**Object Oriented Testing**

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-[unit testing](https://www.geeksforgeeks.org/unit-testing-software-testing/), small “units”, or modules of the software, are tested separately with focus on testing the code of that module. In higher, order testing (e.g, [acceptance testing](https://www.geeksforgeeks.org/acceptance-testing-software-testing/)), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behaviour of the system.

As information systems are becoming more complex, the object-oriented paradigm is gaining popularity because of its benefits in analysis, design, and coding. Conventional testing methods cannot be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message, passing, polymorphism, concurrency, etc.  
Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behaviour, while a class does not have an input-output behaviour specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these  
approaches.

According to Davis the dependencies occurring in conventional systems are:

* Data dependencies between variables
* Calling dependencies between modules
* Functional dependencies between a module and the variable it computes
* Definitional dependencies between a variable and its types.

But in Object-Oriented systems there are following additional dependencies:

* Class to class dependencies
* Class to method dependencies
* Class to message dependencies
* Class to variable dependencies
* Method to variable dependencies
* Method to message dependencies
* Method to method dependencies

**Issues in Testing Classes:**

Additional testing techniques are, therefore, required to test these dependencies. Another issue of interest is that it is not possible to test the class dynamically, only its instances i.e, objects can be tested. Similarly, the concept of inheritance opens various issues e.g., if changes are made to a parent class or super class, in a larger system of a class it will be difficult to test subclasses individually and isolate the error to one class.

In object-oriented programs, control flow is characterized by message passing among objects, and the control flow switches from one object to another by inter-object communication. Consequently, there is no control flow within a class like functions. This lack of sequential control flow within a class requires different approaches for testing. Furthermore, in a function, arguments passed to the function with global data determine the path of execution within the procedure. But, in an object, the state associated with the object also influences the path of execution, and methods of a class can communicate among themselves through this state because this state is persistent across invocations of methods. Hence, for testing objects, the state of an object has to play an important role.

Techniques of object-oriented testing are as follows:

1. **Fault Based Testing:**  
   This type of checking permits for coming up with test cases supported the consumer specification or the code or both. It tries to identify possible faults (areas of design or code that may lead to errors.). For all of these faults, a test case is developed to “flush” the errors out. These tests also force each time of code to be executed.

This method of testing does not find all types of errors. However, incorrect specification and interface errors can be missed. These types of errors can be uncovered by function testing in the traditional testing model. In the object-oriented model, interaction errors can be uncovered by scenario-based testing. This form of Object oriented-testing can only test against the client’s specifications, so interface errors are still missed.

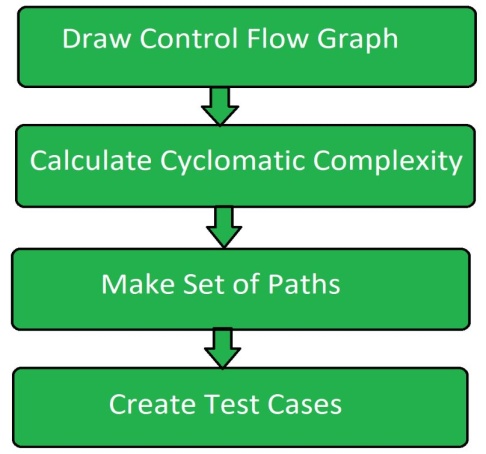
1. **Class Testing Based on Method Testing:**  
   This approach is the simplest approach to test classes. Each method of the class performs a well defined cohesive function and can, therefore, be related to unit testing of the traditional testing techniques. Therefore all the methods of a class can be involved at least once to test the class.
2. **Random Testing:**  
   It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behaviour of the categories.
3. **Partition Testing:**  
   This methodology categorizes the inputs and outputs of a category so as to check them severely. This minimizes the number of cases that have to be designed.
4. **Scenario-based Testing:**  
   It primarily involves capturing the user actions then stimulating them to similar actions throughout the test.  
   These tests tend to search out interaction form of error.

**Path Testing**

Path Testing is a method that is used to design the test cases. In path testing method, the control flow graph of a program is designed to find a set of linearly independent paths of execution. In this method Cyclomatic Complexity is used to determine the number of linearly independent paths and then test cases are generated for each path.

It gives complete branch coverage but achieves that without covering all possible paths of the control flow graph. McCabe’s Cyclomatic Complexity is used in path testing. It is a structural testing method that uses the source code of a program to find every possible executable path.

