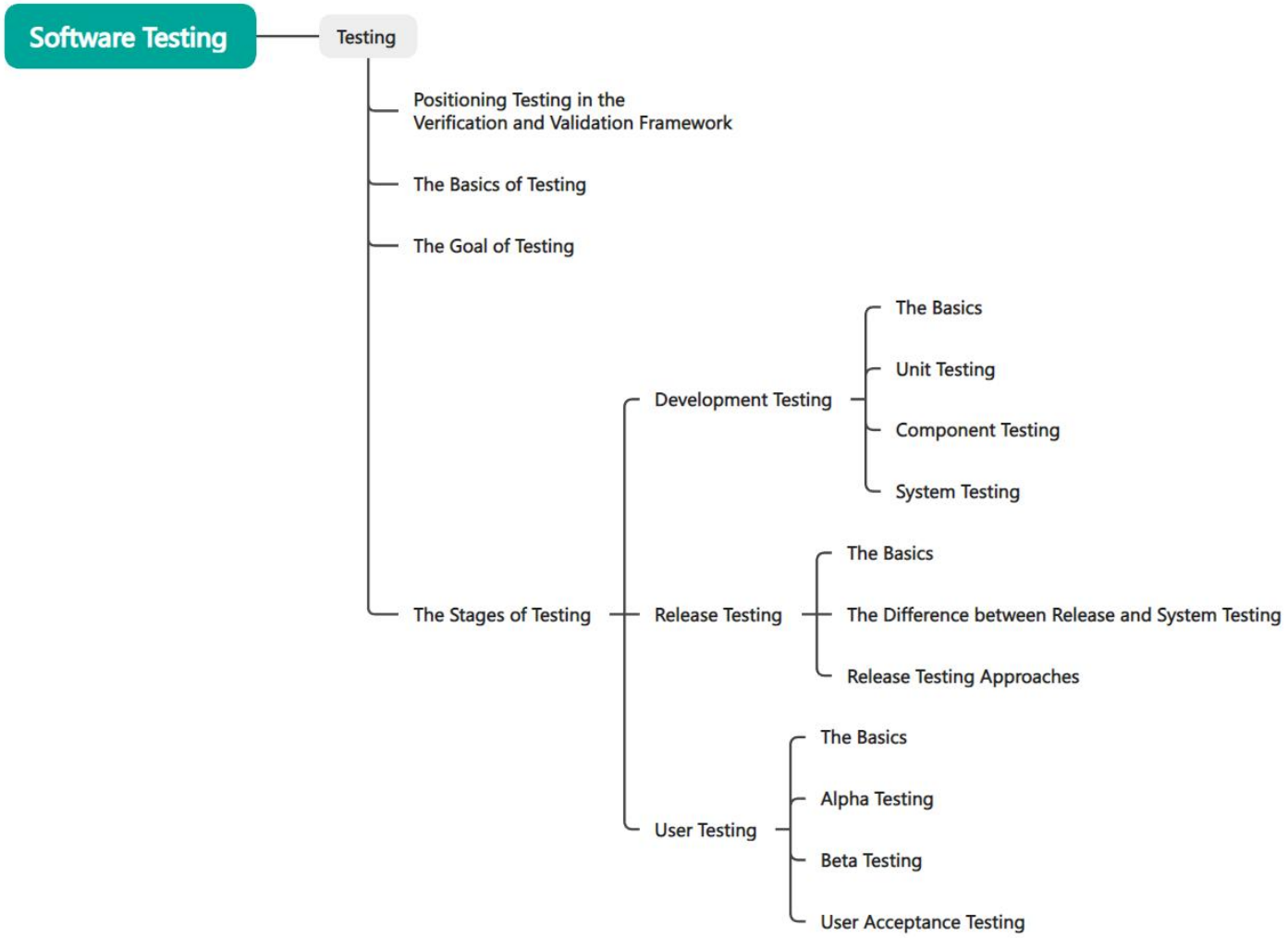
Software Engineering

Week 10 Software Testing - P1

AY 25/26

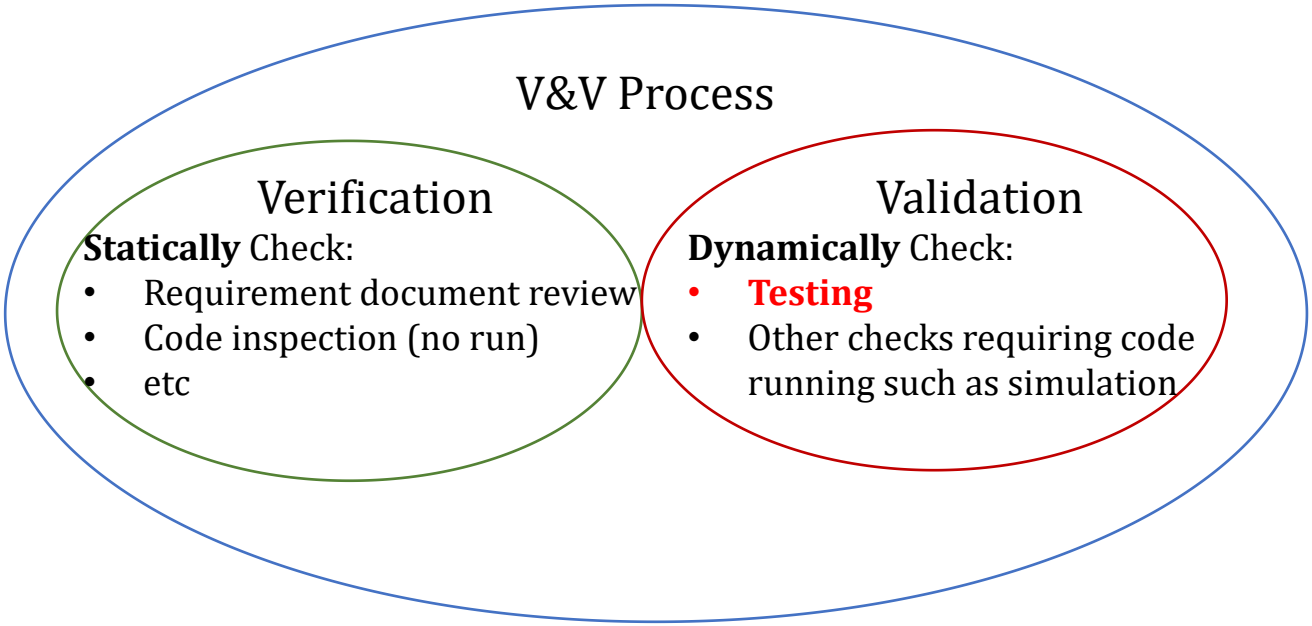
Week 10

Topics covered



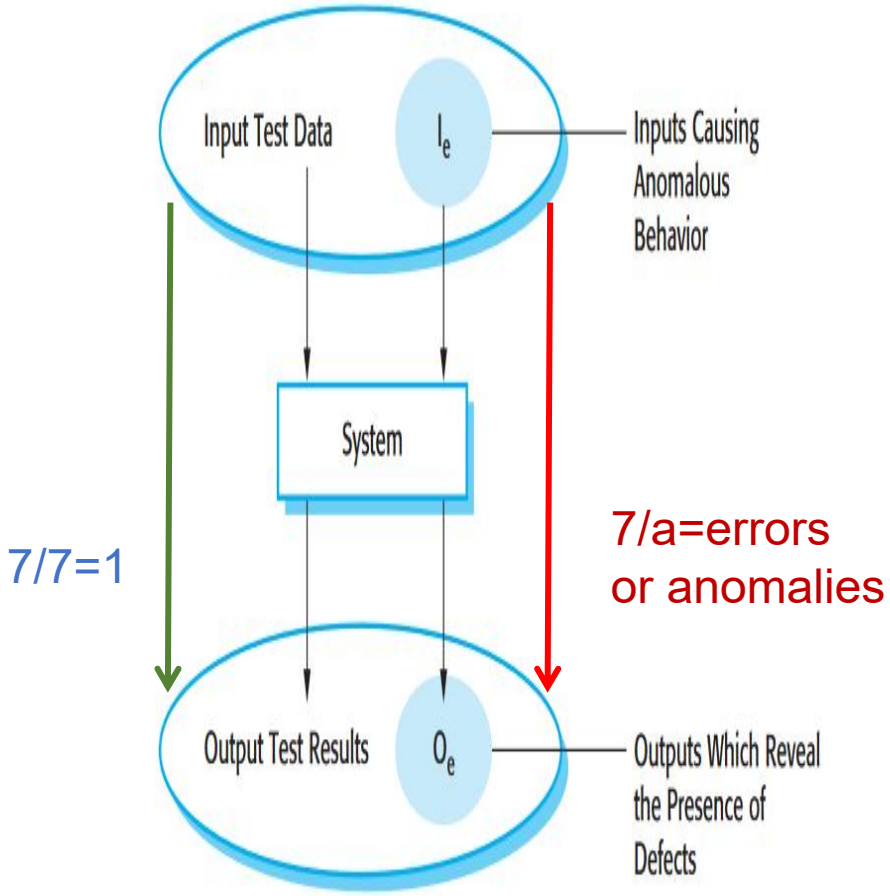
1.1 Testing - A Note

- From a general perspective, testing is part of a broader process of software **Verification** and **Validation** (V & V)
  - Verification checks that software being developed meets its specification
  - Validation checks delivers the functionality expected by the people paying for the software.



1.2 Testing - The Basics

- We use testing to check a program (1) does what it is intended to do and to (2) discover program defects before it is put into use.
- When you test software, you execute a program using **artificial data (data we made on our own)**.
- You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.



1.3 Testing - 2 Goals

On the one hand, to demonstrate to the developer and the customer that the software functionally works as expected.

