Software Engineering

Week 10 Software Testing - P1

AY 24/25

Week 10

Topics covered

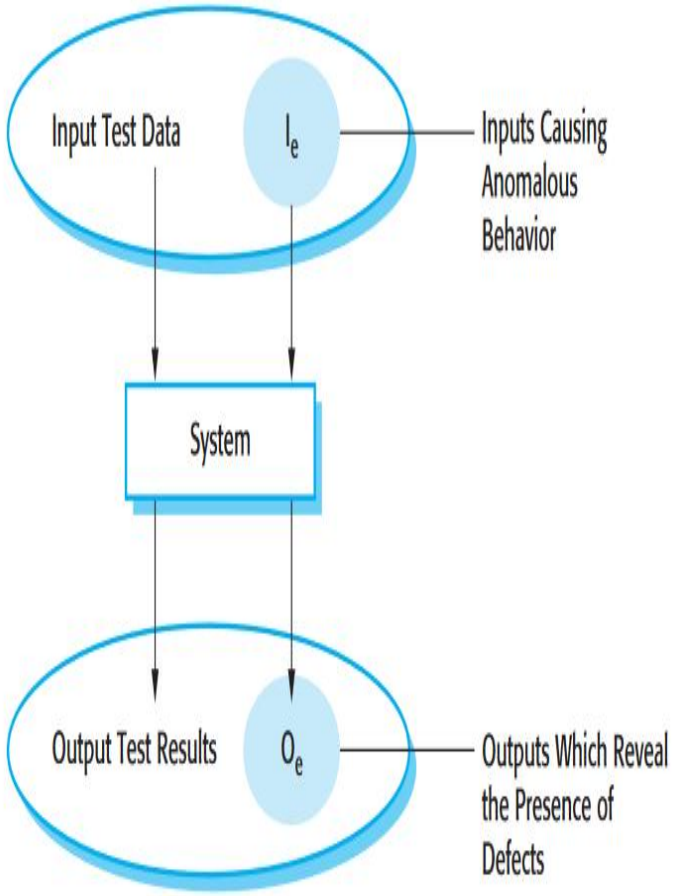
1. Testing
2. Development testing
3. Release testing
4. User testing

1.1 Testing - A Note

- From a general perspective, testing is part of a broader process of software verification and validation (V & V), checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software.
- These checking processes start as soon as requirements become available and continue through all stages of the development process
- Two Techiniques:
  - Static: Inspections concerned with analysis of the static system representation to discover problems (e.g., check document and code analysis)
  - Dynamic: Testing concerned with exercising and observing product behaviour. The system is executed with test data and its operational behaviour is observed.

1.2 Testing - The Basics

- Testing, from a practical, execution-focused view, is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- When you test software, you execute a program using **artificial data**.
- 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

- To demonstrate to the developer and the customer that the software meets its requirements, for **validation**
  - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification, for **finding defects**
  - It is concerned with rooting out 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.
  - System designers and programmers are likely to be involved
- Release testing, where a separate testing team test a complete version of the system before it is released to users.
  - Check that the system meets the requirements of system stakeholders
- User testing, where users or potential users of a system test the system in their own environment.
  - The ‘user’ may be an internal marketing group
  - Acceptance testing where the customer formally tests

2. Development testing - The Basics

✧ Development testing includes all testing activities that are carried out by the team developing the system.

- **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. Component testing should focus on testing component interfaces.
- **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.

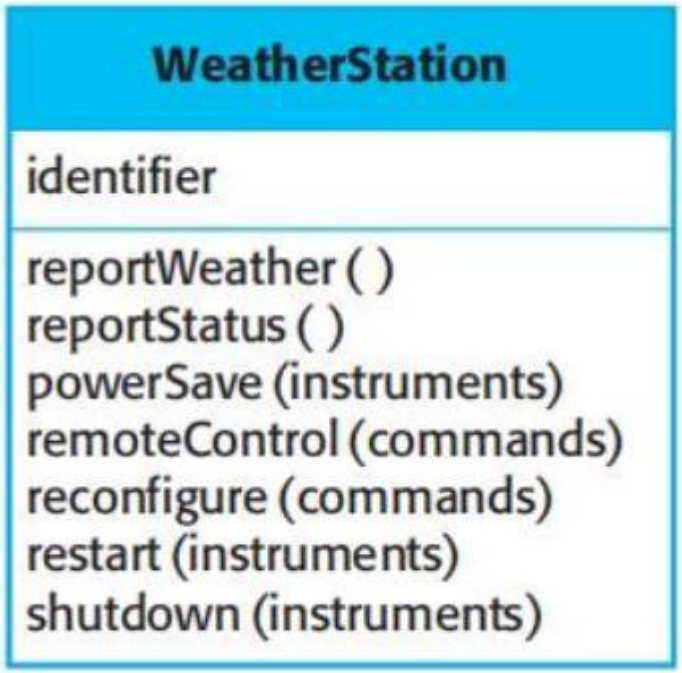
2.1 Unit testing

- Unit testing is the process of testing individual components 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.

