ENGINEERING

EFFICIENCY

Efficiency in software refers to how well the software uses system resources (like CPU, memory, disk, and network) to perform its tasks. Example: A web browser that loads pages quickly and consumes minimal RAM is considered efficient.

RELIABILITY

Reliability refers to the software's ability to consistently perform its intended functions without failure under specified conditions over time. Example: Banking software that processes transactions without errors, even during peak loads, is considered reliable.

REUSABILITY

Reusability refers to software's ability to use existing modules in new applications without modification. Example: A library of reusable functions for processing user data can be integrated into multiple projects without rewriting the code.

MAINTAINABILITY

Maintainability is the ease with which software can be updated, fixed, or improved to meet new requirements or correct defects. Example: A software where developers can quickly add new features or fix bugs is maintainable.



Thank You



Software Processes

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Software Processes
- ▶ Framework Activities
- ▶ Umbrella Activities
- ▶ SDLC
- ▶ Process Flow

Software Process

A software process is a collection of

- ▶ activities,
- ▶ actions, and
- ▶ tasks

that are performed when some work product is to be created.

4

- An **activity** strives to achieve a **broad objective** and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied. Ex: Gathering and documenting what the software should do.
- **Actions** are **smaller, specific steps** performed within an activity. They focus on achieving a particular goal or outcome within that phase. Ex: Preparing a Software Requirements Specification document.
- **Tasks** are the **smallest units of work**, often performed by individual team members. They are specific, measurable, and actionable. Ex: Write a use case for "User Login."

Process Framework

- ▶ A process framework establishes the foundation for a complete software engineering process by identifying a small number of **framework activities** that are applicable to all software projects, regardless of their size or complexity.
- ▶ In addition, the process framework encompasses a set of **umbrella activities** that are applicable across the entire software process.

Software process

Process framework

Umbrella activities

framework activity # 1

software engineering action #1.1

Task sets

- work tasks
- work products
- quality assurance points
- project milestones

⋮

software engineering action #1.k

Task sets

- work tasks
- work products
- quality assurance points
- project milestones

⋮

framework activity # n

software engineering action #n.1

Task sets

- work tasks
- work products
- quality assurance points
- project milestones

⋮

software engineering action #n.m

Task sets

- work tasks
- work products
- quality assurance points
- project milestones

Framework Activities

Framework activities are the **core, essential activities** in the software process.

A generic process framework for software engineering encompasses five activities:

- ▶ Communication
- ▶ Planning
- ▶ Modeling
- ▶ Construction
- ▶ Deployment

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

- **Communication.** Before any technical work can commence, it is critically important to **communicate and collaborate with the customer** (and other stakeholders) The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "**map**" that helps guide the team as it makes the journey. The map—called a **software project plan**—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

- **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
- **Construction.** This activity combines **code generation** (either manual or automated) and **the testing** that is required to uncover errors in the code.
- **Deployment.** The software (as a complete entity or as a partially completed increment) is **delivered to the customer** who evaluates the delivered product and provides feedback based on the evaluation.

Umbrella Activities

Umbrella activities are supportive activities that span across the entire software process to enhance the quality, manageability, and maintainability of the project. Typical umbrella activities include:

- ▶ Software project tracking and control
- ▶ Risk management
- ▶ Software quality assurance
- ▶ Technical reviews
- ▶ Measurement
- ▶ Software configuration management (effects of changes)
- ▶ Reusability management
- ▶ Work product preparation and production

SDLC

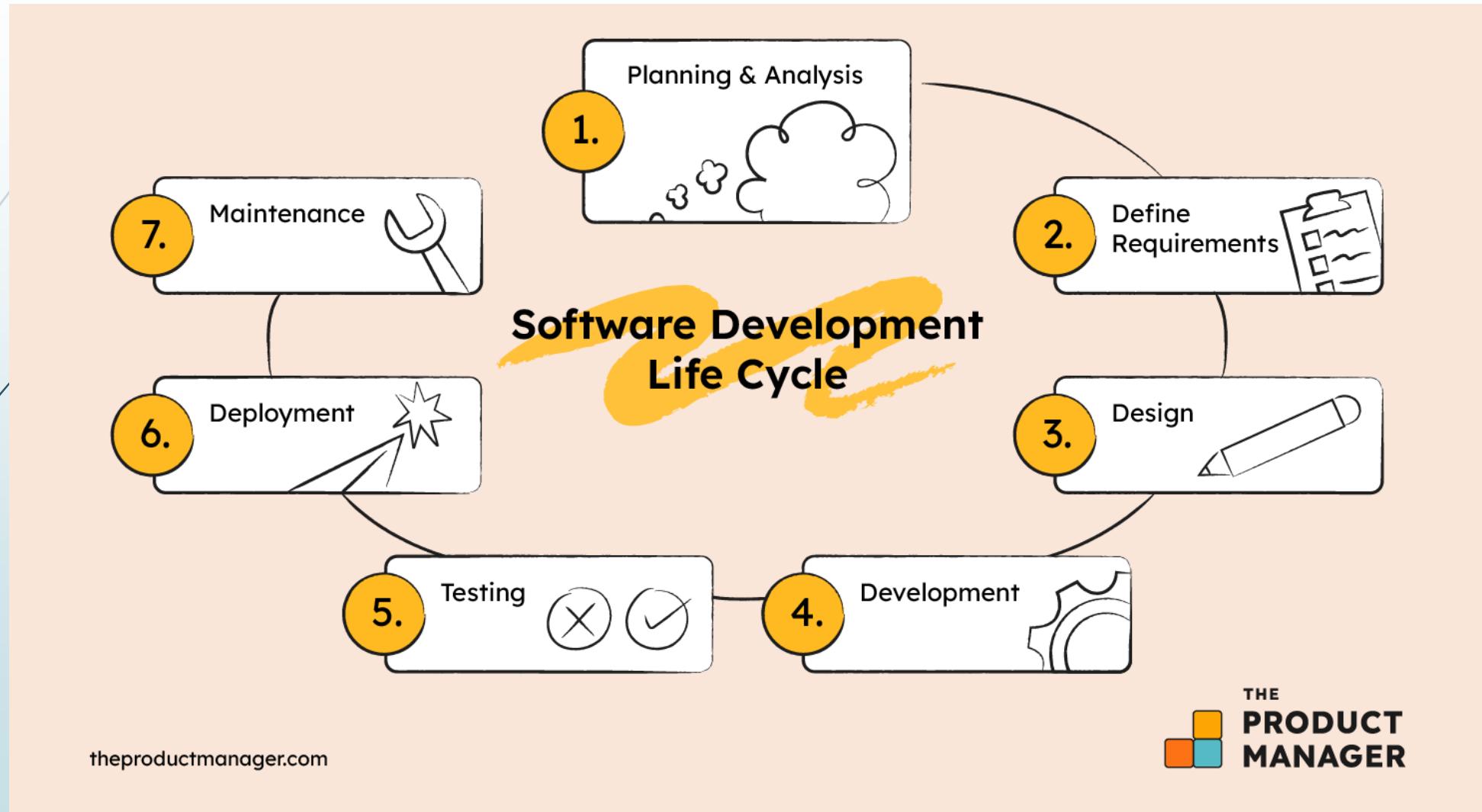
Software development life cycle (SDLC) is a structured process that is used to **design, develop, and test good-quality software**.

SDLC, or software development life cycle, is a methodology that defines the entire procedure of software development step-by-step.

Goal of SDLC:

- The goal of the SDLC life cycle model is to deliver high-quality, maintainable software that meets the user's requirements.
- SDLC in software engineering models outlines the plan for each stage so that each stage of the software development model can perform its task efficiently to deliver the software at a low cost within a given time frame that meets users' requirements.

7 Phases of SDLC



1. Planning & Analysis

The first phase of the SDLC is the project planning stage where you are gathering business requirements from your client or stakeholders. This phase is when you evaluate the feasibility of creating the product, revenue potential, the cost of production, the needs of the end-users, etc.

To properly decide what to make, what not to make, and what to make first, you can use a feature prioritization framework that takes into account the value of the software/update, the cost, the time it takes to build, and other factors.

Once it is decided that the software project is in line with business and stakeholder goals, feasible to create, and addresses user needs, then you can move on to the next phase.

2. Define Requirements

This phase is critical for converting the information gathered during the planning and analysis phase into clear requirements for the development team.

This process guides the development of several important documents: a software requirement specification (SRS) or product specification, a Use Case document, and a Requirement Traceability Matrix document.

3. Design

The design phase is where you put pen to paper—so to speak. The original plan and vision are elaborated into a software design document (SDD) that includes the system design, programming language, templates, platform to use, and application security measures. This is also where you can flowchart how the software responds to user actions.

In most cases, the design phase will include the development of a prototype model. Creating a pre-production version of the product can give the team the opportunity to visualize what the product will look like and make changes without having to go through the hassle of rewriting code.

4. Development

The actual development phase is where the development team members divide the project into software modules and turn the software requirement into code that makes the product.

This SDLC phase can take quite a lot of time and specialized development tools. It's important to have a set timeline and milestones so the software developers understand the expectations and you can keep track of the progress in this stage.

In some cases, the development stage can also merge with the testing stage where certain tests are run to ensure there are no critical bugs.

Keep in mind, different types of product development software will have different specialties so you'll want to pick the one that suits you best.

5. Testing

Before getting the software product out the door to the production environment, it's important to have your quality assurance team perform validation testing to make sure it is functioning properly and does what it's meant to do. The testing process can also help hash out any major user experience issues and security issues.

In some cases, software testing can be done in a simulated environment. Other simpler tests can also be automated.

6. Deployment

During the deployment phase, your final product is delivered to your intended user. You can automate this process and schedule your deployment depending on the type.

For example, if you are only deploying a feature update, you can do so with a small number of users (canary release). If you are creating brand-new software, you can learn more about the different stages of the software release life cycle (SRLC).

7. Maintenance

The maintenance phase is the final stage of the SDLC if you're following the waterfall structure of the software development process. However, the industry is moving towards a more agile software development approach where maintenance is only a stage for further improvement.

In the maintenance stage, users may find bugs and errors that were missed in the earlier testing phase. These bugs need to be fixed for better user experience and retention. In some cases, these can lead to going back to the first step of the software development life cycle.

The SDLC phases can also restart for any new features you may want to add in your next release/update.

Process Flow

- ▶ Process flow—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

There are different process flows:

1. Linear process flow
2. Iterative process flow
3. Evolutionary process flow
4. Parallel process flow

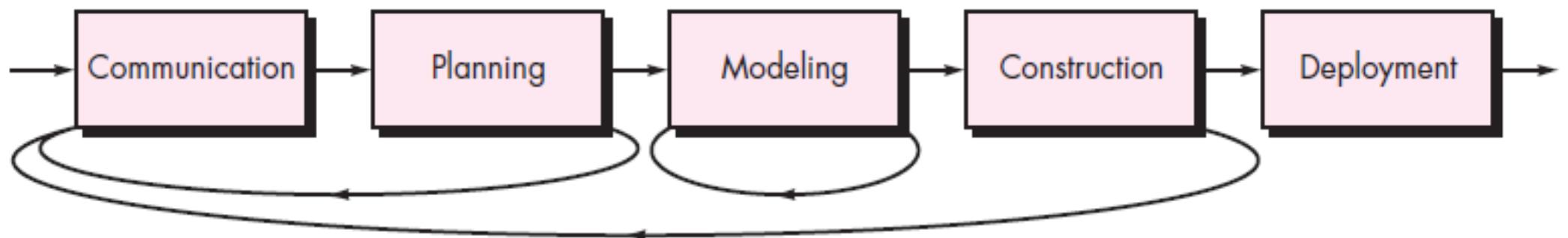
1. Linear Process Flow:

A linear process flow executes each of the five framework activities in **sequence**, beginning with communication and culminating with deployment



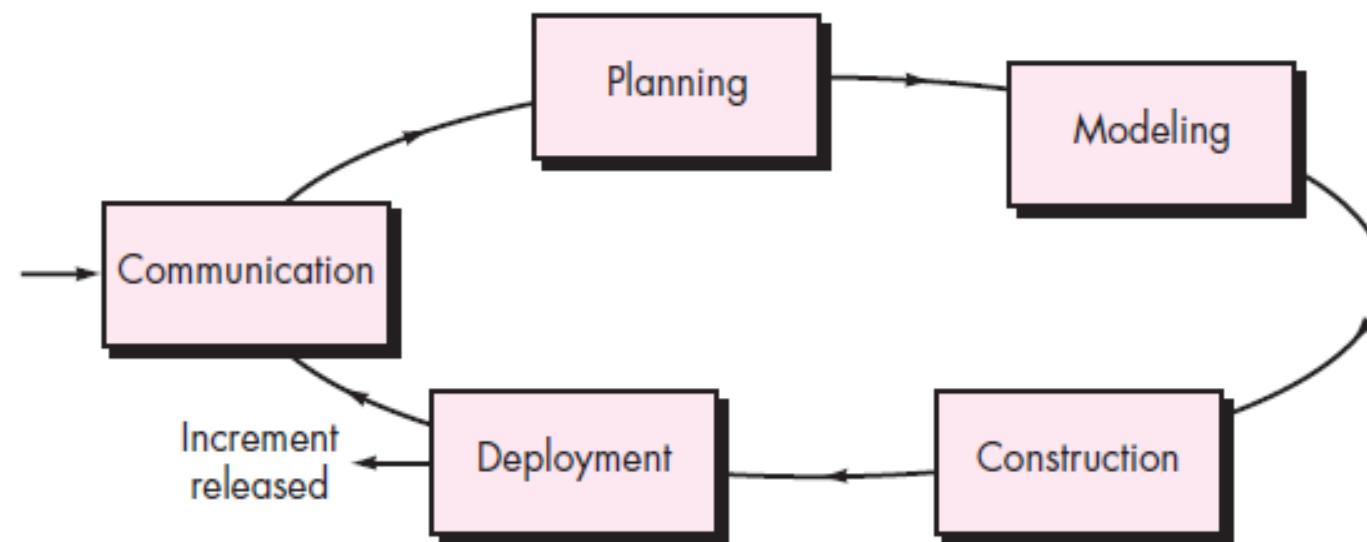
2. Iterative Process Flow:

An iterative process flow repeats one or more of the activities before proceeding to the next



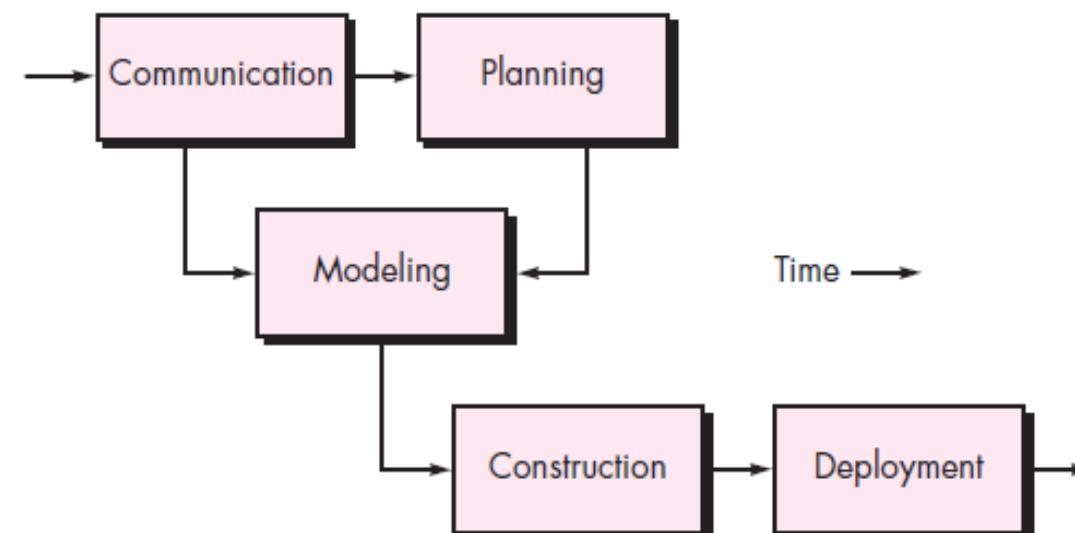
3. Evolutionary Process Flow:

An evolutionary process flow executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software.



4. Parallel Process Flow:

A parallel process flow executes one or more activities in **parallel** with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).





Thank You



Software Process Models

By:

Ms. Ashima Tyagi

Outline

- ▶ Waterfall Model
- ▶ Incremental Model
- ▶ Iterative Model
- ▶ Evolutionary Model
- ▶ RAD Models
- ▶ Agile Model and its methodologies

Software Process Models

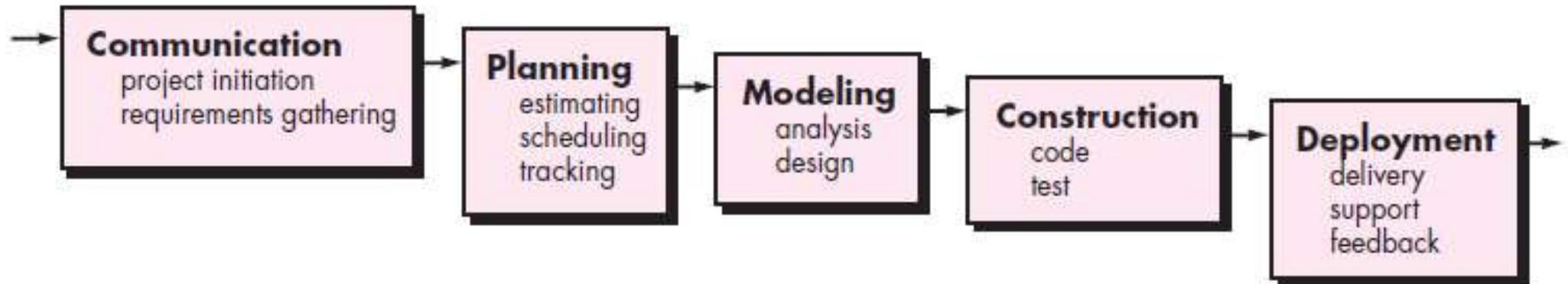
Based on the process flow, there are 4 types of software process models:

- ▶ Waterfall Model
- ▶ Incremental Model
- ▶ Iterative Model
- ▶ Evolutionary Model

Waterfall Model

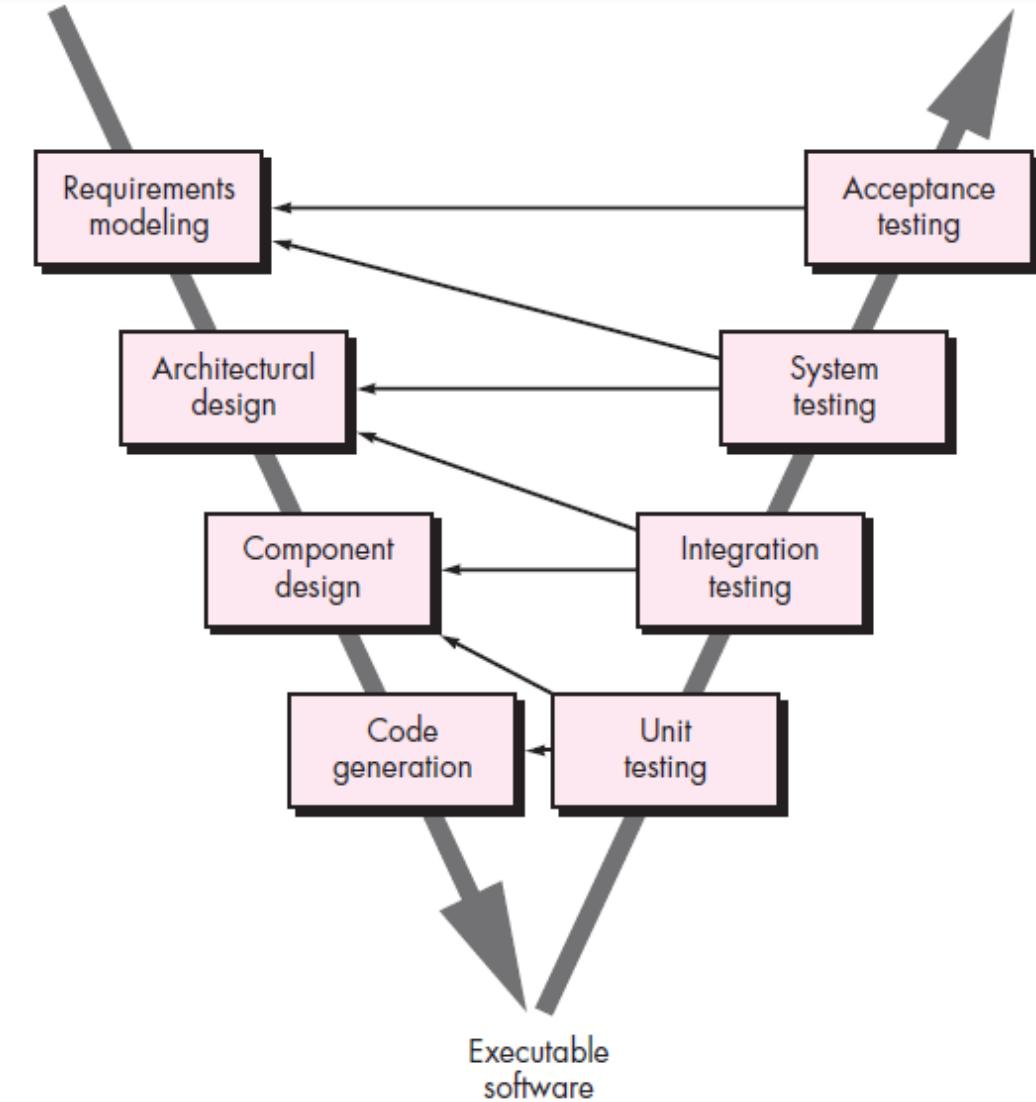
- When work flows from communication through deployment in a reasonably linear fashion.
- The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software

5



V-model:

A variation in the representation of the waterfall model is called the V-model



V-model:

- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution. Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- In reality, there is no fundamental difference between the classic life cycle and the V-model. The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

Advantages of Waterfall Model:

- ▶ **Simple** and Easy to Understand
- ▶ **Well-Structured and Organized:** Because the Waterfall model follows a sequential order, it ensures that there is a systematic approach to software development.
- ▶ **Emphasis on Documentation:** Each phase of the Waterfall model requires documentation before moving on to the next phase. This thorough documentation ensures that there is a formal record of the development process, which can be useful for future maintenance and enhancement.
- ▶ **Easy to Manage:** Due to its linear nature, the Waterfall model can be easier to manage, with clear milestones and deliverables. Project managers can easily monitor progress and resource allocation through the distinct phases.
- ▶ **Facilitates Early Identification of Issues:** With its emphasis on requirement analysis and design before coding begins, the Waterfall model can help in identifying potential issues early on. This can save time and resources by preventing significant changes in later stages.
- ▶ **Ideal for Stable Requirement Projects:** For projects with well-defined, unchanging requirements, the Waterfall model is particularly effective. It allows for a focused development effort without the need for constant revisions.
- ▶ **Clear Customer Expectations:** Since the Waterfall model requires detailed requirements at the beginning and has distinct phases with specific deliverables, it sets clear expectations for clients and stakeholders about what the product will do, thereby reducing the scope of misunderstanding or unexpected outcomes

Disadvantages of Waterfall Model:

- ▶ It is difficult to **induce changes** in the software.
- ▶ It is often difficult for the customer to **state all requirements explicitly**. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
- ▶ The **customer must have patience**. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

Incremental Model

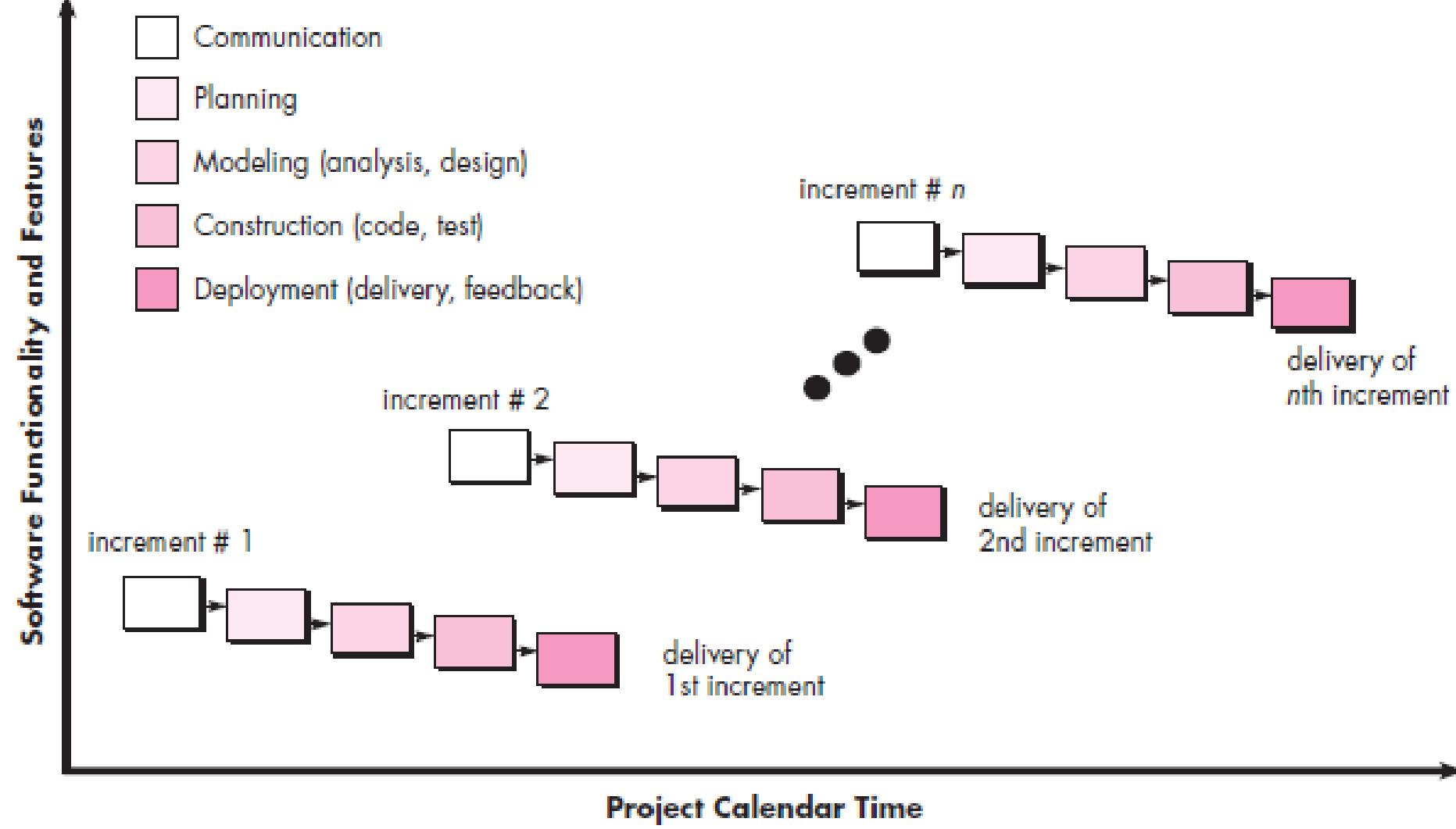
- ▶ There may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.
- ▶ In such cases, you can choose a process model that is designed to produce the software in increments.
- ▶ *The incremental model combines elements of linear and parallel process flows.*
- ▶ Each linear sequence produces deliverable “increments” of the software.
- ▶ In the Incremental model of software development, increments or partial products are delivered to the customer, not just the development team. This approach divides the software development process into smaller, manageable segments, allowing each segment or increment to be developed, tested, and delivered sequentially.



For example, word-processing software developed using the incremental paradigm might deliver basic file management, editing, and document production functions in the first increment; more sophisticated editing and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment.

It should be noted that the process flow for any increment can incorporate the prototyping paradigm.

In the **Incremental Model**, **parallel process flow** occurs because different increments (or modules) of the system can be developed **simultaneously** by separate teams. This parallelism accelerates development and enables incremental delivery of functional parts of the product.



- ▶ When an incremental model is used, the first increment is often a core product.
- ▶ That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation).
- ▶ As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- ▶ This process is repeated following the delivery of each increment, until the complete product is produced.

Advantages of Incremental Model:

- ▶ It is **easy for breakdown of tasks** because of divide and conquer approach used.
- ▶ It has **lowers initial delivery cost**.
- ▶ It has incremental resource deployment.
- ▶ It can **deduct errors** easily because core modules are used by the customer from the beginning of the phase and then 5. These are tested thoroughly.
- ▶ It is good to use when **requirements are known up-front**.
- ▶ It is good to use when **projects having lengthy developments schedules**.
- ▶ It generates **working software quickly** and early during the software life cycle.
- ▶ It is more **flexible and less costly** to change scope and requirements.
- ▶ It is **easy to manage risk** because of iterations.

Disadvantages of Incremental Model:

- ▶ It requires a good planning designing.
- ▶ It is costlier than waterfall model.
- ▶ Definition of system should be complete and clear.
- ▶ **Increased complexity:** As the project is developed incrementally, it can become more complex as each increment is added. This can make it harder to manage and maintain, as well as increase the risk of errors and bugs.
- ▶ **Higher costs:** Since each increment requires its own planning, design, coding, testing, and deployment, the overall cost of the project can be higher than with other development methodologies.
- ▶ **Difficulty in tracking progress:** With multiple increments being developed simultaneously, it can be challenging to track the progress of the project as a whole. This can make it harder to identify potential issues early on and take corrective action.
- ▶ **Increased communication overhead:** With each increment being developed by a different team or individual, there can be a significant increase in the communication overhead required to ensure that everyone is on the same page.
- ▶ **More time spent on testing:** With each increment requiring its own testing phase, more time may be spent on testing overall, which can delay the project's completion.

Iterative Model

- In this Model, you can start with some of the software specifications **and develop the first version of the software.**
- After the first version if there is a need to change the software, then a new version of the software is created with a new iteration.
- Every release of the Iterative Model finishes in an exact and fixed period that is called iteration.
- The Iterative Model allows the accessing earlier phases, in which the variations made respectively.
- The final output of the project renewed at the end of the Software Development Life Cycle (SDLC) process.