- Prove the software can do what it should do.

On the other hand, to discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification, for finding defects.

- So that we can solve the undesirable system behavior, such as:
  - system crashes,
  - unwanted interactions with other systems,
  - incorrect computations and data corruption.

1.4 Stages of testing

Development testing, where the system is tested during development to discover bugs and defects.

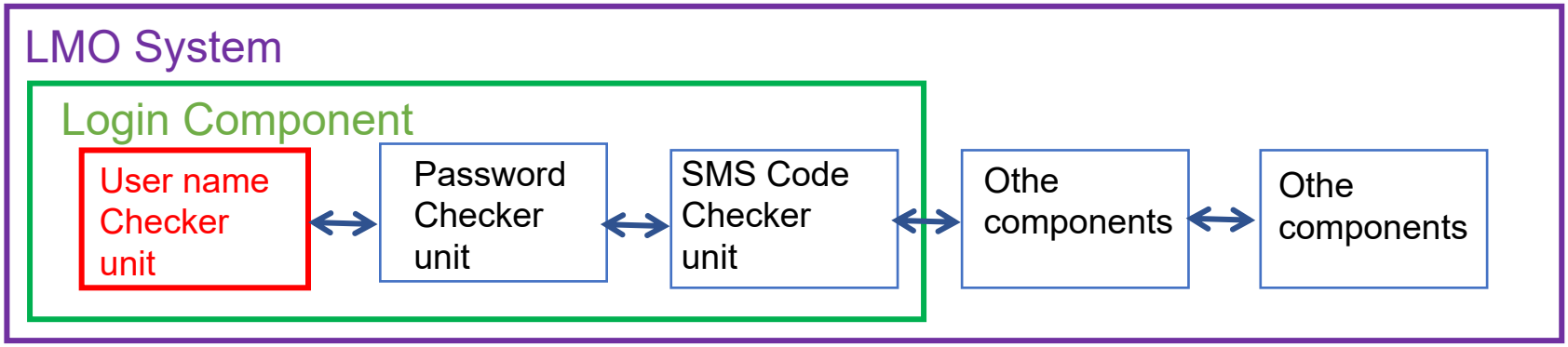
Release testing, where a separate testing team test a complete version of the system before it is released to users.

User testing, where users or potential users of a system test the system in their own environment.

2. Development Testing - The Basics

From small to large, the Development Testing tests at 3 levels:

- **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
- **Component testing**, where several individual units are integrated to create composite components.
- **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.



