UNIT 5: CODING AND TESTING

TESTING

- ❐ **Software testing** is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.
- ❐ Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation.
- ❐ Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects).
- ❐ It involves the execution of a software component or system component to evaluate one or more properties of interest.

ISSUES IN OO TESTING:

- ✰ In procedural software, a unit is a single program, function, or procedure.
- ✰ In object oriented software, a unit is a class.
- ✰ Typical OO software characteristics that impact testing:
    1. State dependent behavior
    2. Encapsulation
    3. Inheritance
    4. Polymorphism and dynamic binding
    5. Abstract and generic classes
    6. Exception handling

The various complexities in OO testing are due to:

1. Research confirms that testing methods proposed for procedural approach are not adequate for OO approach.
2. Thus, we need less emphasis on unit testing and more on integration testing.
3. Another factor is that the **relationships in object-oriented components tend to be very complex**.

   The compositional relationships of inheritance and aggregation especially when combined with polymorphism, introduce new kinds of faults and require new methods for testing.

   This is because the way classes and components are integrated is different in object-oriented languages
4. Testing time for OO software found to be increased compared to testing procedural software

There are various issues in object oriented testing. Following questions lead to various issues in object oriented testing.

1. **What should be the units for object oriented testing?**
2. **What are the implications of composition, encapsulation, inheritance and polymorphism?**
3. **What are the levels of object oriented testing?**
4. **How to perform data flow testing?**

I. Units for object oriented testing:

- A unit is the smallest program component that can be compiled and executed. A unit is a program component that would be developed by one person. Could be a sub-part of one class
- The class is an encapsulation of data attributes and corresponding set of operations. In object oriented testing, focus of unit testing is considered as a class or object.
- There are three main advantages of choosing class as a unit of testing:
    - The class is used in the state chart in which the behaviour of the object is represented.
    - Due to testing classes under the unit testing, the integration testing has clear goals.
    - They are useful for identifying test cases.

II. Implications of Composition and Encapsulation:

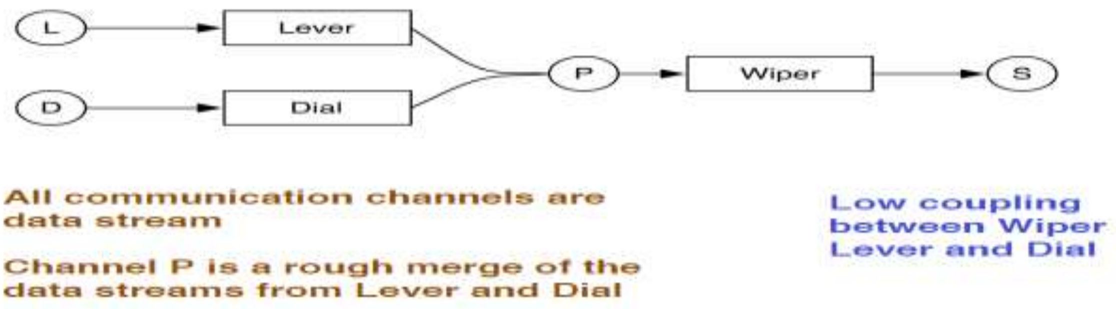
When an object contains the other object, if the contained object cannot exist without the existence of other object, then it is called **the composition**. The composite relationship define the **“has-a”** relationship.

Together with the **goal of reuse**, composition creates the need for unit testing.

If the encapsulation of data and methods in the class is good, then such classes can be easily composed and tested.

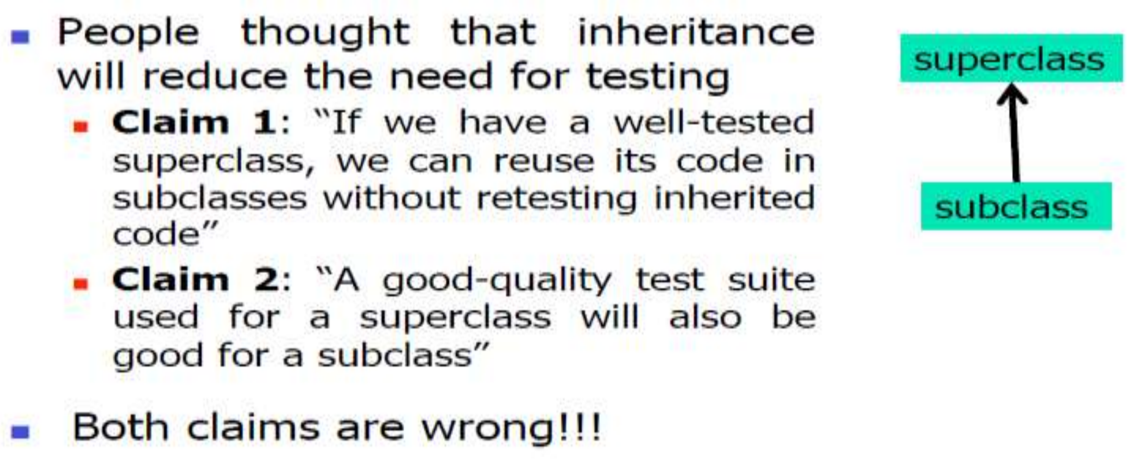
Reality is that the burden is still on integration testing.

Consider the following example:


L
Lever
D
Dial
P
Wiper
S
All communication channels are data stream
Channel P is a rough merge of the data streams from Lever and Dial
Low coupling between Wiper Lever and Dial

III. Implications of inheritance:

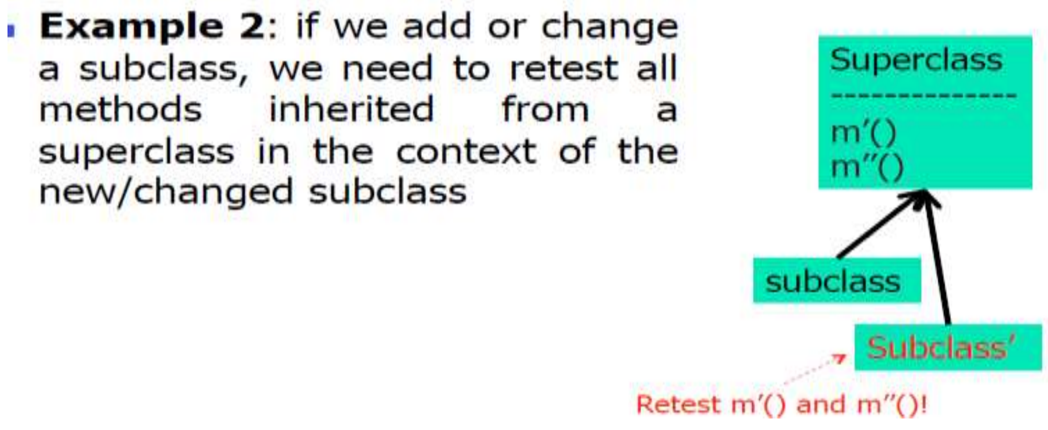
- The **inheritance relationship** is a kind of relationship in which the specialised classes can be derived from the generalised class.
- If in object oriented testing, the class is considered as the unit of unit testing method, then it creates problem for the object oriented design having inheritance.