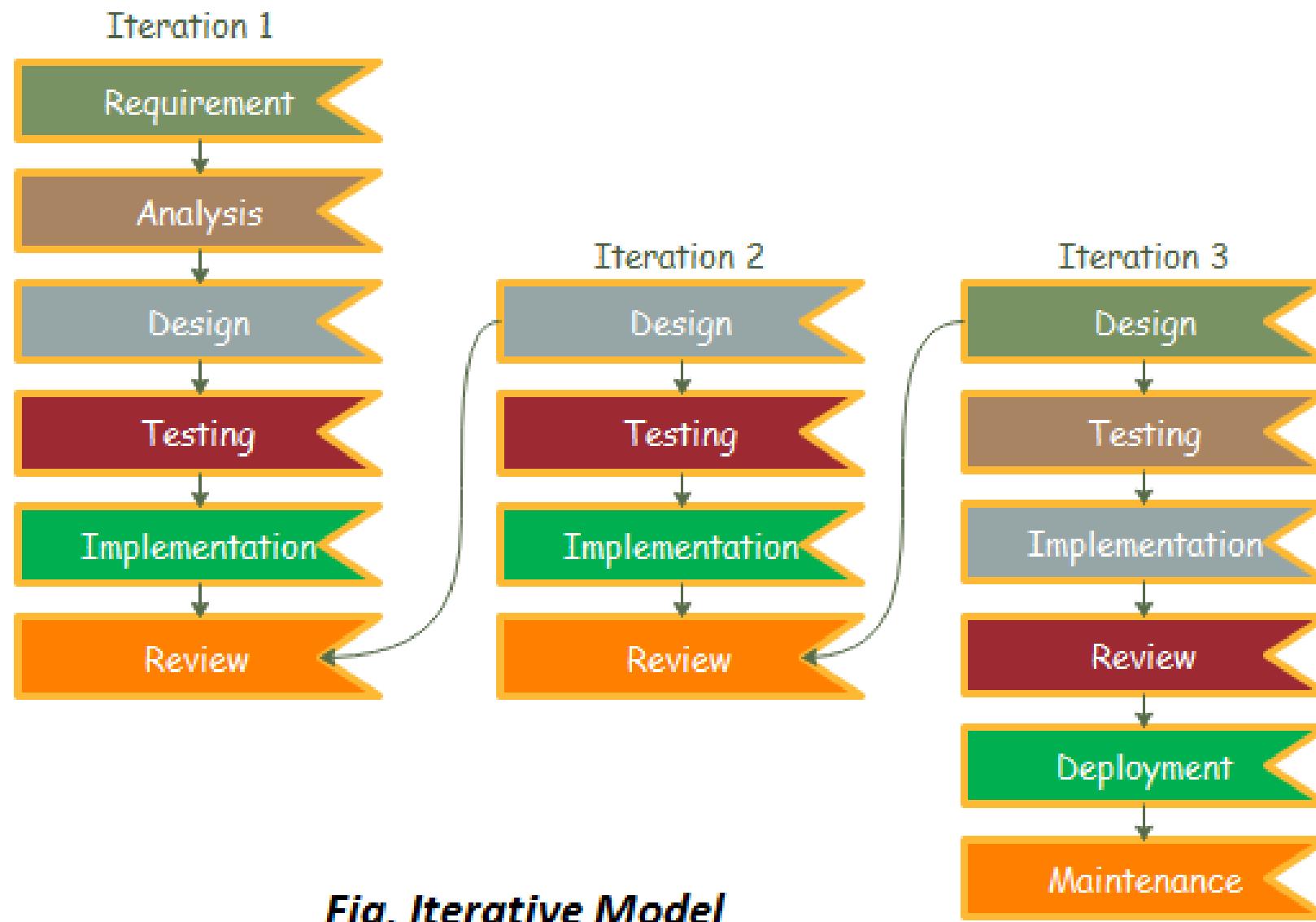


Fig. Iterative Model

Advantages of Iterative Model:

- ▶ Testing and debugging during smaller iteration is easy.
- ▶ It is easily acceptable to ever-changing needs of the project.
- ▶ Risks are identified and resolved during iteration.
- ▶ Limited time spent on documentation and extra time on designing.

Disadvantages of Iterative Model:

- ▶ It is not suitable for smaller projects.
- ▶ More Resources may be required.
- ▶ Design can be changed again and again because of imperfect requirements.
- ▶ Requirement changes can cause over budget.
- ▶ Project completion date not confirmed because of changing requirements.

**Iterative model
in Software
Development**

VS

**Incremental model
in Software
Development**

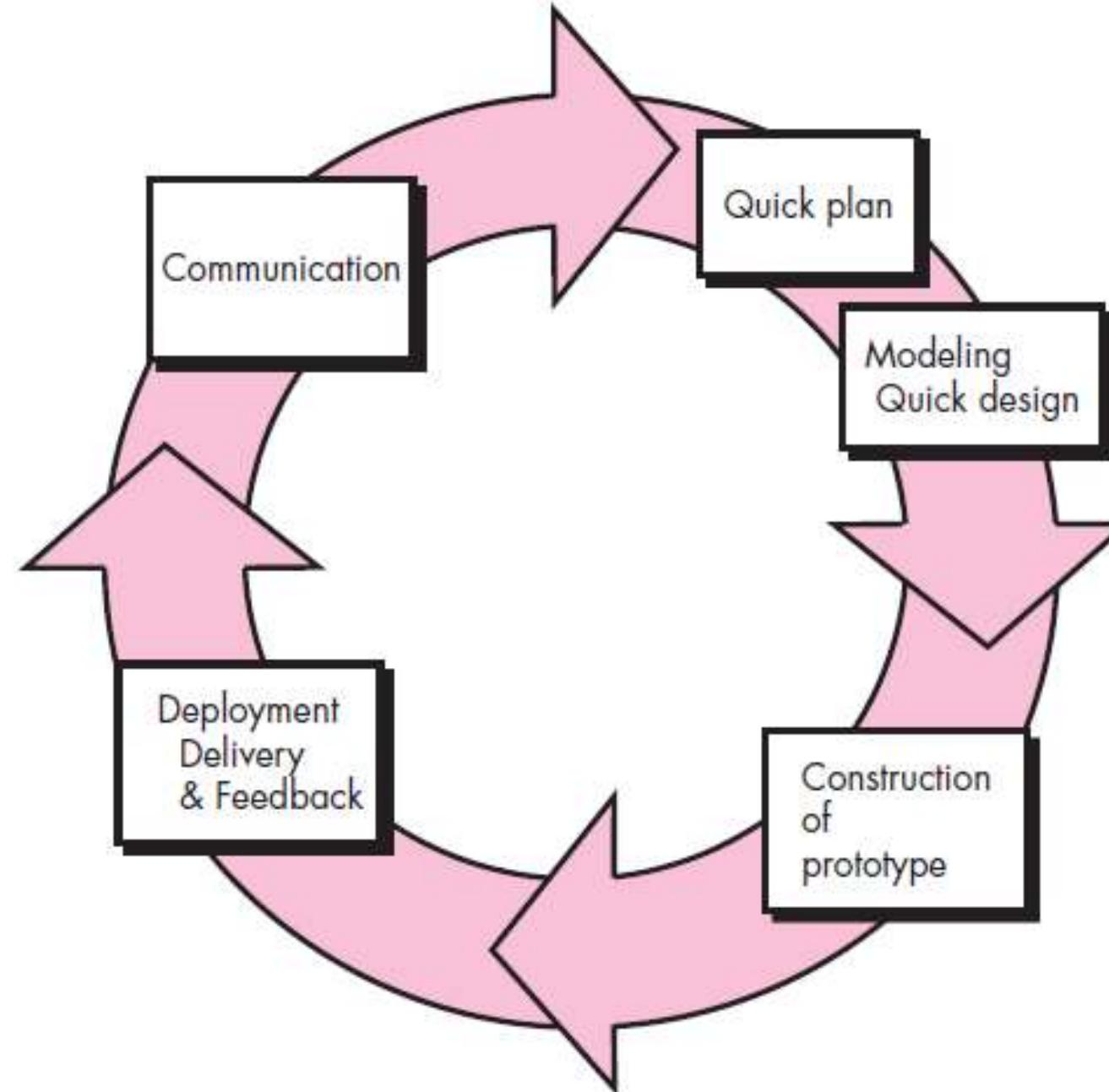
Aspect	Iterative Model	Incremental Model (Iterative Enhanced Model)
Development Approach	Repetitive cycles with continuous refinement	Delivering the software in functional parts
Flexibility	Highly flexible and adaptive to changes	Offers flexibility but less than iterative
Testing	Testing is integrated throughout the cycle	Testing is done for each increment
Delivery of Features	Complete features are developed in each iteration	Features are delivered incrementally
Risk Management	Risks are identified and addressed in each cycle	Risks are managed as increments are delivered
Client Feedback	Feedback is collected and incorporated regularly	Feedback is obtained after each increment
Project Visibility	Provides a clearer view of the project's progress	Offers visible progress with each delivered increment
Dependency Management	Dependencies can be identified and resolved in each iteration	Dependencies between increments need careful management
Completion Time	May take longer to deliver the complete product	Allows for partial delivery in shorter timeframes
Example Analogy	Writing a draft and refining it multiple times	Building a house floor by floor

Evolutionary Process Models

- ▶ Software, like all complex systems, evolves over a period of time.
- ▶ Business and product requirements often change as development proceeds.
- ▶ You need a process model that has been explicitly designed to accommodate a product that evolves over time.
- ▶ *Evolutionary models are iterative. But in evolutionary it doesn't require a useable product at the end of each cycle.*
- ▶ They are characterized in a manner that enables you to develop increasingly more complete versions of the software.
- ▶ There are two common evolutionary process models:
 - Prototyping
 - Spiral

1. Prototyping Model:

- ▶ The prototyping paradigm begins with communication.
- ▶ You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- ▶ A **prototyping iteration** is planned quickly, and modeling (in the form of a “quick design”) occurs.
- ▶ A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).
- ▶ The **quick design** leads to the construction of a prototype.
- ▶ The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.
- ▶ The prototype is developed quickly to test ideas or gather feedback, not necessarily intended for production.



- ▶ In most projects, the first system built is barely usable.
 - ▶ It may be too slow, too big, awkward in use or all three.
 - ▶ There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved.
-
- ▶ Both stakeholders and software engineers like the prototyping paradigm.
 - ▶ Users get a feel for the actual system, and developers get to build something immediately.
 - ▶ After the prototype is finalized, the system is typically developed from scratch (throwaway prototyping)

Advantages of Prototype Model:

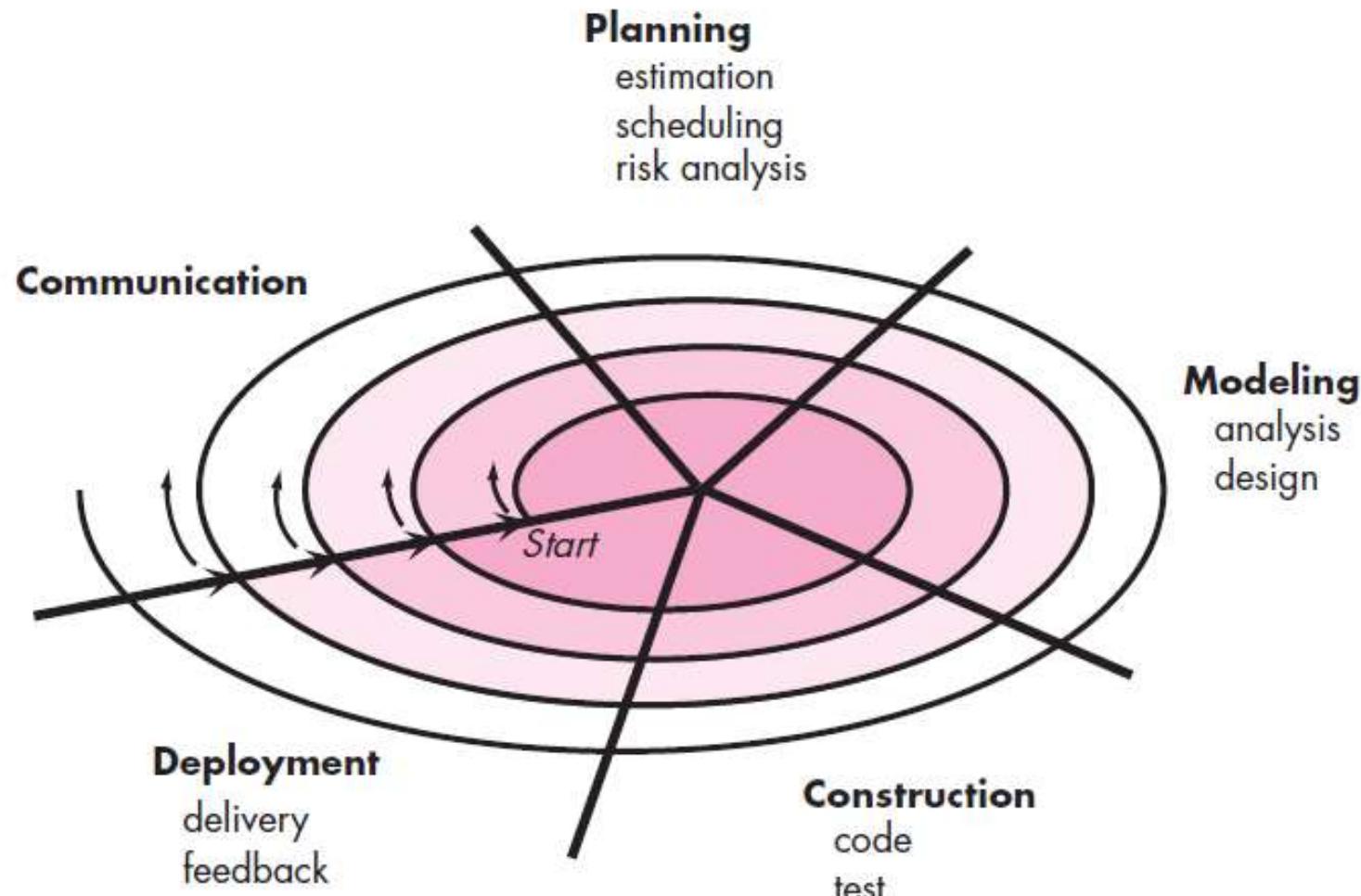
- ▶ A prototype of the working model is delivered early in the project timeline.
- ▶ Changes can made at any point.
- ▶ Error found and fixed easily.

Disadvantages of Prototype Model:

- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability. When informed that the product must be rebuilt so that high levels of quality can be maintained, stakeholders cry foul and demand that "a few fixes" be applied to make the prototype a working product. Too often, software development management relents.
- As a software engineer, you often make implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

2. Spiral Model:

- ▶ The spiral model is an evolutionary software process model that couples the iterative nature of prototyping.
- ▶ Using the spiral model, software is developed in a series of evolutionary releases.
- ▶ During early iterations, the release might be a model or prototype.
- ▶ During later iterations, increasingly more complete versions of the engineered system are produced.
- ▶ The spiral model demands a **direct consideration of technical risks at all stages of the project** and, if properly applied, should reduce risks before they become problematic.
- ▶ To combine iterative development with a systematic focus on **risk management** and continuous refinement.



- ▶ As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center.
- ▶ Risk is considered as each revolution is made.
- ▶ Anchor point milestones—a combination of work products and conditions that are attained along the path of the spiral—are noted for each evolutionary pass.
- ▶ The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Advantages of Spiral Model:

- ▶ A prototype of the working model is delivered early in the project timeline.
- ▶ Changes can made at any point.
- ▶ Error found and fixed easily.
- ▶ Risk is handled at any time.
- ▶ Customer satisfaction
- ▶ Suitable for high risk projects

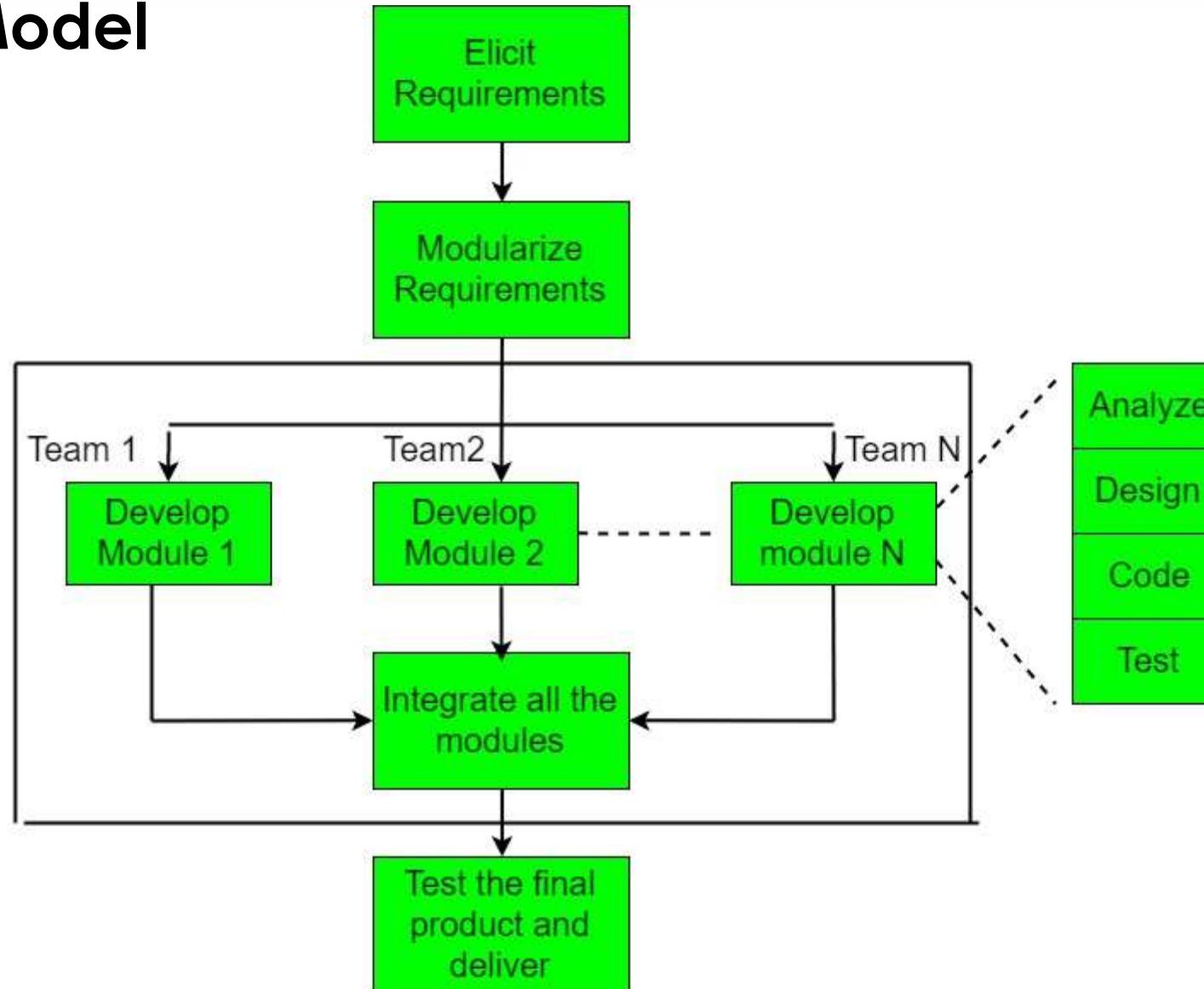
Disadvantages of Spiral Model:

- ▶ Not suitable for small projects
- ▶ Complex
- ▶ Spiral may go on indefinitely
- ▶ Not suitable for low risk projects.

RAD Model

- The **Rapid Application Development (RAD)** Model is a type of software development methodology that emphasizes **quick development** and delivery of high-quality software.
- It focuses on **iterative development** and active user involvement, enabling rapid production of prototypes or working systems.
- It focusses on **minimal planning** in favor of rapid prototyping.

RAD Model



Advantages of RAD Model:

- ▶ Faster Development Cycle: The RAD model emphasizes rapid prototyping, leading to quicker delivery of functional software.
- ▶ High Flexibility: Changes in requirements can be easily accommodated at any stage of development.
- ▶ Increased User Involvement: Regular feedback from users ensures the final product aligns closely with user needs.
- ▶ Reduced Risk: Issues and bugs can be identified and addressed early due to iterative development and frequent testing.
- ▶ Encourages Reusability: Reusable components and prototypes save time and effort in development.
- ▶ Improved Quality: Continuous testing and user feedback enhance the software quality.

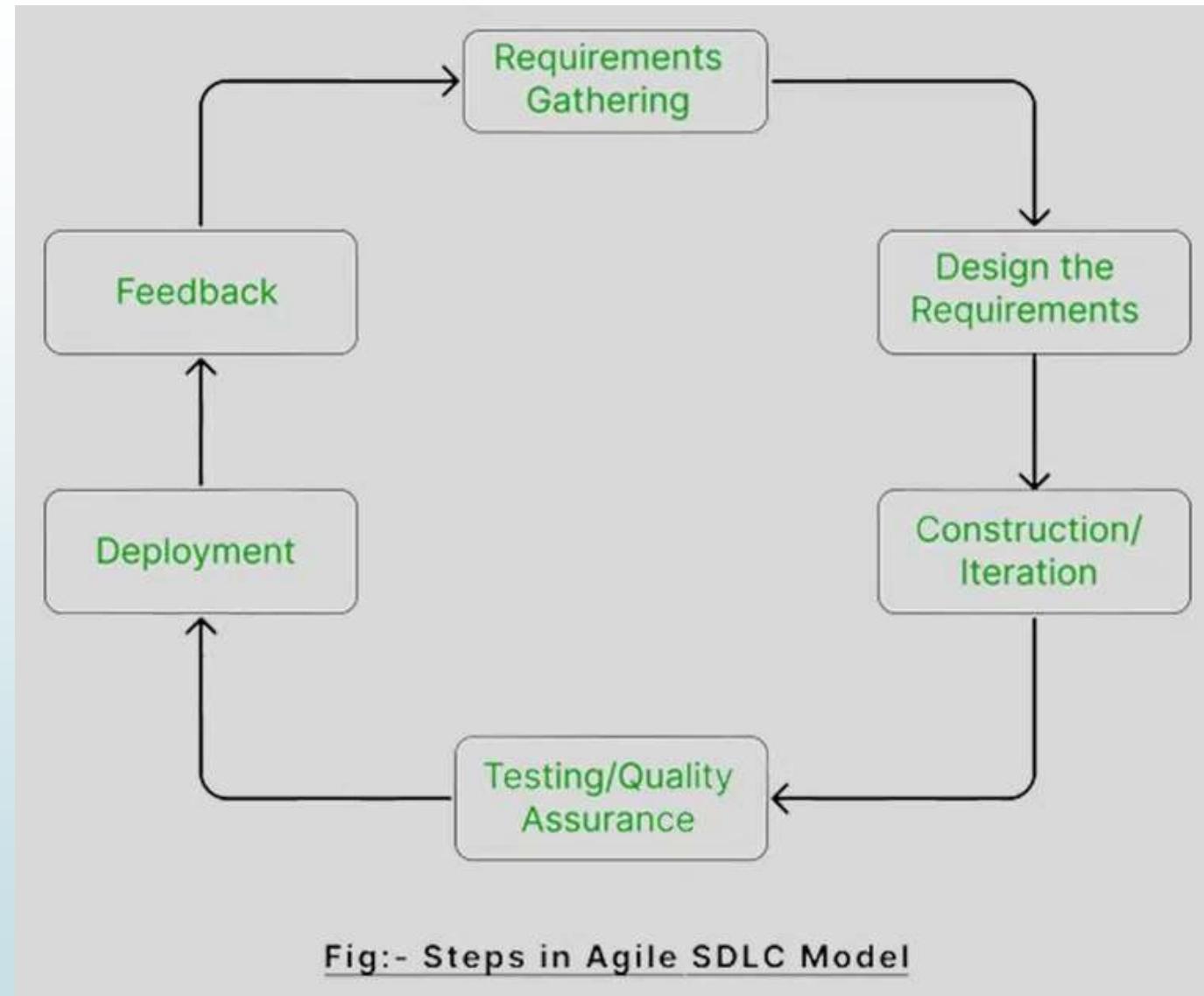
Disadvantages of RAD Model:

- ▶ Dependence on User Availability: Requires consistent and active involvement from users, which may not always be feasible.
- ▶ Not Suitable for All Projects: Works best for projects with clearly defined requirements and small to medium-sized teams. It may not handle large, complex projects well.
- ▶ High Cost: Requires skilled developers, tools for rapid prototyping, and frequent user involvement, increasing the cost.
- ▶ Limited Documentation: Focus on rapid prototyping may lead to insufficient documentation, complicating future maintenance.
- ▶ Team Dependency: Success depends heavily on the skills and collaboration of the team.
- ▶ Poor Performance with Large Teams: Coordination and communication challenges arise in large, distributed teams..

Agile Model

- The main difficulties developers faced were handling customer change requests during project development and the high cost and time required to incorporate these changes.
- To overcome these drawbacks of the Waterfall Model, in the mid-1990s the Agile Software Development model was proposed.

- Agile model is iterative by design
- The Agile Model was primarily designed to help a project **adapt quickly to change requests.**
- So, the main aim of the Agile model is to facilitate **quick project completion.**
- To accomplish this task, agility is required. Agility is achieved by fitting the process to the project and **removing activities that may not be essential for a specific project.**
- Also, anything that is a waste of time and effort is avoided.





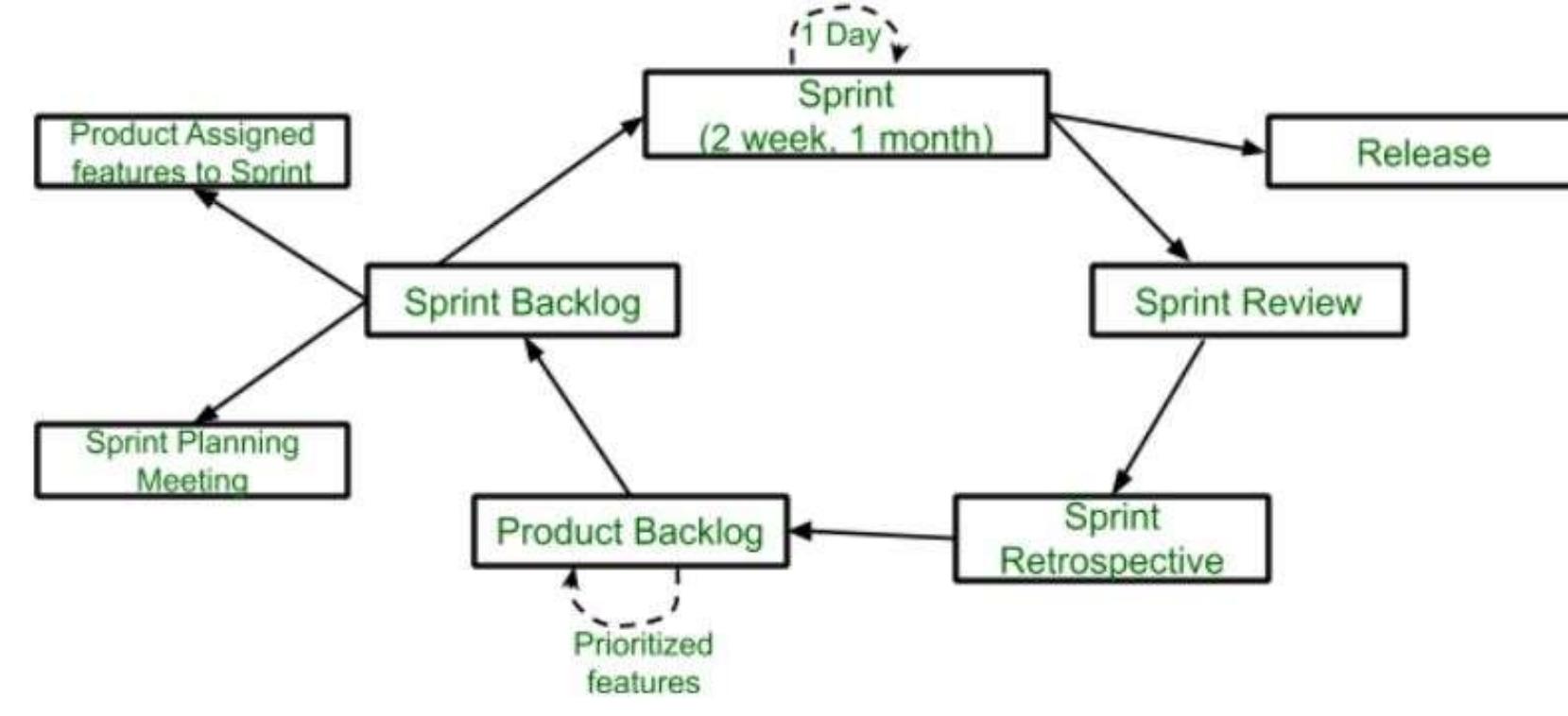
The Agile Model refers to a group of development processes. These processes share some basic characteristics but do have certain subtle differences among themselves.

1. Scrum
2. XP
3. Lean
4. Kanban

1. Scrum Methodology:

- ▶ The word SCRUM is inspired by a scrum in the sport of rugby. In rugby, the team comes together in what they call a scrum to work together to move the ball forward. In this context, Scrum is where the team comes together to move the product forward.
- ▶ Think of Scrum as a way to get work done as a team in small pieces at a time, with continuous experimentation and feedback loops along the way to learn and improve as you go.
- ▶ Scrum helps people and teams deliver value incrementally in a collaborative way. As an agile framework, Scrum provides just enough structure for people and teams to integrate into how they work, while adding the right practices to optimize for their specific needs.
- ▶ The Scrum framework is fairly simple being made up of a Scrum Team consisting of a Product Owner, a Scrum Master and Developers, each of which have specific accountabilities.

Lifecycle of Scrum



Sprint: A Sprint is a time box of one month or less. A new Sprint starts immediately after the completion of the previous Sprint.

After every sprint, the product increment is typically shown to the customer or stakeholders during the Sprint Review. This is a key part of the Scrum process in Agile methodology.

Release: When the product is completed, it goes to the Release stage.

Lifecycle of Scrum

- ▶ **Sprint Review:** If the product still has some **non-achievable features**, it will be checked in this stage and then passed to the Sprint Retrospective stage.
- ▶ **Sprint Retrospective:** In this stage **quality or status** of the product is checked.
- ▶ **Product Backlog:** According to the prioritize features the product is organized.
- ▶ **Sprint Backlog:** Sprint Backlog is divided into two parts Product assigned features to sprint and Sprint planning meeting.
- ▶ **Product-Assigned Features:** The Product Owner collaborates with the team to prioritize features or user stories from the Product Backlog that are to be included in the Sprint Backlog.
- ▶ **Sprint Backlog:** is a subset of the Product Backlog. **It contains the list of tasks or user stories that the development team commits to completing during the sprint.** It is developed during the Sprint Planning Meeting.