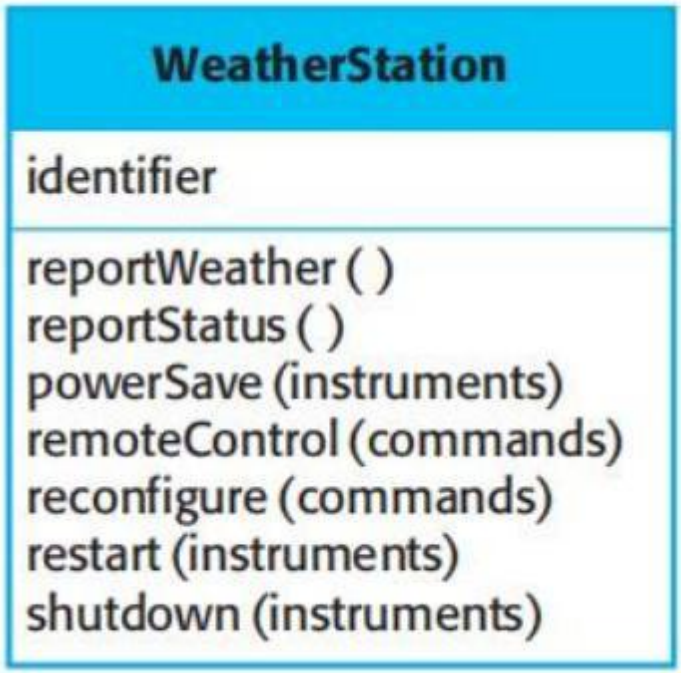
2.1 Unit testing

- Unit testing is the process of testing individual **unit** in isolation.

- Units can be defined at different levels (e.g., a function / method), while, in this class, it is considered to be object classes with several attributes and methods.

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

2.1.1a Unit testing - The weather station example

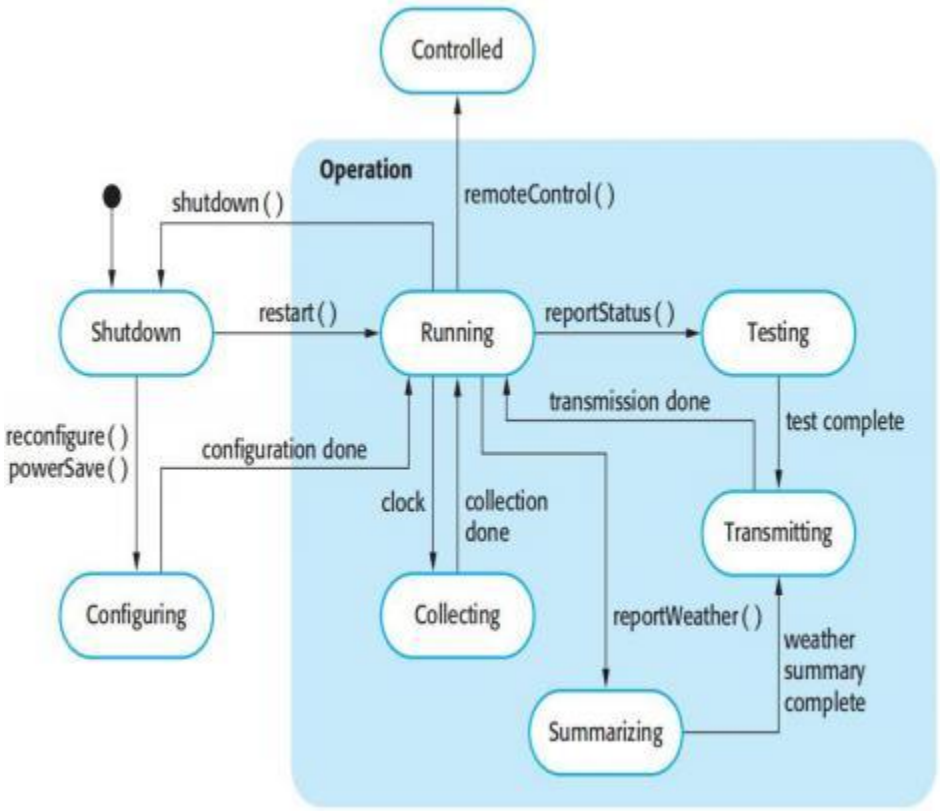


- **Setting and interrogating attributes:** This Object class has a single attribute, which is its identifier. Only need a test that checks if it has been properly set up.

- **Testing all operations:** Define test cases for all of the methods associated with the object such as reportWeather, reportStatus, etc.

Ideally, you should test each methods in isolation but, in some cases, some test sequences are necessary (e.g., restart/shutdown)