**Path Testing Process:**

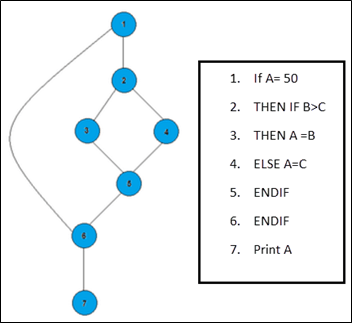


* **Control Flow Graph:**  
  Draw the corresponding control flow graph of the program in which all the executable paths are to be discovered.
* **Cyclomatic Complexity:**  
  After the generation of the control flow graph, calculate the cyclomatic complexity of the program using the following formula.
* McCabe's Cyclomatic Complexity = E - N + 2P
* Where,
* E = Number of edges in control flow graph
* N = Number of vertices in control floe graph

P = Program factor

* **Make Set:**  
  Make a set of all the path according to the control floe graph and calculated cyclomatic complexity. The cardinality of set is equal to the calculated cyclomatic complexity.
* **Create Test Cases:**  
  Create test case for each path of the set obtained in above step.

Here we will take a simple example, to get a better idea what is basis path testing include.

[](https://www.guru99.com/images/3-2016/032216_1047_LearnPathBa1.png)

In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that needs to be tested to get the output,

* **Path 1**: 1,2,3,5,6, 7
* **Path 2**: 1,2,4,5,6, 7
* **Path 3**: 1, 6, 7

## Steps for Basis Path testing

The basic steps involved in basis path testing include

* Draw a control graph (to determine different program paths)
* Calculate Cyclomatic Complexity  (metrics to determine the number of independent paths)
* Find a basis set of paths
* Generate test cases to exercise each path

**Path Testing Techniques:**

* **Control Flow Graph:**  
  The program is converted into control flow graph by representing the code into nodes and edges.
* **Decision to Decision path:**  
  The control flow graph can be broken into various Decision to Decision paths and then collapsed into individual nodes.
* **Independent paths:**  
  Independent path is a path through a Decision to Decision path graph which cannot be reproduced from other paths by other methods.

**Advantages of Path Testing:**

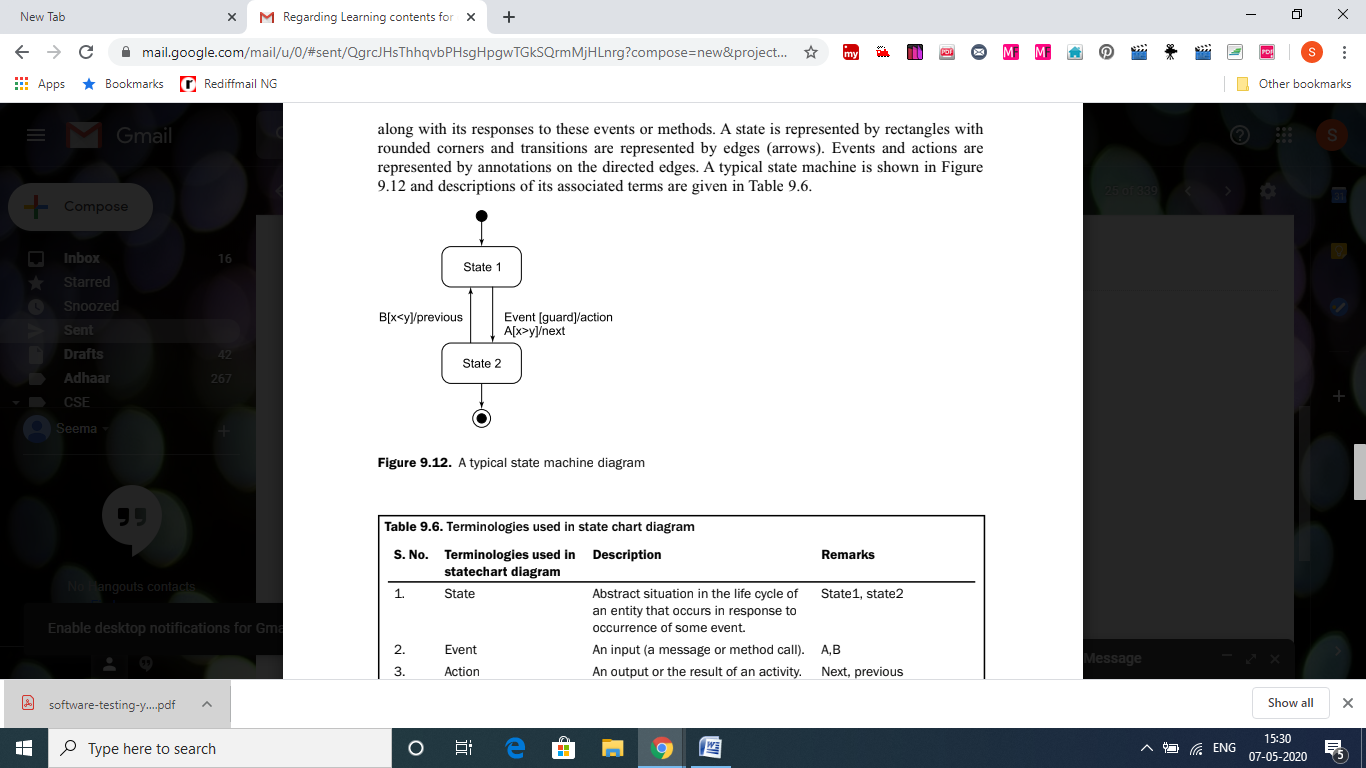
* + 1. Path testing method reduces the redundant tests.
    2. Path testing focuses on the logic of the programs.
    3. Path testing is used in test case design.