2. XP Methodology:

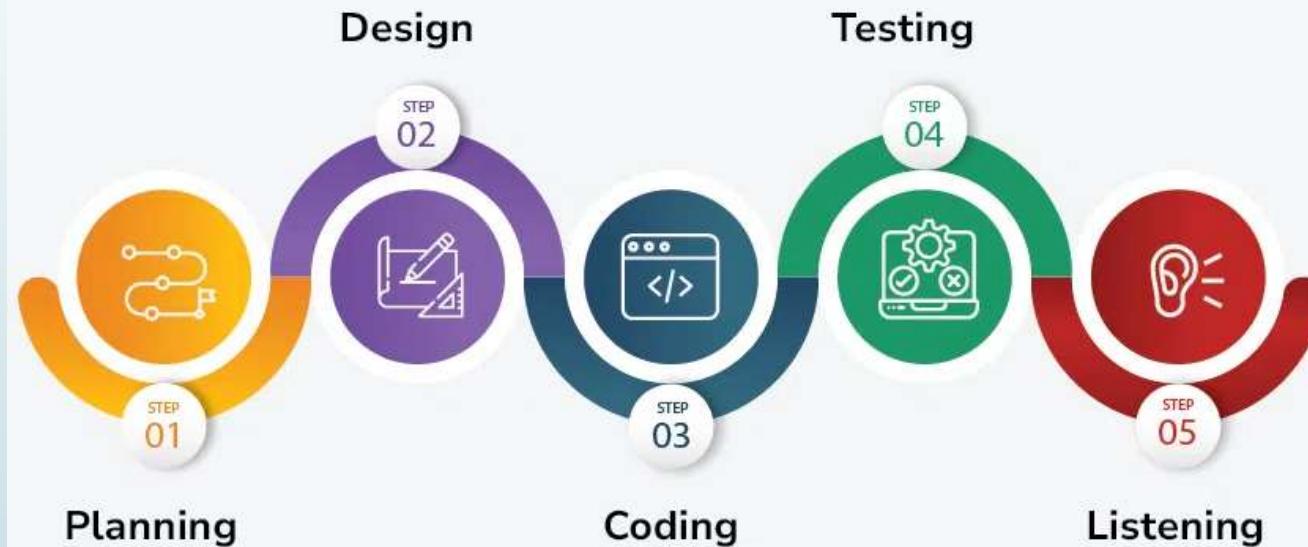
- Extreme Programming (XP) is an Agile software development methodology designed to improve software quality and responsiveness to changing customer requirements.
- XP is built on values like communication, simplicity, feedback, courage, and respect, with a strong focus on engineering practices and customer collaboration.

Core Values of XP:

- ▶ **Communication:** Promotes constant interaction between team members and the customer.
- ▶ **Simplicity:** Encourages building only what is needed and avoiding unnecessary complexity.
- ▶ **Feedback:** Gathers frequent feedback from customers and tests to adapt to changes quickly.
- ▶ **Courage:** Refactoring involves restructuring or improving code without changing its external behavior, which can sometimes be risky or challenging. Courage promotes boldness in refactoring, discarding outdated code, and responding to changes.
- ▶ **Respect:** Ensures a positive team environment where contributions are valued.

Lifecycle of XP

Life Cycle of Extreme Programming (XP)



The Extreme Programming Life Cycle consists of five phases:

- ▶ **Planning:** The first stage of Extreme Programming is planning. During this phase, clients define their needs in concise descriptions known as user stories. The team calculates the effort required for each story and schedules releases according to priority and effort.
- ▶ **Design:** The team creates only the essential design needed for current user stories, using a common analogy or story to help everyone understand the overall system architecture and keep the design straightforward and clear.
- ▶ **Coding:** Extreme Programming (XP) promotes pair programming and developers work together at one workstation, enhancing code quality and knowledge sharing. They write tests before coding to ensure functionality from the start (TDD), and frequently integrate their code into a shared repository with automated tests to catch issues early.
- ▶ **Testing:** Extreme Programming (XP) gives more importance to testing that consist of both unit tests and acceptance test. Unit tests, which are automated, check if specific features work correctly. Acceptance tests, conducted by customers, ensure that the overall system meets initial requirements. This continuous testing ensures the software's quality and alignment with customer needs.
- ▶ **Listening:** In the listening phase regular feedback from customers to ensure the product meets their needs and to adapt to any changes.

3. Lean:

- Lean methodology in Agile is an approach focused on **maximizing value while minimizing waste** in the software development process.

Lean methodology or Lean Software Development(LSD) is based on 7 key principles:

1. Eliminating the Waste
2. Fast delivery
3. Amplify Learning
4. Build Quality
5. Respect Teamwork
6. Delay the Commitment
7. Optimizing the Whole System

Principles of Lean:

1. **Eliminating the Waste:** To identify and eliminate wastes e.g. unnecessary code, delay in processes, inefficient communication, issues with quality, data duplication, more tasks in the log than completed, etc. regular meetings are held by Project Managers. This allows team members to point out faults and suggest changes in the next turn.
2. **Fast Delivery:** Previously long-time planning used to be the key to success in business, but with time, it has been found that engineers spend too much time on building complex systems with unwanted features. So they came up with an **Minimum Viable Product (MVP)** strategy which resulted in building products quickly that included a little functionality and launching the product to market and seeing the reaction. Such an approach allows them to enhance the product based on customer feedback.
3. **Amplify Learning:** Learning is improved through ample code reviewing and meetings that are cross-team applicable. It is also ensured that particular knowledge isn't accumulated by one engineer who's writing a particular piece of code so paired programming is used.

- 
- 4. Builds Quality:** LSD is all about preventing waste and keeping an eye on not sacrificing quality. Developers often apply **test-driven programming to examine the code before it is written**. Quality can also be gained by getting constant feedback from team members and project managers.
 - 5. Respect Teamwork:** LSD focuses on empowering team members, rather than controlling them. Setting up a **collaborative atmosphere**, keeping perfect balance when there are short deadlines and immense workload. This method becomes very important when new members join a well-established team.
 - 6. Delay the Commitment:** **Delay decisions until you have enough information** to make informed choices. This flexibility allows better handling of changing requirements. This methodology always constructs software as flexible, so new knowledge is available and engineers can make improvements.
 - 7. Optimizing the Whole System:** Lean's principle allows managers to break an issue into small constituent parts to optimize the team's workflow, create unity among members, and inspire a sense of shared responsibility which results in enhancing the team's performance. **Look at the entire system rather than optimizing individual parts.** Ensure that all steps in the process contribute to delivering value efficiently.

LSD Process

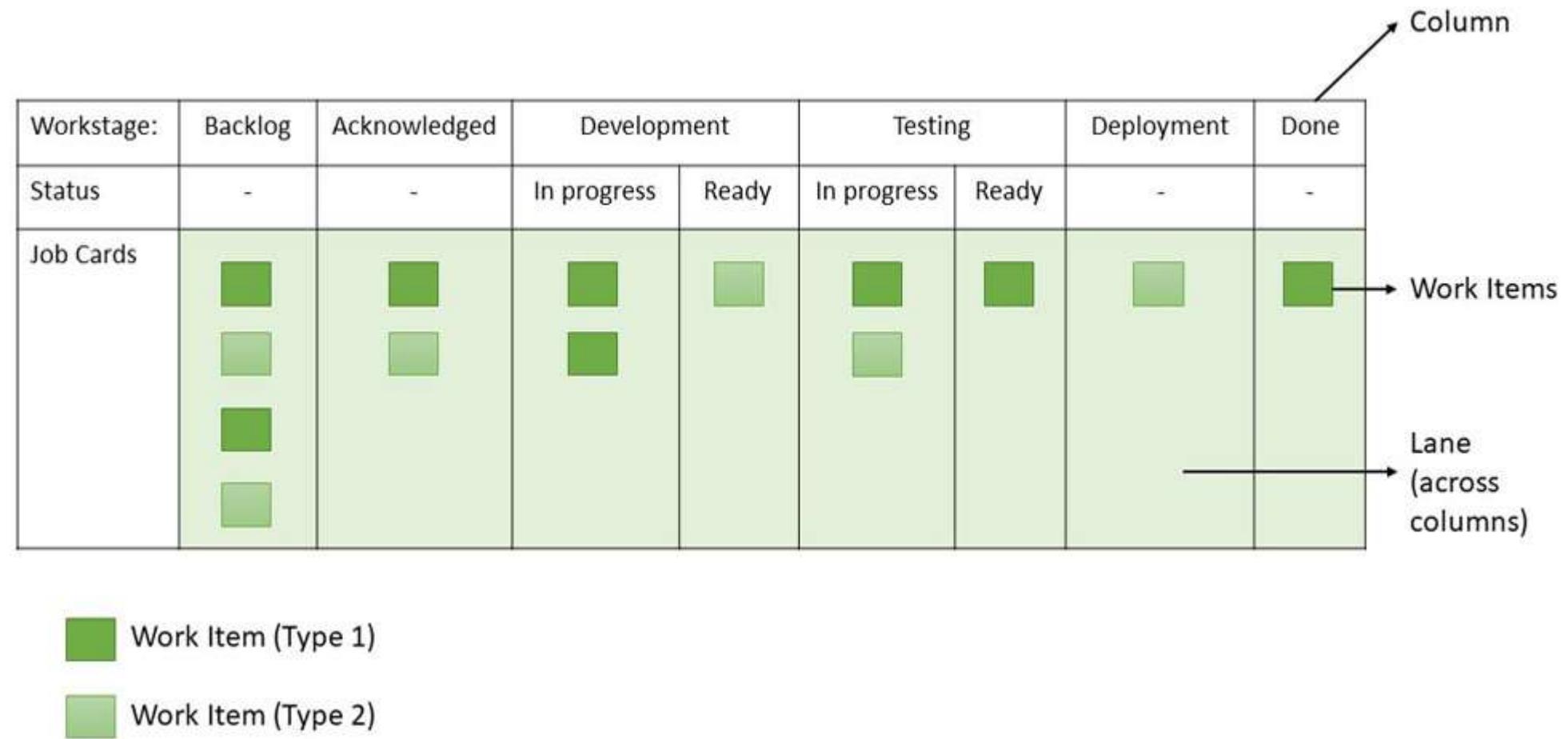
Here is the overview of the lean software development process:

1. **Identify Value:** Understand the customer values and focus on delivering features that meet these needs.
2. **Map the Value Stream:** This involves mapping out the entire software development process to identify and **eliminate wasteful activities** that do not add value.
3. **Create Flow:** Ensure a smooth and **continuous flow of work** by minimizing delays and interruptions.
4. **Establish Pull:** Develop features based on **customer demand** rather than pushing features through the process.
5. **Seek Perfection:** Regularly **review and refine** the development process. Always encourage the team members to identify the areas of improvement and implement changes iteratively.
6. **Build Quality In:** Use practices such as **test-driven development (TDD)** and continuous integration to integrate quality assurance throughout the development process.
7. **Empower Teams:** Empower development teams by providing them with the necessary tools, resources, and autonomy to make decisions.

4. Kanban:

- ▶ Kanban is a popular framework in the Agile methodology that **emphasizes visualizing workflow, limiting work in progress (WIP), and improving efficiency** through continuous delivery.
- ▶ It is highly adaptable and focuses on managing and optimizing the flow of work.

Sample Kanban Board



From the sample kanban board, these can be inferred about Kanban Boards or Cards:

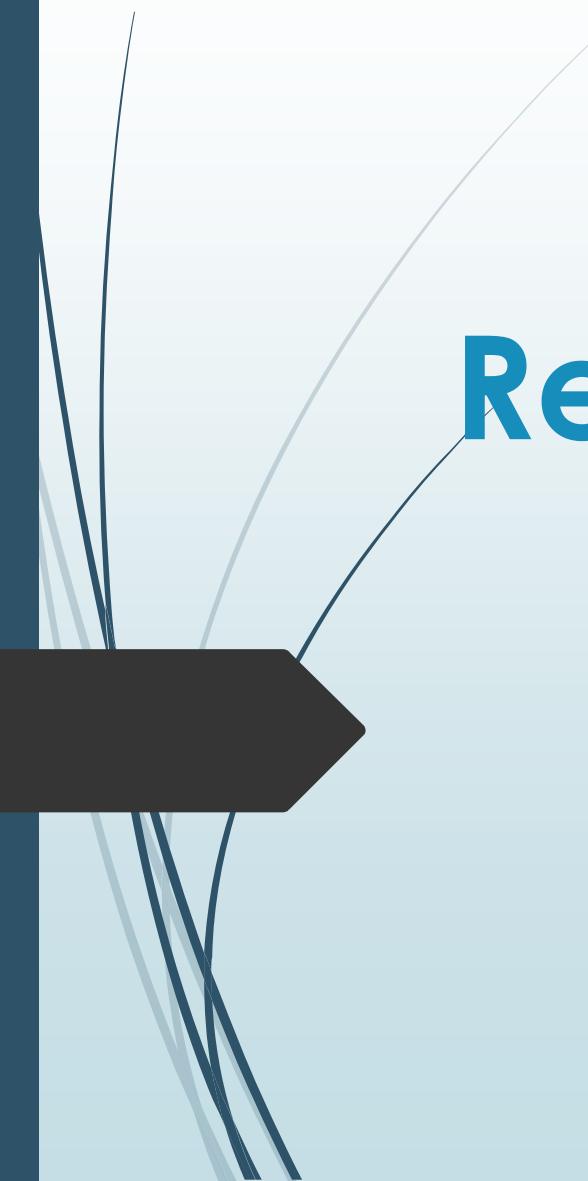
- ▶ **Workflow:** Backlog -> Acknowledged -> Development -> Testing -> Deployment/UAT -> Done
- ▶ **Work stages:** Acknowledged, Development (In Progress), Development (Ready), Testing (In Progress), Testing (Ready), Deployment/UAT, Backlog (Arrival Queue), Done (Finished Stage)
- ▶ **Work Items:** Two types of tasks (Type 1 and Type 2) are represented by work items

Principles of Kanban:

1. **Visualize Workflow:** Use a **Kanban board** to display tasks as they move through stages, typically represented as columns (e.g., To Do, In Progress, Done). Helps identify bottlenecks and provides a clear view of the current workload.
2. **Limit Work in Progress (WIP):** Restrict the number of tasks being worked on simultaneously. Prevents overloading the team, ensuring focus and faster completion of tasks.
3. **Focus on Flow:** Optimize the movement of tasks through the system to achieve a smooth and predictable flow. Monitor key metrics like cycle time (time to complete a task).
4. **Make Process Policies Explicit:** Clearly define rules and guidelines for how tasks move between stages. Ensures alignment and understanding across the team.
5. **Implement Feedback Loops:** Regularly review the process and performance through meetings like daily stand-ups and retrospectives. Encourage continuous learning and improvement.
6. **Improve Collaboratively and Evolve Experimentally:** Use feedback, metrics, and experimentation to refine processes incrementally.



Thank You



Requirement Engineering

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Requirement
- ▶ Types of Requirements
- ▶ Requirement Engineering Tasks
- ▶ Requirements documentation

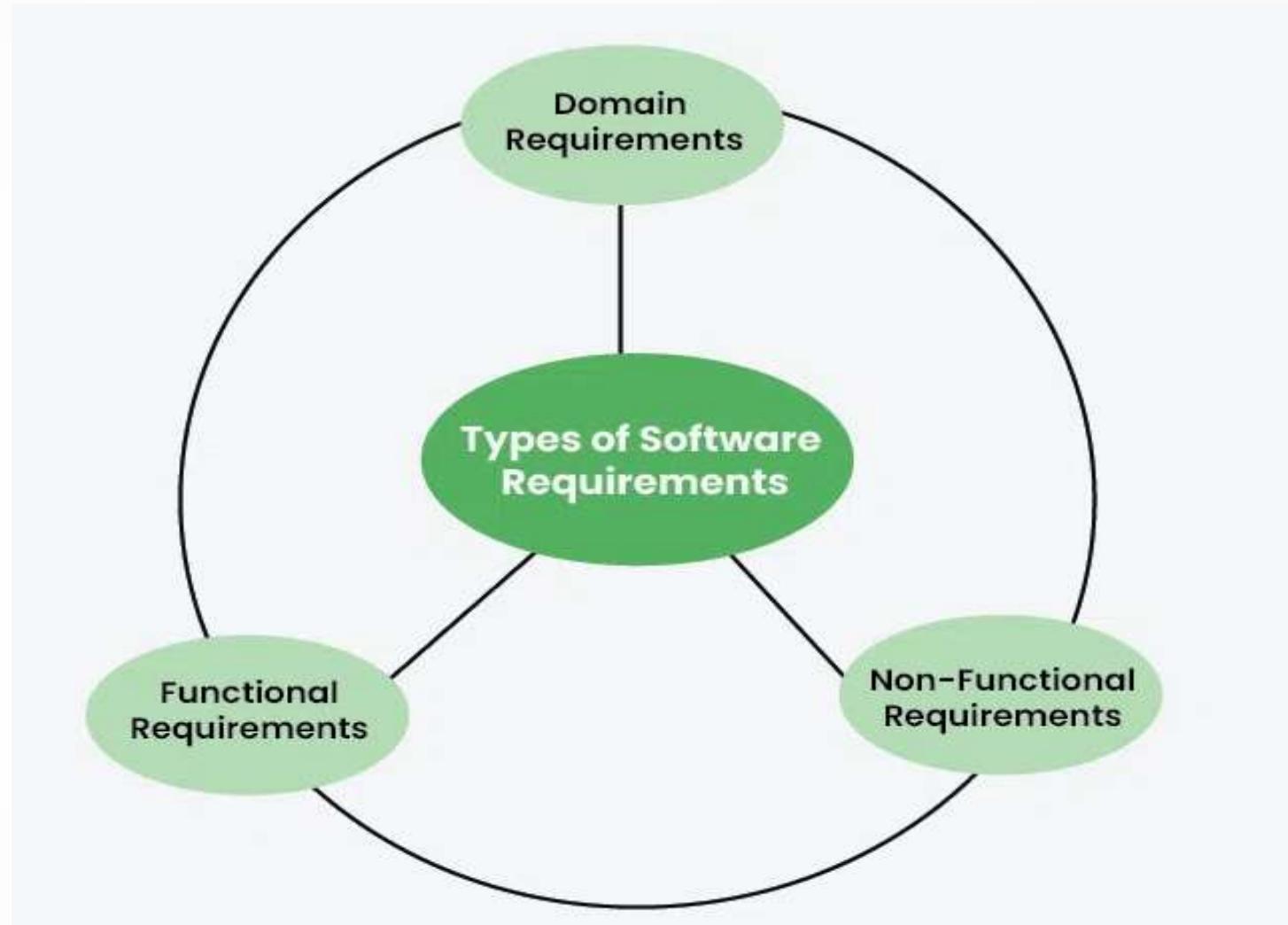
Requirements

A software requirement is a rule the software must conform to: what it must do, how well, and within what constraints or limits.

Types of Requirements

There are two types of requirements:

- ▶ Functional
- ▶ Non-functional
- ▶ Domain



1. Functional requirements

- ▶ “A description of a behavior that a system will exhibit under specific conditions”.
- ▶ For example, “**If the user activates the ‘log in’ button, the login page will appear.**” Functional requirements answer the question, “**What must the software do?**”
- ▶ These are the requirements that the end user specifically demands as basic facilities that the system should offer.

Example:

- ▶ What are the features that we need to design for this system?
- ▶ What are the edge cases we need to consider, if any, in our design?

Functional requirements include:

- **User Requirements:** What the end users expect from the system. These can be high-level and may describe interactions between the system and its users.

Example: "The system should allow users to log in using their email and password."

- **System Requirements:** Detailed specifications of what the system must do, usually describing the internal workings or features.

Example: "The system must authenticate users against the database using SHA-256 hashing."

2. Non-functional requirements

- ▶ “A description of a property or characteristic that a system must exhibit or a constraint that it must respect”
- ▶ For example, “**If the user activates the ‘log in’ button, the login page will appear within 500 milliseconds.**” This nonfunctional requirement has a characteristic that the system must exhibit: **responsiveness**. Responsiveness is also called a quality attribute.
- ▶ Example about respecting a constraint is, “The GUI toolkit must be able to display non-rectangular windows.”

Non-functional requirements include:

They are also called non-behavioral requirements. They deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Example:

- ▶ Each request should be processed with the minimum latency?
- ▶ System should be highly valuable.

Figure shows a simple example of a design to build a table for a house.

Sketch of house



Two Failed Requirements

Note. This rolling table fails the nonfunctional requirement of fitting through an average door and the functional requirement of having four legs.

Requirement Types

Functional Requirements	Non Functional Requirements
A functional requirement defines a system or its component.	A non-functional requirement defines the quality attribute of a software system.
It specifies "What should the software system do?"	It places constraints on "How should the software system fulfill the functional requirements?"
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.
Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
Example 1) Authentication of user whenever he/she logs into the system. 2) System shutdown in case of a cyber attack. 3) A Verification email is sent to user whenever he/she registers for the first time on some software system.	Example 1) Emails should be sent with a latency of no greater than 12 hours from such an activity. 2) The processing of each request should be done within 10 seconds 3) The site should load in 3 seconds when the number of simultaneous users are > 10000

3. Domain requirements

Domain requirements are specific to the domain or industry in which the software operates. They include terminology, rules, and standards relevant to that particular domain.

Examples:

- ▶ Healthcare: The software must comply with HIPAA regulations for handling patient data.
- ▶ Finance: The system should adhere to GAAP standards for financial reporting.
- ▶ E-commerce: The software should support various payment gateways like PayPal, Stripe, and credit cards.

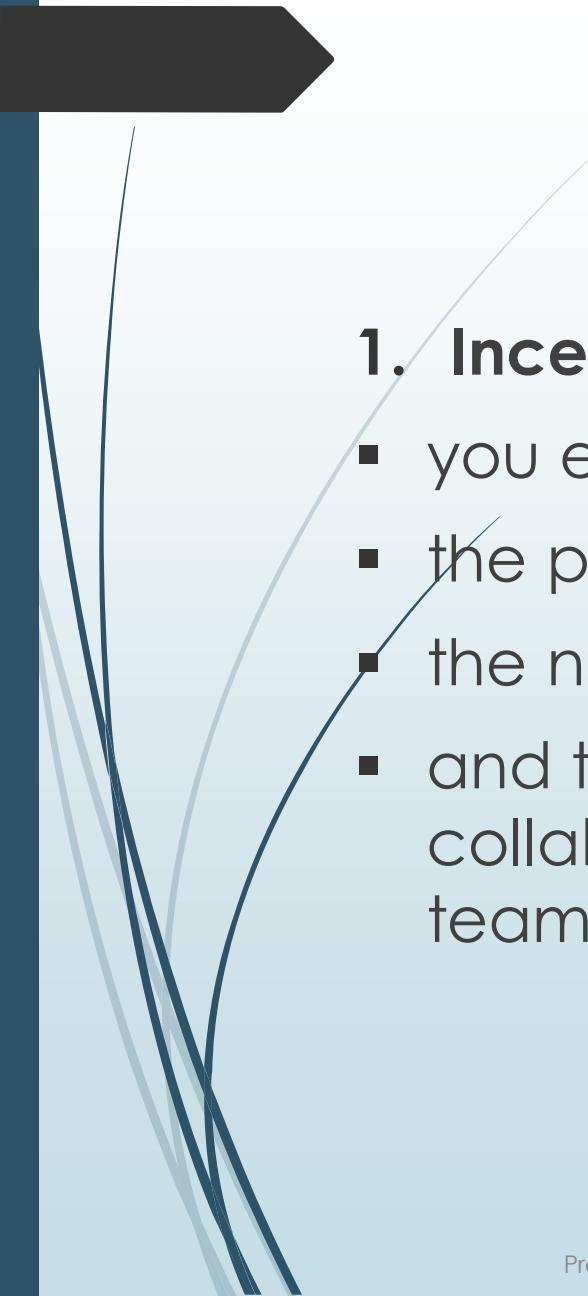
Domain requirements reflect the unique needs and constraints of a particular industry. They ensure that the software is relevant and compliant with industry-specific regulations and standards.

Requirement Engineering

► The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering.

It encompasses seven **distinct tasks/crucial steps**:

1. Inception,
2. Elicitation,
3. Elaboration,
4. Negotiation,
5. Specification,
6. Validation, And
7. Management

- 
1. **Inception:** At project inception,
 - you establish a basic understanding of the problem,
 - the people who want a solution,
 - the nature of the solution that is desired,
 - and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

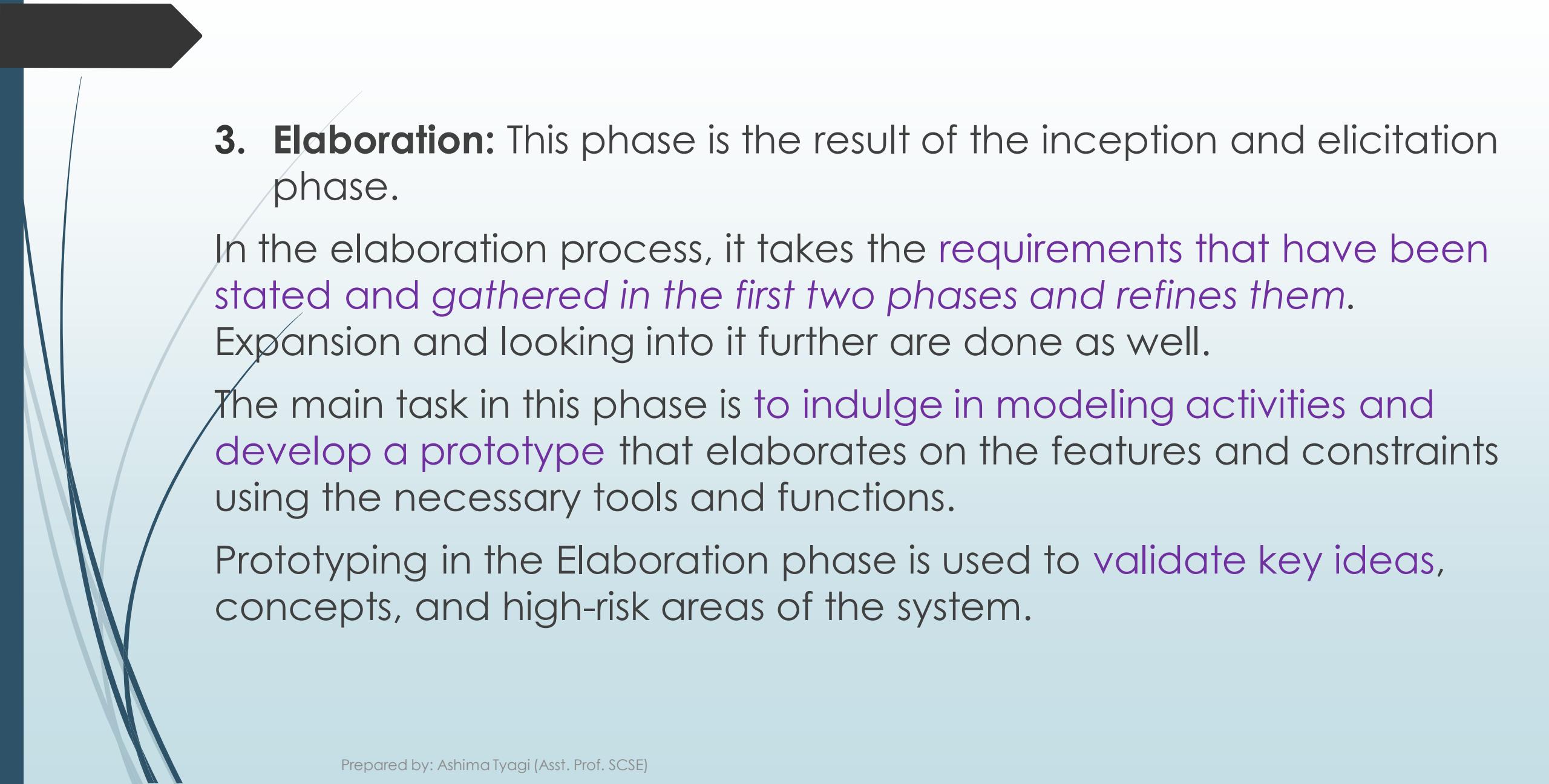
- 
2. **Elicitation:** This phase focuses on *gathering the requirements from the stakeholders*. One should be careful in this phase, as the requirements are what establishes the key purpose of a project. Understanding the kind of requirements needed from the customer is very crucial for a developer.

Types of Elicitation Techniques:

- I. Interviews
- II. Surveys and Questionnaires
- III. Workshops
- IV. Observation
- V. Prototyping
- VI. Use Cases and User Stories
- VII. Document Analysis and Review
- VIII. Brainstorming

The following problems can occur in the elicitation phase:

- Problem of Scope
- Problem of Understanding
- Problem of Volatility



3. **Elaboration:** This phase is the result of the inception and elicitation phase.

In the elaboration process, it takes the requirements that have been stated and gathered *in the first two phases and refines them.*

Expansion and looking into it further are done as well.

The main task in this phase is *to indulge in modeling activities and develop a prototype* that elaborates on the features and constraints using the necessary tools and functions.

Prototyping in the Elaboration phase is used to validate key ideas, concepts, and high-risk areas of the system.

- 
4. **Negotiation:** This phase **emphasizes discussion and exchanging conversation on what is needed and what is to be eliminated.** In the negotiation phase, negotiation is between the developer and the customer and they dwell on how to go about the project with limited business resources. Customers are asked to prioritize the requirements and make guesstimates on the conflicts that may arise along with it. Risks of all the requirements are taken into consideration and negotiated in a way where the customer and developer are both satisfied with reference to the further implementation.

The following are discussed in the negotiation phase:

- ▶ Availability of Resources.
- ▶ Delivery Time.
- ▶ Scope of requirements.
- ▶ Project Cost.
- ▶ Estimations on development.

5. **Specification:** In the specification phase, the requirements engineer gathers all the requirements and **develops a working model**. This final working product will be the basis of any functions, features or constraints to be observed. The models used in this phase include *ER (Entity Relationship) diagrams*, *DFD (Data Flow Diagram)*, *FDD (Function Decomposition Diagrams)*, and *Data Dictionaries*.

A software specification document is submitted to the customer in a language that he/she will understand, to give a glimpse of the working model.