2.1.1b Unit testing - The weather station example

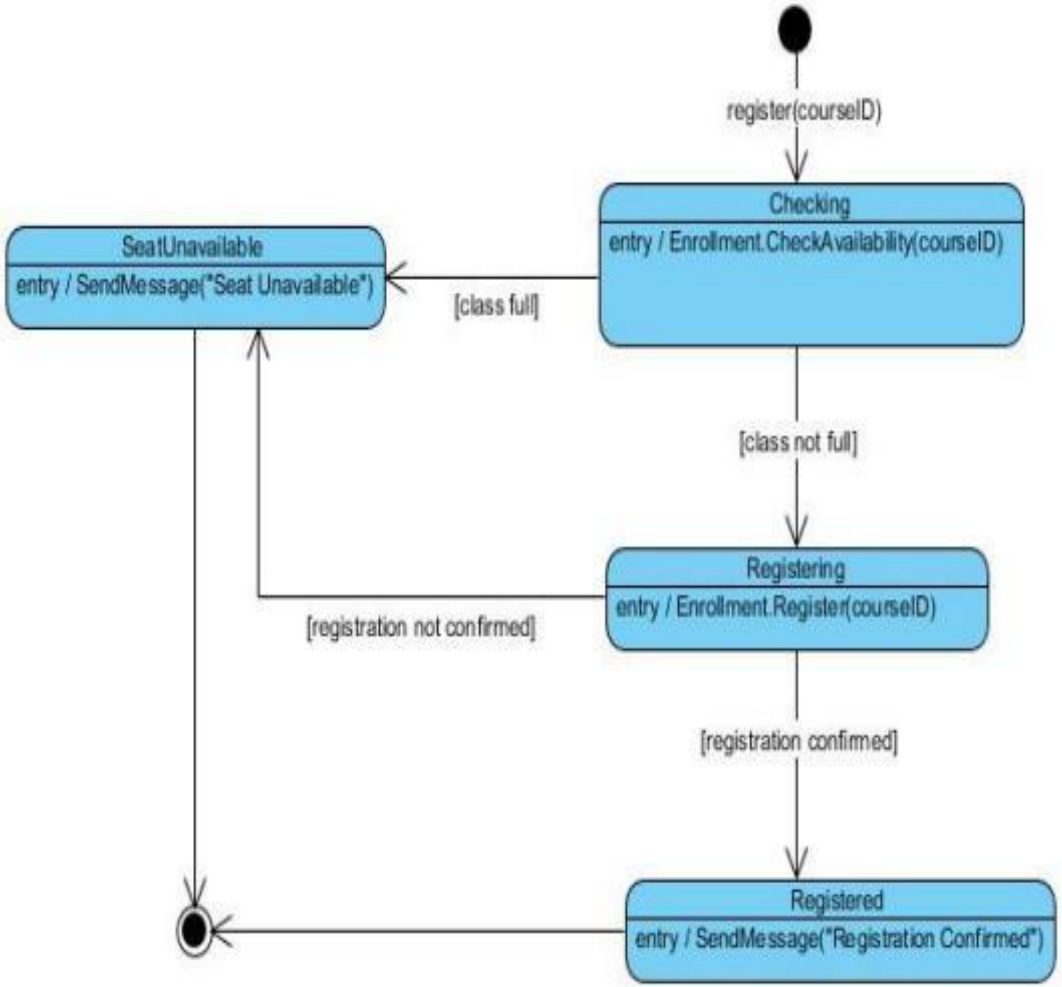


- **Exercising the object in all possible states:** Using a state model, identify flow of state transitions to be tested and the event sequences to cause these transitions

For example:

- Shutdown -> Running-> Shutdown
- Configuring-> Running-> Testing -> Transmitting -> Running
- Running-> Collecting-> Running-> Summarizing ->Transmitting -> Running
- and so on, till the tested flows cover all the possible state

2.1.2a Unit testing - Another Example



Flow of state 1: Start > Checking > SeatUnavailable > Terminate

Flow of state 2: Start > Checking > Registering > Registered >Terminate

Flow of state 3: Start > Checking > Registering > SeatUnavailable > Terminate

By testing all these 3 flows, we exercise the object in ***all possible states***

2.1.3 Automated Unit Testing

Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.

A setup part, where you initialize the system with the test case (e.g., initialize the object under test)

A call part, where you call the object or method to be tested.

An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

```
12 import static org.junit.Assert.*;
13 import org.junit.Before;
14 import org.junit.Test;
15
16 public class CalculatorTest {
17
18     private Calculator calculator;
19
20     // Setup part
21     public void setUp() {
22         calculator = new Calculator();
23     }
24
25     // Test for the add method
26     @Test
27     public void testAdd() {
28         // Call part
29         int result = calculator.add(5, 3);
30
31         // Assertion part
32         assertEquals("5 + 3 must be 8", 8, result);
33     }
34 }
```

2.1.4 Choosing A Unit Test Case

- Testing is expensive and time-consuming, so it is important to design effective unit test cases.
- Effective tests should verify (1) that the methods/functions/system unde test behave correctly under normal, expected usage, and (2) should also reveal any defects when abnormal or problematic inputs are used.
- Therefore, unit test cases should include **both tests** that reflect normal operation and tests that use unusual or abnormal inputs to ensure the methods/functions/system unde test handle these input properly without failing or crashed.
- But how should we choose among so many possible test cases/inputs?

2.1.5a Partition Strategy

```
1   int classifyAge(int age) {  no usages
2       if (age < 0) return ERROR;
3       else if (age < 18) return CHILD;
4       else return ADULT;
5   }
```

Strategy 1: Partition testing, where we:

(1) identify possible groups of inputs that should be equivalent partitions, that is the group member (the detailed values) in these groups should have common characteristics and should be processed in the same way by the system;

(2) we can choose representative cases from within each of these groups.

Equivalent Partition:

- **Negative ages** -> all members are negative integer (same characteristic) and will result in ERROR (processed in the same way by the system)
- **Age between 0 and 17** -> all members are integers between 0 and 17, resulting in CHILD
- **Age ≥ 18** -> all members are integers ≥ 18, resulting in ADULT

So we get 3 equivalent partitions:

...,-3,-2,-1	0,1,2,...,17	18,19,20,...

Because they are of equivalent (same charactertics, and same way to be processed), we can simply choose examples from each of the partitions, instead using all the possible input values, so we can test the following:

- -2, 1, 20
- And 0, 18 (the boundaries between partition)

2.1.5b Partition Strategy

Exercise:

There is a program that **accepts 4 to 10 inputs** which are **five-digit integers ranging from 10,000 to 99,999.**

How do we do the partition testing?

Hints

- You identify partitions by using the program specification or user documentation (in this case, the description of the system) and from experience where you predict the classes of input value that are likely to detect errors.
- Select example inputs from the identified partitions
- Do not forget about the boundaries of the partitions

2.1.6 Guideline Strategy

- Strategy 2: Guideline-based testing, where you use testing guidelines to choose test cases.
- These guidelines reflect **previous experience** of the kinds of errors that programmers often make when developing component.