**STATE BASED TESTING**

State based testing is used as one of the most useful object oriented software testing techniques. It uses the concept of state machine of electronic circuits where the output of the state machine is dependent not only on the present state but also on the past state. A state represents the effect of previous inputs. Hence, in state machine, the output is not only dependent on the present Inputs but also on the previous inputs. In electronic circuits, such circuits are called sequential circuits. If the output of a state is only dependent on present inputs, such circuits are called combinational circuits. In state based testing, the resulting state is compared with the expected state.

**What is a State Machine?**

State machines are used to model the behaviour of objects. A state machine represents various states which an object is expected to visit during its lifetime in response to events or methods along with its responses to these events or methods. A state is represented by rectangles with rounded corners and transitions are represented by edges (arrows). Events and actions are represented by annotations on the directed edges. A typical state machine is shown in Figure.



We consider an example of a process i.e. program under execution that may have the following states:

**» New**: The process is created

**» Ready**: The process is waiting for the processor to be allocated.

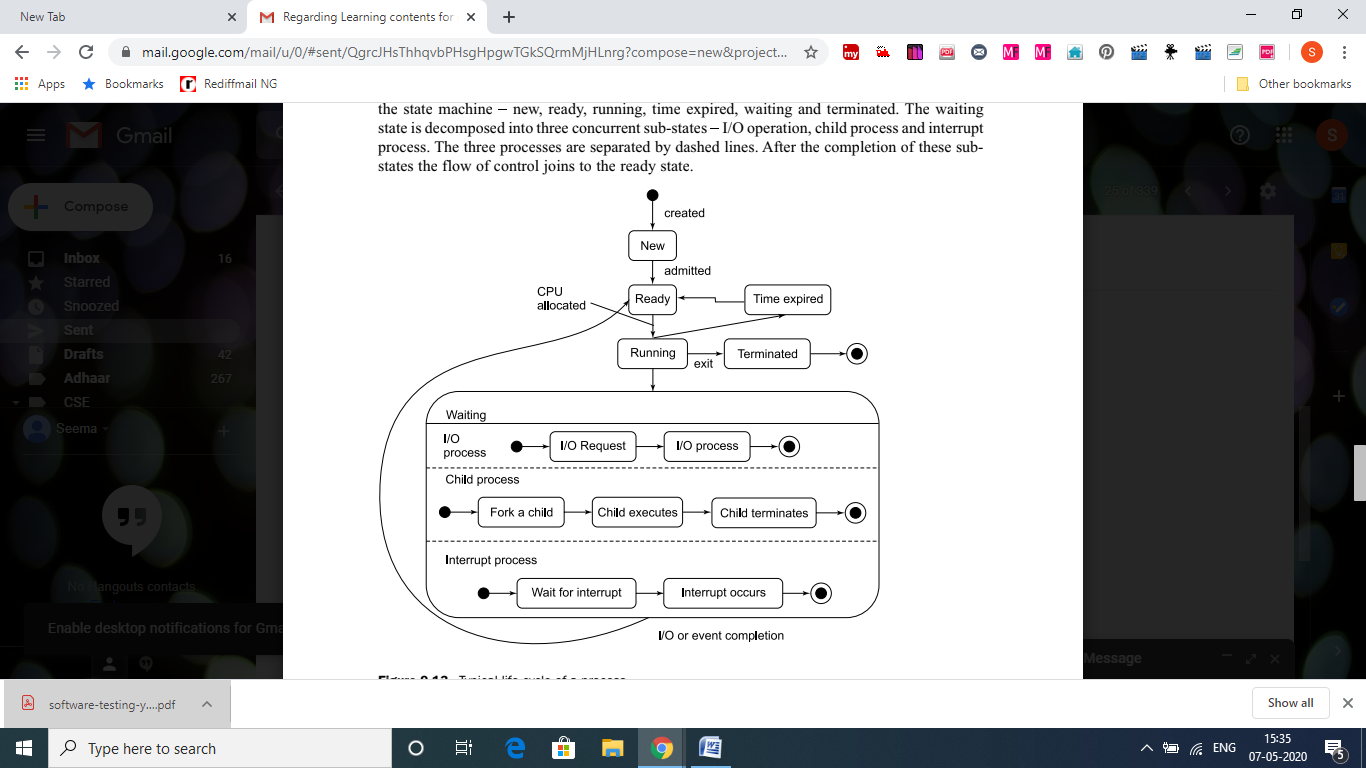
**» Running**: The process is allocated to the processor and is being executed.

