Software Testing

Topics Covered:

1. Development testing
2. Test-driven development
3. Release testing
4. User testing

Program Testing: Testing refers to show that program does what it is intended to do and to discover program defects before it is executed. When we test software, we execute program with the test/ artificial data. Testing is part of general verification and validation process.

Validation and Defect Testing:

- First goal leads to validation testing: you expect the system to perform correctly using a given sort of test cases
- Second goal leads to defect testing: the test cases are designed to expose defects.

Program Testing Goals:

- Validation testing: To demonstrated to the developer and the customer that software meet its requirement
- Defect Testing: To discover situation in which the behavior of software is incorrect, or does not conform to its specification. A successfully test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Verification vs Validation:

- Verification: The software should conform/ obey to its specification. (Are we budding the product right?)
- Validation: The software should do what the user really requires (Are we building the right product)

Aim to V & V confidence:

- Aim of V & V is to establish confidence that the system is “fit for purpose”.
- Depend on different variables:
  - System/ Software purpose: the level of confidence depend on how critical the software is to an organization.
  - User expectations: users may have low expectation of certain kinds of software.
  - Marketing environment: Getting product in market early may be more important that finding defect in program.

Inspection and Testing:

- **Software inspections** concerned with analysis of the static system representation to discover problems (static verification): may be supplement by tool-based document and code analysis.
- **Software testing** concerned with exercising and observing product behaviors (dynamic verification): system is executed with test data and its behavior is observed.

Software inspection:

- Involve people examining with aim of discovering anomalies and defect. Inspections not require executing of a system so may be used before implementation.

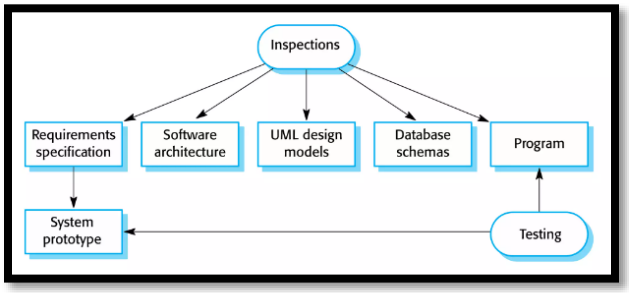
Advantages of Inspection:

- During testing, error can hide other errors, as because inspection is static process and do not have to be concern with interaction between the errors.
- Incomplete version of system can be inspected without additional cost.
- Apart of searching for program defect, inspection consider other attribute of system such as portability & maintainability.

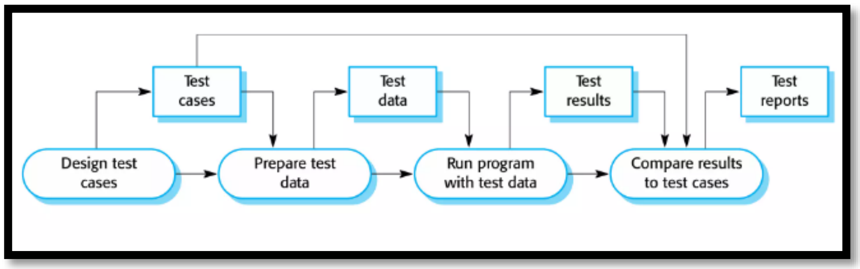
Inspection and Testing:

- Both should be used during the V & V process. Botha re complementary and do not oppose verification technique.
- Inspections cannot check non-functional characteristic such as perform, usability, etc.

Inspection:



Software Testing Process:



Stages of testing:

- Development Testing: where system is tested during development to discover bugs and defects
- Release testing: where a separating testing team test a complete version of system before it's released to users
- User testing: where user of a system test the system in their own environment.

Development Testing:

It includes all testing activities that carried out by the team developing the system.
There are three kind of testing:

- **Unit testing**: Where individual program units or object classes are tested. It is mainly focused on testing the <u>functionality of object or method.</u>
- **Component Testing**: where several individual units are interrogate to create composite component and testing should focus on testing <u>component interfaces</u>.
- **System Testing**: where some or all components in a system are integrated and the system is tested as a whole. Focus on testing <u>component interaction</u>.