- Test with empty input (e.g., empty list/file, “”, missing from field, etc)
- Test with repeated or duplicated values
- Test with invalid format (e.g., abc@@xjtlu.edu.cn)
- Test with extrem values (e.g., age=10000000)
- Test real-world ‘dirty inputs’ (e.g., emoji, mixed language --- the UTF-8 issue)

2.2 Component testing

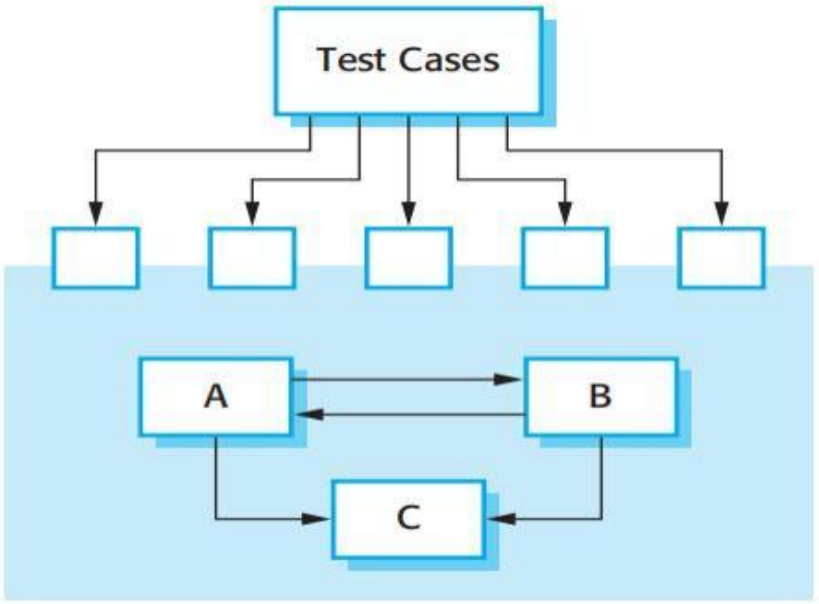
Software components are often composite components that are made up of **several interacting units**.

The functionality of the component is based on the these units interacting through interface.

Testing composite components should therefore focus on showing that the component interface behaves according to its specification.

- You can assume that unit tests on the individual objects within the component have been completed.

2.2.1 Testing Interface in the Component



- Unit A, B, and C have been integrated to create a larger component or subsystem.
- The test cases are not applied to the individual units but rather to the interface of the composite component created by combining these components.

Why not testing every single unit involved: *Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component*

2.2.2 Types of Interface Errors

Interface misuse

- Wrong parameter order, wrong data type, etc.
  - *Expect: transfer(fromAccount, toAccount, amount)*
  - *Get: transfer(toAccount, fromAccount, amount)*

Interface misunderstanding

- The caller makes incorrect assumptions about how the interface behaves.
  - Caller assumes: *getUser(id)* never returns null
  - But the interface does return null when the user does not exist, leading to null pointer crash

Timing errors

- The caller and the function being called operate at different speeds or out of sync.
  - Click ‘pay’ button 2 times and 2 orders generated in the system

2.2.3 General Guidelines for Interface Testing

- **Test parameter boundaries and extreme values**
  - Age 9999, file size 100 GB, etc
- **Test with empty or missing values**
  - processCustomerData(null);
- **Test invalid or malformed inputs**
  - "2024-31-02", "02-2024-31"
- **Test under stress and high load**
  - 1000 requests sent at once
- **Test different orderings of component interactions**
  - Call updateProfile() before getProfile()

2.3 System testing

- System testing during development involves integrating components to create a version of the system and then **testing the integrated system**.
- The focus in system testing is testing the **interactions between components.**
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.

2.3.1 System v.s. Component Testing

System testing obviously overlaps with component testing but they are different:

Component testing focuses on verifying that a single component or module works correctly in isolation.

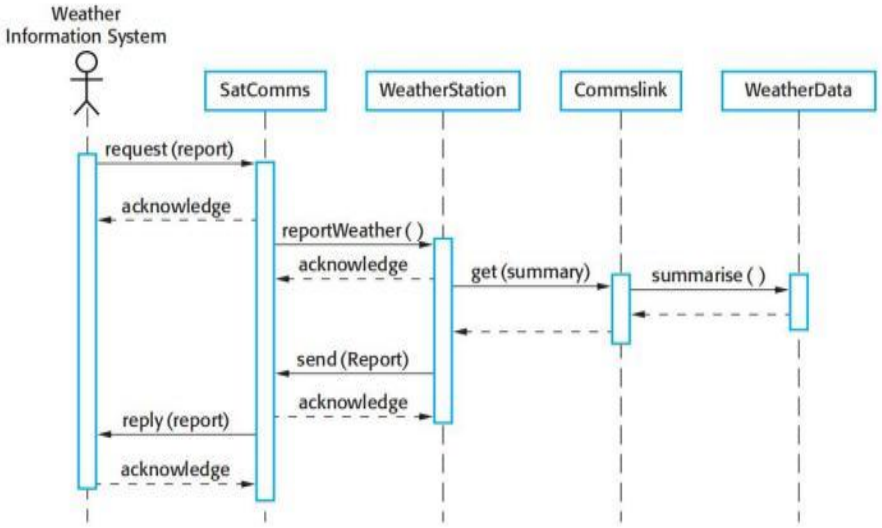
- At this stage, the component is typically tested by the developer who wrote it.
- The main goal is to confirm that the component's internal logic, interfaces, and handling of inputs and outputs are all correct before it is integrated with others.

System testing evaluates the entire integrated system as a complete whole.

- All components, like newly developed modules, reused libraries, third-party services, are combined and tested together.
- It is typically carried out collaboratively among different programmers or by a specific team rather than by individual developers
- It aims to ensure the overall system meets its functional and non-functional requirements

2.3.2 Use case-based Approach

- Because of system testings focus on interactions, use case–based testing is an effective approach to system testing.
- Becasue, typically, each use case is implemented by several components or objects in the system.
- To better identify the interactions involved in different use cases, we can use the sequence diagrams to identify operations that will be tested and to help design the test



SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary) → WeatherData:summarize

Test Case

2.3.3 System Testing Guidelines

- Exhaustive system testing is impossible so we need guidelines for effective test case selection, for example:
  - **Test every major user-visible function**, that is anything the user can click, open, view, or trigger must work (e.g., Menu items, toolbar buttons, page navigation).
  - **Test combinations of functions** that users commonly use together as System-level bugs often appear only when features interact (e.g., Formatting text: Bold + Italic + Underline applied together = ***abc***)
  - **Test end-to-end workflows**, such as from Create Account to Login to Update Profile to Logout.
  - **Test under realistic load and environment conditions**:
    - Many users logged in at once
    - Slow networks

3. Release testing

- Release testing is the process of testing a particular release of a system that is intended for **use outside of the development team.**
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
- Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.