**» Time expired**: The time allocated to the process in execution expires.

**» Waiting**: The process is waiting for some I/O or event to occur.

**» Terminated**: The process under execution has completed.

The state machine for life cycle of a process is shown in following Figure. There are six states in the state machine new, ready, running, time expired, waiting and terminated. The waiting state is decomposed into three concurrent sub-states I/O operation, child process and interrupt process. The three processes are separated by dashed lines. After the completion of these sub-states the flow of control joins to the ready state.



**State Chart Diagram**

In Unified Modeling Language (UML), a state machine is graphically represented by a state chart diagram. It shows the flow of control from one state to another state. Here too, states are represented by rectangles with rounded corners and transitions are represented by edges (arrows).

Two special states are used i.e. (alpha) and (omega) state for representing the constructor and destructor of a class. These states may simplify testing of multiple constructors, exception handling and destructors. Binder [BIND99] has explained this concept very effectively as:

“The alpha state is a null state representing the declaration of an object before its construction. It may accept only a constructor, new, or a similar initialization

message. The state is reached after an object has been destructed or deleted, or has gone out of scope. It allows for explicit modelling and systematic testing of destructors, garbage collection, and other termination actions.” Alpha and omega states are different from start state and end state of a state chart diagram.

These are additional states to make things more explicit and meaningful. We considers an example of a class ‘stack’ where two operations push and pop, are allowed. The functionality of a stack suggests three states empty, holding and full. There are four events **new, push, pop** and **destroy,** with the following purposes:

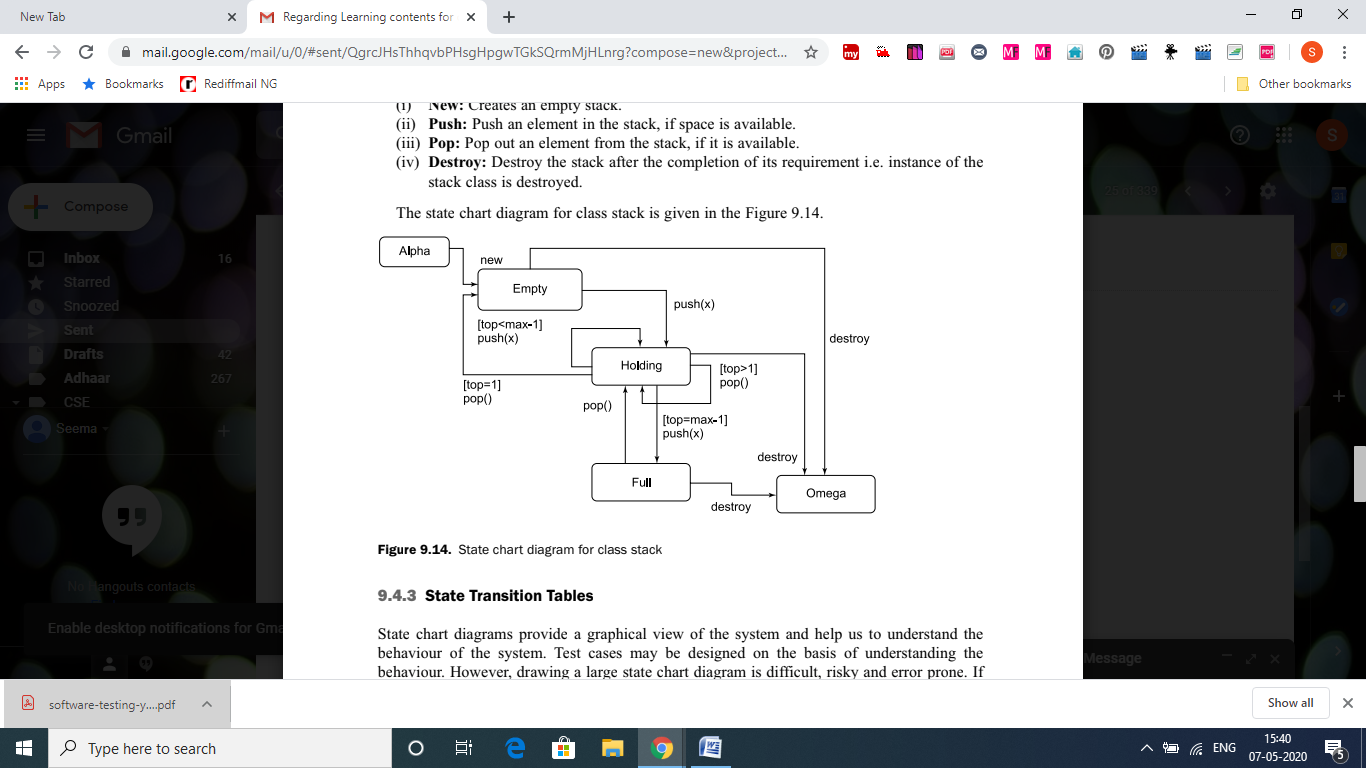
(i) **New**: Creates an empty stack.

(ii) **Push:** Push an element in the stack, if space is available.

(iii) **Pop**: Pop out an element from the stack, if it is available.

(iv)**Destroy**: Destroy the stack after the completion of its requirement i.e. instance of the stack class is destroyed.

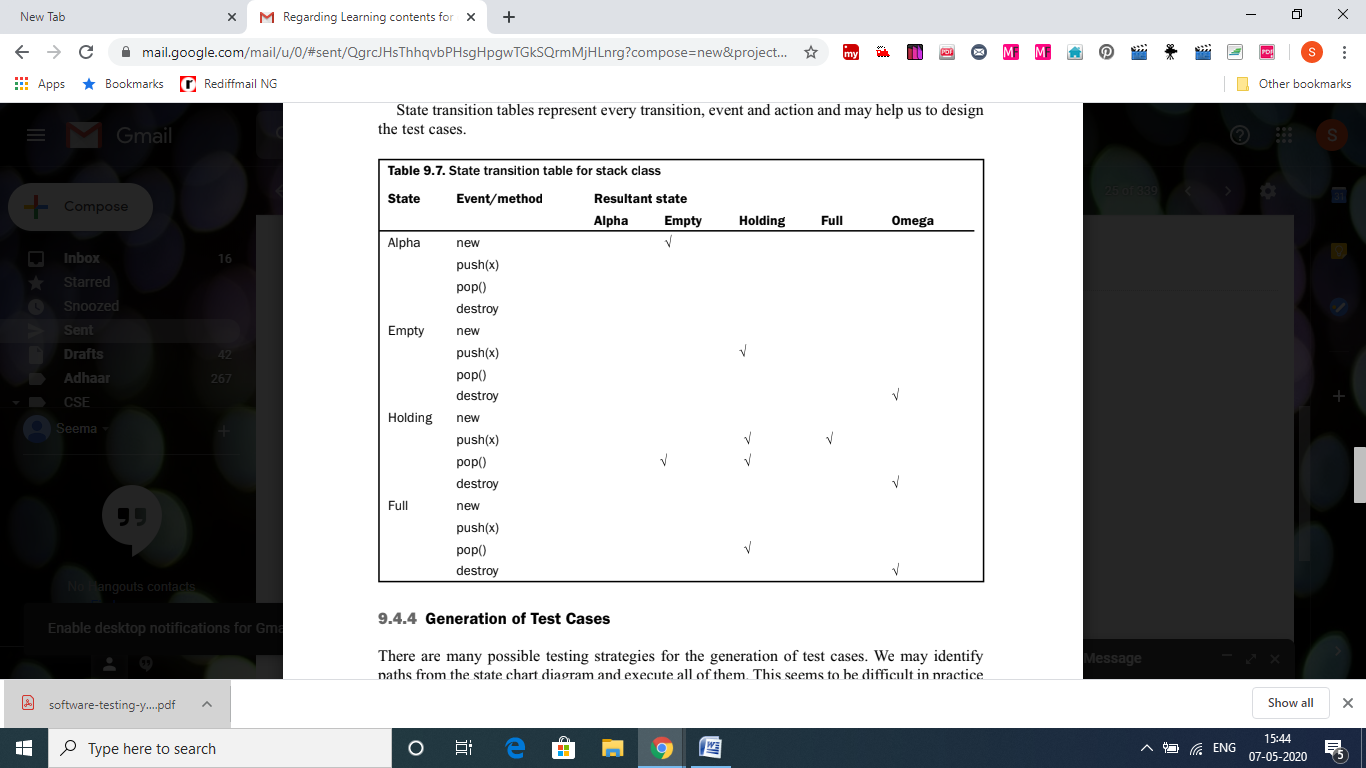
The state chart diagram for class stack is given in the Figure.