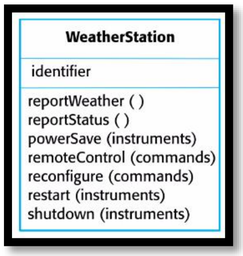
Unit testing: it is a defect testing process where individual program units or object classes are tested in isolation.

- Units may be: individual functions or method, object classes with several attributes, and composite component.

Object class testing: complete test coverage of class involves testing all operations associated with an object, and setting all object attributes. Inheritance make it more difficult to design object class test as information to be tested is not localizes.

Example: Weather Station:

- Need to define test cases of ReportWeather, calibrate, test, startup, shutdown.
- Using a state model, identify sequence of state transition to be tested
  - Example:
  - Shutdown → Running → Shutdown
  - Configuring → Running → Testing → Transmitting → Running



Automated Testing: Whenever possible, unit testing should be automated so that tests are run and checked without manual

We can make use of a testing automation framework (**Junit**) to write and run program test.

These framework provide generic test classes that you extend to create specific test cases

Automated test components:

- A setup part: where we initialize the system with the test cases, namely the input and expected output.
- A call part: where we call the object or method to be tested.
- An assertion part: where we compare the result of the call with expected result. (if true then test has been successful)

Choosing Unit Test cases:

The test cases should show that the component that you are testing does what it is supposed to do
If there are defect in the component, it should be revealed by test cases

This lead to 2 types of unit test case:

- The first should reflect normal operation of program and should how that the component works as expected
- The other kind of test case based on testing experience of where common problem arises. It should use abnormal inputs to check that there are properly processes and do not crash the component.

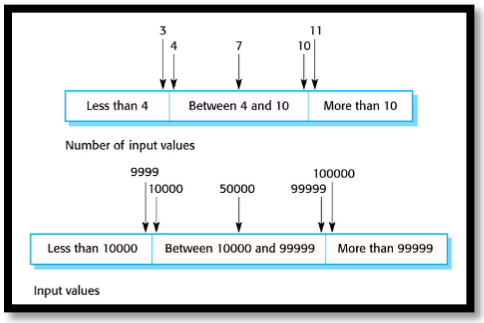
Testing Strategies:

- **Partition Testing**: where we identify groups of inputs that have common characteristics should be processed in same way.
- **Guideline-based testing**: where we use testing guideline to choose test cases. These guideline reflect previous experience of the kinds of errors that programmers often make when developing the component.

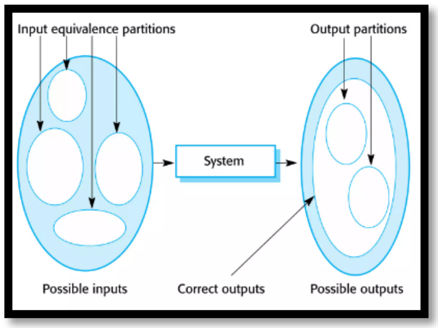
Partition Testing:

- Input data and output result often fall into difference classes where all member of class are related.
- Each of these classes is an Equivalence partition where the program behave in an Equivalent way for each member
- Test cases should be chosen from each partition.

Equivalence Partitions:



Equivalence Partitioning:



Testing Guidelines (Sequences):

- Test software with sequences which have only a single value
- Use sequences of difference size in difference test
- Test with sequences of zero length.

General Testing Guidelines:

- Choose input that force the system to generate all error messages
- Design input that cause input buffers to overflow
- Force invalid outputs to be generate
- Force computation results to be too large or too small.