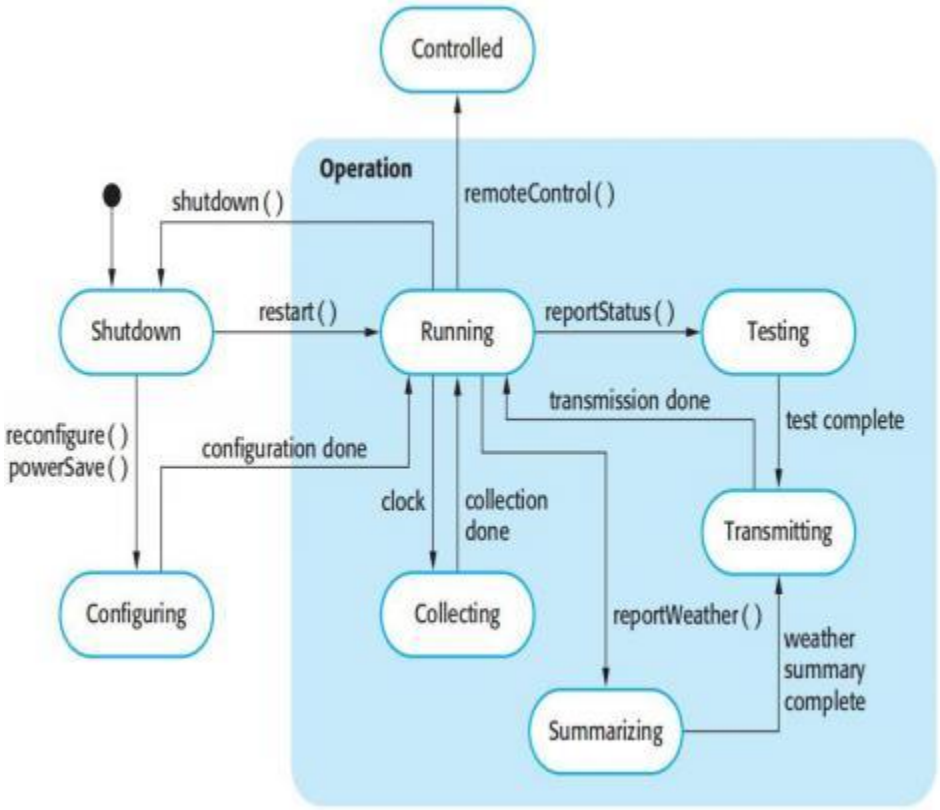
- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states.