This phase specifies the following:

- ▶ Written document.
- ▶ A set of models.
- ▶ A collection of use cases.
- ▶ A prototype.

6. **Validation:** This phase focuses on checking for errors and debugging. In the validation phase, the developer scans the specification document and checks for the following:

- ▶ All the requirements have been stated and met correctly
- ▶ Errors have been debugged and corrected.
- ▶ Work product is built according to the standards.

This requirements validation mechanism is known as the **formal technical review**. The review team that works together and validates the requirements include software engineers, customers, users, and other stakeholders. Everyone in this team takes part in checking the specification by examining for any errors, missing information, or anything that has to be added or checking for any unrealistic and problematic errors.

Some of the validation techniques are the following-

- ▶ Requirements reviews/inspections.
- ▶ Prototyping.
- ▶ Test-case generation.
- ▶ Automated consistency analysis.

7. **Requirements Management:** is a set of activities where the entire team takes part in identifying, controlling, tracking, and establishing the requirements for the successful and smooth implementation of the project.

In this phase, the team is responsible for managing any changes that may occur during the project. New requirements emerge, and it is in this phase, responsibility should be taken to manage and prioritize as to where its position is in the project and how this new change will affect the overall system, and how to address and deal with the change. Based on this phase, the working model will be analyzed carefully and ready to be delivered to the customer.

Requirements Documentation

- ▶ Requirements Documentation is a detailed record of all the requirements for a software system.
- ▶ It serves as a communication tool between stakeholders, including business analysts, developers, designers, and clients.
- ▶ The goal is to document what the system must do (functional) and how it must perform (non-functional).
- ▶ A well-prepared requirements document helps in preventing misunderstandings, reduces ambiguity, and sets a clear baseline for development.

Understanding the Nature of Software

- ▶ The nature of software refers to the inherent characteristics and properties that define software as a product.
- ▶ These properties influence how software is **developed, maintained, tested, and used**.
- ▶ Unlike physical products, software is intangible, flexible, and constantly evolving, which brings both advantages and challenges.

Software Requirement Specification (SRS)

- Software Requirement Specification (SRS) is a document that describes the functional and non-functional requirements of a software system.
- It serves as a blueprint for the development team and other stakeholders, outlining what the software should do and how it should behave.



IEEE Standard for SRS

Source: Adapted from IEEE-STD-830-1993 See also, van Viet 1999, pp226-231

1 Introduction

Purpose

Scope

Definitions, acronyms, abbreviations

Reference documents

Overview

Identifies the product, & application domain

2 Overall Description

Product perspective

Product functions

User characteristics

Constraints

Assumptions and Dependencies

Describes contents and structure of the remainder of the SRS

Describes all external interfaces: system, user, hardware, software; also operations, site adaptation, and hardware constraints

Summary of major functions

3 Specific Requirements

Appendices

Index

Anything that will limit the developer's options (e.g. regulations, reliability, criticality, hardware limitations, parallelism, etc)

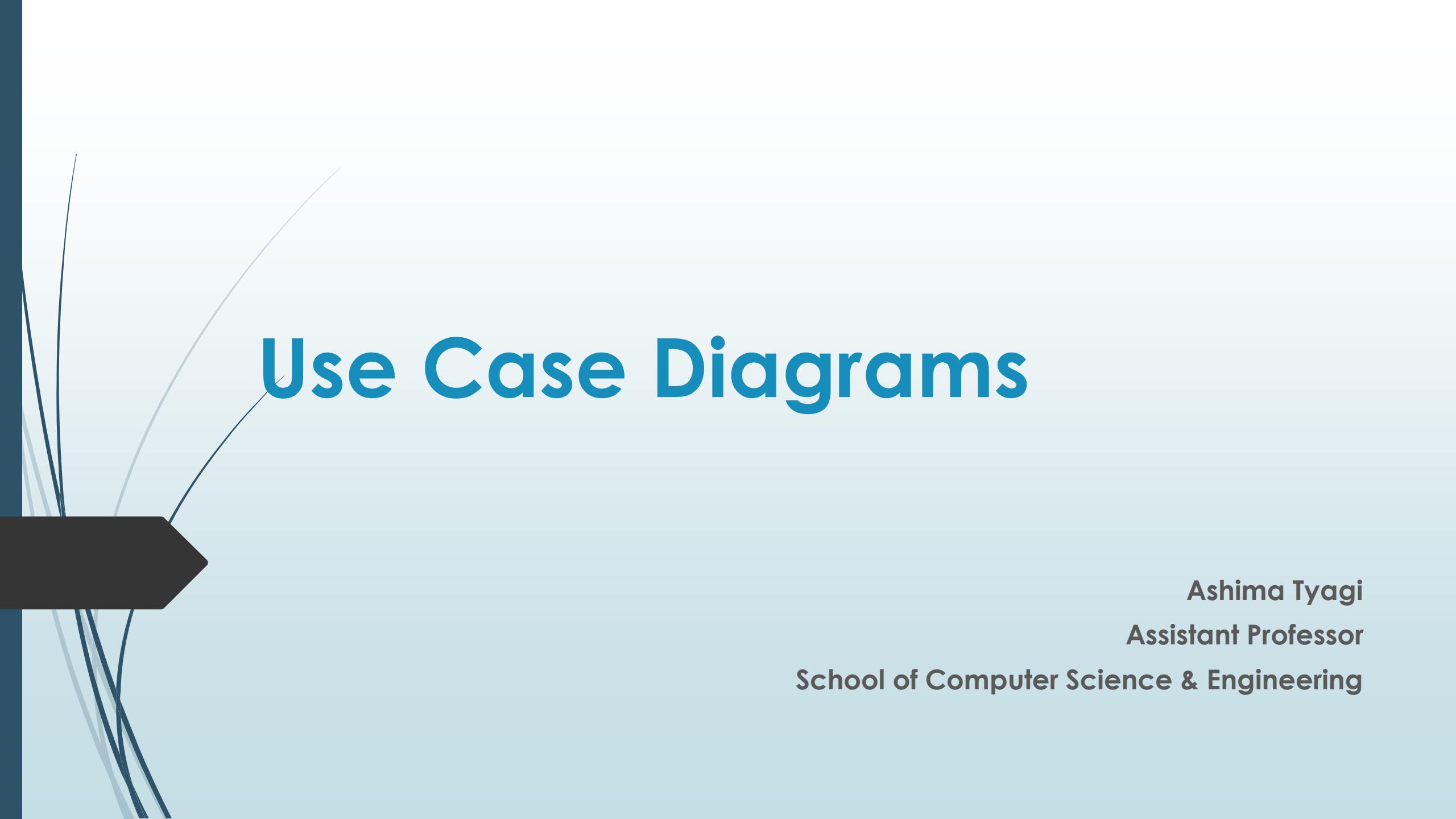
All the requirements go in here (I.e. this is the body of the document). IEEE STD provides 8 different templates for this section

The SRS document contains the following information:

1. **Introduction:** Provides an overview of the software, its purpose, and scope.
2. **Functional Requirements:** Describes the specific functions and features of the software, including inputs, outputs, and behavior.
3. **Non-Functional Requirements:** Specifies the quality attributes of the software, such as performance, security, and usability.
4. **Design Constraints:** Specifies any limitations or restrictions that apply to the software.
5. **Performance Requirements**
6. **System Features:** Provides a detailed description of each feature or module of the software.
7. **User Stories or Use Cases:** Describes the interactions between the software and its users, including specific scenarios and workflows.
8. **Assumptions and Dependencies:** Lists any assumptions made during the requirements gathering process and dependencies on other systems or components.
9. **Preliminary Schedule and Budget**



Thank You



Use Case Diagrams

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Use Case Diagrams

- ▶ A Use Case Diagram represents the **interaction between actors (users or external systems)** and a system under consideration to accomplish specific goals.
- ▶ It provides a high-level view of the system's functionality by illustrating the various ways users can interact with it.

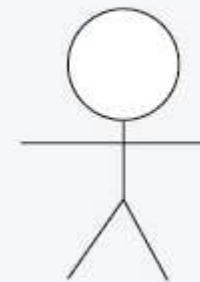
Purpose:

- ▶ It defines the dynamic aspect of a system.
- ▶ It gathers the system's needs.
- ▶ It depicts the external view of the system.
- ▶ It recognizes the internal as well as external factors that influence the system.
- ▶ It represents the interaction between the actors.

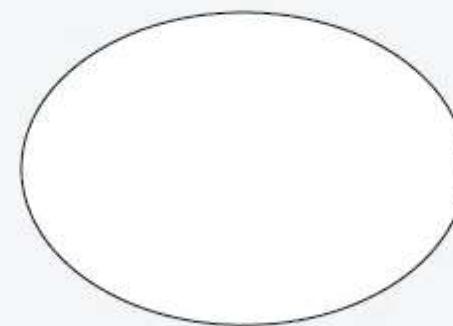
3

Notations in Use case diagrams

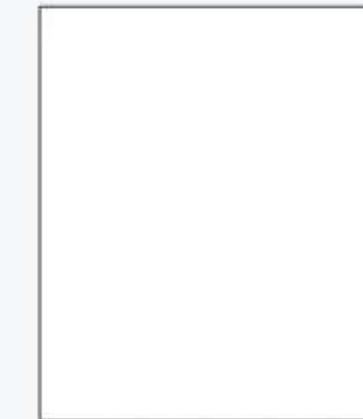
Use Case Diagram Notations



Actor



UseCase

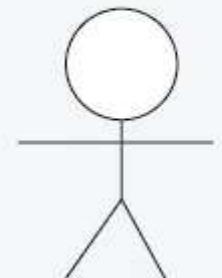


System Boundary

Actors:

- ▶ Actors are external entities that interact with the system. These can include users, other systems, or hardware devices.
- ▶ In the context of a Use Case Diagram, actors initiate use cases and receive the outcomes.
- ▶ Proper identification and understanding of actors are crucial for accurately modeling system behavior.

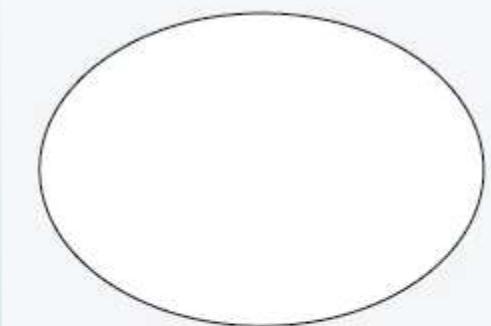
Actor



Actors are typically represented by stick figures

Use Cases:

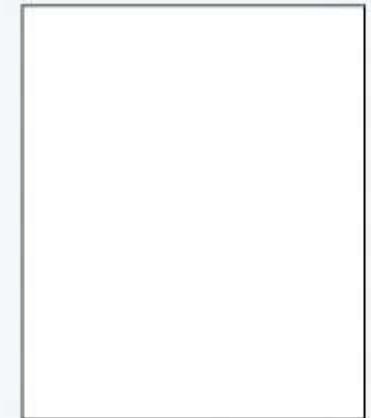
- ▶ Use cases are like **scenes** in the play.
- ▶ They represent specific activities your system can do.
- ▶ In the online shopping system, examples of use cases could be “**Place Order**,” “**Track Delivery**,” or “**Update Product Information**”.
- ▶ Use cases are represented by ovals.



Use cases are represented by ovals.

System Boundary:

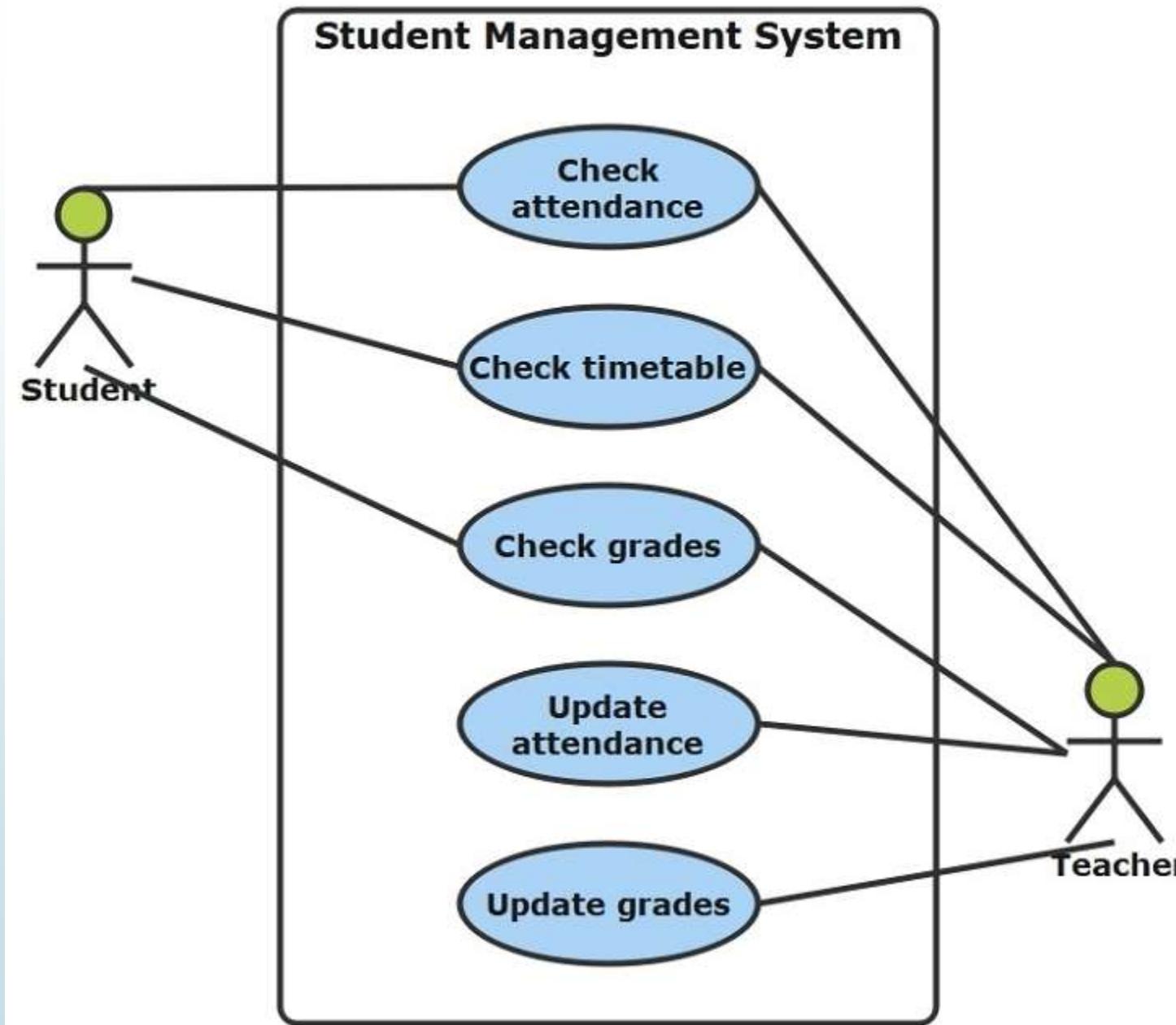
- The system boundary is a visual representation of the **scope or limits of the system** you are modeling.
- It defines what is inside the system and what is outside.
- The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it.
- The system boundary is typically represented by a rectangular box that surrounds all the use cases of the system.



System boundary is represented by a rectangular box.

7

► Example1:



Relationships in use case diagram:

The relationship between actor and the use case can be defined using:

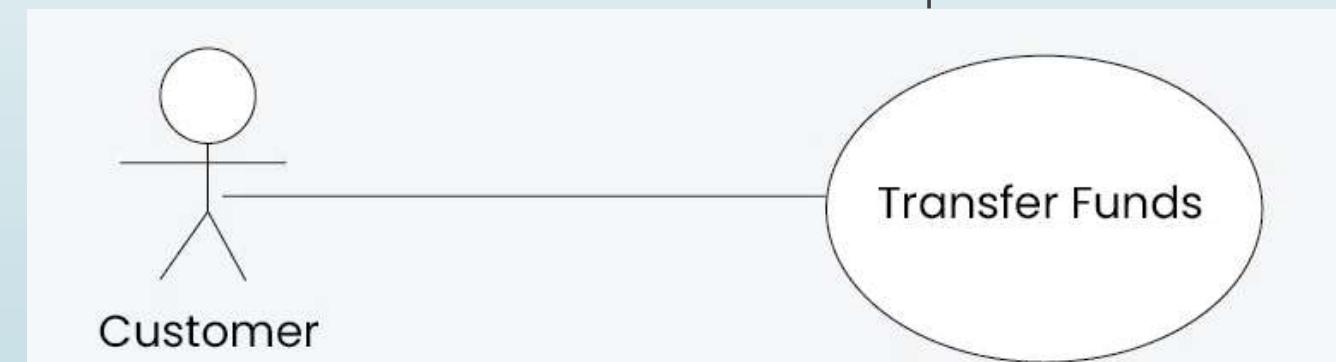
- ▶ Association
- ▶ Include
- ▶ Extend
- ▶ Generalisation

Association Relationship:

The Association Relationship represents a *communication or interaction between an actor and a use case*.

Example: Online Banking System

- ▶ Actor: Customer
- ▶ Use Case: Transfer Funds
- ▶ Association: A line connecting the “Customer” actor to the “Transfer Funds” use case, indicating the customer’s involvement in the funds transfer process.



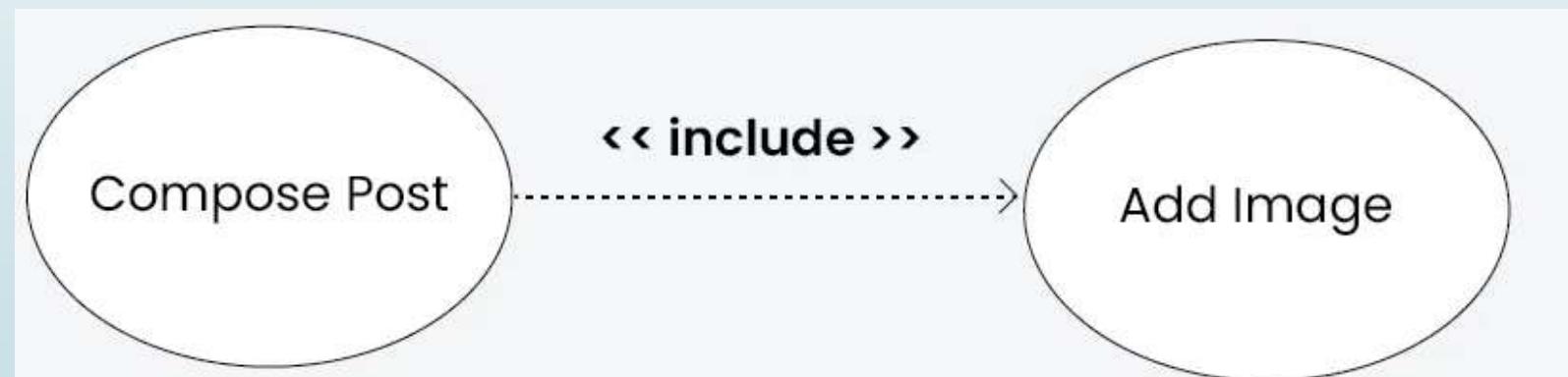
Represents a communication or interaction between an actor(Customer) and a use case(Transfer Funds)

Include Relationship:

The Include Relationship indicates that a use case *includes the functionality of another use case.*

Example: Social Media Posting

- ▶ Use Cases: Compose Post, Add Image
- ▶ Include Relationship: The “Compose Post” use case includes the functionality of “Add Image.” Therefore, composing a post includes the action of adding an image.



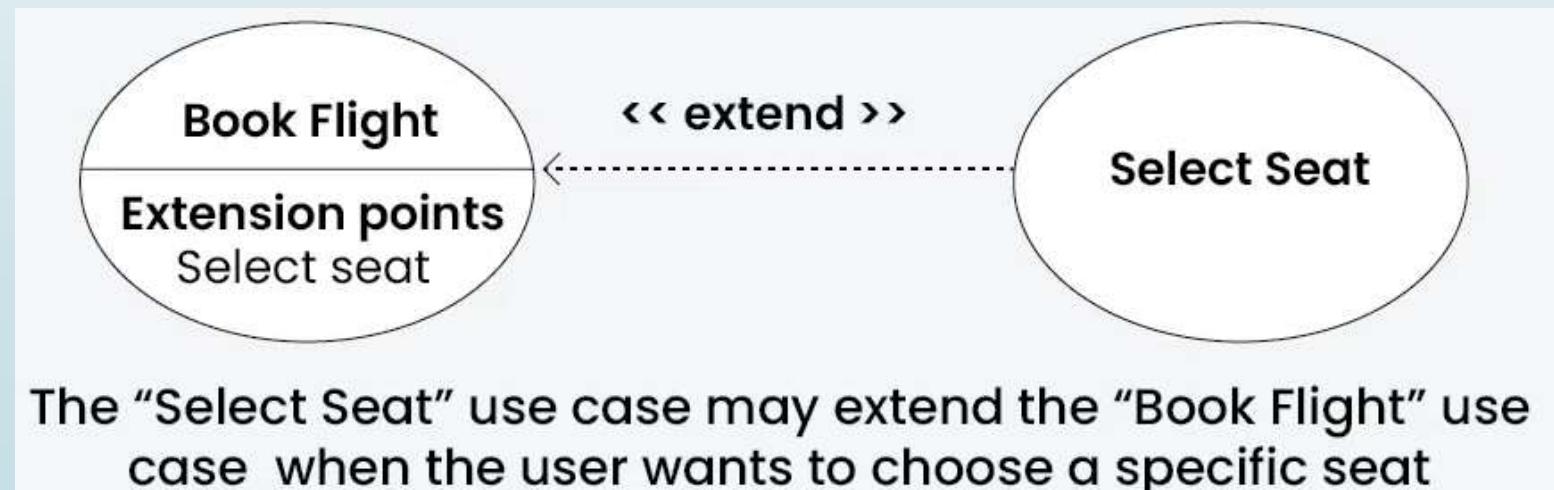
"Compose Post" includes the functionality of another "Add Image"

Extend Relationship:

The Extend Relationship illustrates that *a use case can be extended by another use case under specific conditions*. This relationship is useful for handling optional or exceptional behavior.

Example: Flight Booking System

- ▶ Use Cases: Book Flight, Select Seat
- ▶ Extend Relationship: The “Select Seat” use case may extend the “Book Flight” use case when the user wants to choose a specific seat, but it is an **optional step**.

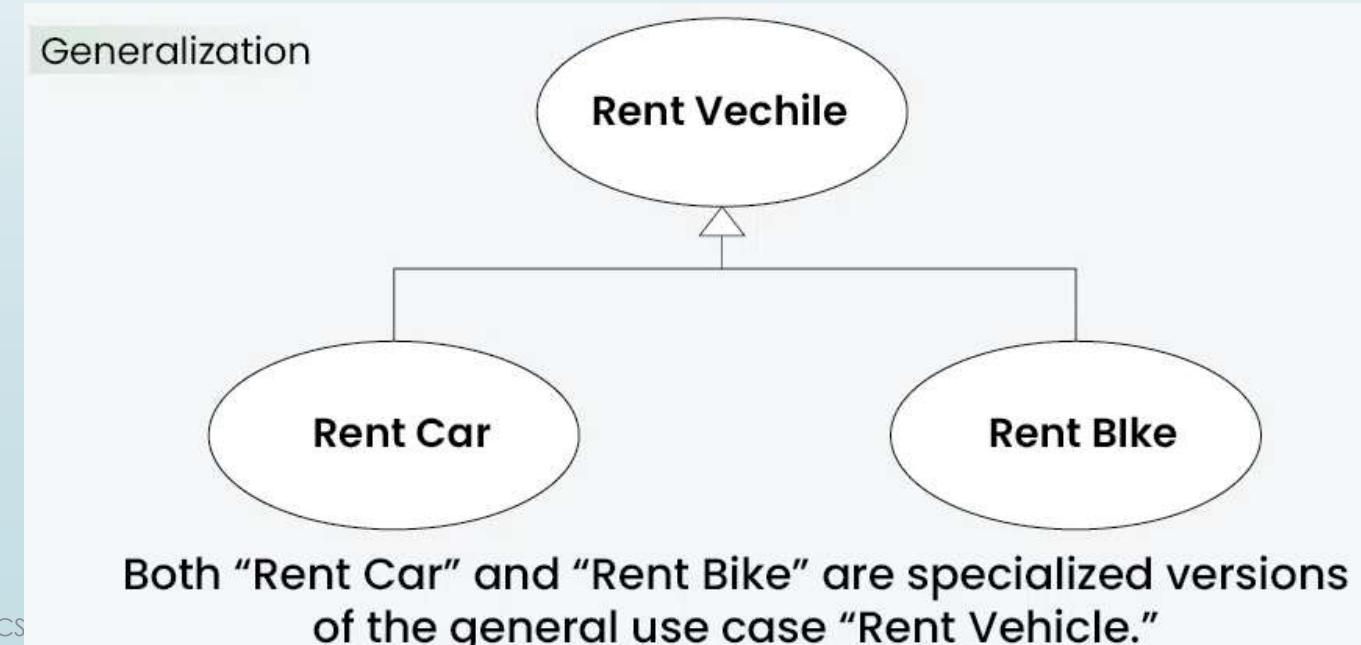


Generalization Relationship:

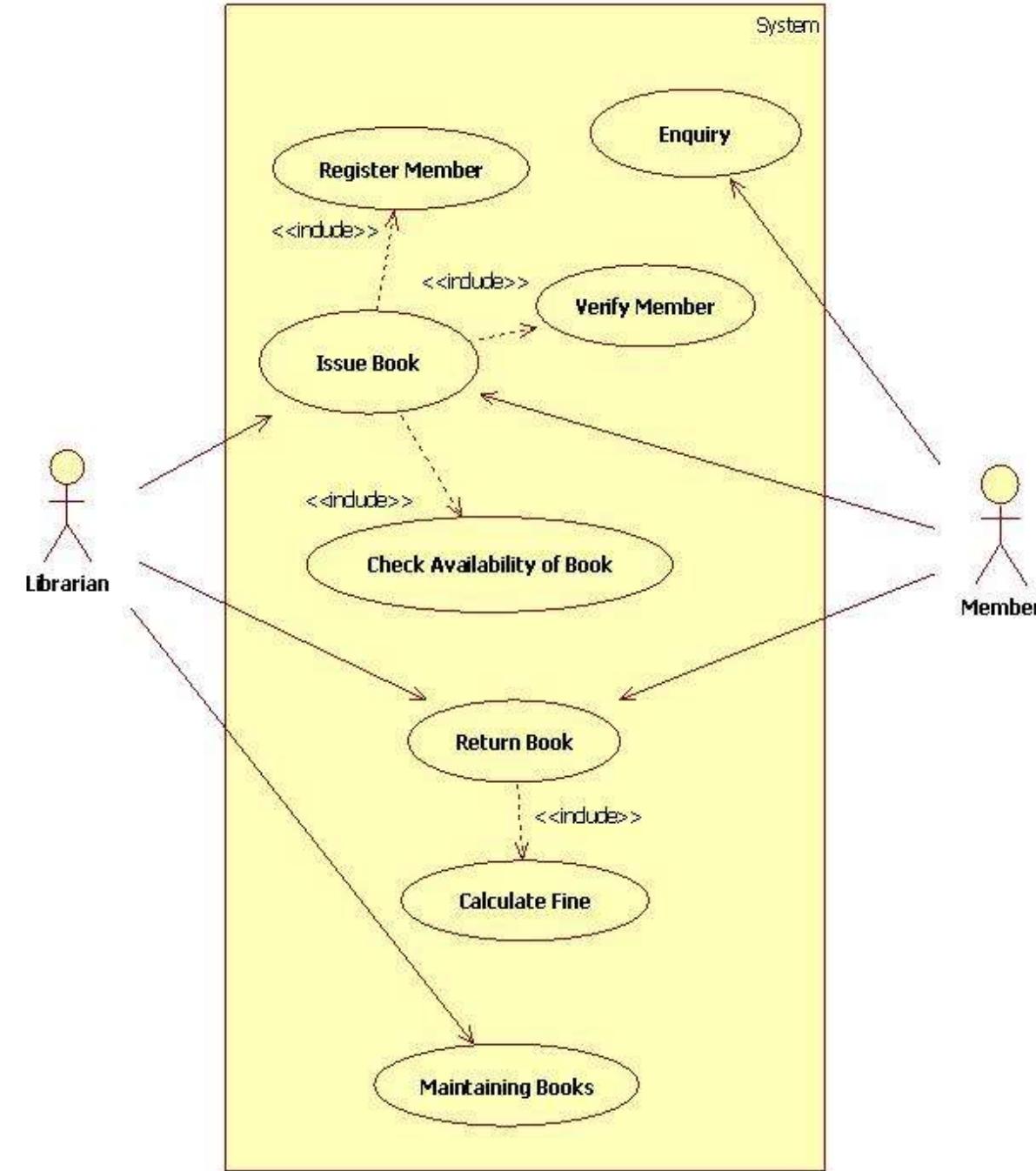
The Generalization Relationship establishes *an “is-a” connection between two use cases*, indicating that one use case is a specialized version of another.

Example: Vehicle Rental System

- ▶ Use Cases: Rent Car, Rent Bike
- ▶ Generalization Relationship: Both “Rent Car” and “Rent Bike” are specialized versions of the general use case “Rent Vehicle.”