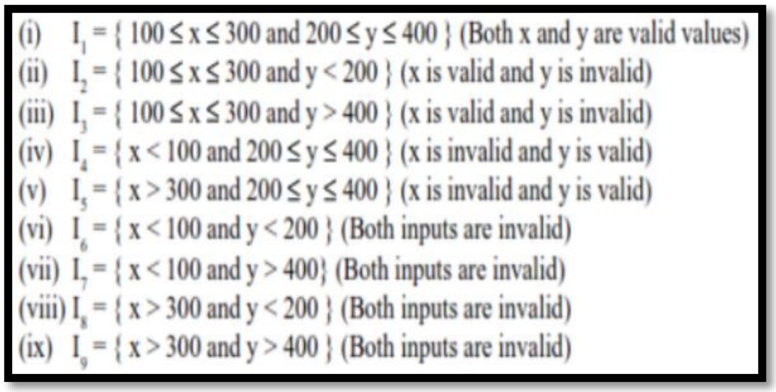
Creation of Equivalence Testing:
The entire input domain can be divide into at least two equivalence classes (one contain all valid and other invalid)

Equivalence class testing:
It is strategy to efficiently cover a wide range of scenarios while minimizing the number of test cases needed by partitioning input and expected output. Instead of executing every possible test case, testers groups input into categories that are expected to exhibit similar behavior.
So, by selecting just one representative test case from each category, testers can still achieve a reasonable level of coverage.
This approach relies on the assumption that is one test in that group works correctly the rest test cases will work correctly.

Applicability:
The basic requirement is that input or output must be partitioned based on the requirements and every partition will give a test case. If ne test case catches a bug, the other probably will too and vice versa**.**

Example for addition test cases:

100 <= x <= 300
200 <= y <= 400



(i) $I_1 = \{ 100 \le x \le 300 \text{ and } 200 \le y \le 400 \}$ (Both x and y are valid values)
(ii) $I_2 = \{ 100 \le x \le 300 \text{ and } y < 200 \}$ (x is valid and y is invalid)
(iii) $I_3 = \{ 100 \le x \le 300 \text{ and } y > 400 \}$ (x is valid and y is invalid)
(iv) $I_4 = \{ x < 100 \text{ and } 200 \le y \le 400 \}$ (x is invalid and y is valid)
(v) $I_5 = \{ x > 300 \text{ and } 200 \le y \le 400 \}$ (x is invalid and y is valid)
(vi) $I_6 = \{ x < 100 \text{ and } y < 200 \}$ (Both inputs are invalid)
(vii) $I_7 = \{ x < 100 \text{ and } y > 400 \}$ (Both inputs are invalid)
(viii) $I_8 = \{ x > 300 \text{ and } y < 200 \}$ (Both inputs are invalid)
(ix) $I_9 = \{ x > 300 \text{ and } y > 400 \}$ (Both inputs are invalid)

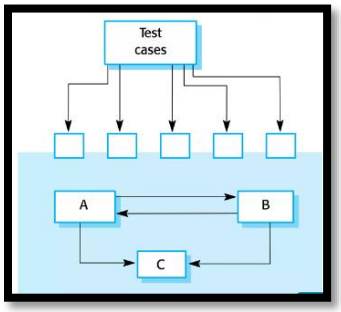
Component Testing:

Software components are often composite components that are made up of several interacting objects. We access the functionality of these objects through the defined component interface. It focus on showing that the component interface behaves according to its specification.

Interface testing: objectives are to detect faults due to interface errors or invalid assumptions about interfaces.

Interface types:

- **Parameter interfaces**: Data passed from one method to another
- **Shared memory interfaces**: block of memory shared between procedure or method
- **Procedural interface**: Sub system encapsulates a set of procedures to be called by other sub systems
- **Message passing interfaces**: sub-systems request service from other sub-systems.



Interface errors:

- **Interface misuse**: a calling component calls another component and makes an error in its use.
    - Example: (parameter in wrong order)
- **Interface Misunderstanding**: A calling component embeds assumption about the behavior of the called component which are incorrect.
- **Timing errors**: the called and calling component operated at difference speeds and out dated information is accessed.