- People thought that inheritance will reduce the need for testing
  - **Claim 1**: "If we have a well-tested superclass, we can reuse its code in subclasses without retesting inherited code"
  - **Claim 2**: "A good-quality test suite used for a superclass will also be good for a subclass"
- Both claims are wrong!!!

superclass
subclass

- To avoid this, classes are flattened.
- That is, all the attributes or methods that needs to be inherited from superclass to subclass need to be included in the subclasses explicitly.
- "But these flattened classes cannot be tested thoroughly"
- To overcome this problem, the specialised method known as testing method can be added in each of the flattened classes.
- One solution could be:



- **Example 2**: if we add or change a subclass, we need to retest all methods inherited from a superclass in the context of the new/changed subclass

Superclass
m'()
m''()
subclass
Subclass'
Retest m'() and m''()!

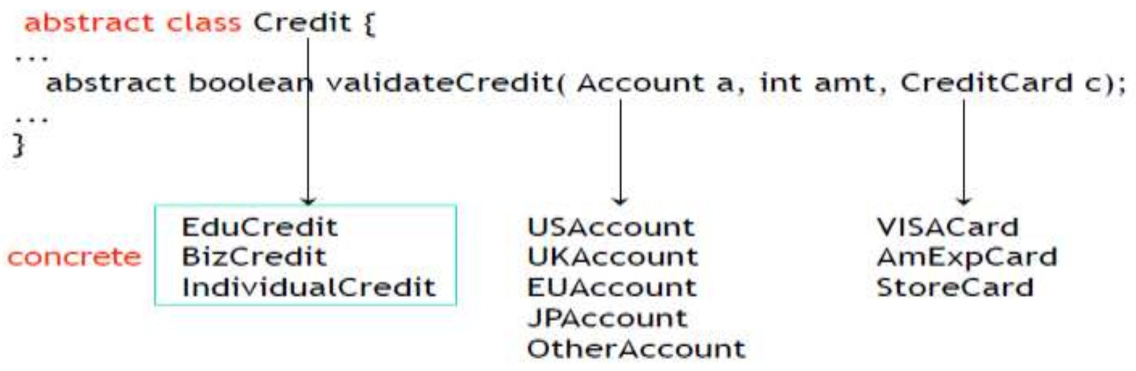
IV. Implications of polymorphism:

Polymorphism means same method can be used by different objects to achieve the same functionality.

Polymorphism is the capability of an operation exhibiting different behaviour in different instances.

The behaviour depends upon the type of data used.

1. It leads to Combinatorial Explosion problem



```
abstract class Credit {
...
  abstract boolean validateCredit( Account a, int amt, CreditCard c);
...
}
```

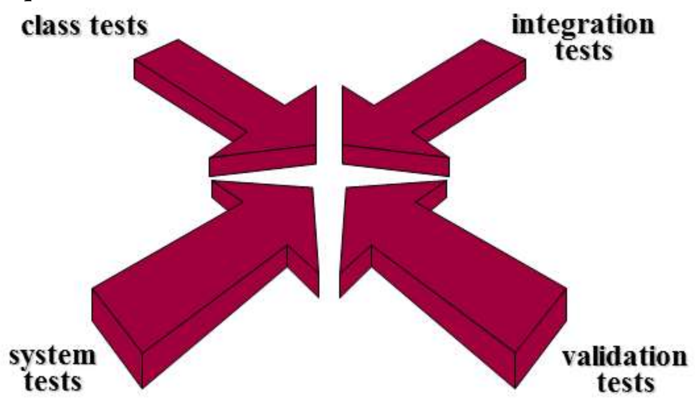
The combinatorial problem: 3 x 5 x 3 = 45 possible combinations of dynamic bindings (just for this one method!)

2. It introduces **undecidability concern** in class based testing as it can lead to messages sent to wrong object.

V. Levels of object testing:

There are four levels of object oriented testing:

a) **Method or operation testing:** Similar to conventional unit testing of procedural software.
b) **Class testing :** classes are tested individually
    a. Also called Intraclass Integration Testing
c) **Integration testing:** interactions among the previously tested classes are tested.
    a. Also called Interclass Integration testing
d) **System testing:** System Testing (ST) is a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements.

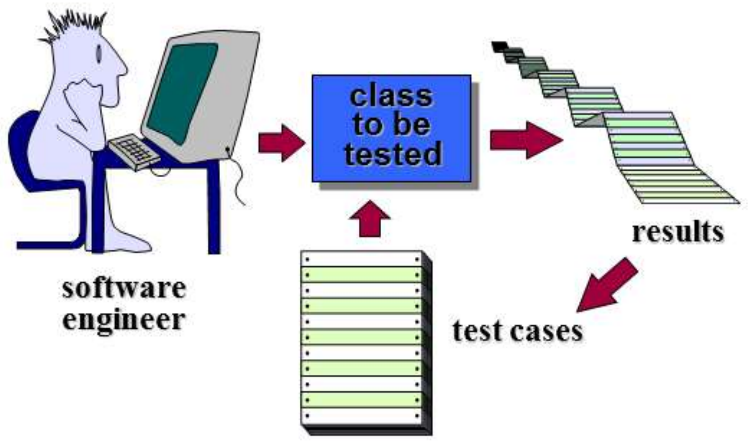


VI. Dataflow Testing :

Data flow testing is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects.
Need analogue to dataflow testing of units in traditional programs.
The solution would be to **use a revised Petri net definition** to handle method calls between classes.