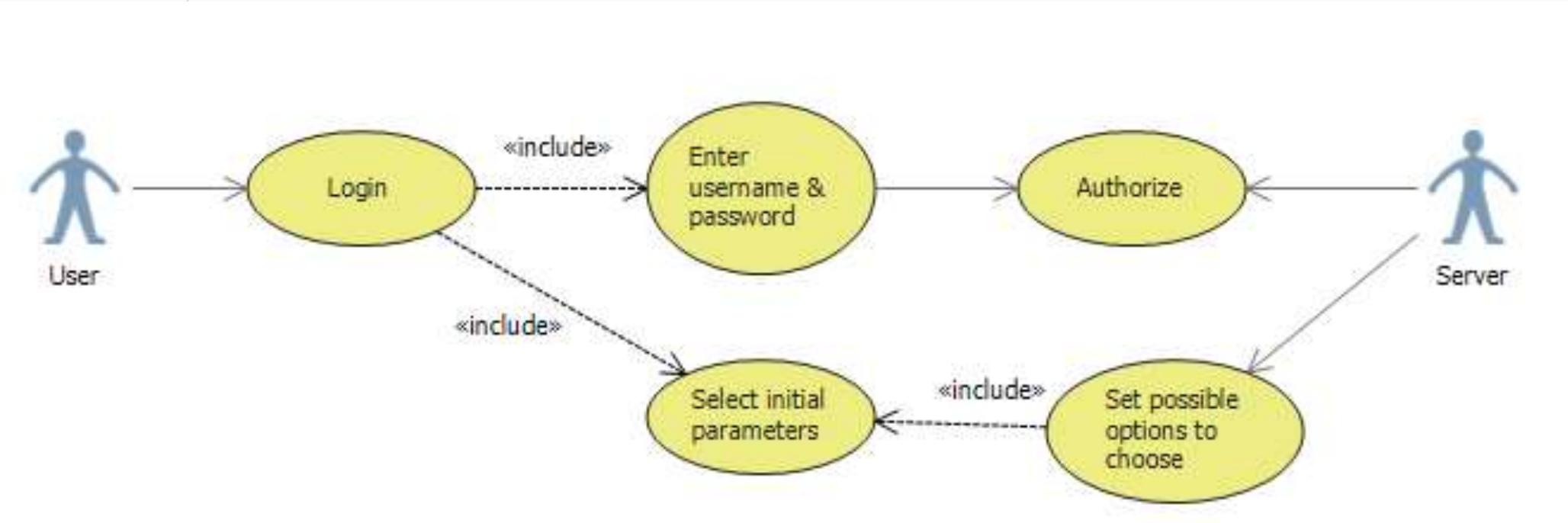


Library Management Use Case diagram

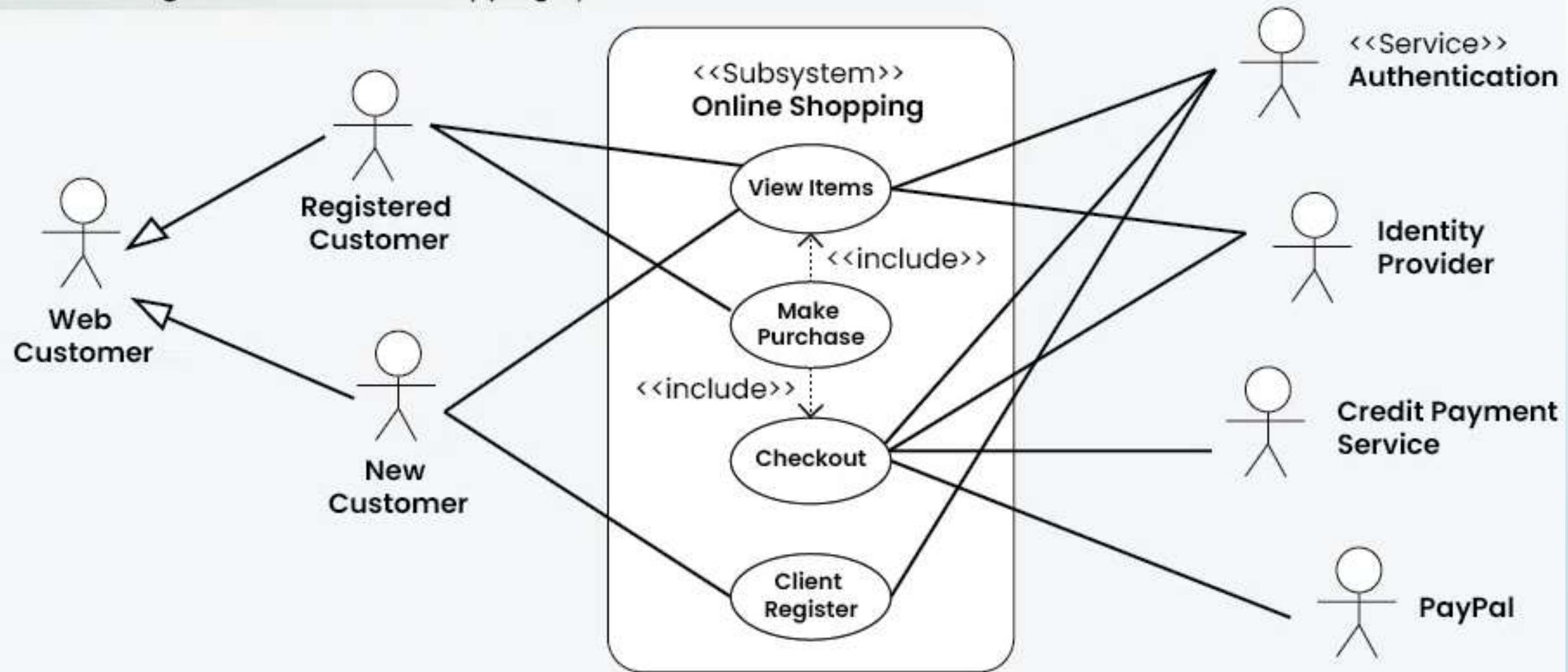


► Use case diagram of user login

14

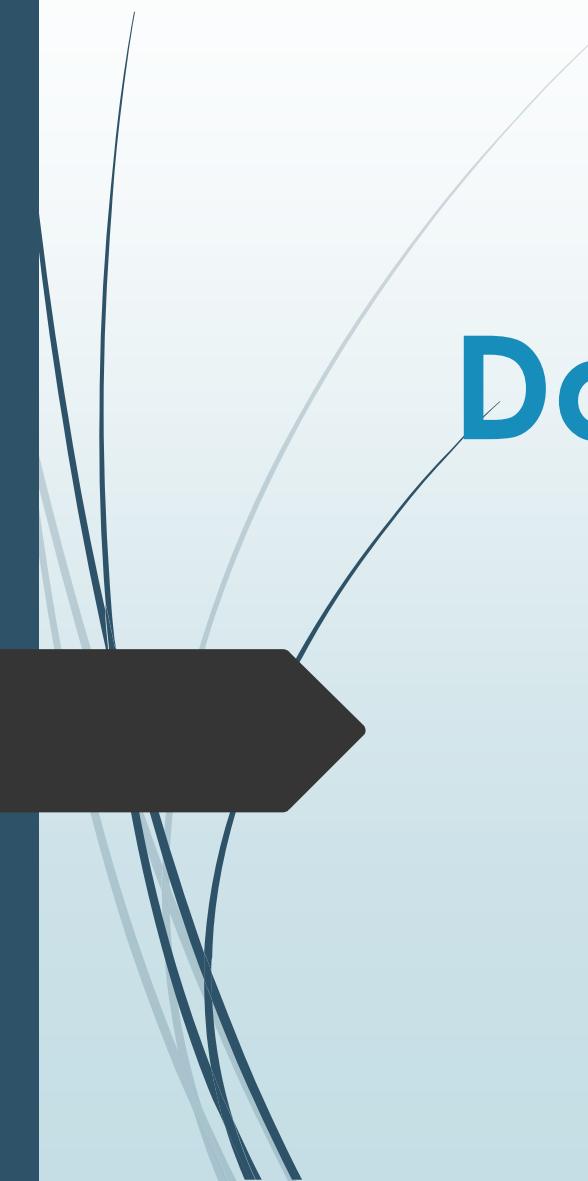


Use Case diagram of an Online Shopping System





Thank You



Data Flow Diagrams

Ashima Tyagi

Assistant Professor

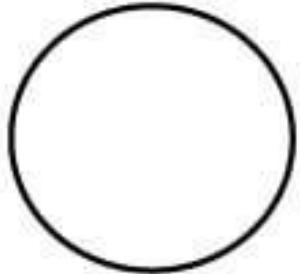
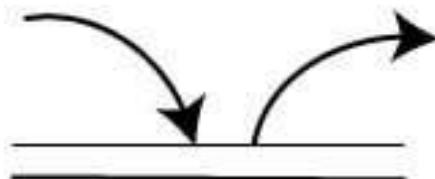
School of Computer Science & Engineering

Outline

► DFD

Data Flow Diagrams

- ▶ The flow of data of a system or a process is represented by DFD.
- ▶ It shows how data enters and leaves the system, what changes the information, and where data is stored.
- ▶ The objective of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesigning a system. The DFD is also called as a data flow graph or bubble chart.

Symbol	Name	Function
	Data flow	Used to Connect Processes to each other , to sources or Sinks; the arrow head indicates direction of data flow.
	Process	Perfroms Some transformation of Input data to yield output data.
	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

Symbols for Data Flow Diagrams

5

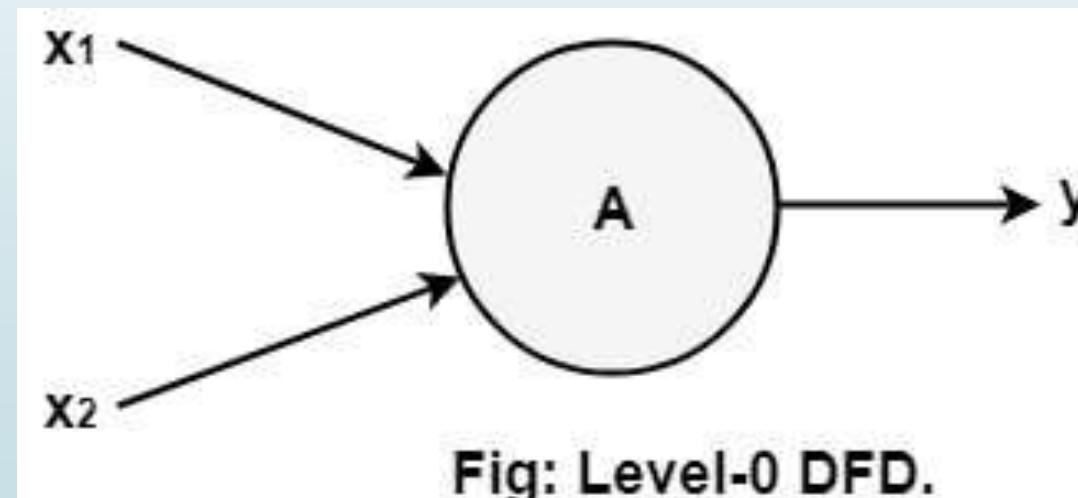
Levels in Data Flow Diagrams (DFD)

- The DFD may be used to perform a system or software at any level of abstraction.
- Infact, DFDs may be partitioned into levels that represent increasing information flow and functional detail.
- Levels in DFD are numbered 0, 1, 2 or beyond.
- Here, we will see primarily three levels in the data flow diagram, which are: 0-level DFD, 1-level DFD, and 2-level DFD.

6

0-level DFD

- It is also known as fundamental system model, or context diagram represents the entire software requirement as a single bubble with input and output data denoted by incoming and outgoing arrows.
- It is essential to preserve the number of inputs and outputs between levels.
- Thus, if bubble "A" has two inputs x_1 and x_2 and one output y , then the expanded DFD, that represents "A" should have exactly two external inputs and one external output as shown in fig:



7

- The Level-0 DFD, also called context diagram of the result management system is shown in fig. As the bubbles are decomposed into less and less abstract bubbles, the corresponding data flow may also be needed to be decomposed.

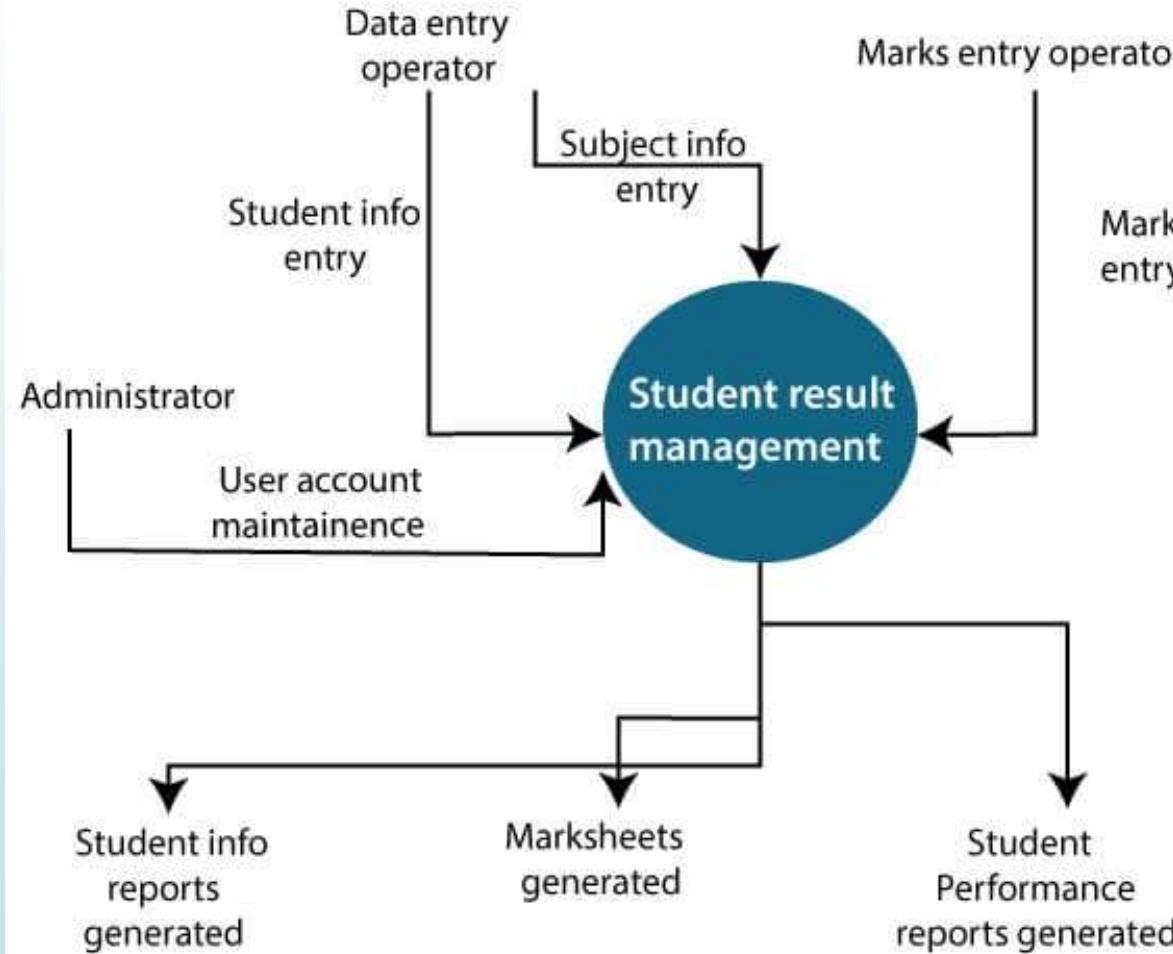


Fig: Level-0 DFD of result management system

8

Level-1 DFD

- In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes. In this level, we highlight the main objectives of the system and breakdown the high-level process of 0-level DFD into subprocesses.

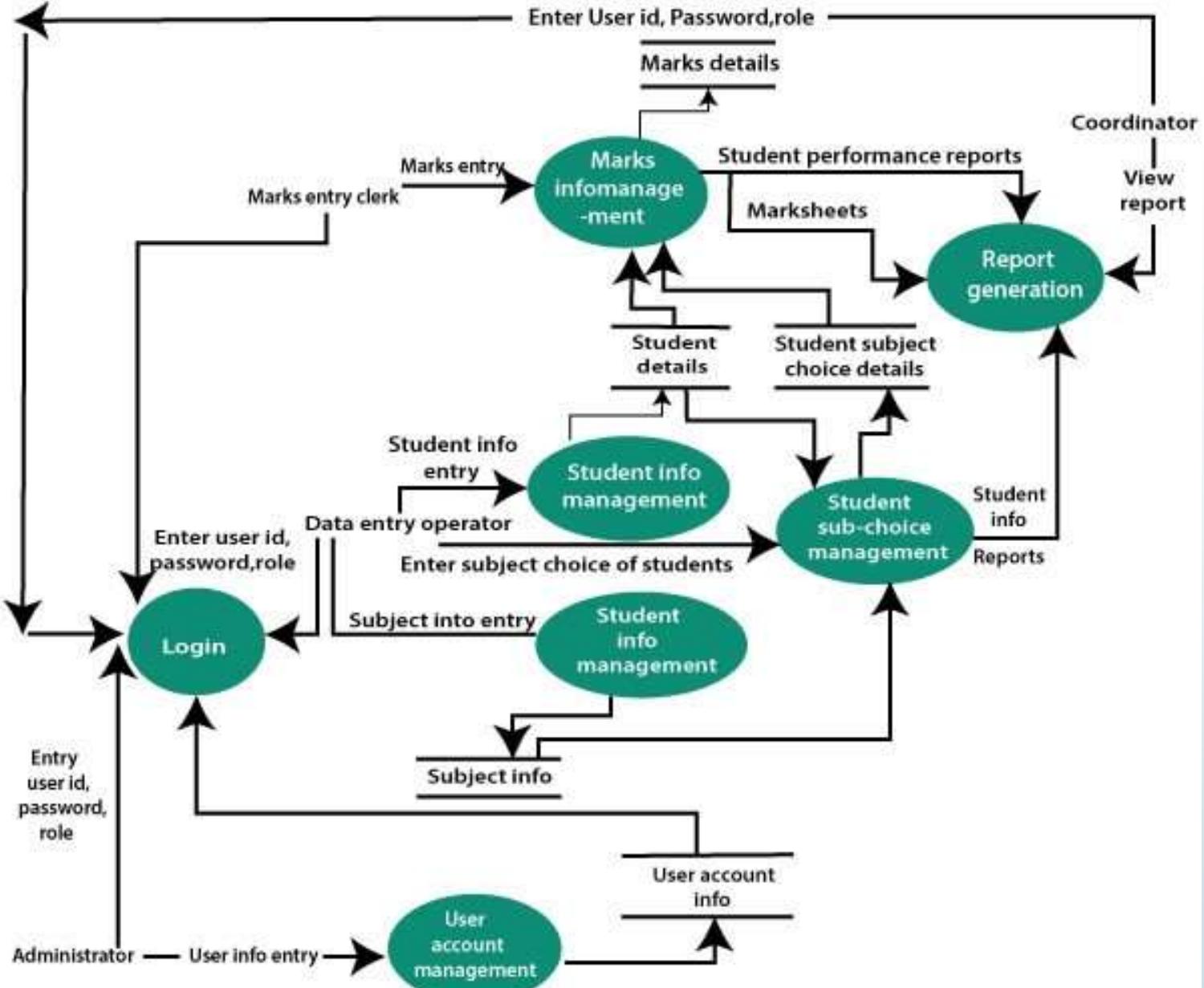
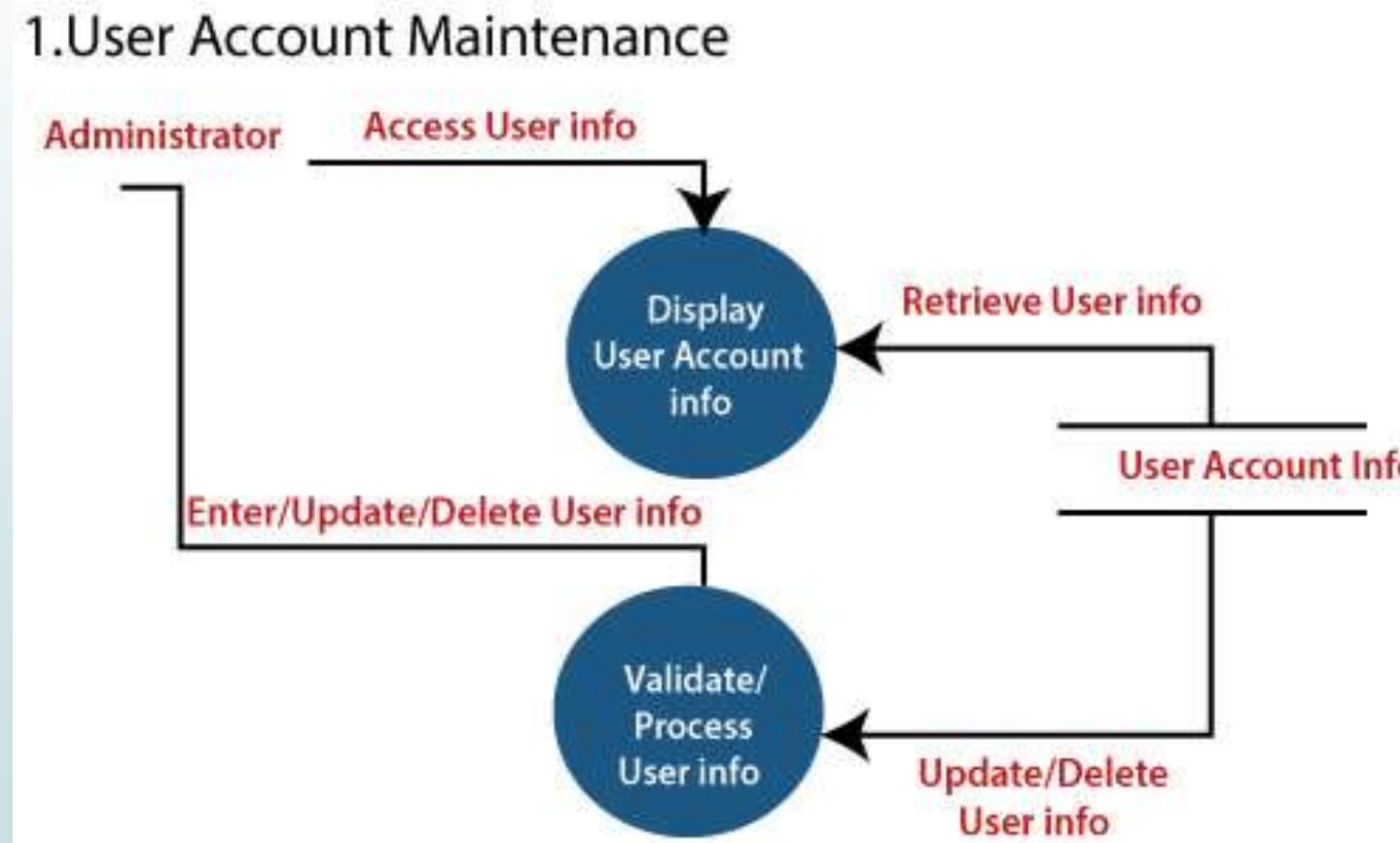


Fig: Level-1 DFD of result management system

10

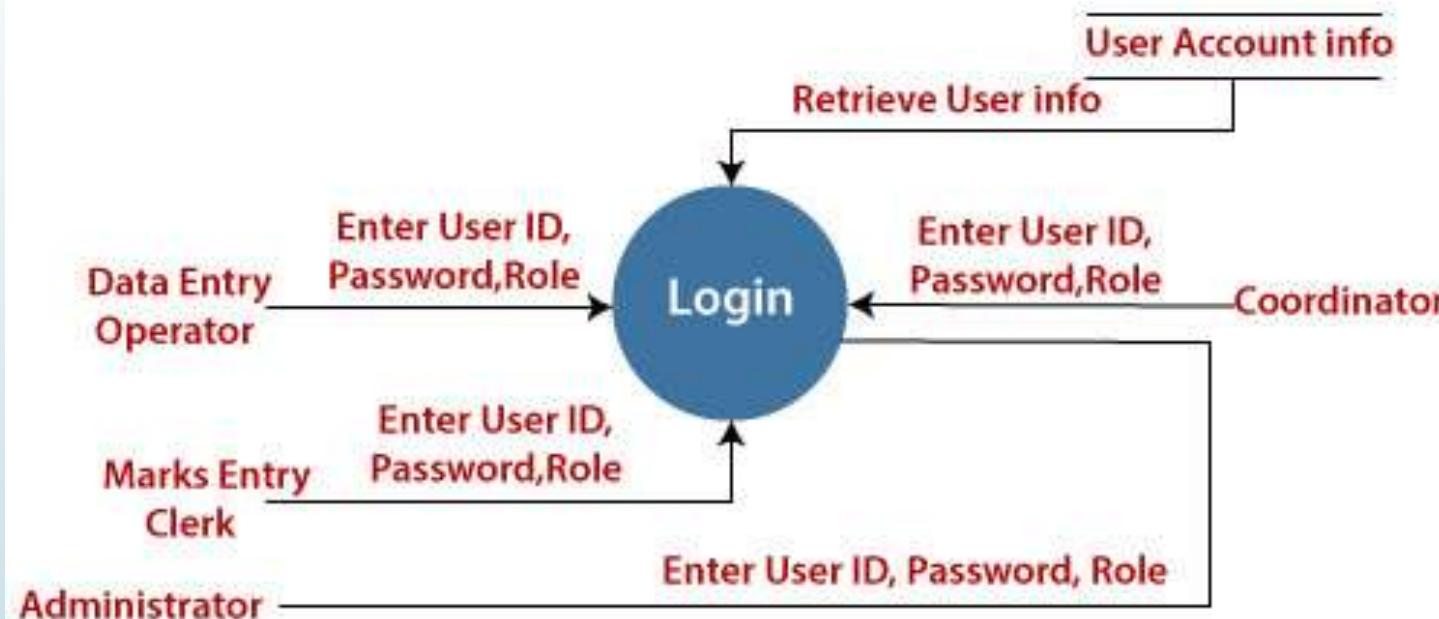
Level-2 DFD

- 2-level DFD goes one process deeper into parts of 1-level DFD. It can be used to project or record the specific/necessary detail about the system's functioning.

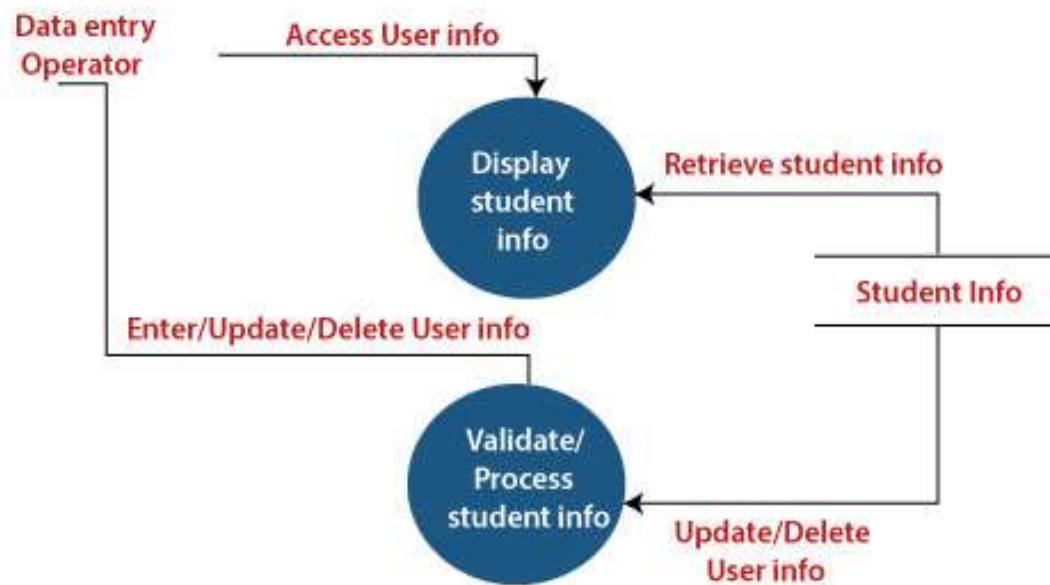


2. Login

The level 2 DFD of this process is given below:

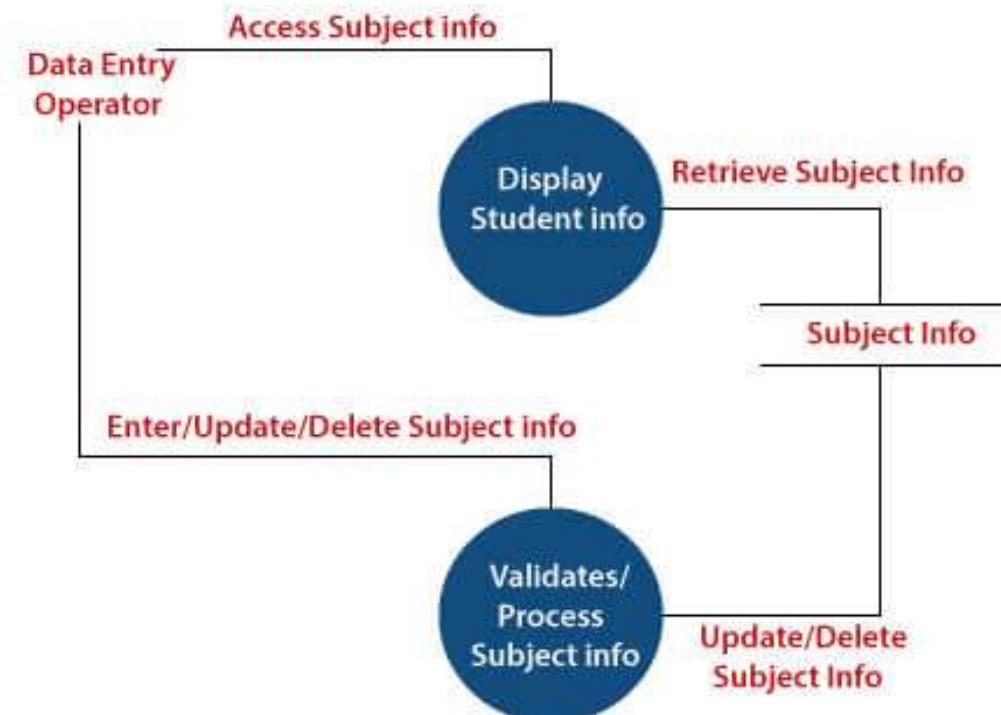


3. Student Information Management



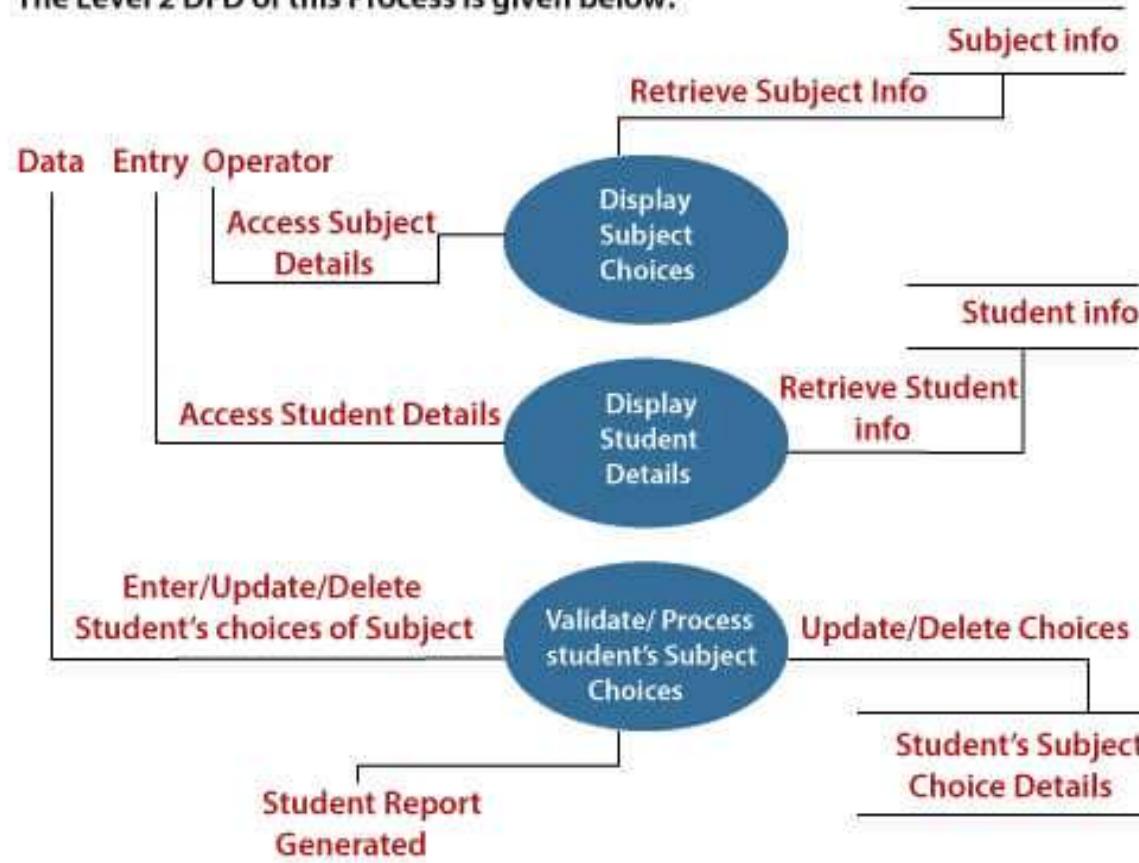
4. Subject Information Management

The level 2 DFD of this process is given below:



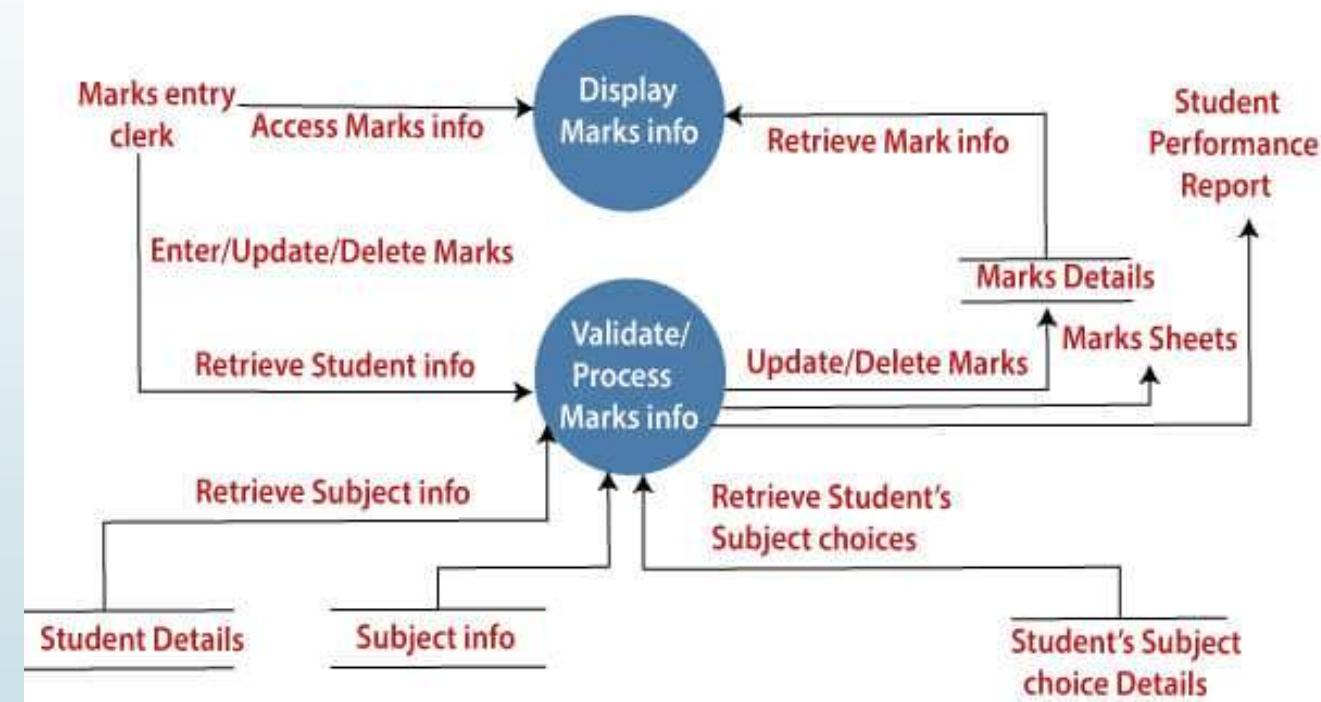
5. Student's Subject Choice Management

The Level 2 DFD of this Process is given below:



6. Marks Information Management

The Level 2 DFD of this Process is given below:





Thank You



Software Design

Ashima Tyagi
Assistant Professor
School of Computer Science & Engineering

Outline

- ▶ Software Design
- ▶ Software design concepts
- ▶ Effective modular design
- ▶ Cohesion and Coupling
- ▶ Architectural design
- ▶ Object Oriented Design
- ▶ User Interface Design
- ▶ Detailed Design Transaction Transformation
- ▶ Refactoring of designs

Software Design

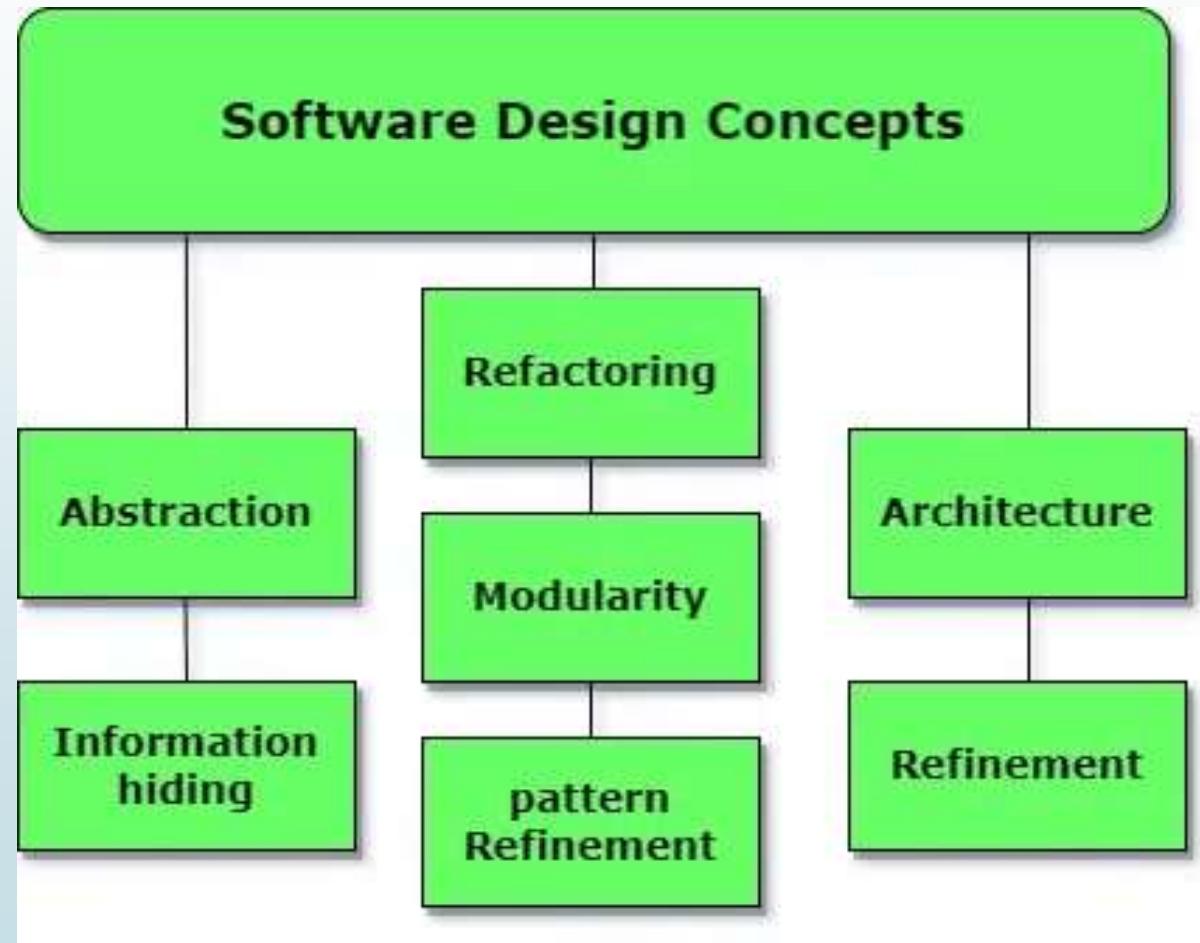
The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.

4

Software design challenges

1. Complexity
2. Maintaining Quality
3. Meeting Deadlines
4. Keeping up with new technologies
5. Managing changing requirements
6. Debugging
7. Balancing short-term and long-term goals

Software Design Concepts



Software design concepts

1. **Abstraction (Hide Irrelevant data):** Abstraction simply means to **hide the details to reduce complexity and increase efficiency or quality.** Abstraction is used to simplify complex systems by hiding unnecessary details and exposing only the relevant functionality. This makes software easier to develop, use, maintain, and scale. Abstraction allows developers to focus on what a module does rather than how it does it. By defining abstract methods or interfaces, the same code can be used in different scenarios without modification.
2. **Modularity (subdivide the system):** Modularity simply means **dividing the system or project into smaller parts to reduce the complexity of the system or project.** In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays, there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we can divide the system into components then the cost would be small.

- 3. Architecture (design a structure of something):** Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
- 4. Refinement (removes impurities):** Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
- 5. Pattern (a Repeated form):** A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
- 6. Information Hiding (Hide the Information):** Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.
- 7. Refactoring (Reconstruct something):** Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't impact the behavior of the design and improves the internal structure".

Effective Modular Design

The process of breaking down a software into multiple independent **modules** where each module is developed separately is called Modularization.

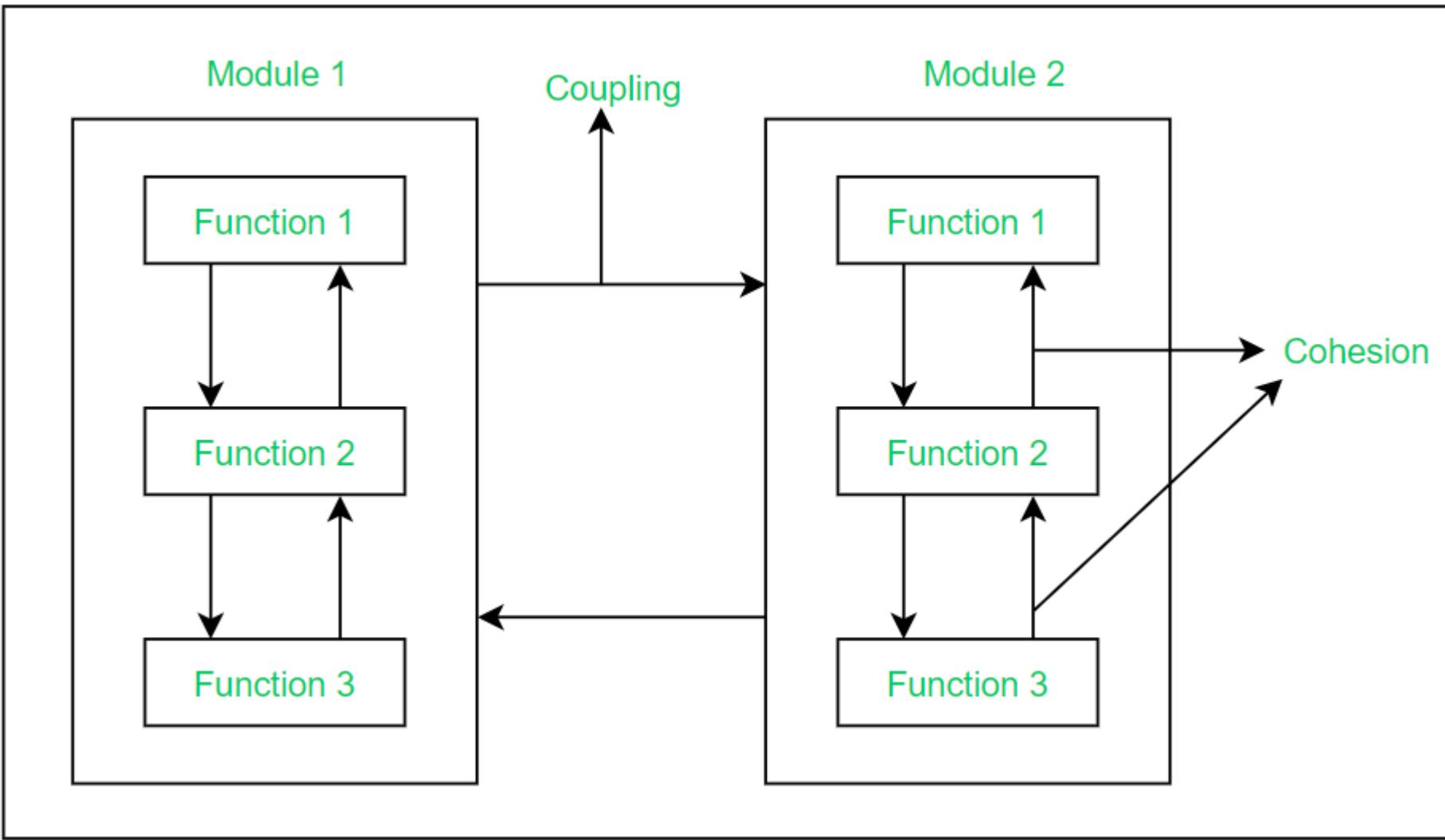
Effective modular design can be achieved if the partitioned modules are separately solvable, modifiable as well as compilable. Here separate compilable modules means that after making changes in a module there is no need of recompiling the whole software system.

- In order to build a software with effective modular design there is a factor "**Functional Independence**" which comes into play. The meaning of Functional Independence is that a **function is atomic in nature** so that it performs only a single task of the software without or with least interaction with other modules. **Functional Independence is considered as a sign of growth in modularity** i.e., presence of larger functional independence results in a software system of good design and design further affects the quality of the software.

Benefits of Independent modules/functions in a software design:

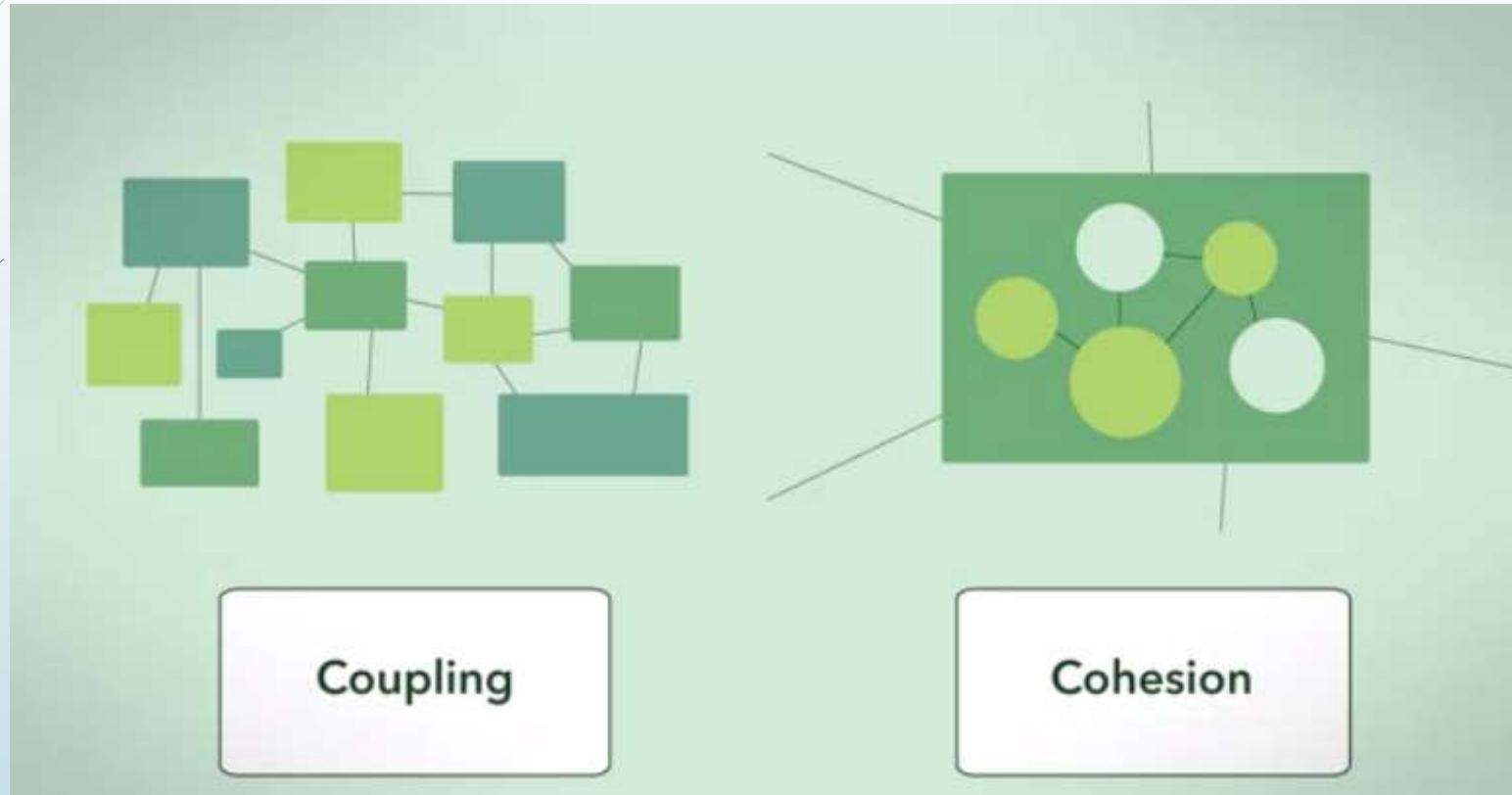
- Since the functionality of the software have been broken down into atomic levels, thus developers **get a clear requirement of each and every functions** and hence designing of the software becomes easy and error free.
- As the modules are independent they have limited or **almost no dependency** on other modules. **So, making changes in a module without affecting the whole system is possible in this approach.**
- **Error propagation from one module to another** and further in whole system can be neglected and it saves time during testing and debugging.
- Independence of modules of a software system can be measured using 2 criteria : **Cohesion, and Coupling.**

Cohesion Vs Coupling



Cohesion Vs Coupling

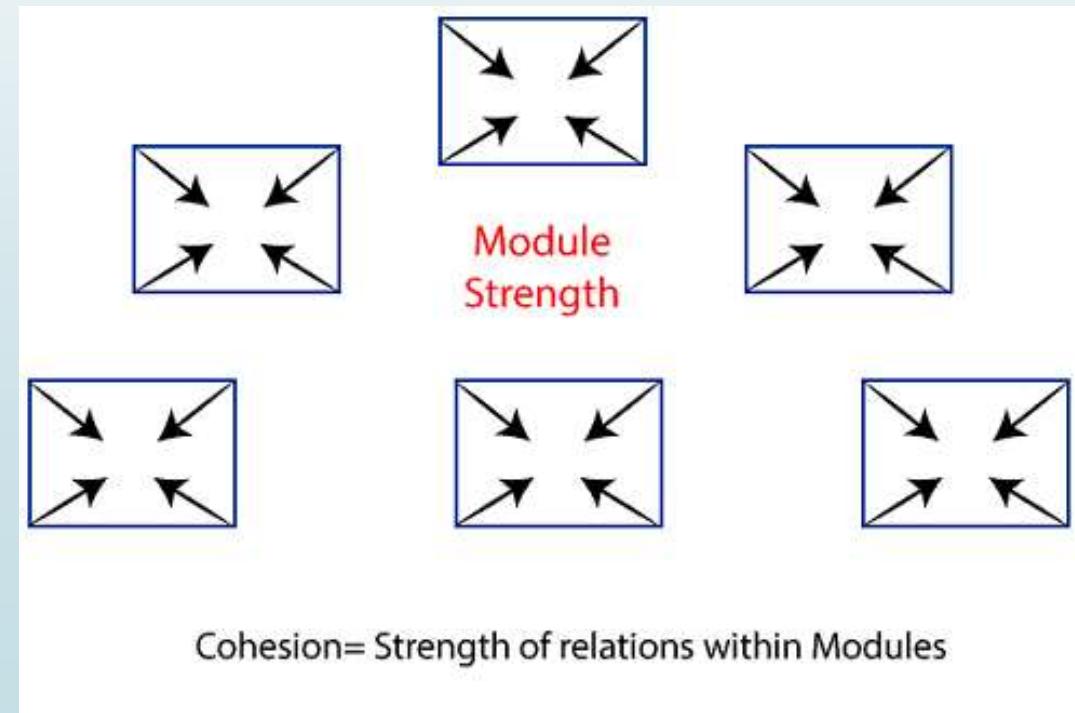
- We want: High cohesion and low coupling



Cohesion

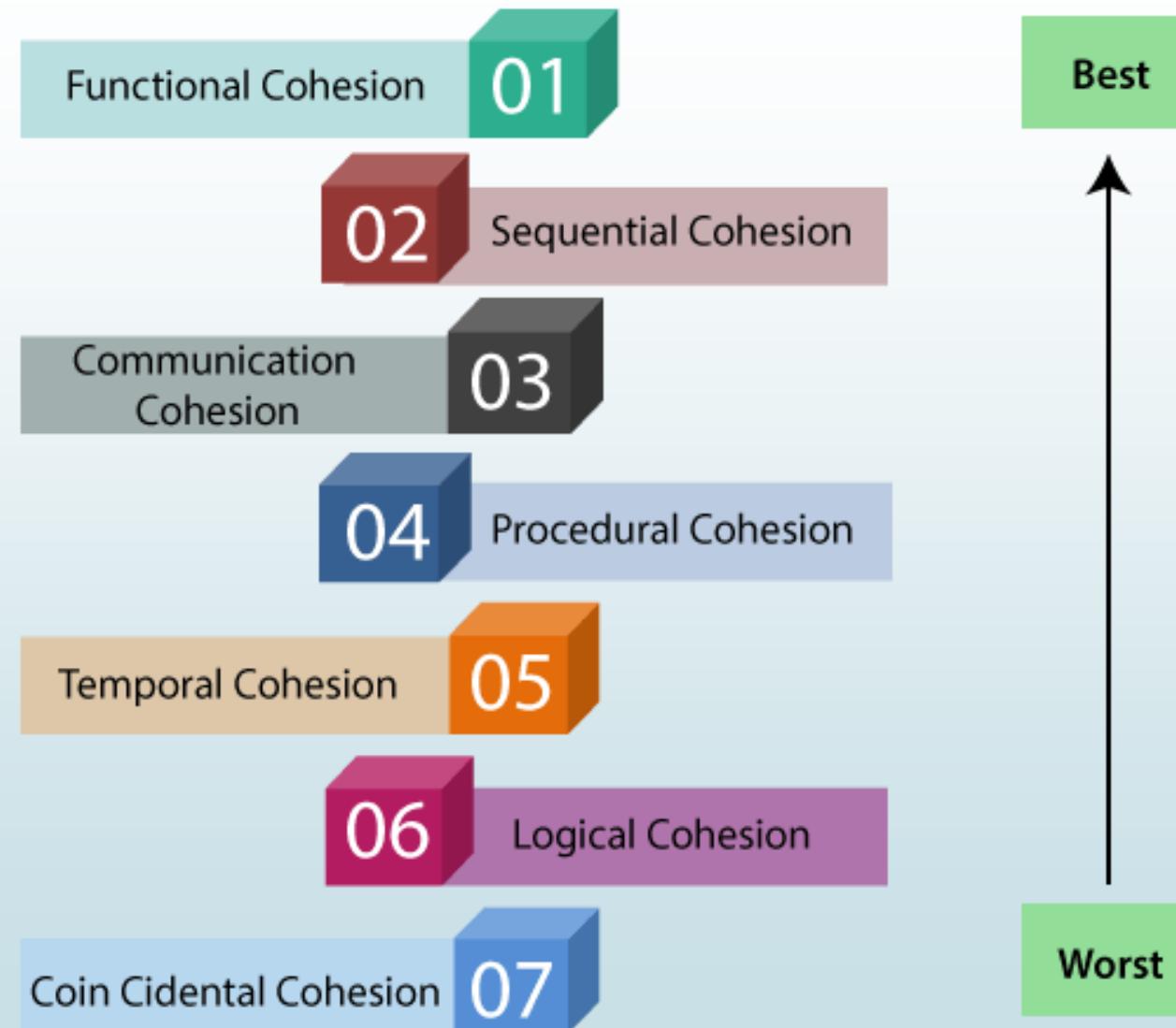
12

- ▶ Cohesion defines to the degree to which the **elements of a module belong together.**
- ▶ Thus, cohesion measures the strength of relationships between pieces of functionality within a given module.
- ▶ For example, in highly cohesive systems, functionality is strongly related.



Types of Modules Cohesion

13



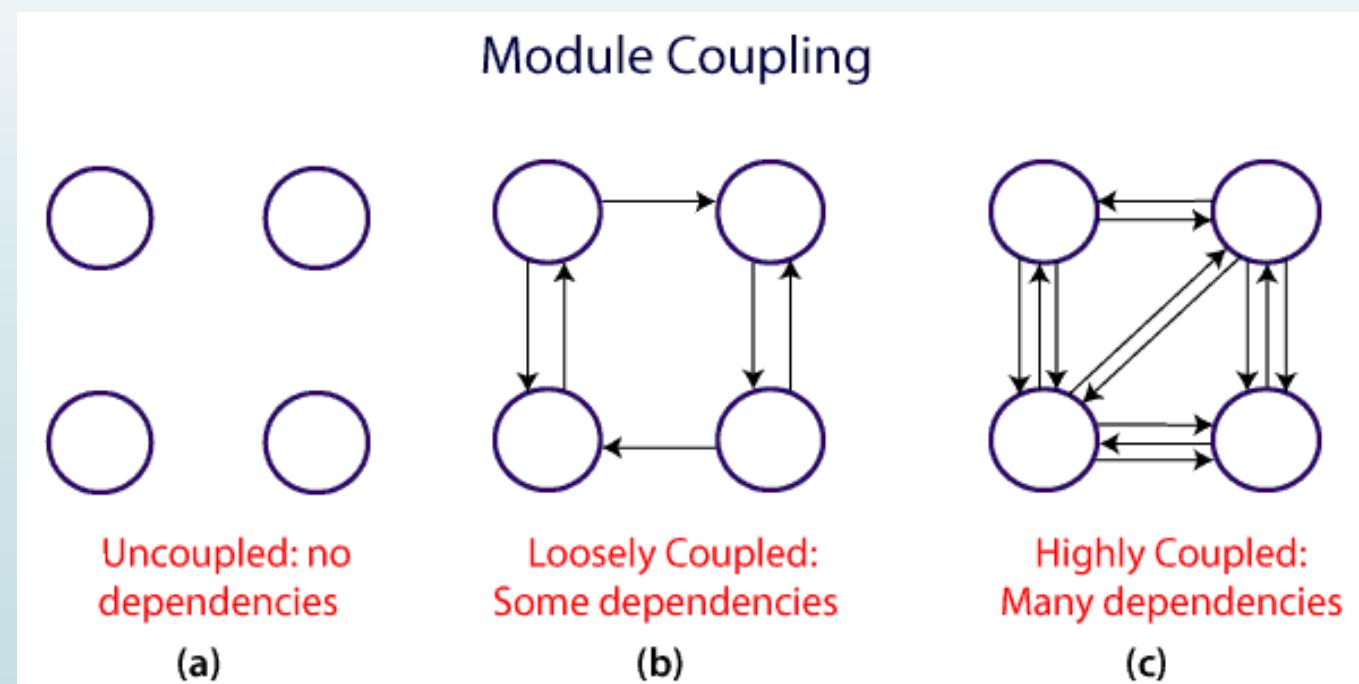
1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Coupling

15

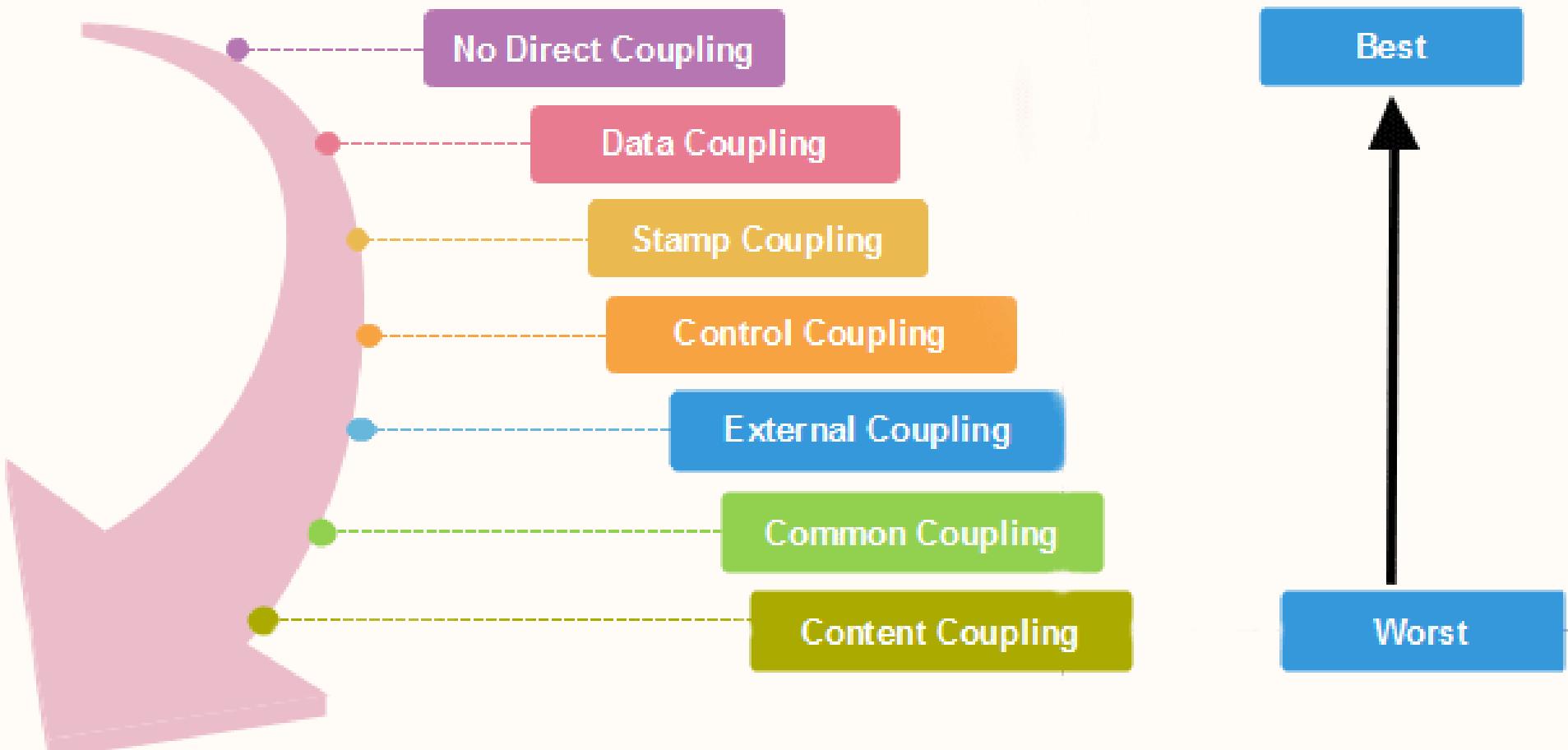
- The coupling is the degree of interdependence between software modules.
- Two modules that are tightly coupled are strongly dependent on each other.
- However, two modules that are loosely coupled are not dependent on each other.
- Uncoupled modules have no interdependence at all within them.