Interface Testing Guidelines:

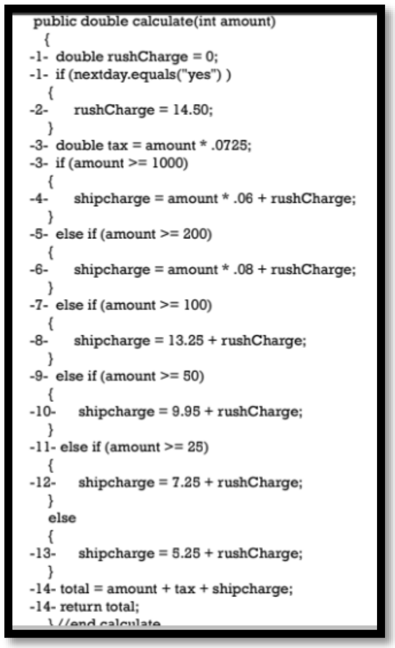
- Design test so that parameter to a called procedure are at extreme ends of their ranges
- Always test pointer parameters with the null pointers
- Testing test which cause the component to fail
- Use stress testing in message passing system
- In shared memory vary the order in which components are activated.

Path Testing:

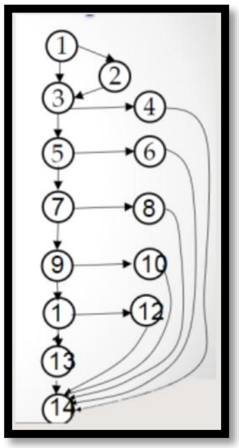
A systematic approach aimed at ensuring that every possible path through program is traversed by executing a set of test cases. The starting point for path testing is a program flow graph that shows (nodes = program decision and arc = flow of control)

Basic steps of path testing:

1. Draw a control graph (to determine different program paths)
2. Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
3. Find a basis set of paths
4. Generate test cases to exercise each path.



```
 public double calculate(int amount)
   {
-1-  double rushCharge = 0;
-1-  if (nextday.equals("yes") )
   {
-2-     rushCharge = 14.50;
   }
-3-  double tax = amount * .0725;
-3-  if (amount >= 1000)
   {
-4-     shipcharge = amount * .06 + rushCharge;
   }
-5-  else if (amount >= 200)
   {
-6-     shipcharge = amount * .08 + rushCharge;
   }
-7-  else if (amount >= 100)
   {
-8-     shipcharge = 13.25 + rushCharge;
   }
-9-  else if (amount >= 50)
   {
-10-    shipcharge = 9.95 + rushCharge;
   }
-11- else if (amount >= 25)
   {
-12-    shipcharge = 7.25 + rushCharge;
   }
   else
   {
-13-    shipcharge = 5.25 + rushCharge;
   }
-14- total = amount + tax + shipcharge;
-14- return total;
   } //end calculate
```



Step-01

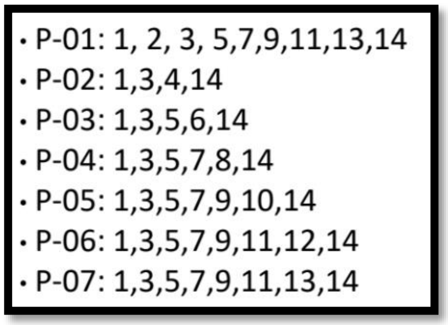
Step-02:

V = E – N +2

V = 19 – 14 + 2

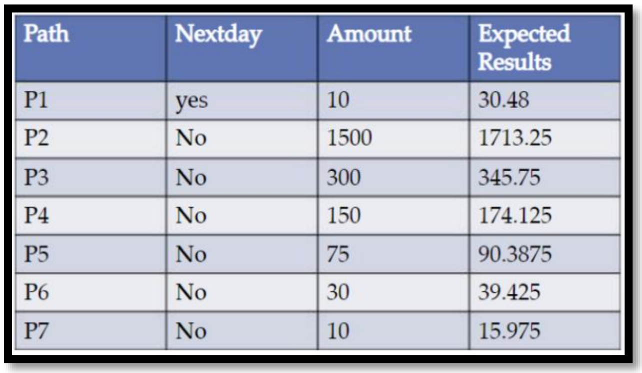
V = 7

Step-03:



- P-01: 1, 2, 3, 5,7,9,11,13,14
- P-02: 1,3,4,14
- P-03: 1,3,5,6,14
- P-04: 1,3,5,7,8,14
- P-05: 1,3,5,7,9,10,14
- P-06: 1,3,5,7,9,11,12,14
- P-07: 1,3,5,7,9,11,13,14

Step-04:



Path	Nextday	Amount	Expected Results
P1	yes	10	30.48
P2	No	1500	1713.25
P3	No	300	345.75
P4	No	150	174.125
P5	No	75	90.3875
P6	No	30	39.425
P7	No	10	15.975

System Testing:

System testing during development involves integrating all component and then test the integrated system. The focus in system testing is testing the interactions between components. System testing checks that component are compatible, interact correct and transfer right data at right time across the interface.