CLASS TESTING

- Class testing is the equivalent of unit testing in conventional software
- Smallest testable unit is the encapsulated class
- A single operation needs to be tested as part of a class hierarchy because its context of use may differ subtly
- Approach:
  - Methods within the class are tested
  - The state behavior of the class is examined
- Unlike conventional unit testing which focuses on input-process-output, class testing focuses on designing sequences of methods to exercise the states of a class
- But white-box methods can still be applied



Ways to test a class:

Code can be tested effectively by

- inspection (preferable when construction of a test driver is too difficult)
- execution of test cases (lends itself to easy regression testing later on)

Remember: When you test a class, you are really:

- creating instances of that class and

- Testing the behavior of those instances.

How is class testing done?

Create a test driver that

- creates instances of the class
- sets up suitable environment around those instances to run a test case
- sends one or more messages to an instance as specified by a test case
- checks the outcome based on a reply value, changes to the instance, or parameters to the message
- deletes any instances it creates if responsible for storage allocation

Two types of class testing:

The two types are:

1. **Functional Testing:** Here , the test methods as black boxes and tests based on specification
2. **Structural Testing:** Here, 'Set' and 'Get' methods are used to update and retrieve values for attributes

Constructing test Cases:

- Identify requirements for test cases for all possible combinations of situations in which
  - a precondition can hold
  - post conditions can be achieved
- Create test cases for those requirements
  - specific input values – typical and boundary
  - determine correct outputs
  - eliminate conditions that are not meaningful
- Add test cases to show what happens when a precondition is violated

Object Interaction:

- Object interaction is:
  - request by a sender object to a receiver object
  - to perform one of the receiver's operations
  - all of the processing performed by the receiver to complete the request
- Includes messages between
  - an object and its components
  - an object and other objects with which it is associated

Consider the Windshield Wiper Example:

1. The lever has four positions: OFF, INT, LOW and HIGH
2. Dial is only relevant when lever is on INT

c2.Lever	OFF	INT	INT	INT	LOW	HIGH
c2.Dial	n/a	1	2	3	n/a	n/a
a1.Wiper	0	4	6	12	30	30

3. To maintain the state of lever and dial
   - 'Sense' methods for lever and dial
   - Get/Set operations for each variable

```
class windshieldWiper

        private wiperSpeed
        private leverPosition
        private dialPosition

        windshieldWiper(wiperSpeed,
                leverPosition, dialPosition)

        getWiperSpeed()
        setWiperSpeed()

        getLeverPosition()
        setLeverPosition()

        getDialPosition()
        setDialPosition()

        senseLeverUp()
        senseLeverDown()

        senseDialUp()
        senseDialDown()

End class windshieldWiper
```

```
senseLeverUp()
    Case leverPosition Of
        Case 1: Off
            leverPosition = Int
            Case dialPosition Of
                Case 1: 1
                    wiperSpeed = 4
                Case 2: 2
                    wiperSpeed = 6
                Case 3: 3
                    wiperSpeed = 12
            EndCase 'dialPosition
        Case 2: Int
            leverPosition = Low
            wiperSpeed = 30
        Case 3: Low
            leverPosition = High
            wiperSpeed = 60
        Case 4: High
            (impossible; error condition)
    EndCase 'leverPosition
```

Methods for testing windshield wiper

Test the get/set methods

- ☆ wiper Speed
- ☆ leverPosition
- ☆ dialPosition

Test Case	Preconditions	Method	Expected Value of leverPos
1	windshieldWiper(0,Off,1)	senseLeverUp()	INT
2	windshieldWiper(0,Int,1)	senseLeverUp()	LOW
3	windshieldWiper(0,Low,1)	senseLeverUp()	HIGH
4	windshieldWiper(0,High,1)	senseLeverDown()	LOW
5	windshieldWiper(0,Low,1)	senseLeverDown()	INT
6	windshieldWiper(0,Int,1)	senseLeverDown()	OFF

Pros and Cons of Class Testing

1. Creation of drivers and stubs add to overhead.

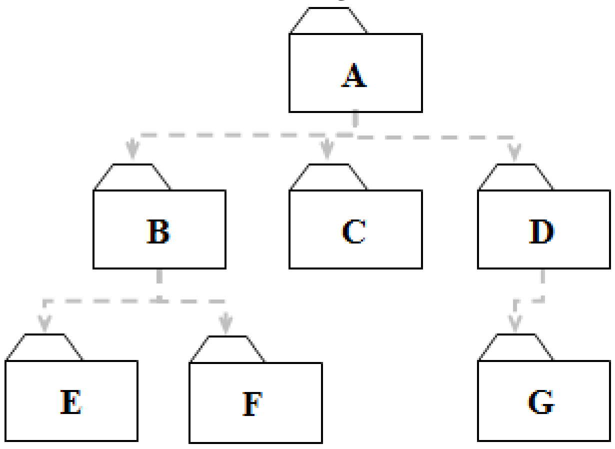
OO INTEGRATION TESTING

Need for Integration Testing:

- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- Often many Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

Definition:

- ❒ Integration testing is the testing performed to catch any errors when two or more individually developed components are aggregated to execute their functionalities.
- ❒ The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.
- ❒ Integration testing is the phase in software testing in which individual software modules are combined and tested as a group.
- ❒ It occurs after unit testing and before validation testing.



- ❒

- ❒ Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing
- ❒ Integration applied three different incremental strategies
    - **Thread-based testing:** integrates classes required to respond to one input or event
    - **Use-based testing:** integrates classes required by one use case
    - **Cluster testing:** integrates classes required to demonstrate one collaboration

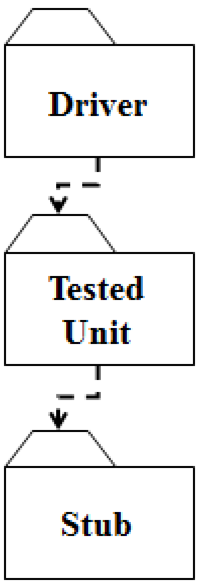
Components of Integration testing:

Here are some views about integration testing:

1) The gradual replacement of stubs and drivers by separately tested units
2) bottom-up integration guided by the functional decomposition tree
3) top-down integration guided by the functional decomposition tree
4) “big bang” integration where all units are thrown together at once