2.1.1a Unit testing - The weather station example



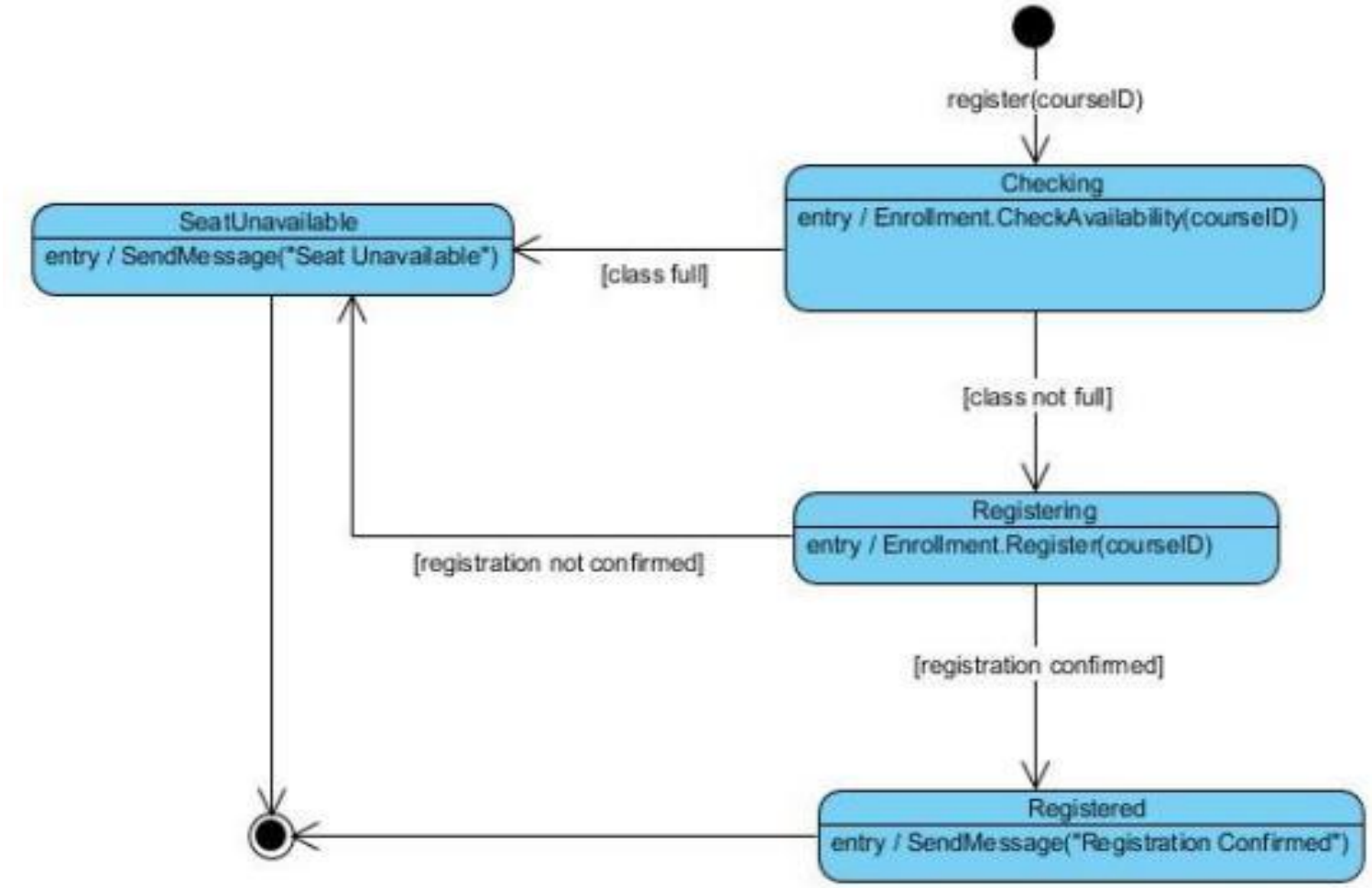
- It has a single attribute, which is its identifier. Only need a test that checks if it has been properly set up.
- Define test cases for all of the methods associated with the object such as reportWeather, reportStatus, etc.
- Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary (e.g., restart/shutdown).

2.1.1b Unit testing - The weather station example



- Using a state model, identify sequences 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

2.1.2a Unit testing - Another Example



2.1.2b Unit testing - Another Example

Initial State	Inputs	Next State
Start	register(courseID)	Checking
Checking	[class not full]	Registering
Checking	[class full]	SeatUnavailable
Registering	[registration confirmed]	Registered
Registering	[registration not confirmed]	SeatUnavailable
SeatUnavailable		Terminate
Registered		Terminate

- Test Case 1:Start > Checking > Registering > Registered >Terminate
- Test Case 2:Start > Checking > SeatUnavailable > Terminate
- Test Case 3:Start > Checking > Registering > SeatUnavailable > Terminate

2.1.3a 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.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

2.1.3b Automated Unit Testing

- A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
- 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.

2.1.3c Automated Unit Testing


```
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 }
```

2.1.4a Choosing A Test Case

- Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:
  - The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
  - If there are defects in the component, these should be revealed by test cases.
- This leads to 2 types of unit test case:
  - The first of these should reflect normal operation of a program and should show that the component works as expected.
  - The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

2.1.4b Strategies of Choosing A Test Case

Strategy 1: Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.