System and Component testing:

- During system testing, reusable components that are separated developed may be integrate with newly developed component. The complete system is then tested.
- Component developed by different team members may be integrated at this stage, system testing is collective testing rather than individual process.

Use-case testing:

The use-case developed to identify system interactions can be used for basis of system testing. Each use case involve several system components so testing the use case enable to test interaction occurred between components.
Sequence diagram associated with the use case document the components and interaction that are being tested.

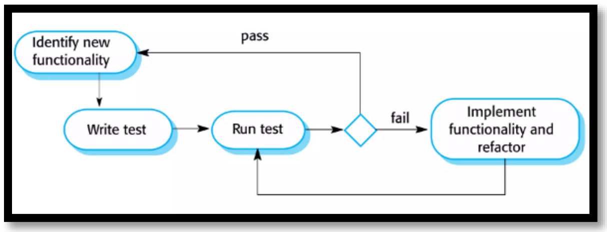
Testing Policies:

These policies designed the required system test coverage. Example of testing policies are:

- All system functionality that are accessed through menus should be tested
- Combination of functions (e.g. text formatting) that are accessed through the same menu must be tested
- Where user input is provided, all functions must be test with both correct input and incorrect input

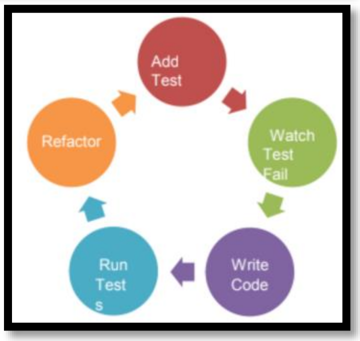
Test-Driven Development

TDD is an approach to program development in which you inter-leave testing and code development. Test are written before code. We develop code incrementally, along with the test for that increment.
TDD was introduced as part of agile method such as extreme programming.


Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

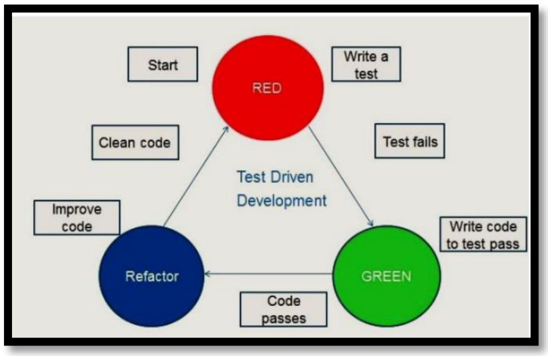
TDD Process Activities:

1. Start by identifying the increment of functionality that is required
2. Write a test for this functionality and implement this as automated test
3. Run the test, along with all other test that have been implemented.
4. Implement the functionality and re-run the test
5. Once all steps run successfully, you move on the implementing the next chunk of functionality.



Benefits of test-driven development:

- **Code coverage**: every code segment that you write has at least one associated test.
- **Regression testing**: suite is developed incrementally as a program is developed
- **Simplified debugging**: when test fails, it should be obvious where the problem lies. So that newly written code needs to be checked and modified.
- **System documentation**: The test themselves are a form of documentation that describe what code should be doing.



Test- Driven Deployment

Regression Testing: is testing the system to check that changes have not broken previously working code.

- **Manual testing process**: regression testing is expensive but, with automated testing, it is simple and straightforward. All test are re run every time a change is made of the program.
- Tests must run successfully before the change is committed.

Release Testing

It is process of testing a particular release of system that is intended for use outside of the development team. The primary goal of release testing process is to convince the supplier of the system that it is good enough for us. Therefore has to show that the system delivers its specified functionality, performance and dependability, and that is does not fails in normal use.