Stubs and Drivers

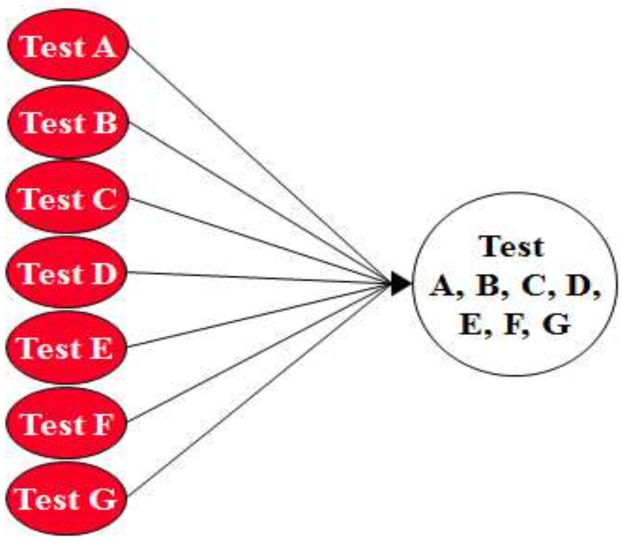
- Driver:
    - A component, that calls the TestedUnit
    - Controls the test cases
- Stub:
    - A component, the TestedUnit depends on
    - Partial implementation
    - Returns fake values.



Big Bang Approach:

- ★ In this approach, most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing.
- ★ The Big Bang method is very effective for saving time in the integration testing process.

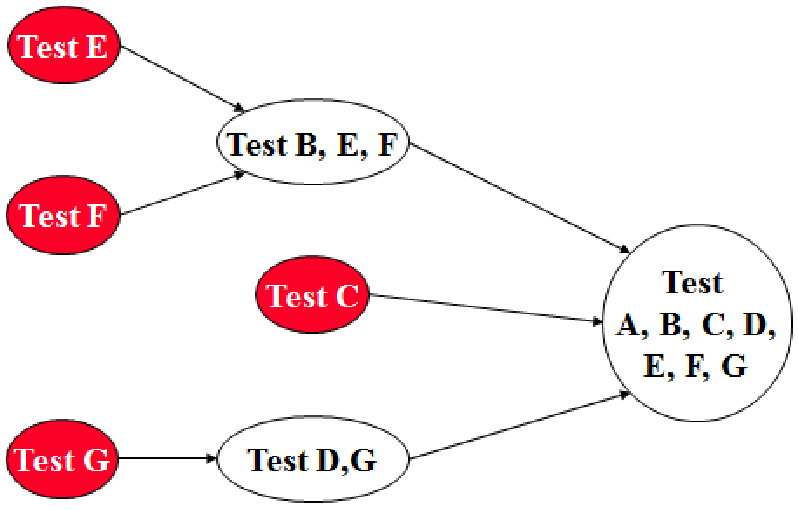
★ However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing.



Bottom up testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is repeated until all subsystems are included

Drivers are needed.



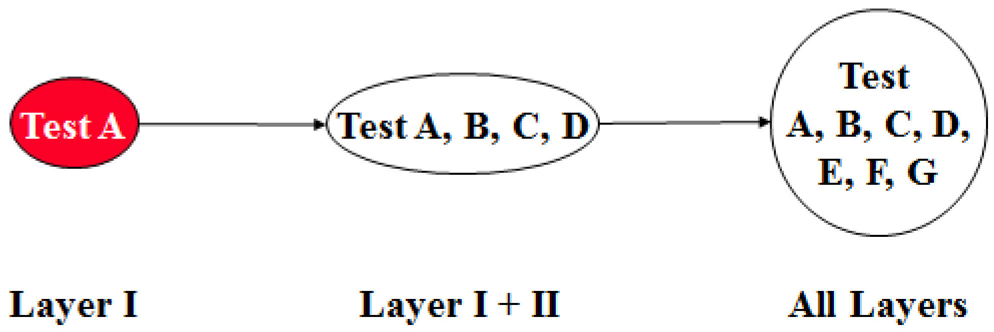
Pros and Cons:

- Con:
  - Tests the most important subsystem (user interface) last
  - Drivers needed
- Pro
  - No stubs needed

- Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements.

Top down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Stubs are needed to do the testing.



Pros and Cons:

Pro

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

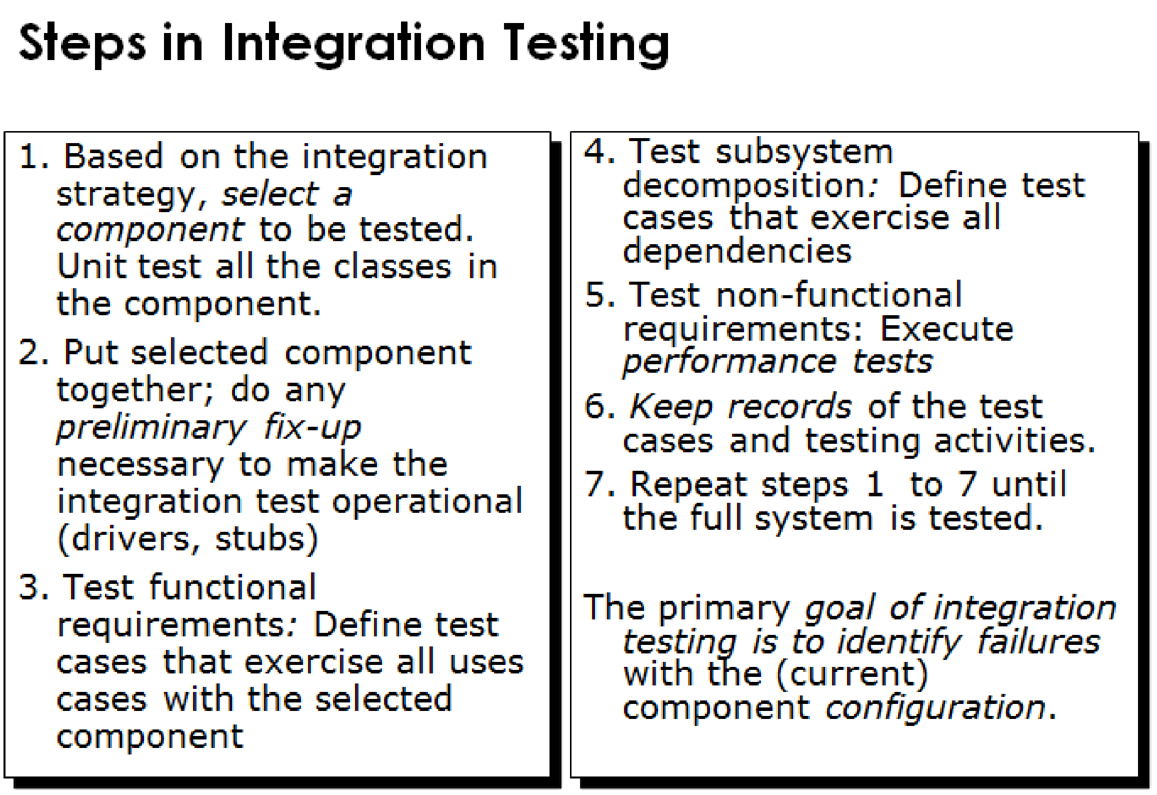
Cons

- Writing stubs is difficult: Stubs must allow all possible conditions to be tested.
- Large number of stubs may be required, especially if the lowest level of the system contains many methods.

Some interfaces are not tested separately

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- The system is viewed as having three layers
    - A target layer in the middle
    - A layer above the target
    - A layer below the target
- Testing converges at the target layer.



Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements*:* Define test cases that exercise all uses cases with the selected component
4. Test subsystem decomposition*:* Define test cases that exercise all dependencies
5. Test non-functional requirements: Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify failures* with the (current) component *configuration*.

Common Integration Testing Techniques:

a) **State based Testing:** State-based testing techniques rely on the construction of a finite-state machine (FSM) or state-transition diagram to represent the change of states of the program under test.

b) **Event-Based techniques:** Instead of using the state-based approach, the synchronization sequence for a concurrent program can also be viewed as relationships between pairs of synchronization events.

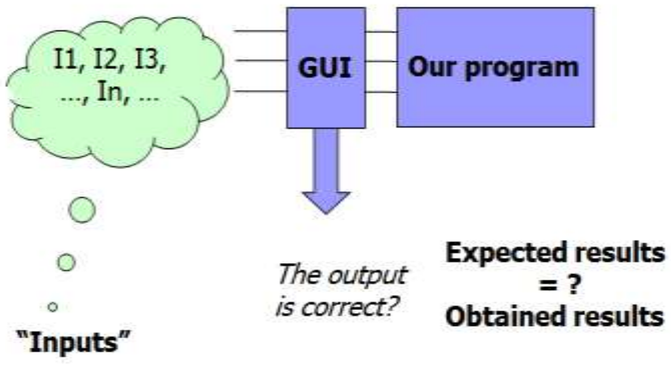
c) **Integrated formal methods** appear to provide a promising track. However, the complexity of the combined formal models may be a serious concern for people in the industry.

d) **Deterministic testing** forces the synchronization to be executed in desirable orders so that a deterministic test oracle can be applied and checked with the deterministic result. Because of dynamic binding, the same synchronization may result in different binding effects at different points of input, thus causing confusion

e) For **fault-based testing**, the effectiveness of mutation rules for integration testing of object-oriented programs has yet to be studied.

GUI TESTING

Graphical User interface testing is used to identify the presence of defects is a product/software under test by using Graphical user interface [GUI].



What is GUI? :

- ❒ A GUI (Graphical User Interface) is a **hierarchical, graphical front end** to a software system.
- ❒ A GUI contains graphical objects w, called **widgets**, each with a set of properties p, which have discrete values v at run-time.
- ❒ A **graphical event e** is a state transducer, which yields the GUI from a state S to the next state S'.

Types of GUI Testing:

GUI testing falls into two categories.

1. **Usability testing** refers to assessing how usable the interface is, using principles from user interface design.
2. **Functional testing** refers to assessing whether the user interface works as intended.

GUI Testing Process:

1) Identify the testing objective by defining a coverage criteria
2) Generate test cases from GUI structure, specification, model
    a. Generate sequences of GUI events
    b. Complete them with inputs and expected oracles
3) Define executable test cases
4) Run them and check the results

GUI Testing difficulties:

- ☆ GUI test automation is difficult
- ☆ Often GUI test automation **is technology-dependent**
- ☆ Observing and trace GUI states is difficult
- ☆ UI **state explosion problem**
    - o A lot of possible states of the GUI
- ☆ Controlling GUI events is difficult

Problems with Test Case generation for GUI Testing:

To generate a set of test cases, test designers attempt to cover all the functionality of the system and fully exercise the GUI itself. The difficulty in accomplishing this task is twofold: to deal with domain size and with sequences.

1. **Domain size:** Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A relatively small program such as Microsoft WordPad has 325 possible GUI operations. In a large program, the number of operations can easily be an order of magnitude larger.
2. **Sequencing Problem:** Some functionality of the system may only be accomplished with a sequence of GUI events. For example, to open a file a user may have to first click on the File Menu, and then select the Open operation, use a dialog box to specify the file name, and focus the application on the newly opened window. Increasing the number of possible operations increases the sequencing problem exponentially. This can become a serious issue when the tester is creating test cases manually.

GUI Testing - Approaches:

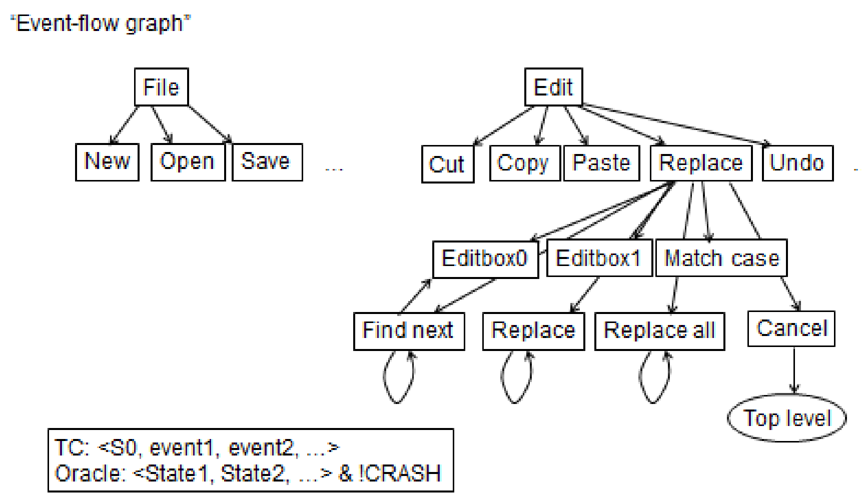
The various approaches in GUI testing are:

1. **Manual based :** Based on the domain and application knowledge of the tester
2. **Capture and Replay** : Based on capture and replay of user sessions
3. **Model-based testing :** Based on the execution of user sessions selected from a model of the GUI
    i. Which type of model to use?
        a. Event-based model
        b. State-based model
        c. Domain model

Model Based Testing:

- Event-based model - Based on all events of the GUI need to be executed at least once.
- State-based model - "all states" of the GUI are to be exercised at least once.
- Domain model - Based on the application domain and its functionality.