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 componen

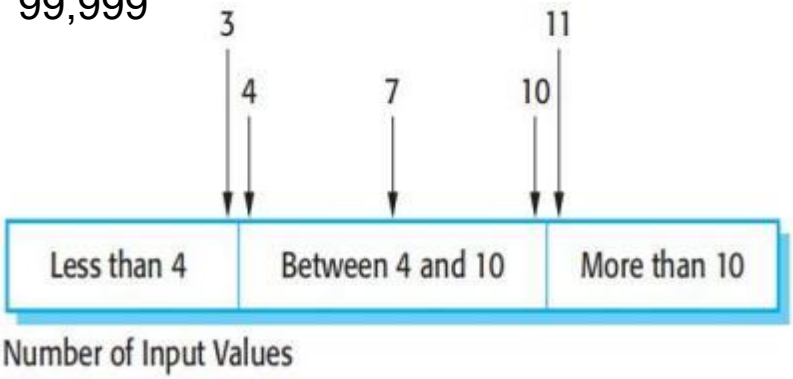
2.1.5a Partition Strategy

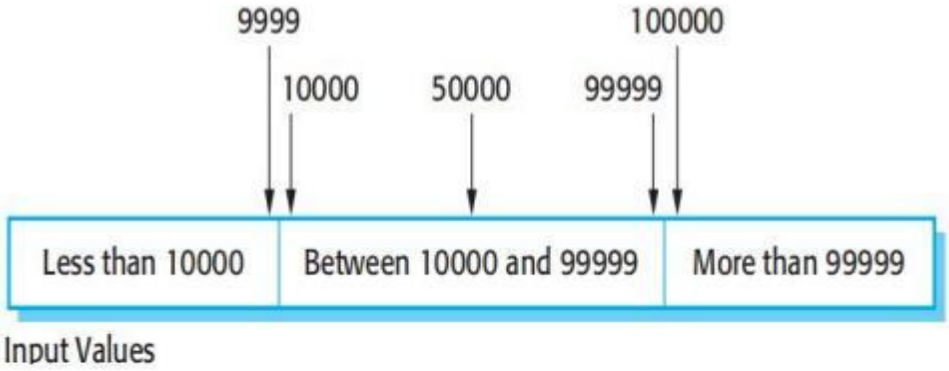
- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Partition - Positive Integer	Partition - Negative Integer
1,2,3,4	-1,-2,-3,-4

2.1.5b Partition Strategy

Program accepts 4 to 10 inputs which are five-digit integers ranging from 10,000 to 99,999





A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition.

You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors.

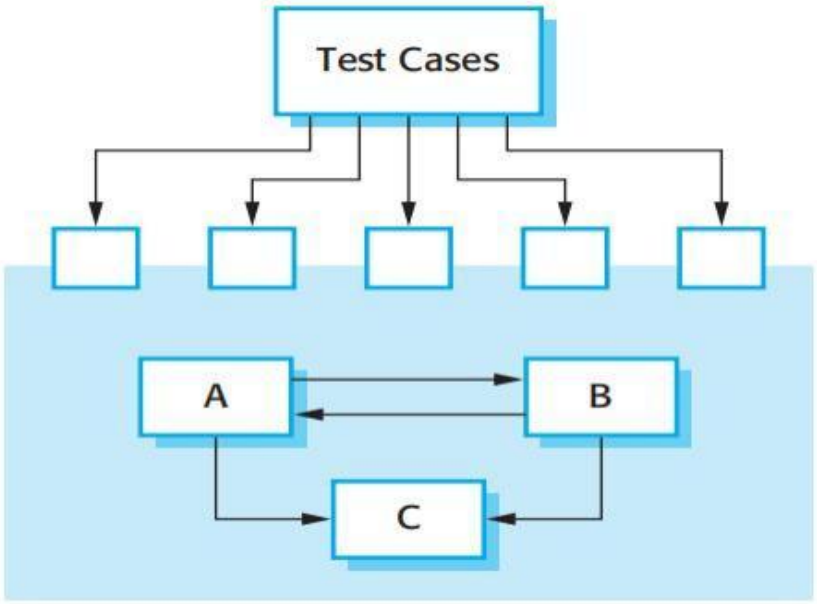
2.1.6 Guideline Strategy

- Test software with sequences/arrays/list which have only a single value.
  - Programmers naturally think of sequences as made up of several values and sometimes they embed this assumption in their programs. Consequently, if presented with a single-value sequence, a program may not work properly
- Use sequences/arrays/list of different sizes in different tests.
  - [1], [1,2], [1,2,3] and so on
- Derive tests so that the first, middle and last elements of the sequence are accessed.
  - e.g., the partition strategy
- Test with sequences/arrays/list of zero length.
  - e.g., []

2.2 Component testing

- Software components are often composite components that are made up of several interacting objects.
- You access the functionality of these objects through the defined component 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