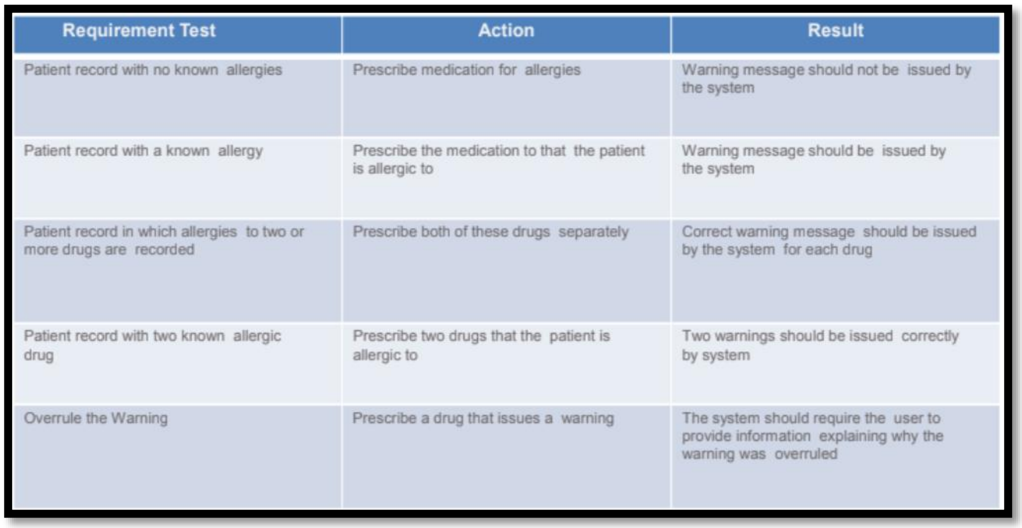
Release testing and system testing: Release testing is a form of system testing, important difference:

- **Release testing**: A separate team that has not been involved in the system development should be responsible.
- **System testing**: development team should focus on finding bugs in system and check the system meet its requirement.

Requirements based testing: it involves examining each requirement and developing a test or test for it.
Example: Mentcare system requirements:

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

Example:



Requirement Test	Action	Result
Patient record with no known allergies	Prescribe medication for allergies	Warning message should not be issued by the system
Patient record with a known allergy	Prescribe the medication to that the patient is allergic to	Warning message should be issued by the system
Patient record in which allergies to two or more drugs are recorded	Prescribe both of these drugs separately	Correct warning message should be issued by the system for each drug
Patient record with two known allergic drug	Prescribe two drugs that the patient is allergic to	Two warnings should be issued correctly by system
Overrule the Warning	Prescribe a drug that issues a warning	The system should require the user to provide information explaining why the warning was overruled

Performance Testing:
Part of release testing may involve testing the properties of as system such as performance and reliability
Test should reflect the profile of use of the system. Performance tests usually involve series of tests where the lead is steadily increase until the system performance becomes unacceptable.
Stress testing is a form of performance testing where the system is overloaded to test its failure behavior.

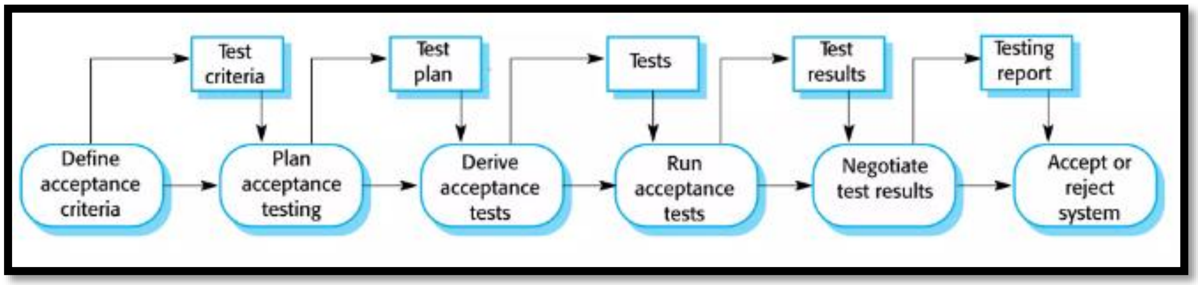
User Testing

User or customer testing is a stage in which user provide input and advice on system testing. User testing is essential even comprehensive system and release testing is done.
The reason for user testing is because of the influence of users working environment have major effect on performance, usability, reliability of system.