1. **Event-Based Model:**
    - Model the space of GUI event interactions as a graph
    - Given a GUI:
        - create a graph model of all the possible sequences that a user can execute
        - use the model to generate event sequences



In this example, the event types:

- ☐ Structural events (Edit, Replace)
- ☐ Termination events (Ok, cancel)
- ☐ System interaction events (Editbox0, Find next)

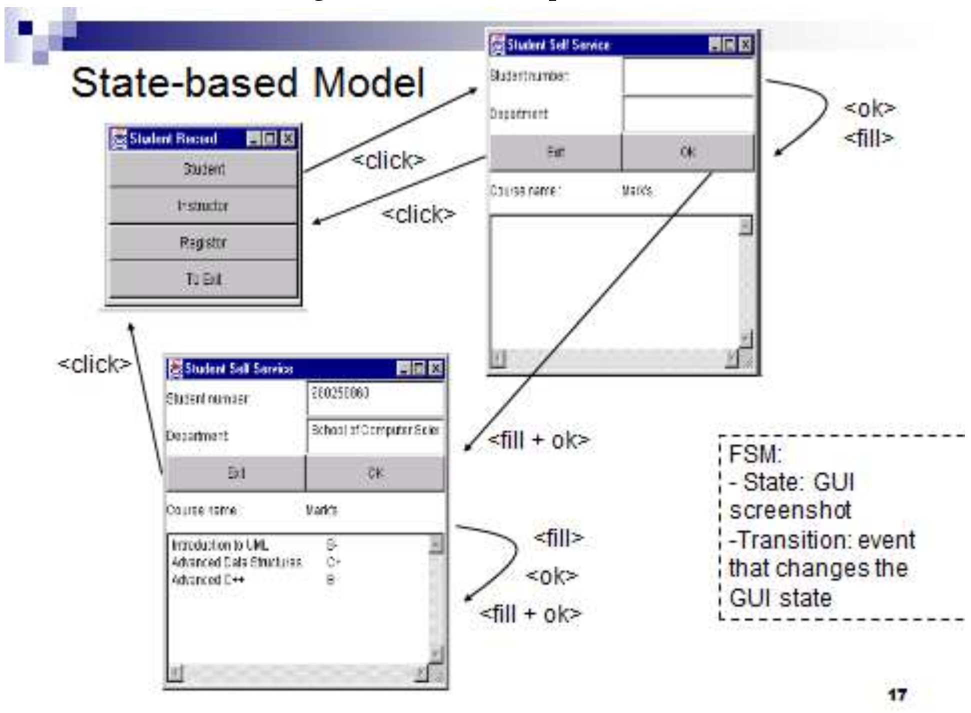
2. **State Based Model**

   Model the space of GUI event interactions as a state model, e.g., by using a finite state machine (FSM):

   - States are screenshot/representation of the GUI
   - Transitions are GUI events that change the GUI state

   Given a GUI:

   1. create a FSM of the possible sequences that a user can execute, considering the GUI state
   2. use the FSM to generate event sequences



GUI Testing Checklist:

- Check Screen Validations
- Verify All Navigations
- Check usability Conditions
- Verify Data Integrity
- Verify the object states

Coverage Criteria for GUI testing:

Possible coverage criteria include:

- Event-coverage: all events of the GUI need to be executed at least once
- State-coverage: "all states" of the GUI need to be exercised at least once
- Functionality-coverage: using a functional point of view

Running the test cases:

At first the strategies were migrated and adapted from the CLI testing strategies.

Mouse position capture

Capture playback is a system where the system screen is "captured" as a bitmapped graphic at various times during system testing. This capturing allowed the tester to "play back" the testing process and compare the screens at the output phase of the test with expected screens.

Event capture

By capturing the window 'events' into logs the interactions with the system are now in a format that is decoupled from the appearance of the GUI. Now, only the event streams are captured. There is some filtering of the event streams necessary since the streams of events are usually very detailed and most events aren't directly relevant to the problem

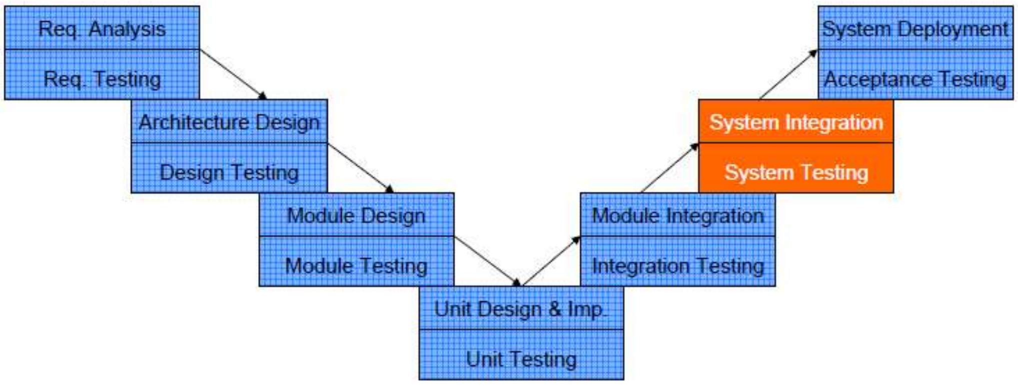
SYSTEM TESTING

Definition:

- **OO System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements?**
- System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

- In System testing, the functionalities of the system are tested from an end-to-end perspective.
- System Testing is usually carried out by a team that is independent of the development team in order to measure the quality of the system unbiased.
- It includes both functional and Non-Functional testing.

System Testing in OO testing:



OO System testing:

Therefore OO system testing includes:

- Functional Testing - Validates functional requirements
- Performance Testing -Validates non-functional requirements
- Acceptance Testing - Validates clients expectations

1. Functional testing :

Goal: Test functionality of system

- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box
- Unit test cases can be reused, but new test cases have to be developed as well.