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

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

2.2.3 General Guidelines for Interface Testing

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges

- Always test pointer parameters with null pointers
  - processCustomerData(*some para*) -> processCustomerData(NULL)

- Design tests which cause the component to fail
  - parseDate(String dateStr) in format "YYYY-MM-DD” -> parseDate("2024-31-02") or parseDate("02-2024-31")

- Use Stress stress testing
  - Test how the system handles high traffic and data load

- Where several components interact through shared memory, design tests that vary the order in which these components are activated
  - In a multi-threaded application where T1, T2, T3 threads access and modify shared variables, write tests performing T1, T3, T2 or so

2.3 System testing

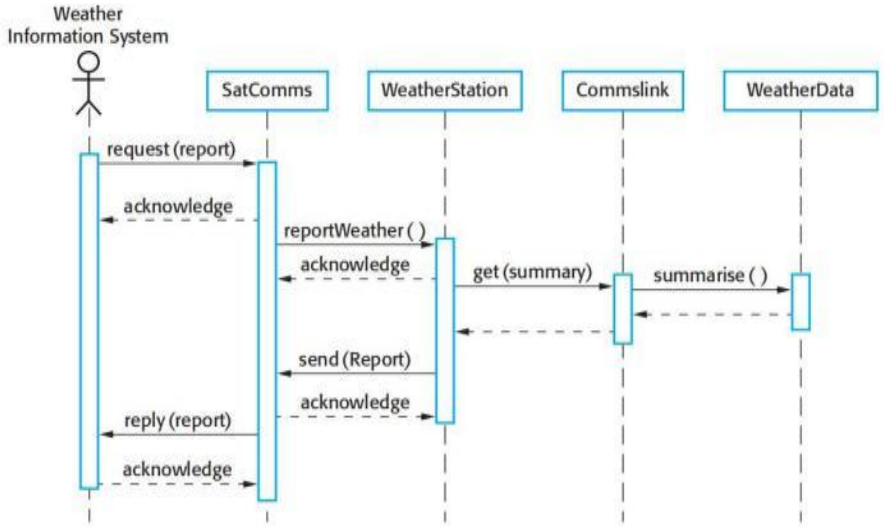
- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer 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 thereare two important differences
  - During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
  - Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
    - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

2.3.2 Use case-based Approach

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- You can use the sequence diagrams to identify operations that will be tested and to help design the test cases to execute .



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

Test Case

2.3.3 System Testing Policies

- Exhaustive system testing is impossible so testing policies, which define the required system test coverage, may be developed.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
    - e.g., Under the "File" menu, there are "Open", "Save", "Exit" buttons, ensure all these buttons interacting with users can work
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
    - e.g., Selecting text and applying multiple formatting options together, e.g., Bold + Italic, to ensure they can work in combination without issues
  - Where user input is provided, all functions must be tested with both correct and incorrect input.
    - e.g., username + password

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 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 .

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.

SET TEST

- 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 two 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

3.3a Scenario Approach

- Scenario approach devises 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.

3.3c Scenario Approach

- ✧ Authentication by logging on to the system.
- ✧ Downloading and uploading of specified patient records to a laptop.
- ✧ Home visit scheduling.
- ✧ Encryption and decryption of patient records on a mobile device.
- ✧ Record retrieval and modification.
- ✧ Links with the drugs database that maintains side-effect information.
- ✧ The system for call prompting.

3.4a Testing Performance

- 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

- To conduct performance test:
  - An operational profile: a set of tests that reflect the actual mix of work that will be handled by the system
  - Construct: If 90% of the transactions in a system are of type A; 5% of type B; and the remainder of types C, D, and E
    → design the operational profile so that the vast majority of tests are of type A

3.4c Testing Performance

✧ 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 transaction

  → Increase the load on the system 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.
  - The reason for this is that 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 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.**