A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the **amount of shared data is large**. Thus, it can be said that a design with high coupling will have more errors.



Types of Modules Coupling

There are various types of module Coupling are as follows:



- 1. No Direct Coupling:** There is no direct coupling between M1 and M2. In this case, modules are subordinates to different modules. Therefore, no direct coupling.
- 2. Data Coupling:** When data of one module is passed to another module, this is called data coupling.
- 3. Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.
- 4. Control Coupling:** Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).
- 5. External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.
- 6. Common Coupling:** Two modules are common coupled if they share information through some global data items.
- 7. Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Coupling Vs Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative independence between the modules.	Cohesion shows the module's relative functional strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Architectural Design

- Software architecture is the high-level structure of a software system that defines how different components interact.
- It serves as a **blueprint** that guides developers in designing and implementing software systems efficiently.

20

The software needs an architectural design to represent the design of the software.

IEEE defines architectural design as “the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.”

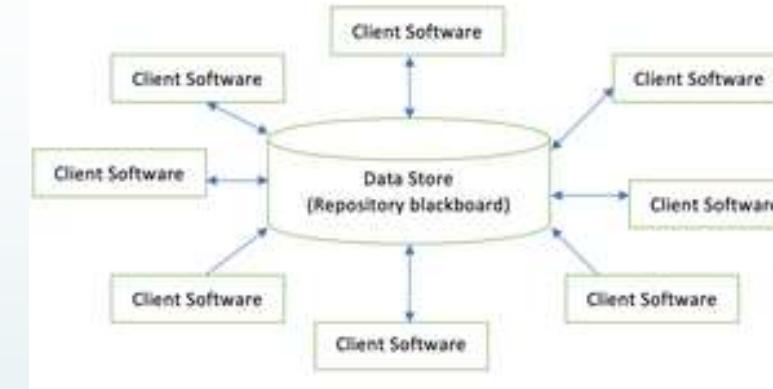
The software that is built for computer-based systems can exhibit one of these many architectural styles.

System Category Consists of

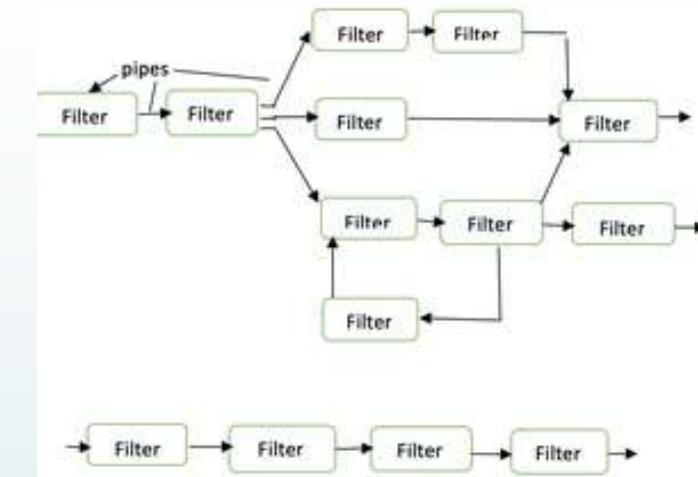
- ▶ A *set of components* (eg: a database, computational modules) that will perform a function required by the system.
- ▶ The *set of connectors* will help in coordination, communication, and cooperation between the components.
- ▶ *Conditions* that defines how components can be integrated to form the system.
- ▶ *Semantic models* that help the designer to understand the overall properties of the system.
- ▶ The *use* of architectural styles is to establish a structure for all the components of the system.

Different Architecture Styles:

- Data centered architectures
- Data flow architectures
- Call and Return architectures
- Object Oriented architecture
- Layered architecture



Data centered architectures



Data flow architectures

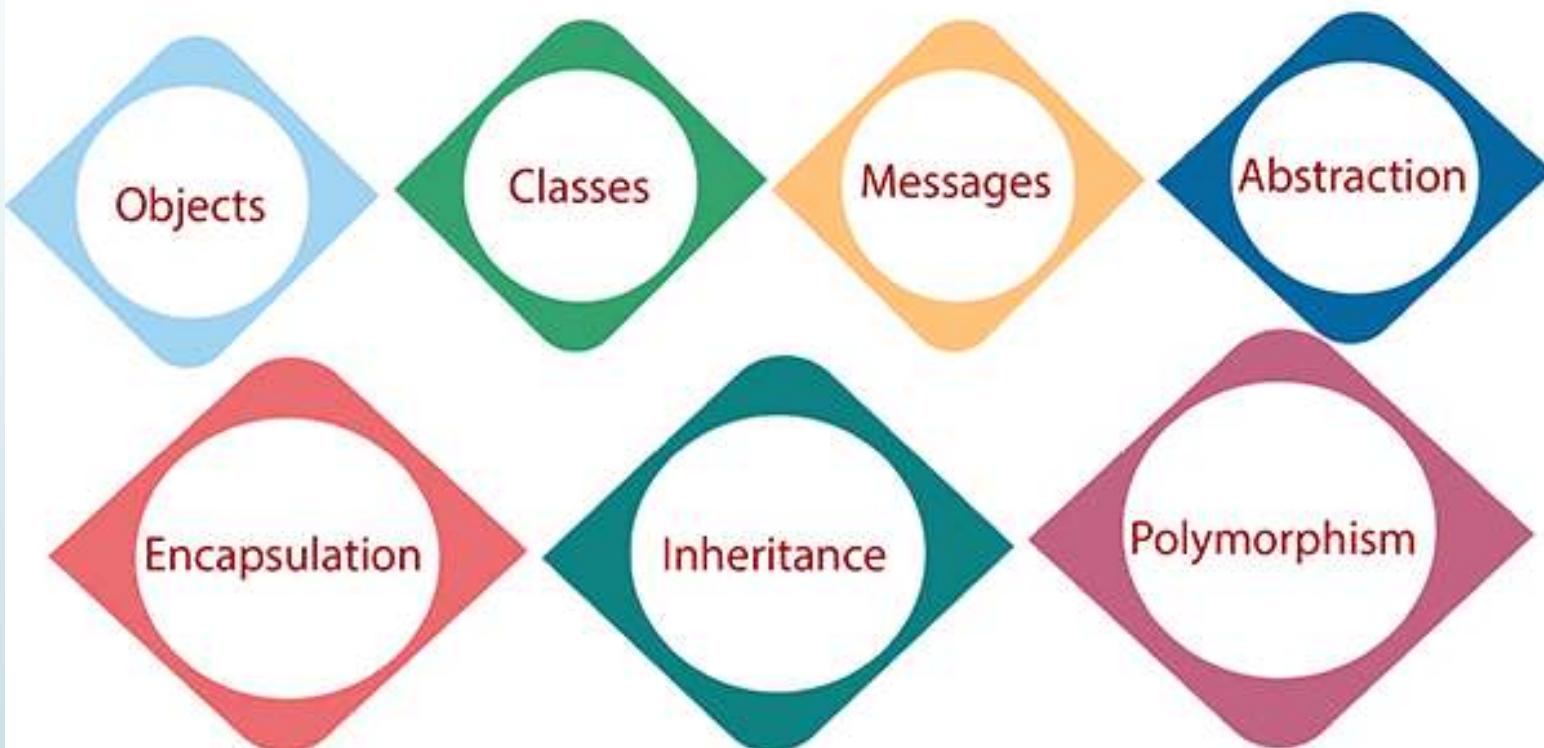


Call and Return architectures

Object-Oriented Design

- In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities).
- The state is distributed among the objects, and each object handles its state data.
- For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.
- In other words, each object is a member of some class. Classes may inherit features from the superclass.

Object Oriented Design



Advantages:

1. **Modularity:** OOD promotes modularity by organizing software into objects that represent real-world entities. This makes it easier to maintain and update software components independently.
2. **Reusability:** Objects can be reused in different parts of an application or in different applications altogether, which can significantly reduce development time and effort.
3. **Encapsulation:** Wrapping data (variables) and methods (functions) that operate on the data into a single unit (a class).
4. **Abstraction:** OOD allows developers to create abstract representations of real-world entities, which can simplify the design and implementation of complex systems.
5. **Inheritance:** Inheritance allows new classes to inherit attributes and methods from existing classes, enabling code reuse and promoting the creation of hierarchies of related classes.
6. **Polymorphism:** Polymorphism allows objects to be treated as instances of their parent class, which can simplify code and make it more flexible and adaptable to change.

Challenges:

1. **Complexity:** Object-oriented designs can become complex, especially in large-scale systems with many interacting objects. Managing this complexity can be challenging and require careful design and planning.
2. **Performance Overhead:** Object-oriented programming languages often introduce a performance overhead compared to procedural languages due to features like dynamic dispatch and memory allocation for objects.
3. **Learning Curve:** Object-oriented concepts such as inheritance, polymorphism, and encapsulation can be challenging for developers new to OOD, leading to longer learning curves.
4. **Design Patterns:** Choosing the right design patterns for a given problem can be challenging, and using them incorrectly can lead to overly complex or inefficient designs.
5. **Testing and Debugging:** Object-oriented designs can be more difficult to test and debug compared to procedural designs, especially when dealing with complex interactions between objects.
6. **Maintenance:** While OOD promotes modularity and reusability, poorly designed object-oriented systems can be difficult to maintain and extend, especially if the original design did not anticipate future changes.

User-Interface Design

The user interface is the **front-end application view** to which the user interacts to use the software. The software becomes more popular if its user interface is:

- ▶ Attractive
- ▶ Simple to use
- ▶ Responsive in a short time
- ▶ Clear to understand
- ▶ Consistent on all interface screens

UI Design Principles



Elements:

► Layout

- **Grid systems** help align elements properly.
- Responsive design ensures adaptability on different devices.

► Typography

- Use readable fonts like **Sans-serif** (e.g., **Roboto, Open Sans**).
- Maintain **consistent font sizes** for headings, body text, and buttons.

► Color Scheme

- Use a **primary color palette** and a few accent colors.
- Ensure **contrast** for readability (e.g., black text on a white background).

► Buttons & Icons

- Use **clear labels and recognizable icons**.
- Ensure buttons have a **hover/press state**.

► Navigation

- Keep navigation **simple and intuitive**.
- Use a **hamburger menu** for mobile devices.

► Forms & Inputs

- Provide **clear labels** and placeholders.
- Show error messages when users enter incorrect data.

Refactoring of designs

- ▶ Refactoring in software design refers to the process of modifying an existing system's structure without changing its external behavior.
- ▶ The goal is to improve code maintainability, readability, and performance while reducing technical debt.

Refactoring improves software by:

- ▶ Enhancing readability (simplifies complex code).
- ▶ Improving maintainability (reduces dependencies and code duplication).
- ▶ Boosting performance (removes unnecessary computations).
- ▶ Reducing complexity (eliminates redundant logic).
- ▶ Ensuring scalability (better modularity for future changes).

Refactoring should be done when:

- ✓ Code is difficult to understand or modify.
- ✓ The system has repeated code (duplication).
- ✓ Adding new features takes too much time due to tight coupling.
- ✓ The code has long methods or large classes.
- ✓ The system experiences performance bottlenecks.

Refactoring **is NOT done** when software is close to a deadline or when major redesigns are required (in such cases, rewriting may be a better option).

Types of Refactoring in Design

- 1. Code-Level Refactoring:** Refactoring the structure of code without changing functionality.
 - ▶ Extract Method: Break large functions into smaller, reusable functions. ✓ Improves readability and maintainability.
 - ▶ Reduce Nested Loops: ✓ Improves performance.
- 2. Architectural Refactoring:** Involves restructuring software components for better scalability and performance.
 - ▶ Reduce coupling
 - ▶ Layered Architecture Refactoring

3. Performance-Based Refactoring: Refactors inefficient code to improve performance.

- Optimize Loops ✓ Avoid redundant calculations inside loops.

4. UI/UX Design Refactoring: Refactoring user interfaces to improve usability.

- Reduce Clutter: Remove unnecessary UI elements.
- Improve Navigation: Simplify menu structures.
- Optimize Load Times: Reduce API calls and optimize image sizes.

Example:

- ◆ Before Refactoring – A long form with 10 fields on one page.
- ◆ After Refactoring – Break into multi-step forms to improve user experience.

Detailed Design Transaction Transformation

- ▶ Transaction transformation is a software design approach that deals with processing input transactions into output results by applying a structured sequence of operations.
- ▶ It is commonly used in business applications, banking systems, and real-time data processing.

What is Transaction Transformation?

It refers to the structured way of handling transactions (user requests or system inputs) that trigger a series of operations to produce a required output.

Transaction Processing Flow

A transaction typically follows these steps:

1. Input Transaction Processing

- The system receives an input request from a user or another system. Example: A customer places an online order.

2. Validation & Pre-Processing

- Checks if the input data is valid (e.g., username/password, stock availability). Example: Verify if an item is in stock before confirming the order.

3. Transformation & Computation

- The system processes the transaction according to business rules. Example: Calculate total cost including taxes and discounts.

4. Database Update

- The transaction modifies relevant data records. Example: Update inventory, deducting purchased quantity.

5. Output Generation

- The system generates results, which could be: UI feedback (e.g., "Order Confirmed!"), A receipt or invoice, A message or notification

Example – Online Banking Transaction

Let's consider an online fund transfer:

1. Input Transaction Processing

- User initiates transfer (inputs account details, amount).

2. Validation

- Check if account exists.
- Verify sufficient balance.

3. Processing

- Deduct amount from sender's account.
- Add amount to receiver's account.

4. Database Update

- Commit changes to ensure consistency.

5. Output Generation

- Show confirmation message (e.g., "Transfer Successful!").
- Send notification via SMS/email.

Error Handling & Recovery

Robust systems handle transaction failures using:

- ✓ **Rollback Mechanism** → Undo changes if a failure occurs.
- ✓ **Compensation Transactions** → Reverse incorrect transactions (e.g., refund processing).
- ✓ **Retry Logic** → Reattempt transaction if it fails due to temporary issues.



Thank You



Software Reliability

Ashima Tyagi

Assistant Professor

School of Computer Science & Engineering

Outline

- ▶ Introduction
- ▶ Hardware reliability vs. Software reliability
- ▶ Reliability metrics
- ▶ Failure and Faults
- ▶ Dependability Concept
- ▶ Reliability and Availability Modeling
- ▶ Reliability Evaluation Testing methods and models
- ▶ Static and Dynamic Analysis

Software Reliability

- ▶ Software reliability is defined as the probability of **failure-free operation** of a software system for a specified time in a specified environment.
- ▶ It is a measure of how dependable and error-free the software is.
- ▶ High reliability means fewer bugs and more stable performance.

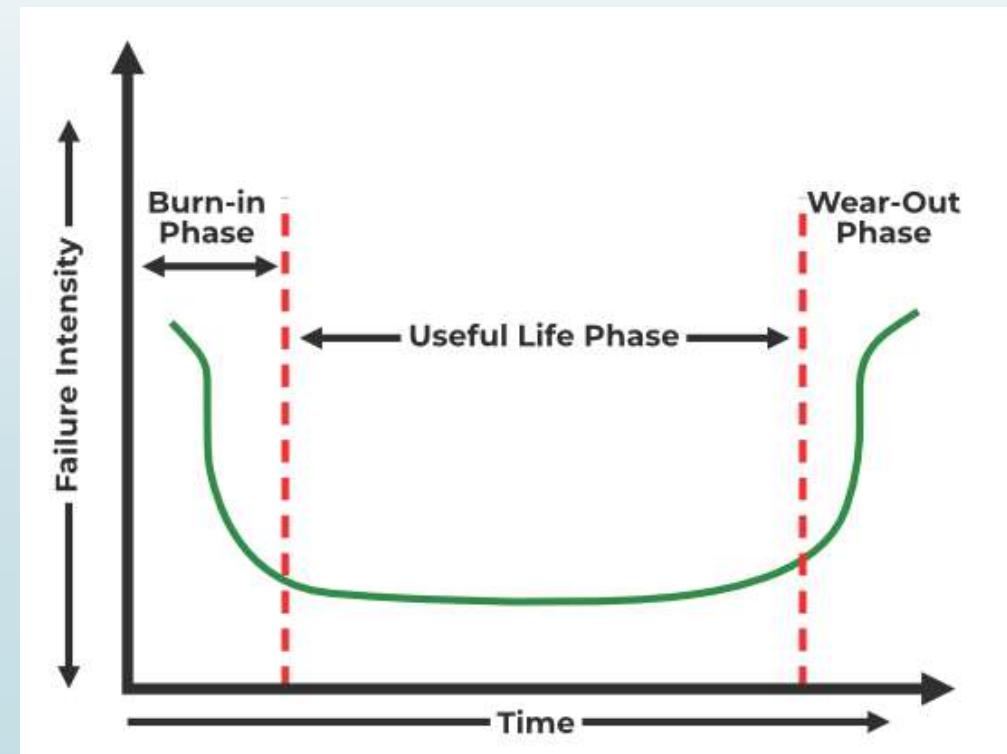
4

Examples:

- ▶ Consider a weather forecasting app. If the app consistently provides accurate weather predictions without crashes or incorrect information, then its reliability is high. If it often crashes or gives wrong data, its reliability is low.
- ▶ **Reliability under stress:** A reliable e-commerce website can handle the increased traffic (a high load) without crashing or slowing down excessively. It will load pages properly, process payments accurately, and allow users to complete transactions. The website can manage a large number of concurrent users (e.g., thousands or millions) without failure. If the website does crash under heavy load, it would be seen as having low reliability under stress.

Hardware Reliability

- ▶ Hardware reliability is the probability that the ability of the hardware to perform its function for some period of time.
- ▶ It may change during certain periods such as initial burn-in or the end of useful life.



6

Burn-in Phase (Early Failures)

- High failure intensity at the beginning occurs due to manufacturing defects, poor-quality components, or assembly errors.
- This phase is also called the infant mortality phase, where weak components fail early.
- Manufacturers often perform stress testing or burn-in testing to eliminate defective parts before reaching customers.

Useful Life Phase (Random Failures)

- Once weak components have failed, the remaining hardware operates reliably.
- The failure rate is relatively constant and low.
- Most failures during this phase are due to random external factors, such as power surges, accidental damage, or unpredictable wear.

Wear-Out Phase (Aging Failures)

- As time progresses, components age and degrade, leading to an increase in failure intensity.
- Common causes include material fatigue, corrosion, and wear.
- Preventative maintenance, replacements, or upgrades are required to extend the product's life.

Causes of Hardware Failures

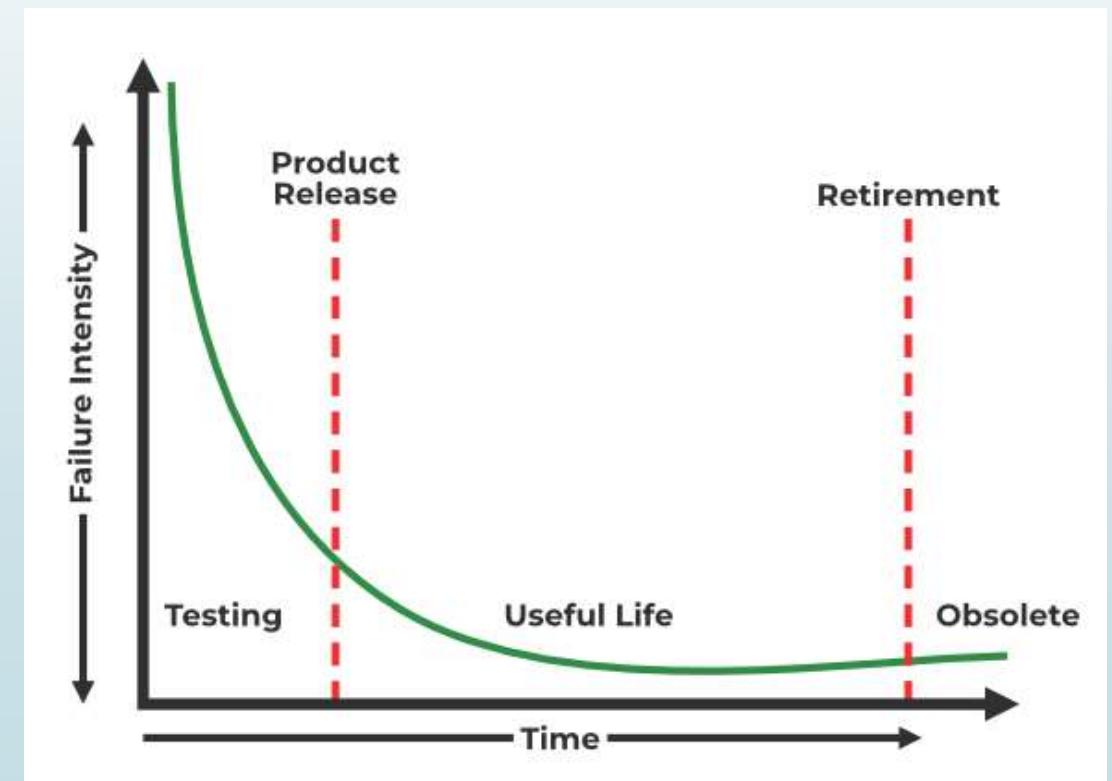
- ▶ Physical Wear and Tear → Components degrade over time.
- ▶ Environmental Factors → Temperature, humidity, dust, and power surges affect performance.
- ▶ Manufacturing Defects → Some components may have defects from production.
- ▶ Aging and Corrosion → Metallic parts may corrode, and circuits may weaken.

Improving Hardware Reliability

- ✓ Redundant Components → Use backup hardware like RAID for storage.
- ✓ Cooling Systems → Prevent overheating with fans and heat sinks.
- ✓ Quality Manufacturing → Use high-quality materials and strict testing.
- ✓ Regular Maintenance → Clean and inspect components regularly.

Software Reliability

- Software reliability is the probability that the software will operate failure-free for a specific period of time in a specific environment.
- It is measured per some unit of time.



9

Testing Phase (High Failure Intensity Initially)

- Failure intensity is high initially because:
 - Many undetected bugs and errors exist.
 - Code may not be optimized for real-world use.
 - Software engineers actively identify and fix defects.
- Over time, failures decrease as bugs are found and fixed.

Product Release (Start of Useful Life Phase)

- After rigorous testing, the software is released to users.
- Most critical bugs are resolved, but minor or unforeseen issues may still emerge.
- Reliability improves as updates and patches are released.

Useful Life Phase (Stable Period)

- The software operates reliably with minimal failures.
- Bugs and issues are resolved through patches, updates, and maintenance.
- The failure rate remains low and stable, similar to the useful life phase of hardware.

Retirement Phase (End of Software Life)

- The software becomes outdated due to:
 - New technologies and security concerns.
 - Better alternatives (newer software versions).
 - Lack of support from developers.
- The failure rate may slightly increase as updates and security patches are discontinued.
- Eventually, the software becomes obsolete.

Causes of Software Failures

- Bugs and Errors → Programming mistakes or logic errors.
- Inadequate Testing → Software not thoroughly tested for edge cases.
- Complexity → More features increase the chances of defects.
- External Dependencies → Third-party libraries may introduce issues.

Improving Software Reliability

- ✓ Code Reviews and Testing → Perform unit testing, integration testing, and debugging.
- ✓ Error Handling → Use proper exception handling to prevent crashes.
- ✓ Version Control → Maintain stable software versions with updates.
- ✓ User Feedback and Updates → Continuously fix issues based on reports.

FEATURE	HARDWARE RELIABILITY	SOFTWARE RELIABILITY
Nature of Failure	Physical damage or degradation	Logical errors or bugs
Failure Rate Trend	Increases over time (wear and tear)	Remains constant unless fixed
Maintenance	Requires physical repair/replacement	Can be updated or patched
Testing Methods	Stress tests, durability tests	Unit testing, regression testing
MTBF vs. MTTF	Mean Time Between Failures	Mean Time To Failure
Redundancy Use	Extra components for backup	Error handling, backups, and patches
Upgradability	Requires hardware replacement	Software updates and patches

Reliability Metrics

► 1 Mean Time to Failure (MTTF)

Definition: Average time a non-repairable system operates before failing.

Formula:

$$\text{MTTF} = \frac{\text{Total operational time}}{\text{Number of failures}}$$

Example: If a system runs for 500 hours and fails 5 times,
 $\text{MTTF}=500 / 5 = 100 \text{ hours}$

► 2 Mean Time to Repair (MTTR)

- **Definition:** Average time taken to repair a system after failure.

Formula:

$$\text{MTTR} = \frac{\text{Total repair time}}{\text{Number of repairs}}$$

Example: If it takes 20 hours to fix 4 issues,

$$\text{MTTR} = 20 / 4 = 5 \text{ hours}$$

► 3 Mean Time Between Failures (MTBF)

- **Definition:** Average time between consecutive failures in a repairable system.

Formula:

$$\text{MTBF} = \text{MTTF} + \text{Mean Time To Repair (MTTR)}$$

- Example: If MTTF is 100 hours and MTTR is 10 hours,
 $\text{MTBF}=100+10=110 \text{ hours}$

► 4 Failure Rate (λ)

- **Definition:** The number of failures per unit of time.

Formula:

$$\lambda = \frac{1}{MTTF}$$

- Example: If MTTF is 100 hours,

$$\lambda = 1 / 100 = 0.01 \text{ failures per hour}$$

► 5 Availability (A)

- **Definition:** Probability that a system is operational at any given time.

Formula:

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

- Example: If MTBF = 110 hours and MTTR = 10 hours,

$$A = 110 / (110 + 10) = 110 / 120 = 0.9167 (\approx 91.67\%)$$

► 6 Probability of Failure-Free Operation (P)

- **Definition:** Probability that the system operates without failure over time t .

Formula:

$$P(t) = e^{-\lambda t}$$

- Example: If $\lambda = 0.01$ and time = 50 hours,

$$P(50) = e^{-0.01 \times 50} \approx 0.6065$$

► 7 Downtime

- **Definition:** Downtime is the total time a system is non-operational or unavailable due to failures, maintenance, or other interruptions.

Formula:

$$\text{Downtime} = \text{Total Time} \times (1 - \text{Availability})$$

► Where:

- **Availability** is typically expressed as a decimal (e.g., 0.95 for 95%).
- **Total Time** is the time frame considered (e.g., total hours in a year).

► 8 Uptime

- **Definition:** Uptime refers to the total time a system is operational and available without failure.

Formula:

$$\text{Uptime} = \text{Total Time} - \text{Downtime}$$

► Where:

- **Total Time** is the entire period considered (e.g., hours in a day, month, or year).
- **Downtime** is the time when the system was non-operational

► Where Are Downtime and Uptime Used?

- 1. Availability Calculation:** Measures system reliability and operational performance (used in servers, networks, and cloud services).

$$\text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}$$

- 2. Mean Time Between Failure:** Predicts when a system might fail next and helps in maintenance scheduling.

$$\text{MTBF} = \text{Uptime} + \text{Downtime}$$

- 3. Mean Time To Repair:** Assesses how quickly a system can recover after a failure—essential for service level agreements (SLAs).

$$\text{MTTR} = \frac{\text{Downtime}}{\text{Number of Failures}}$$

Practice questions on reliability metrics

► **Check a Separate doc file.**

Error and Bug

🔍 Error

Definition: An error is a human mistake made by the developer while writing the code, designing the system, or configuring the environment.

- ▶ When It Happens: During the development phase.
- ▶ Example: A developer accidentally writes $x = y + z$ instead of $x = y - z$.
- ▶ Misconfiguring a database connection string.



Definition: A bug is a flaw or fault in the software that causes it to behave unexpectedly or incorrectly.

- ▶ When It Happens: Detected during testing or when the software runs.
- ▶ Example: The program crashes when a user clicks a button.
- ▶ A feature doesn't work as intended because of a coding error.

Feature	Error	Bug
What it is	Human mistake	Result of that mistake
Detected	During development	During testing or execution
Cause	Coding, logic, or design flaw	Code not working as expected
Example	Typing wrong logic in code	App crashes due to that logic

Fault and Failure

⌚ A **fault** (or defect/bug) is an underlying flaw or error in a system's code, design, or architecture. A fault may or may not lead to a failure.