3.1 Release testing v.s System testing

Important differences:

- A separate team that has not been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on discovering technical bugs in the system. The objective of release testing is to check that the system meets its requirements and is good enough for external use .
- System testing focus: Does the entire system **function correctly** as a whole?
- Release testing focus: Is the system **acceptable for real users and real-world use**?

3.2 Requirements Based Approach

Requirements-based approach involves examining each requirement and developing a test or tests for it.

MHC-PMS requirements:

- If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
- If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

Based on the requirement description, we design test cases:

- Set up a patient record with no known allergies.
- Set up a patient record with a known allergy.
- Set up a patient record in which allergies to one or more drugs are recorded.
- Prescribe two drugs that the patient is allergic to.
- Prescribe a drug that issues a warning and overrule that warning

Translate the requirements into the tests

3.3a Scenario Approach

- Scenario approach comes up with typical scenarios of use and use these to develop test cases for the system.
- **A scenario is a story** that describes one way in which the system might be used.
- Scenarios should be realistic and real system users should be able to relate to them.

3.3b Scenario Approach

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.

3.4a Testing Performance Approach

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.

- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

3.4b Testing Performance - Use Operational Profile

To conduct performance test, an operational profile should be constructed. This is a set of tests that reflect the actual mix of work that will be handled by the system in the **real world scenarios**:

Suppose we are performance-testing a food delivery system, of which the real user behaviors like:

80% browse restaurants
15% place orders
4% add items to cart
1% write reviews

The operational profile (suppose 1000 tests contained) should be:

800 tests → browse restaurants
150 tests → place orders
40 tests → add to cart
10 tests → write reviews

We design an operational profile so performance tests accurately reflect real-world usage, allowing us to ***detect true bottlenecks, predict real capacity, and ensure the system performs well for actual users***

3.4c Testing Performance - Stress Testing

Stress testing

- Stressing the system by **making demands that are outside the design limits** of the software.
- Example: testing a transaction processing system that is designed to process up to 300 transactions per second.

  → Test with fewer 300 transactions

  → Increase the load on the system (like 400, 500, 1000...) until it fails

  It is important because:

  - It tests the failure behavior of the system.
  - It stresses the system and may cause defects to come to light that would not normally be discovered.

4. User testing

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.

- User testing is essential, even when comprehensive system and release testing have been carried out.

- User test is important because the influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a (simulated) testing environment.

4.1 Types of user testing

Alpha testing

- Users of the software work with the development team to test the software at the developer's site.

Beta testing

- A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

Acceptance testing

- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

4.2 Alpha Testing

- In alpha testing, **users and developers work together** to test a system as it is being developed, becasue users can identify problems and issues that are not readily apparent to the development testing team

- **Testers** may be willing to get involved in the alpha testing process because this gives them early information about new system features that they can exploit.

- **Benefits:**
  - Early Detection of Issues: Problems that are subtle and context-specific are identified early by real users, preventing costly fixes post-launch.
  - Early Adopter Engagement: Users involved in the alpha testing feel a sense of ownership and engagement with the product, making them likely support the product when it is on the market

4.3 Beta Testing

- Beta testing takes place when **an early, sometimes unfinished, release** of a software system is made available to customers and users for evaluation. Beta testing is essential to discover interaction problems between the software and features of the environment where it is used. Also a form of markerting.

- Beta testers may be a **selected group of customers** who are early adopters of the system. Alternatively, the software may be made publicly available for use by **anyone who is interested in it.**

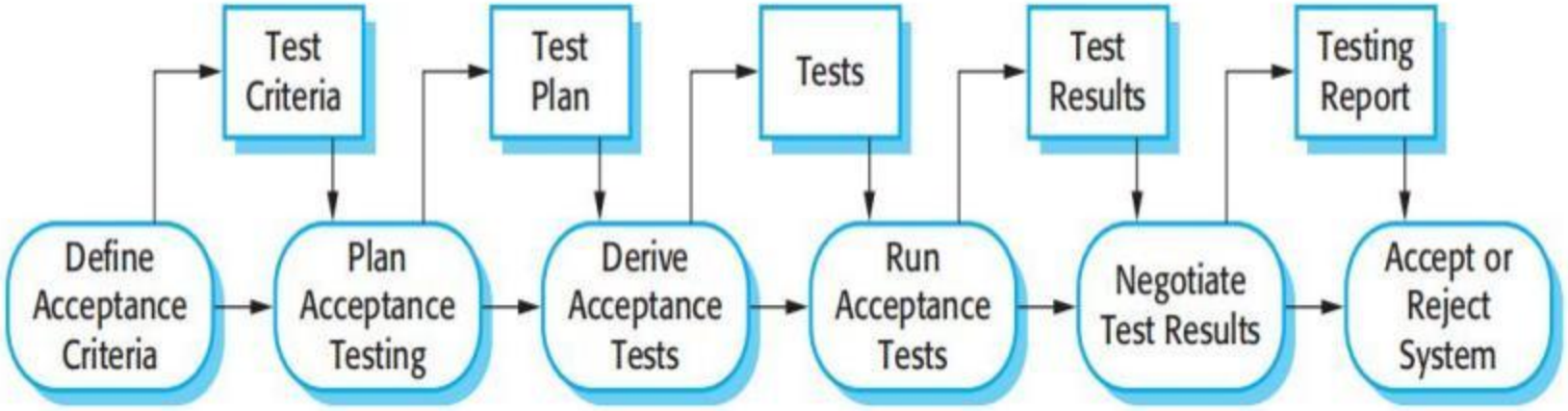
- **Benefits**
  - Usability Feedback: It allows the developers to see how real users interact with the software, which can highlight usability issues that may not have been apparent before.
  - Performance Issues: It helps identify performance bottlenecks and areas where the application may not scale well
  - Marketing Insight: Gathering feedback from beta users helps ensure the product meets the expectations of its target market

4.4 Acceptance Testing

- Acceptance testing is a critical phase in the software development lifecycle, focusing on evaluating whether the system meets the agreed-upon requirements and specifications set by the business or the end-users.