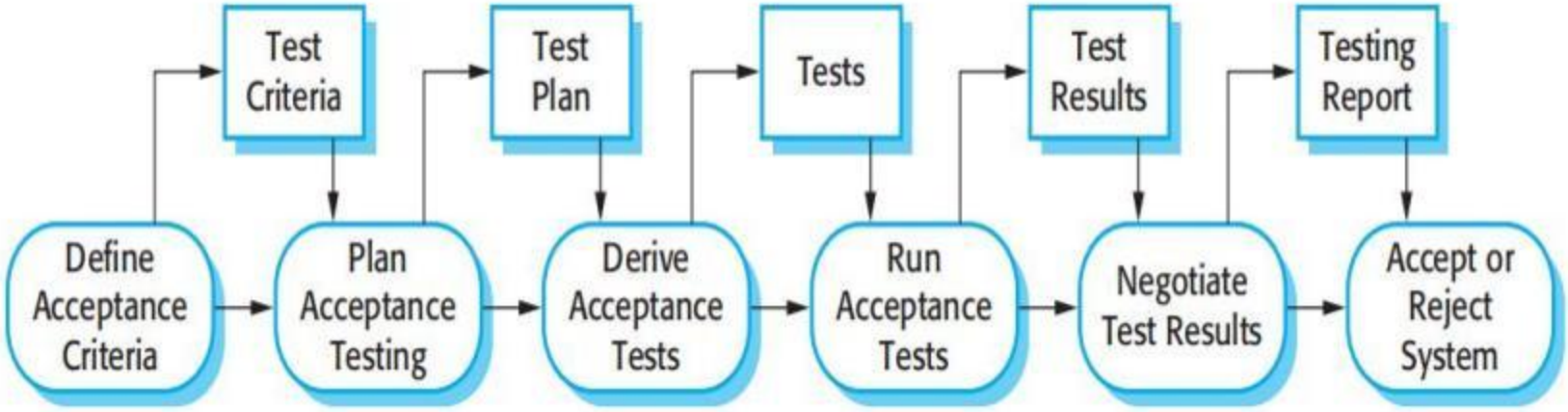
- **Benefits**
  - Usability Feedback: It allows the developers to see how real users interact with the software, which can highlight usability issues that may not have been apparent before.
  - Performance Issues: It helps identify performance bottlenecks and areas where the application may not scale 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.

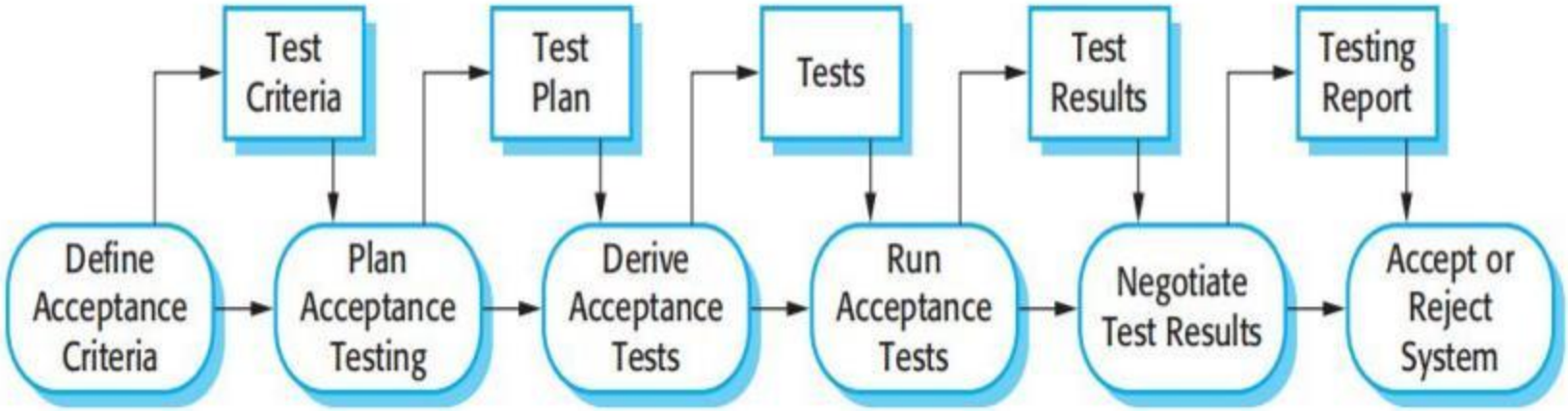
- It **verifies** that the software system meets all business and user requirements, **checks** compliance with regulations, standards, and other criteria agreed upon with the stakeholders, and **ensures** that the system is capable and ready for operational use and is satisfactory to the end-users.