Type of user testing:

- **Alpha testing**: Users of software work with the development team to test the software at the developer's site
- **Beta testing**: a release of software is made available to users to allow them to experiment and raise problems that they discover with the system developers
- **Acceptance testing**: customer test a system to decide whether or not it is ready to be accepted from system developers and deploys in the customer environment. Primarily for custom system.

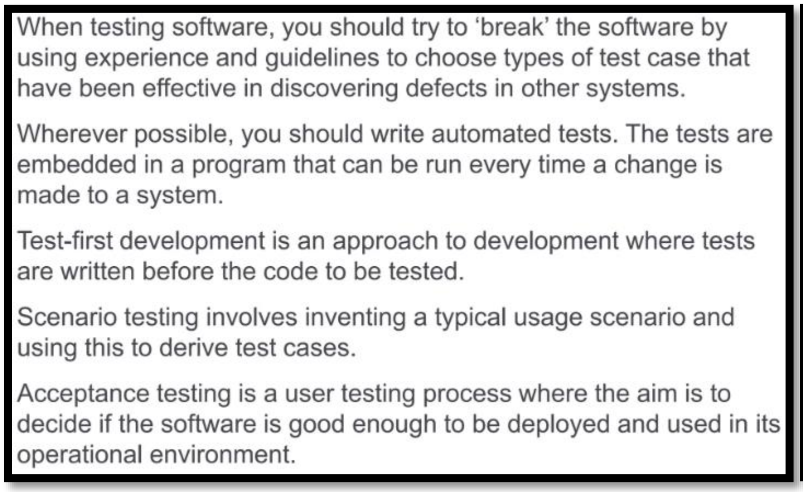
Acceptance Testing:
Define acceptance criteria → Plan acceptance testing → Derive acceptance test → Run acceptance test →
Negotiate test result → Reject/ accept system



Agile methods and acceptance testing:

- In agile method the user is part of development team and is responsible for making decision on acceptability of system
- Test are defined by the user and are integrated with other tests in that they are run automatically when changes are made
- There is no acceptance testing process
- Main problem here is whether or not the embedded user is typical and can represent the interest of all system stakeholder

Some Extra Key Points:

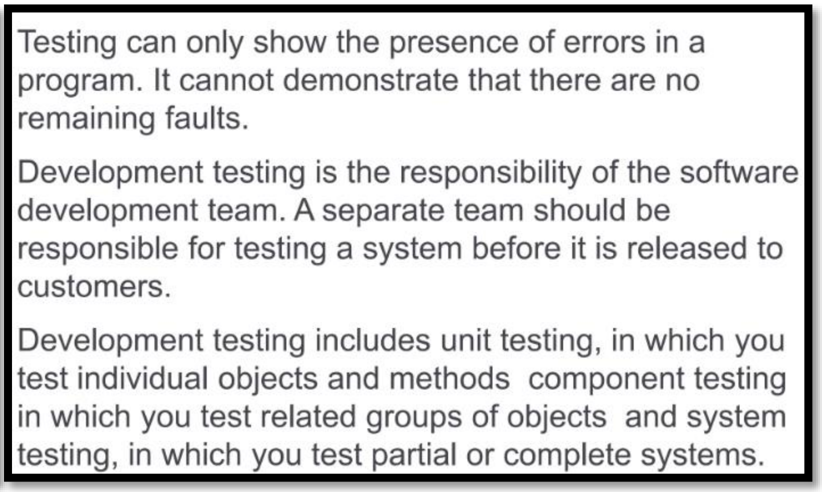

When testing software, you should try to 'break' the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.

Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.

Test-first development is an approach to development where tests are written before the code to be tested.

Scenario testing involves inventing a typical usage scenario and using this to derive test cases.

Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.


Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.

Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.

Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.