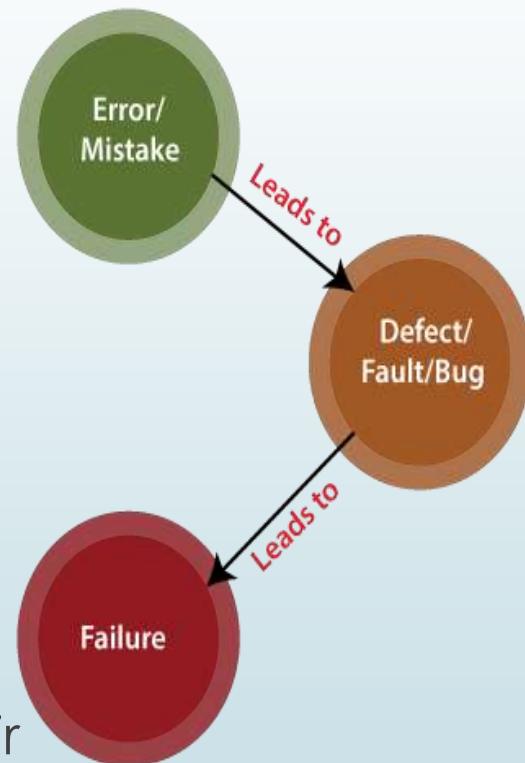
► **Example:**

If a developer writes incorrect logic for a login validation, that's a fault. However, users might not notice it until they input specific incorrect data.

✗ **Failure:** It occurs when a system doesn't perform as expected due to one or more faults being triggered during execution.

► **Example:**

When the faulty login validation prevents users from accessing their accounts even after entering the correct credentials, that's a failure.



✓ 10 Fault Leading to Failure

When a fault gets triggered during execution, causing the system to behave unexpectedly or fail to meet requirements.

Examples:

🔑 Login System Bug

Fault: A developer writes the condition `if (username == storedUsername)` without ignoring case sensitivity.

Failure: The system rejects a valid username if the user enters "JohnDoe" instead of "johndoe" when it was meant to be case-insensitive for convenience.

➊ Division by Zero in Calculator App

Fault: Missing a check for division by zero in the code.

Failure: When a user tries to divide a number by zero, the program crashes unexpectedly.

🚗 Automotive Cruise Control System

Fault: A sensor provides incorrect speed data due to a miscalibration.

Failure: The cruise control accelerates unexpectedly, causing a safety issue.

X 2 Fault Not Leading to Failure

The fault exists in the system but doesn't cause a failure because it's never triggered or has no impact on execution.

Examples:

Dead Code in Calendar Application

Fault: A feature for event notifications has been written but never integrated into the final app flow.

No Failure: The notification feature code is never executed, so users don't experience any issue.

Memory Leak in an Unused Function

Fault: A memory leak exists in a function that allocates memory but doesn't free it properly.

No Failure: Since the function is never called during execution, the memory leak doesn't affect the system.

Incorrect Error Message in Network System

Fault: A wrong error message is set to display when the internet is disconnected.

No Failure: If the network connection remains stable, this fault never triggers and the users don't encounter it.

Fault Prevention

- ▶ Fault Prevention/Avoidance strategies identify all potential areas where a fault can occur and close the gaps.
- ▶ These prevention strategies address system requirements and specifications, software design methods, re-usability, or formal methods.

Techniques:

- Well-defined requirements and design specifications
- Use of coding standards and best practices
- Regular team code reviews and pair programming
- Developer training and upskilling

Example:

Following strict coding guidelines and conducting frequent peer reviews to ensure logical consistency can help prevent faults from occurring in a banking application.

- ▶ But it is difficult to quantify the impact of fault avoidance strategies on system dependability. So, despite fault prevention efforts, faults are created, so fault removal is needed.

Fault Removal

- ▶ Fault removal strategies are dependability-enhancing techniques employed during verification and validation.
- ▶ They improve by detecting existing faults and eliminating the defected faults.
- ▶ They are employed after the development phase of the software to contribute to the validation of the software.

Techniques:

- Static code analysis (detects syntax and structural issues)
- Code inspections and peer reviews
- Comprehensive testing (unit, integration, system testing)
- Debugging during development

▶ Example:

A software team running unit tests to catch and fix off-by-one errors before releasing a file-processing tool.

- ▶ The difficulties encountered in testing programs are often related to the prohibitive costs and exhaustive testing. Therefore, fault removal is imperfect, hence fault tolerance is needed.

Fault Tolerance

- ▶ Fault tolerance includes dependability-enhancing techniques that are used during the validation of software to estimate the presence of faults.
- ▶ It is used to reduce system design faults and enhance the reliability of the software.

Techniques:

- Redundancy (backup systems or duplicate components)
- Exception handling and graceful degradation
- Recovery blocks and failover mechanisms
- Checkpointing for system recovery

Example:

In an e-commerce platform, if a payment gateway fails, the system automatically switches to a backup payment processor to complete transactions without affecting the user experience.

- ▶ Fault tolerance techniques are employed during the development phase of the software which enables the system to tolerate faults remaining in the system after its development and provide operation complying with the requirements specification in spite of faults.
- ▶ Therefore, when a fault occurs it prevents the system failure.

Fault Forecast

- ▶ Predict the occurrence and impact of potential faults to prepare for or prevent failures.

Techniques:

- Reliability modeling and simulations
- Statistical analysis of failure data
- Risk assessment and management
- Predictive analytics

Example:

Using historical server downtime data to predict potential system failures and schedule preventive maintenance for a cloud storage service.

Aspect	Goal	Techniques	Example
Fault Prevention	Avoid introducing faults	Coding standards, peer reviews	Prevent logic errors in financial software
Fault Removal	Detect and eliminate faults	Testing, debugging, code inspections	Fix memory leaks found during testing
Fault Tolerance	Ensure system runs despite faults	Redundancy, failover systems, exception handling	Backup server activation during main server failure
Fault Forecast	Predict and minimize the impact of faults	Risk assessment, reliability models	Predict server failures using historical downtime data

Techniques for detecting software faults

There are several methods used to identify and resolve faults in software engineering, including:

- ▶ **Code Reviews:** A code review is a process in which other developers or team members review the code written by a developer to identify potential errors or areas for improvement. This can be done manually or with automated tools.
- ▶ **Testing:** Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. There are several types of testing, such as unit testing, integration testing, and acceptance testing, which can help identify faults in the software.

- ▶ **Debugging:** Debugging is the process of identifying and resolving faults in the software by analyzing the program's source code, data, and execution. Debugging tools, such as debuggers, can help developers identify the source of a fault and trace it through the code.
- ▶ **Monitoring:** Monitoring is the ongoing process of tracking and analyzing the performance and behavior of a system. Monitoring tools, such as log analyzers, can help identify and diagnose faults in production systems.
- ▶ **Root Cause Analysis:** Root cause analysis is a method used to identify the underlying cause of a fault, rather than just addressing its symptoms. This can help prevent the same fault from occurring in the future.

Techniques for mitigating software faults

- ▶ **Code Reviews:** Code review is a process where code is evaluated by peers to identify potential problems and suggest improvements.
- ▶ **Unit Testing:** Unit testing involves writing automated tests for individual units of code to ensure that each component works as expected.
- ▶ **Test-Driven Development:** Test-driven development (TDD) is a software development process where unit tests are written before writing the code, ensuring that the code meets the requirements.
- ▶ **Continuous Integration and Continuous Deployment (CI/CD):** CI/CD is a software engineering practice where code changes are automatically built, tested, and deployed to production.
- ▶ **Static Code Analysis:** Static code analysis is the process of automatically analyzing code to find potential problems without actually executing it.
- ▶ **Design and Architecture Reviews:** Reviews of design and architecture help ensure that the overall design of the software is correct and scalable.
- ▶ **Error Handling:** Proper error handling helps prevent the spread of faults and ensures that the software can handle unexpected situations gracefully.
- ▶ **Documentation:** Clear and up-to-date documentation can help reduce the likelihood of errors by providing a clear understanding of the software's functionality

Dependability Concept

- Dependability in software engineering refers to the ability of a system to deliver reliable and correct services while handling faults and failures effectively.
- It ensures that the system remains available, secure, and maintains integrity even under failures or external threats.

Key Attributes of Dependability

- 1 **Reliability** – The system operates correctly over a given period.
- 2 **Availability** – The system is operational and accessible when needed.
- 3 **Safety** – The system does not cause unacceptable risk.
- 4 **Integrity** – The system prevents unauthorized data modification.
- 5 **Maintainability** – The system can be updated, repaired, or modified efficiently.

Failure Behavior

Failure behavior describes how a system responds when faults occur. Failures can be classified into different types based on their occurrence and impact:

➤ Types of Failures:

1. **Transient Failures** – Temporary failures that disappear after some time (e.g., network glitches).
2. **Intermittent Failures** – Failures that occur irregularly and unpredictably (e.g., hardware overheating).
3. **Permanent Failures** – Persistent failures that require intervention (e.g., corrupted database).

➤ Failure Models:

- **Fail-Safe** – The system enters a safe state when a failure occurs (e.g., circuit breakers in power grids).
- **Fail-Soft** – The system continues operation at a reduced level (e.g., battery-saving mode in devices).
- **Fail-Stop** – The system halts completely upon failure detection (e.g., emergency shutdown in industrial systems).

◆ Characteristics of Dependability

1. Reliability
2. Availability
3. Safety
4. Security
5. Maintainability
6. Integrity
7. Fault Prevention, Removal, Tolerance, Forecast
8. Recoverability
9. Consistency

🛠 Example Scenarios

◆ Case 1: Banking System Failure Prevention

A banking app prevents transaction failures by **verifying user authentication** before processing payments.

If the system detects an error, it **rolls back** the transaction instead of completing a faulty transaction.

◆ Case 2: Flight Control System Failure Tolerance

A flight control system uses redundant processors. If one processor fails, the backup system takes over, ensuring safe operations.

◆ Case 3: E-commerce Website Failure Forecasting

An online store predicts high traffic during sales and **scales servers in advance** to prevent downtime.

◆ Maintenance Policy in Dependability

Maintenance policies define how a system is maintained to ensure dependability. There are different approaches to maintenance:

📌 1. Corrective Maintenance

Fixing faults after they occur.

Example: Patching security vulnerabilities in an application.

📌 2. Preventive Maintenance

Regular updates and checks to prevent future failures.

Example: Scheduled software updates and database optimizations.

📌 3. Predictive Maintenance

Using monitoring tools to predict and address potential failures.

Example: AI-based failure detection in cloud infrastructure.

📌 4. Adaptive Maintenance

Modifying a system to accommodate changes in the environment.

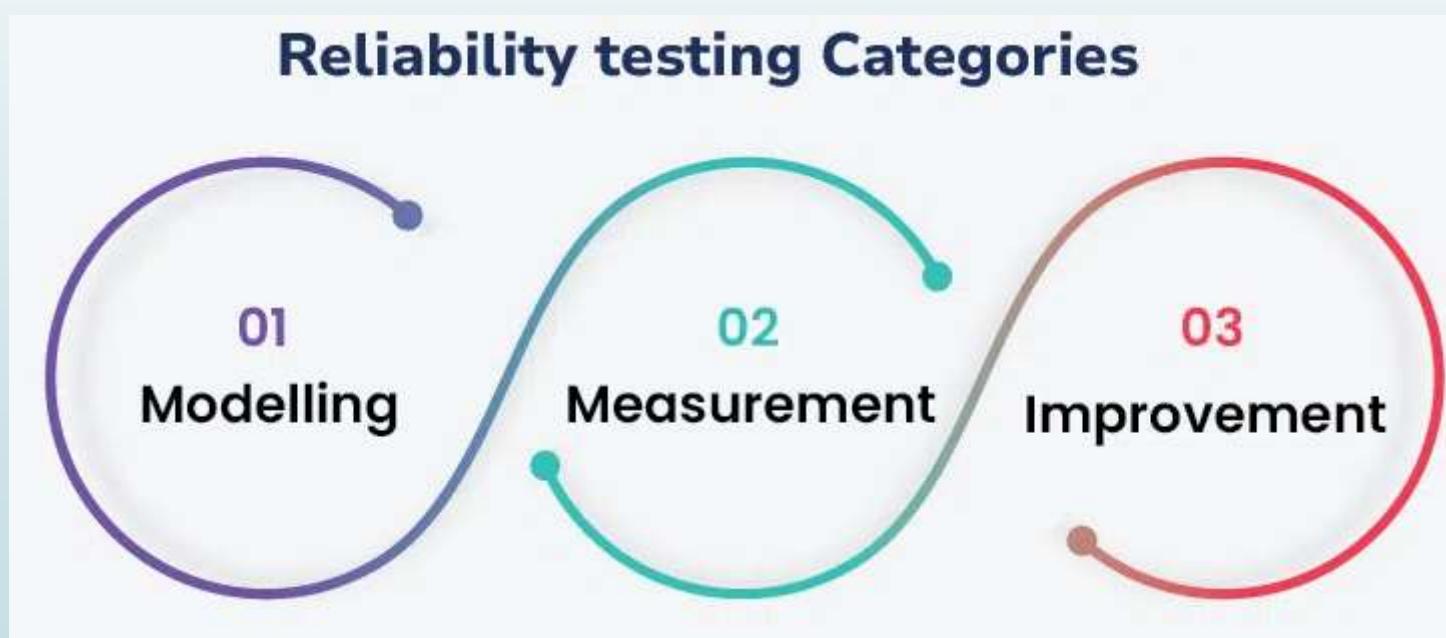
Example: Upgrading software to support new operating systems.

Reliability and Availability Modeling

- Reliability and Availability are two critical attributes of a system, particularly in software engineering, where ensuring a system's ability to function consistently and be accessible is essential for user satisfaction and business continuity.
- Modeling these attributes helps in designing and analyzing systems to ensure they meet the required performance standards.

Reliability Evaluation Testing methods

- Reliability testing is a type of software testing that evaluates the ability of a system to perform its intended function consistently and without failure over an extended period.



1. Modelling

Modelling in reliability testing involves **creating mathematical or statistical representations of how a product or system might fail over time**. It's like making an educated guess about the product's lifespan based on its design and components. This helps predict when and how failures might occur without actually waiting for the product to fail in real life.

Example: Engineers might create a model to estimate how long a new smartphone battery will last before it degrades significantly.

43

2. Measurement

Measurement focuses on collecting real-world data about a product's performance and failures. This involves testing products under various conditions and recording when and how they fail. It's about gathering concrete evidence of reliability rather than just predictions.

Example: A car manufacturer might test drive hundreds of cars for thousands of miles, recording any issues that arise during these tests.

3. Improvement

Improvement uses the insights gained from modelling and measurement to enhance the reliability of a product or system. This involves identifying weak points, redesigning components, or changing manufacturing processes to make the product more reliable.

Example: After finding that a particular part in a washing machine fails frequently, engineers might redesign that part or choose a more durable material to improve its lifespan.

45

Different Ways/Methods to Perform Reliability Testing



46

- **Stress testing:** Stress testing involves subjecting the system to **high levels of load** or usage to identify performance bottlenecks or issues that can cause the system to fail. Example: Testing a ride-booking app (Uber/Ola) during peak hours.
- **Endurance testing:** Endurance testing involves **running the system continuously for an extended period** runs the system continuously for long periods to check for memory leaks, slowdowns, or performance degradation. Example: A video streaming app (Netflix) runs non-stop for 48 hours to check if memory usage increases over time.
- **Recovery testing:** Recovery testing tests the system's ability to **recover from crashes, failures, or network disconnections**. Example: A user is making a money transfer, and the system crashes mid-transaction. The testing checks if the system can restore its last correct state without data corruption.

- **Environmental Testing:** Conducting tests on the product or system in various environmental settings, such as temperature shifts, humidity levels, vibration exposure or shock exposure, helps in evaluating its dependability in real-world circumstances. Example: A self-driving car must operate in extreme temperatures (-30°C in Canada, +50°C in deserts).
- **Performance Testing:** Evaluates if the system meets performance criteria under both normal and peak load conditions. Example: The website must handle millions of users during a Big Billion Day Sale without slowing down.
- **Regression Testing:** After every software update, regression testing ensures that previous features still work correctly. Helps detect new bugs introduced by recent changes. Example: Prevents new updates from breaking existing functionalities.
- **Fault Tree Analysis:** Identifies possible failures and examines how different system components contribute to failures. Uses a tree-like diagram to trace failures back to their root causes. Example: If a navigation failure occurs, fault tree analysis helps find the root cause, such as: GPS signal lost, Sensor failure, Software bug in autopilot mode.

Limits of Reliability Testing

Limitation	Explanation	Example
Time Constraints	Some reliability tests require long durations	Testing a space probe's software for a 10-year mission
Cost Constraints	Large-scale tests can be expensive	Cloud companies spend millions on server reliability tests
Unknown Failure Modes	Not all failure scenarios can be predicted	Unexpected cyberattacks affecting system performance
Environmental Variability	External factors change reliability results	Network reliability varies by location and traffic

Starvation in Reliability Evaluation

Starvation occurs when certain processes or requests are indefinitely delayed because system resources prioritize other operations.

❖ Examples of Starvation in Software Systems

1. A low-priority process may never get CPU time because high-priority processes keep running.
2. A low-priority query may never execute because the database always prioritizes faster queries.
3. In cloud computing, some virtual machines may be starved of resources if others consume too much bandwidth.

◆ How to Prevent Starvation?

- ✓ Priority Aging: Increases the priority of waiting tasks over time.
- ✓ Fair Scheduling: Ensures all requests get equal access to resources.
- ✓ Resource Allocation Strategies: Limits high-priority tasks from consuming all resources.

Coverage in Reliability Testing

Coverage refers to how much of the system is tested for reliability. Higher coverage means more test cases are executed.

📌 Types of Coverage in Reliability Testing:

Type of Coverage	Description	Example
Code Coverage	Ensures all lines of code are tested	Unit testing every function in a Python program
Path Coverage	Tests all possible execution paths	Checking all login scenarios in a banking app
Fault Coverage	Tests different failure cases	Simulating server crashes in cloud computing
Condition Coverage	Checks all if-else conditions	Testing both valid and invalid inputs in forms



Higher coverage ensures fewer undetected failures.

Filtering in Reliability Evaluation

Filtering helps in **removing irrelevant failures or noise** to focus on critical failures.

📌 How Filtering Works?

1. Noise Reduction: Removes false failure reports caused by temporary glitches.
2. Severity-Based Filtering: Focuses only on high-impact failures that disrupt system performance.
3. Duplicate Failure Filtering: Removes duplicate error reports to avoid unnecessary debugging.

🚀 Example: Filtering in Log Analysis

- Web servers (Google, Facebook): If millions of error logs are generated, filtering removes duplicate errors and focuses on critical issues like security breaches.

Reliability Models

- Software reliability models help predict and measure the dependability of software by **analyzing failure behavior, fault rates, and system performance over time.**
- These models are crucial in ensuring software quality, improving maintainability, and reducing system failures.

Some of the Software Reliability Models that are used to achieve reliability in Software Development are:-

- Jelinski and Moranda Model
- Basic Execution Time Model
- Logarithmic Poisson Time Model
- The Bug Seeding Model
- Shooman Model
- Littlewood-Verrall Model
- Goel-Okumoto Model
- Musa-Okumoto Model

Classes of Software Reliability Models

Software reliability models are broadly classified into three main categories:

I. Deterministic Models

- These models assume a fixed number of faults in the software.
- They do not consider randomness in software failures.
- Used primarily for quality assurance and defect estimation.
- Example: Halstead's Software Metric Model (measures code complexity and defect likelihood).

II. Probabilistic Models

- These models assume that software failures occur randomly and follow a probability distribution.
- Used for reliability predictions based on historical failure data.
- Example: Jelinski-Moranda Model (models failures over time using a constant failure rate).

III. Fault-Based Models

- These models focus on fault detection and removal to predict software reliability growth.
- Assumes that debugging reduces failure rates over time.
- Example: Musa-Okumoto Model (models failure intensity and software improvement over time).

Statistical Reliability Models

Statistical models use **failure data** to estimate software reliability. These models fall into two major categories:

I. Time Between Failure (TBF) Models

- Measure time intervals between consecutive failures.
- Assume failure times follow a probability distribution (e.g., exponential or Weibull).
- Example: Musa Basic Model (predicts reliability based on failure times).

II. Failure Count Models

- Focus on the number of failures in a given time period.
- Used when failures occur at discrete time points.
- Example: Goel-Okumoto Model (predicts software failure rates based on observed defect counts).

Reliability Growth Models

The growth model represents the reliability or failure rate of a system as a function of time or the number of test cases.

Models included in this group are as follows:

- ▶ Coutinho Model
- ▶ Wall and Ferguson Model

Coutinho Model

In the log-log paper, Coutinho charted the cumulative number of defects identified and the number of corrective measures taken vs. the cumulative testing weeks. Let $N(t)$ represent the total number of failures and t represent the entire testing duration. The model's failure rate, $\lambda(t)$, may be represented as,

$$\lambda(t) = \frac{N(t)}{t} = \beta_0 t^{-\beta_1}$$

Where,

β_0 and β_1 are model parameters. This model's parameters may be estimated using the least-squares approach.

$N'(t)$ (Failure Rate Function)
 β_0 (Fault Detection Rate Constant)
 $b(t)$ (Fault Detection Function)
 $b'(t)b''(t)b'''(t)$ (Rate of Fault Detection)

57

Wall and Ferguson Model

- The total number of failures at time t , $m(t)$, may be written as,

$$m(t) = a_0 [b(t)]^3$$

a_0 (Initial Number of Faults)
 $b(t)$ (Fault Detection Growth Function)

where α_0 : is unidentified parameter. The number of test cases or total testing time can be used to calculate the function $b(t)$.

- Similarly, at time t , the failure rate function is given by,

$$\lambda(t) = m'(t) = a_0 \beta b'(t) [b(t)]^{\beta-1}$$

a_0 (Initial Failure Rate)
 β (Decay Parameter)

Wall and Ferguson evaluated their model using a variety of software failure data and discovered that the failure data correlated well with the model.

Static Analysis

- ▶ Static analysis involves **no dynamic execution of the software** under test and can detect possible defects in an early stage, before running the program.
- ▶ Static analysis is **done after coding and before executing unit tests**.
- ▶ Static analysis can be done by a machine to automatically “walk through” **the source code and detect noncomplying rules**. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.
- ▶ Static analysis can also be performed by a person who would **review the code to ensure proper coding standards** and conventions are used to construct the program. This is often called Code Review and is done by a peer developer, someone other than the developer who wrote the code.
- ▶ Static analysis is also used to force developers to not use risky or buggy parts of the programming language by setting rules that must not be used.

59

When developers performs code analysis, they usually look for

- Lines of code
- Comment frequency
- Proper nesting
- Number of function calls
- Cyclomatic complexity
- Can also check for unit tests

Quality attributes that can be the focus of static analysis:

- Reliability
- Maintainability
- Testability
- Re-usability
- Portability
- Efficiency

There are several types of static analysis methods-

Control Analysis :-

This software focuses on examining **the controls used in calling structure**, control flow analysis and state transition analysis. The calling structure is related to the model by identifying the calling and call structure. The calling structure can be a **process, subroutine, function, or method**. Control flow analysis checks the sequence of control transfers. Furthermore, it inefficient constructions in the model. A graph of the model is created in which the conditional branches and model junctions are represented by nodes.

Data Analysis :-

Ensures proper operation is applied to data objects such as **data structures and linked lists**. In addition, this method also ensures that the defined data is used properly. Data analysis involves two methods, namely, **data dependency and data-flow analysis**. Data dependency is necessary to assess the accuracy of synchronization across multiple processors. Data flow analysis checks the definition and context of variables.

61

Fault/Failure Analysis :-

It analyzes faults (incorrectly component) and failure (incorrect behavior of model component) in the model. This method uses the input-output transformation description to identify the conditions that **are cause for the failure**. To determine the failures in certain conditions the model design specification is checked.

Interface Analysis :-

This software verifies interactive and distribution simulations to check the code. There are two basic techniques for interface analysis and **user interface analysis** examines sub model interfaces and determines the **accuracy of interface structure**. User interface analysis examines the user interface model and for the precautionary steps taken to prevent errors during the user's interaction with the model. This method also focuses on how accurately the interface is integrated into the overall model and simulation.

Dynamic Analysis

- ▶ In contrast to Static Analysis, where code is not executed, dynamic analysis is based on the system execution, often using tools.
- ▶ Dynamic program analysis is the analysis of computer software that is performed with executing programs built from that software on a real or virtual processor (analysis performed without executing programs is known as static code analysis). Dynamic program analysis tools may require loading of special libraries or even recompilation of program code.
- ▶ The most common dynamic analysis practice is executing Unit Tests against the code to find any errors in code.

Dynamic code analysis advantages:

- ▶ It identifies vulnerabilities in a runtime environment.
- ▶ It allows for analysis of applications in which you do not have access to the actual code.
- ▶ It identifies vulnerabilities that might have been false negatives in the static code analysis.
- ▶ It permits you to validate static code analysis findings.
- ▶ It can be conducted against any application.

Dynamic code analysis limitations:

- ▶ Automated tools provide a false sense of security that everything is being addressed.
- ▶ Cannot guarantee the full test coverage of the source code
- ▶ Automated tools produce false positives and false negatives.
- ▶ Automated tools are only as good as the rules they are using to scan with.
- ▶ It is more difficult to trace the vulnerability back to the exact location in the code, taking longer to fix the problem.

Objectives of Dynamic Testing

- ▶ Find errors and bugs: Through comprehensive testing, find and expose flaws, faults, or defects in the software code and its functionality so that they can be fixed as soon as possible.
- ▶ Verify the behavior of the system: Verify that the software operates as expected and complies with company requirements, industry or regulatory standards, user expectations, and any applicable business regulations.
- ▶ Assessing Performance: To make sure the software satisfies performance requirements, evaluate its performance by monitoring reaction times, throughput, and use of resources under various scenarios.
- ▶ Assure Trustworthiness: Examine the software's dependability by determining how well it performs regularly under typical operating conditions, free of unexpected faults or crashes.
- ▶ Accuracy of Test Data: Verify the precision and consistency of the data handled by the software to guarantee reliable and uniform information handling.
- ▶ Assess Scalability: Examine whether the application can grow to handle more users, workloads, or data volumes without seeing an obvious decline in performance.

66

Dynamic Analysis Methods

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Performance testing
- Security testing



Thank You

Solutions for Practice Questions on Reliability Metrics

1. MTTF Calculation

Question:

A software system has been running for 600 hours and has failed 4 times during this period.

Calculate the Mean Time to Failure (MTTF).

Solution:

The formula for MTTF is:

$$\text{MTTF} = \frac{\text{Total operational time}}{\text{Number of failures}}$$
$$\text{MTTF} = \frac{600}{4} = 150 \text{ hours}$$

Answer: The MTTF is **150 hours**.

2. MTBF and Availability

Question:

A system has an MTTF of 200 hours and an MTTR of 20 hours.

- a) Calculate the **Mean Time Between Failures (MTBF)**.
- b) Find the **availability** of the system.

Solution:

a) MTBF Formula:

$$\text{MTBF} = \frac{\text{MTTF} + \text{MTTR}}{\text{MTTF}}$$
$$\text{MTBF} = \frac{200 + 20}{200} = 220 \text{ hours}$$

b) Availability Formula:

$$A = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$
$$A = \frac{200}{200 + 20} = \frac{200}{220} = 0.9091$$

Convert to percentage:

$$0.9091 \times 100 \approx 90.91\%$$

Answer:

- MTBF = **220 hours**
 - Availability = **91.67%**
-

3. Failure Rate Determination

Question:

If a device has an MTTF of 500 hours, what is its **failure rate**?

Solution:

Failure rate (λ) formula:

$$\lambda = \text{MTTF} / \text{Mean Time to Failure} = \frac{1}{\text{MTTF}} = \frac{1}{500} = 0.002 \text{ failures per hour}$$

Answer: Failure rate is **0.002 failures per hour**.

4. MTTR Calculation

Question:

A machine fails 6 times in a month, and the total repair time is 12 hours. Find the **Mean Time to Repair (MTTR)**.

Solution:

$$\text{MTTR} = \frac{\text{Total repair time}}{\text{Number of repairs}} = \frac{12 \text{ hours}}{6} = 2 \text{ hours}$$

Answer: MTTR is **2 hours**.

5. Probability of Failure-Free Operation

Question:

The failure rate (λ) of a server is 0.002 failures per hour.

What is the probability that the server will operate without failure for 100 hours?

Solution:

$$P(t) = e^{-\lambda t} \\ P(100) = e^{-0.002 \times 100} = e^{-0.2}$$

Using $e^{-0.2} \approx 0.8187 e^{-0.2} \approx 0.8187$

Answer: Probability of failure-free operation is **0.8187** (or **81.87%**).

6. System Downtime Calculation

Question:

A system with an availability of 95% operates continuously for 1 year (365 days).

Calculate the expected **downtime** in hours over the year.

Solution:

Total hours in a year:

$$365 \times 24 = 8760 \text{ hours}$$

Downtime formula:

$$\text{Downtime} = (1 - A) \times \text{Total hours}$$

$$\text{Downtime} = (1 - 0.95) \times 8760 = 0.05 \times 8760 = 438 \text{ hours}$$

Answer: Expected downtime is **438 hours**.

7. Availability with Given Uptime and Downtime

Question:

A system runs for 400 hours and experiences 20 hours of downtime.

What is the system's **availability**?

Solution:

$$A = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}$$

$$A = \frac{400}{400 + 20} = \frac{400}{420} \approx 0.9524$$

Convert to percentage:

$$0.9524 \times 100 = 95.24\%$$

Answer: Availability is **95.24%**.

8. MTBF from Uptime and Downtime

Question:

A device has a total uptime of 800 hours and downtime of 50 hours.

Calculate the **Mean Time Between Failures (MTBF)**.

Solution:

$$MTBF = \text{Uptime} + \text{Downtime}$$

$$MTBF = 800 + 50 = 850 \text{ hours}$$

Answer: MTBF is **850 hours**.

9. Failure Rate and MTTF

Question:

A software product has a failure rate of 0.005 failures per hour.

Find the **Mean Time to Failure (MTTF)**.

Solution:

$$MTTF = \frac{1}{\lambda} = \frac{1}{0.005} = 200 \text{ hours}$$

Answer: MTTF is **200 hours**.

10. Comparing Two Systems Based on Availability

Question:

- System A: MTBF = 150 hours, MTTR = 10 hours
- System B: MTBF = 200 hours, MTTR = 20 hours

Which system is more available?

Solution:

Availability Formula:

$$A = \frac{MTBF}{MTBF + MTTR} = \frac{MTBF}{MTBF + MTTR}$$

For System A:

$$A = \frac{150}{150 + 10} = \frac{150}{160} = 0.9375 = 93.75\%$$

For System B:

$$A = \frac{200}{200+20} = \frac{200}{220} = 0.9091 \quad (90.91\%)$$

Answer: **System A** is more available with **93.75%** availability.