4.4.1 Acceptance Testing - Step1



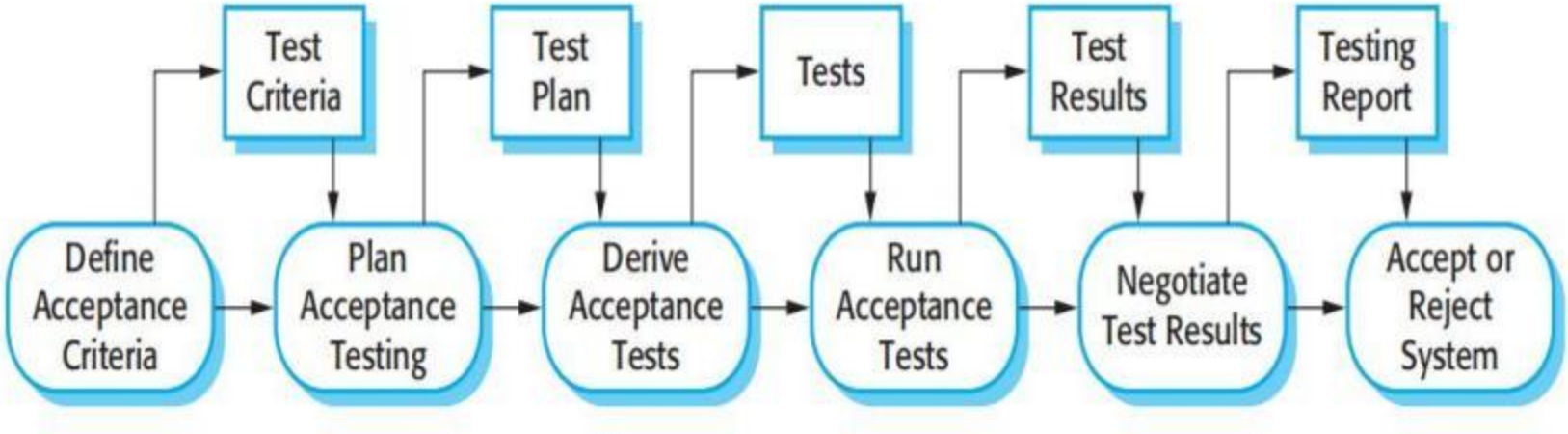
1. **Define acceptance criteria:** Ideally, it should take place early in the process before the contract for the system is signed. In practice, detailed requirements may not be available and there may be significant requirements change during the development process

4.4.2 Acceptance Testing - Step2



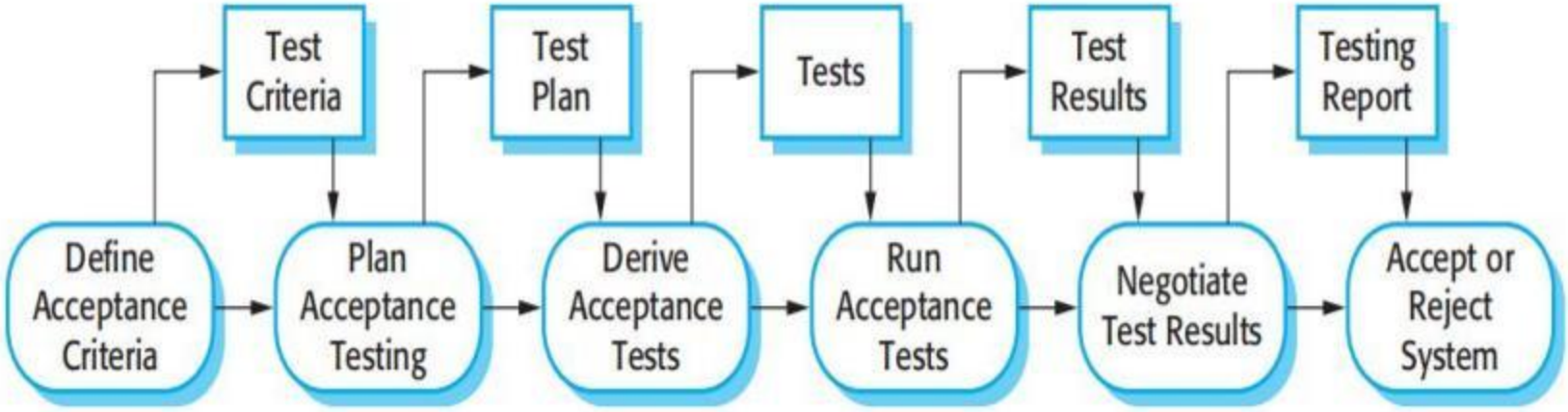
2. **Plan acceptance testing:** It involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. Discussion about (1) the required coverage of the requirements; (2) the order in which system features are tested; and (3) risks to the testing process and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