2. Performance Testing

Goal: Try to violate non-functional requirements

- Test how the system behaves when overloaded.
- Try unusual orders of execution
  - Call a receive() before send()
- Check the system's response to large volumes of data
  - If the system is supposed to handle 1000 items, try it with 1001 items.
- What is the amount of time spent in different use cases?

Are typical cases executed in a timely fashion?

Types of Performance Testing:

1. Compatibility test
   a. Test backward compatibility with existing systems

   b. Compatibility testing is conducted on the application to evaluate the application's compatibility with the computing environment
2. Configuration testing
   a. Test the various software and hardware configurations
3. Human factors testing
   a. Test with end users.
4. Quality testing
   a. Test reliability, maintain- ability & availability
5. Recovery testing
   a. Test system's response to presence of errors or loss of data
   b. Examples of recovery testing:
      i. While an application is running, suddenly restart the computer, and afterwards check the validness of the application's data integrity.
6. Security testing
   a. Try to violate security requirements.
   b. Security testing is a process intended to reveal flaws in the security mechanisms of an information system that protect data and maintain functionality as intended.
7. Stress Testing
   a. Stress limits of system
8. Timing testing
   a. Evaluate response times and time to perform a function
9. Volume testing
   a. Test what happens if large amounts of data are handled.
   b. For example, if you want to volume test your application with a specific database size, you will expand your database to that size and then test the application's performance on it.

Generation of Test cases for OO System Testing:

There are three ways to generate test cases for OO System testing:

1. Use UML description
   – Generate list of System Functions
2. Generate & Expand Use Cases
   – High Level Use Cases (HLUC)
   – Essential Use Cases (EUC)
   – Expanded Essential Use Cases (EEUC)
   – Real Use Cases (RUC)
3. Generate Test Cases from Real Use Cases

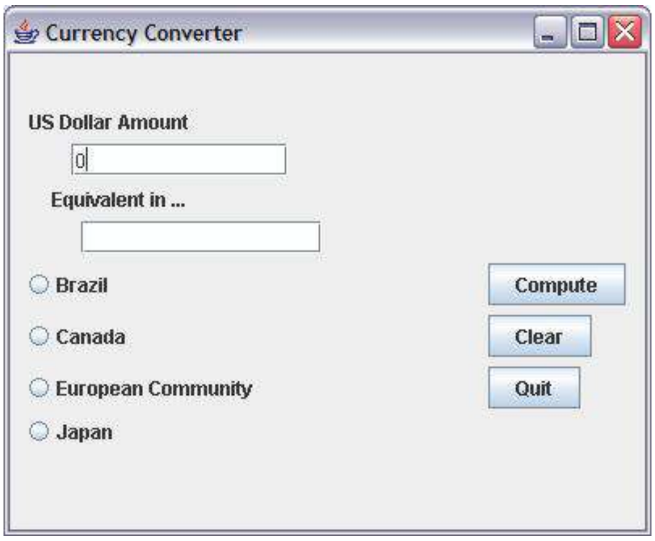
The steps are summarized as:

1. Generated a list of system functions
2. Developed a set of HLUCs
3. Extended these to create a set of EUCs

4. Create a detailed GUI definition
5. Use this to generate the Expanded Essential Use Cases (EEUC)
6. Generate Real Use Cases (RUC)
7. Generate System Test Cases (SysTC)

Consider an example scenario: **A currency Converter Application**

- It Converts US Dollars to:
    - Brazilian real (R$)
    - Canadian dollars (C$)
    - European Union euros (€)
    - Japanese yen (¥)
- User can revise inputs
- User can perform repeated conversions

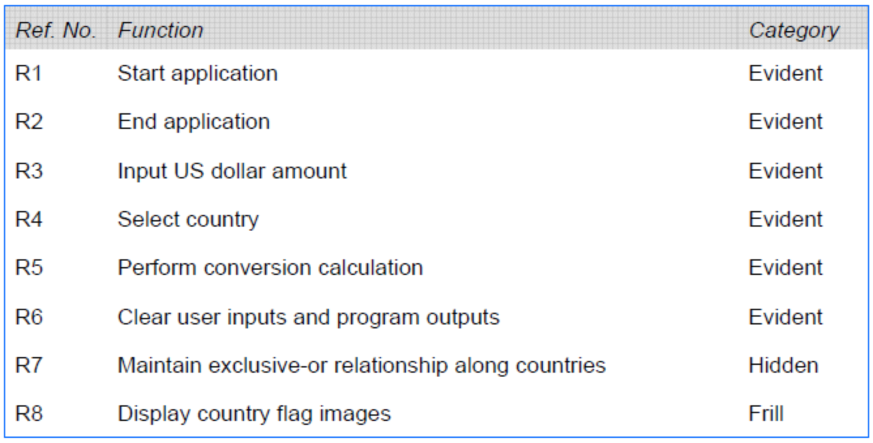


Now the various methods of generating use cases are explained:

1. Generate a list of System Functions

1. In this method, the functions of the system as the user describes them
2. Developed from the UML Specification
3. Identifies 3 types of functions:
    - Evident – Obvious to the user
    - Hidden – Not immediately obvious
    - Frill

So the set of system Functions are identified as:



Ref. No.	Function	Category
R1	Start application	Evident
R2	End application	Evident
R3	Input US dollar amount	Evident
R4	Select country	Evident
R5	Perform conversion calculation	Evident
R6	Clear user inputs and program outputs	Evident
R7	Maintain exclusive-or relationship along countries	Hidden
R8	Display country flag images	Frill

2. Generation and Expansion of Use Cases

- Describe the functional requirements of a system
- Each Use Case describes a scenario
- Shows how the system should interact with the user (actor)
- Several levels of use cases
  - High Level
  - Essential
  - Expanded Essential
  - Real

High Level Use Cases (HLUC)

- Brief description of the main functions of the system
- High level view of program
- Very few details shown
  - Name of function
  - Actors involved
  - Type of use case
  - Description of function