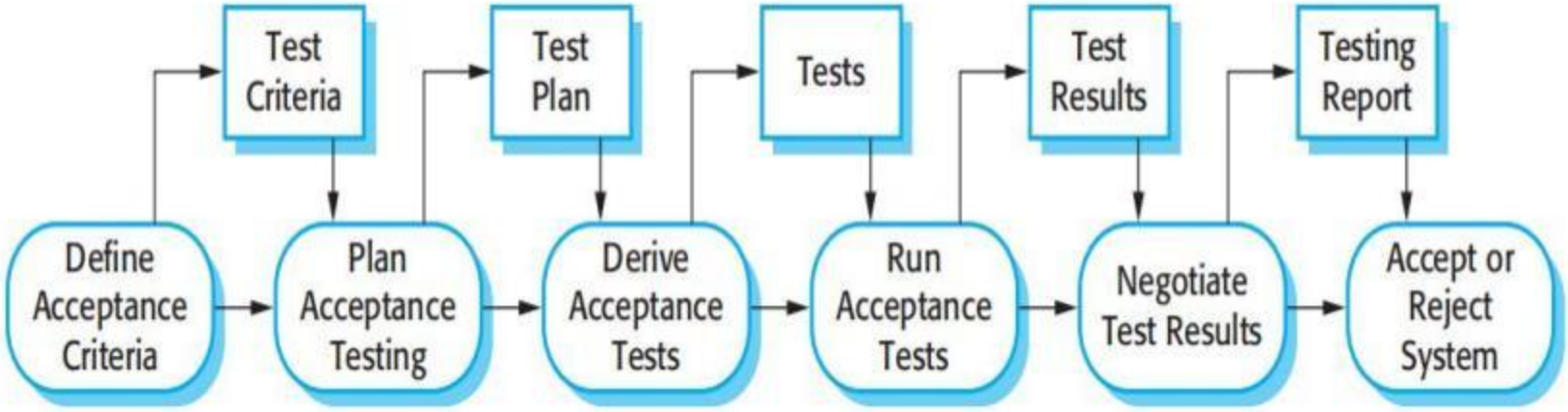
- It **verifies** that the software system meets all business and user requirements, **checks** compliance with regulations, standards, and other criteria agreed upon with the stakeholders, and **ensures** that the system is capable and ready for operational use and is satisfactory to the end-users.

4.4.1 Acceptance Testing - Step1



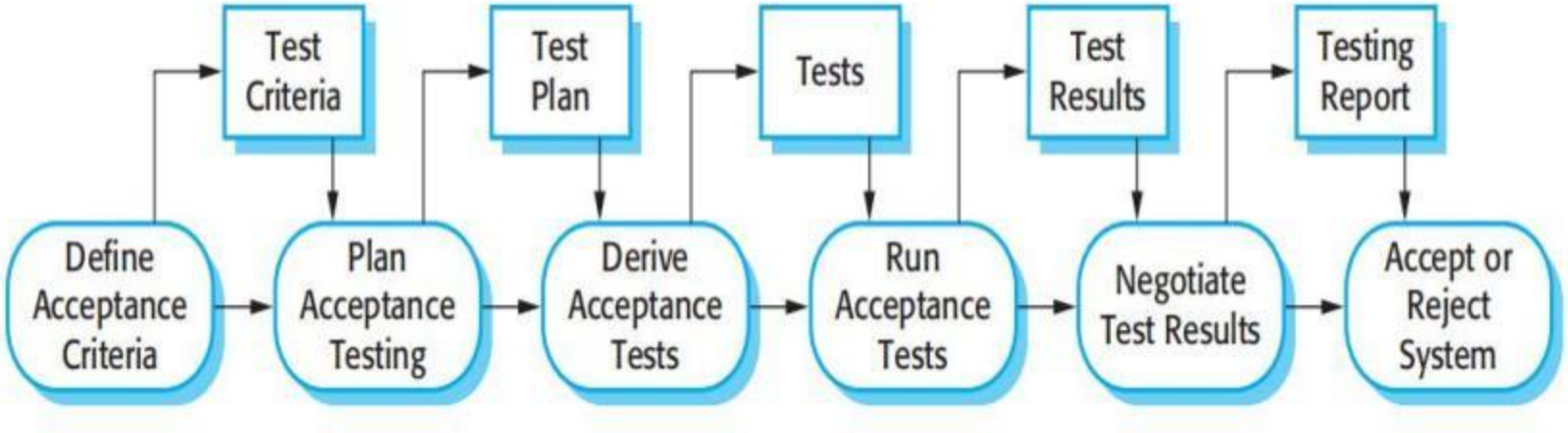
1. **Define acceptance criteria:** Ideally, it should take place early in the process before the contract for the system is signed. In practice, detailed requirements may not be available and there may be significant requirements change during the development process

4.4.2 Acceptance Testing - Step2



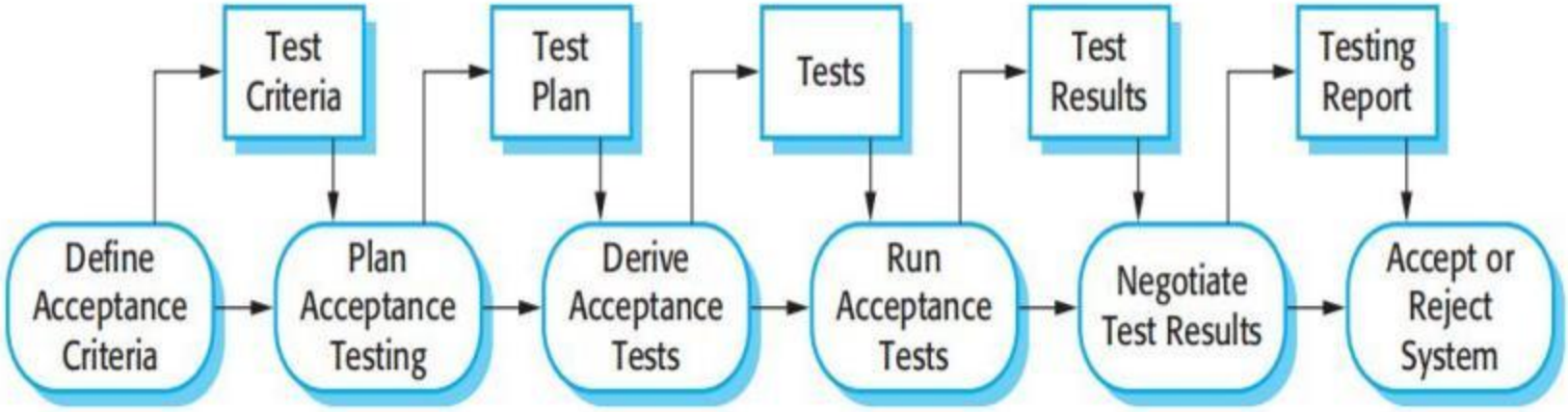
2. **Plan acceptance testing:** It involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. Discussion about (1) the required coverage of the requirements; (2) the order in which system features are tested; and (3) risks to the testing process and how to mitigate them.