4.4.3 Acceptance Testing - Step3



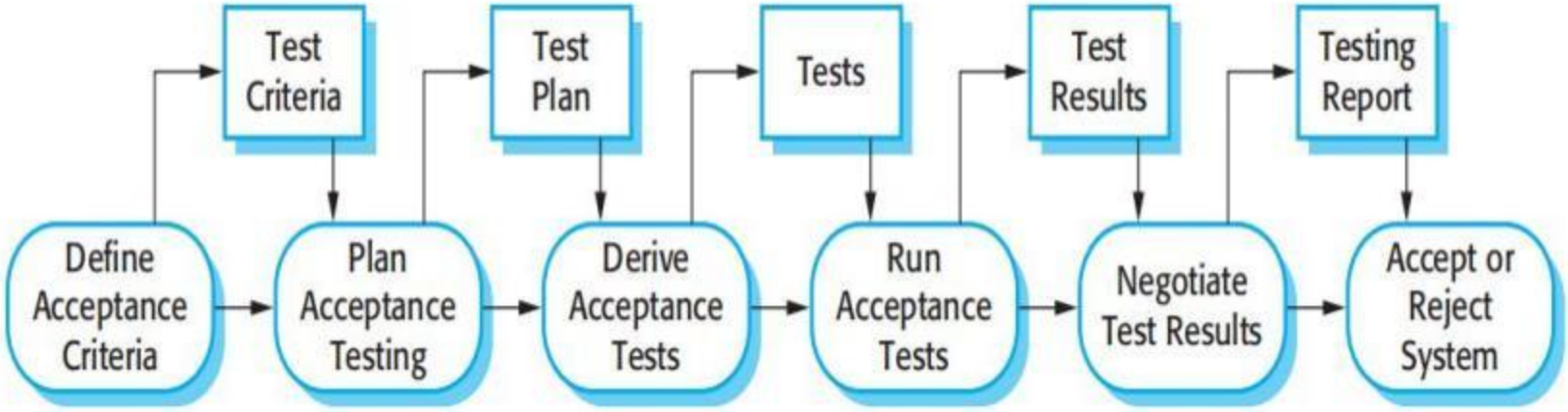
3. **Derive acceptance tests:** Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system

4.4.4 Acceptance Testing - Step4



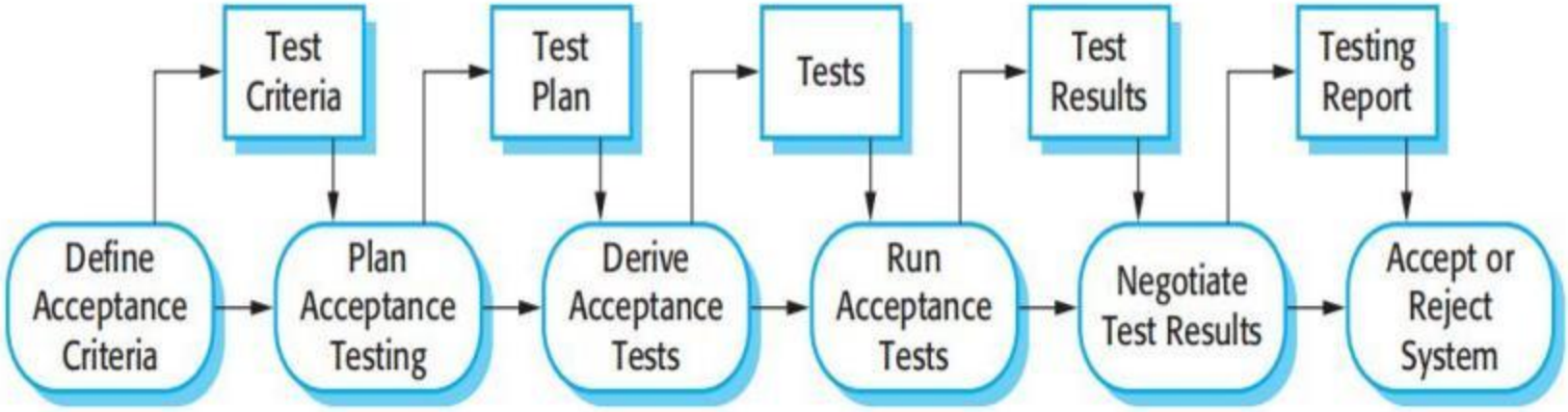
4. **Run acceptance tests:** The agreed acceptance tests are executed on the system. Ideally, take place in the actual environment where the system will be used. Practically, a user testing environment may have to be set up to run these tests.

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.