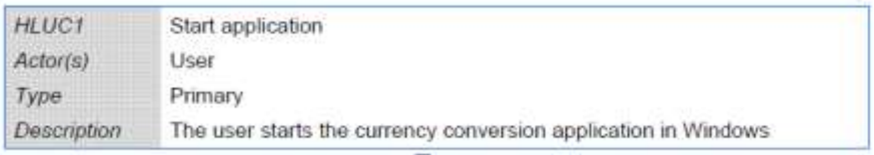
Essential Use Cases (EUC)

- Identifies what user expects to happen
- Adds “actor” and “system” events to the HLUC
- Actions / Responses are numbered
- Numbers show approximate sequence in time

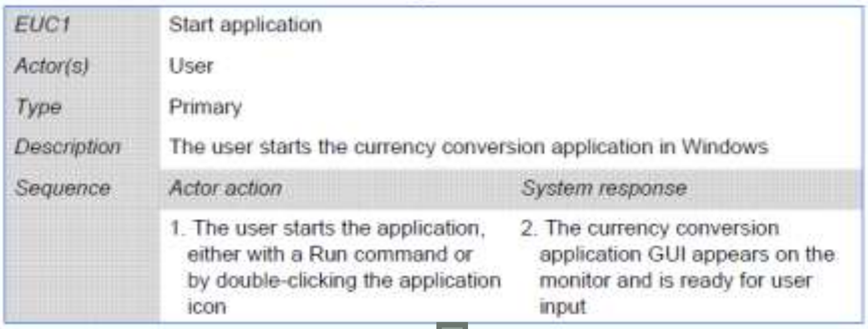
Expanded Essential Use Cases (EEUC)

- Next level of Use Case refinement
- Detailed description of processes involved
- Adds:
  - Pre / Post conditions

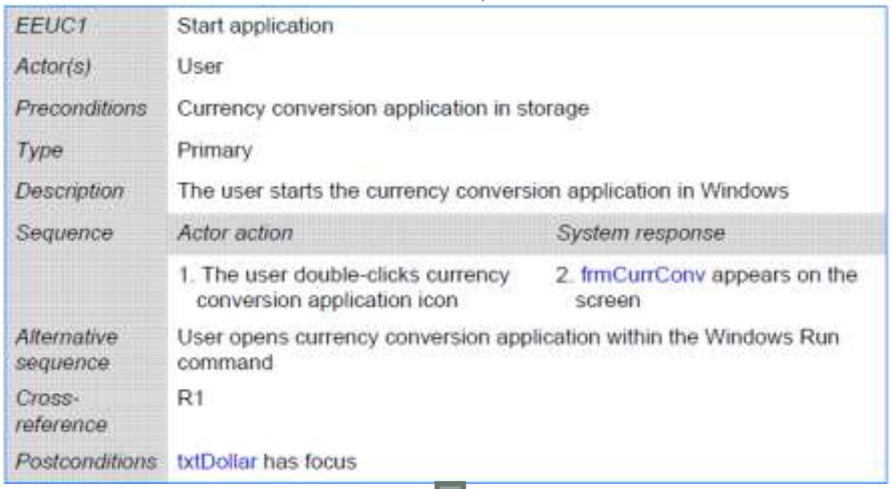
- Alternative sequences of events
- References system functions found earlier



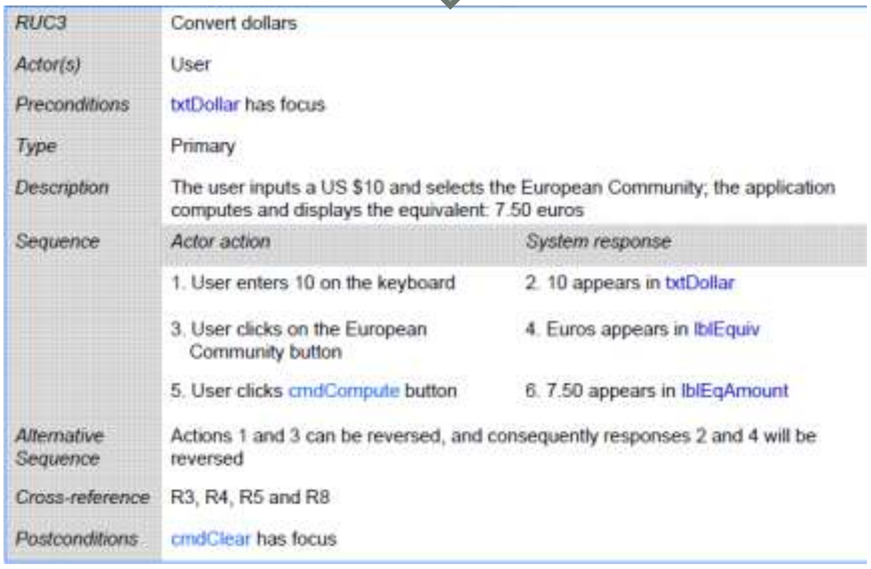
HLUC1	Start application
Actor(s)	User
Type	Primary
Description	The user starts the currency conversion application in Windows



EUC1	Start application	
Actor(s)	User	
Type	Primary	
Description	The user starts the currency conversion application in Windows	
Sequence	*Actor action*	*System response*
	1. The user starts the application, either with a Run command or by double-clicking the application icon	2. The currency conversion application GUI appears on the monitor and is ready for user input



EEUC1	Start application	
Actor(s)	User	
Preconditions	Currency conversion application in storage	
Type	Primary	
Description	The user starts the currency conversion application in Windows	
Sequence	*Actor action*	*System response*
	1. The user double-clicks currency conversion application icon	2. frmCurrConv appears on the screen
Alternative sequence	User opens currency conversion application within the Windows Run command	
Cross-reference	R1	
Postconditions	txtDollar has focus	



RUC3	Convert dollars	
Actor(s)	User	
Preconditions	txtDollar has focus	
Type	Primary	
Description	The user inputs a US $10 and selects the European Community; the application computes and displays the equivalent: 7.50 euros	
Sequence	*Actor action*	*System response*
	1. User enters 10 on the keyboard	2. 10 appears in txtDollar
	3. User clicks on the European Community button	4. Euros appears in lblEquiv
	5. User clicks cmdCompute button	6. 7.50 appears in lblEqAmount
Alternative Sequence	Actions 1 and 3 can be reversed, and consequently responses 2 and 4 will be reversed	
Cross-reference	R3, R4, R5 and R8	
Postconditions	cmdClear has focus	

Thus the test cases are generated for OO System Testing.

Mapping Design to Code:
Refer Technical and pdf
All the best