4.4.3 Acceptance Testing - Step3



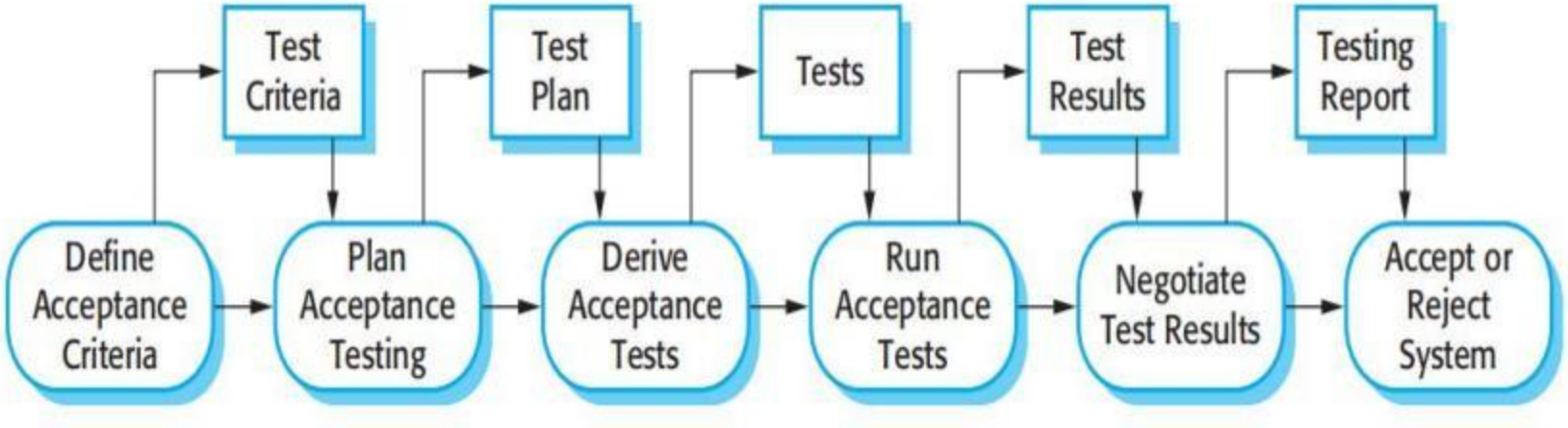
3. **Derive acceptance tests:** Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system

4.4.4 Acceptance Testing - Step4



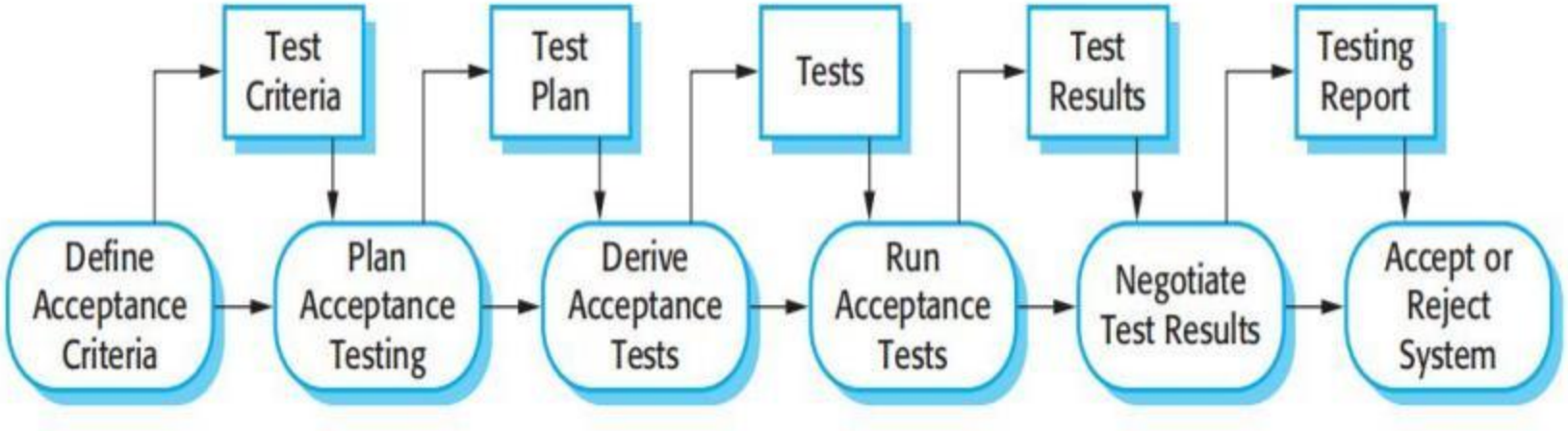
4. **Run acceptance tests:** The agreed acceptance tests are executed on the system. Ideally, take place in the actual environment where the system will be used. Practically, a user testing environment may have to be set up to run these tests.

4.4.5 Acceptance Testing - Step5



5. **Negotiate test results:** It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. The developer and the customer have to negotiate to decide if the system is good enough to be put into use. They must also agree on the developer's response to identified problems.

4.4.6 Acceptance Testing - Step6



6. **Reject/accept system:** This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.