**State Transition Tables**

State chart diagrams provide a graphical view of the system and help us to understand the behaviour of the system. Test cases may be designed on the basis of understanding the behaviour. However, drawing a large state chart diagram is difficult, risky and error prone. If states are more than 10 or 15, it is difficult to keep track of various transitions. In practice, we may have to handle systems with 100 states or more. State transition tables are used when the

number of states is more these tables and provide information in a compact tabular form. In state transition tables, rows represent the present acceptable state and columns represent the resultant state. State transition tables represent every transition, event and action and may help us to design the test cases.



**CLASS TESTING**

A class is very important in object oriented programming. Every instance of a class is known

as an object. Testing of a class is very significant and critical in object oriented testing where

we want to verify the implementation of a class with respect to its specifications. If the

implementation is as per specifications, then it is expected that every instance of the class may behave in the specified way. Class testing is similar to the unit testing of a conventional system. We require stubs and drivers for testing a ‘unit’ and sometimes, it may require significant effort. Similarly, classes also cannot be tested in isolation. They may also require additional source code (similar to stubs and drivers) for testing independently.

**How Should We Test a Class?**

We want to test the source code of a class. Validation and verification techniques are equally

applicable to test a class. We may review the source code during verification and may be able

to detect a good number of errors. Reviews are very common in practice, but their effectiveness is heavily dependent on the ability of the reviewer(s).

Another type of testing is validation where we may execute a class using a set of test cases.

This is also common in practice but significant effort may be required to write test drivers and sometime this effort may be more than the effort of developing the ‘unit’ under test. After writing test cases for a class, we must design a test driver to execute each of the test cases and record the output of every test case. The test driver creates one or more instances of a class to execute a test case. We should always remember that classes are tested by creating instances and testing the behaviour of those instances [MCGR01].

**Issues Related to Class Testing**

How should we test a class? We may test it independently, as a unit or as a group of a system.

The decision is dependent on the amount of effort required to develop a test driver, severity of class in the system and associated risk with it and so on. If a class has been developed to be a part of a class library, thorough testing is essential even if the cost of developing a test driver is very high. Classes should be tested by its developers after developing a test driver. Developers are familiar with the internal design, complexities and other critical issues of a class under test and this knowledge may help to design test cases and develop test driver(s). Class should be testedwith respect to its specifications. If some unspecified behaviours have been implemented, we may not be able to test them. We should always be very careful for additional functionalities which are not specified. Generally, we should discourage this practice and if it has been implemented in the SRS document, it should immediately be specified. A test plan with a test suite may discipline the testers to follow a predefined path. This is particularly essential when developers are also the testers.

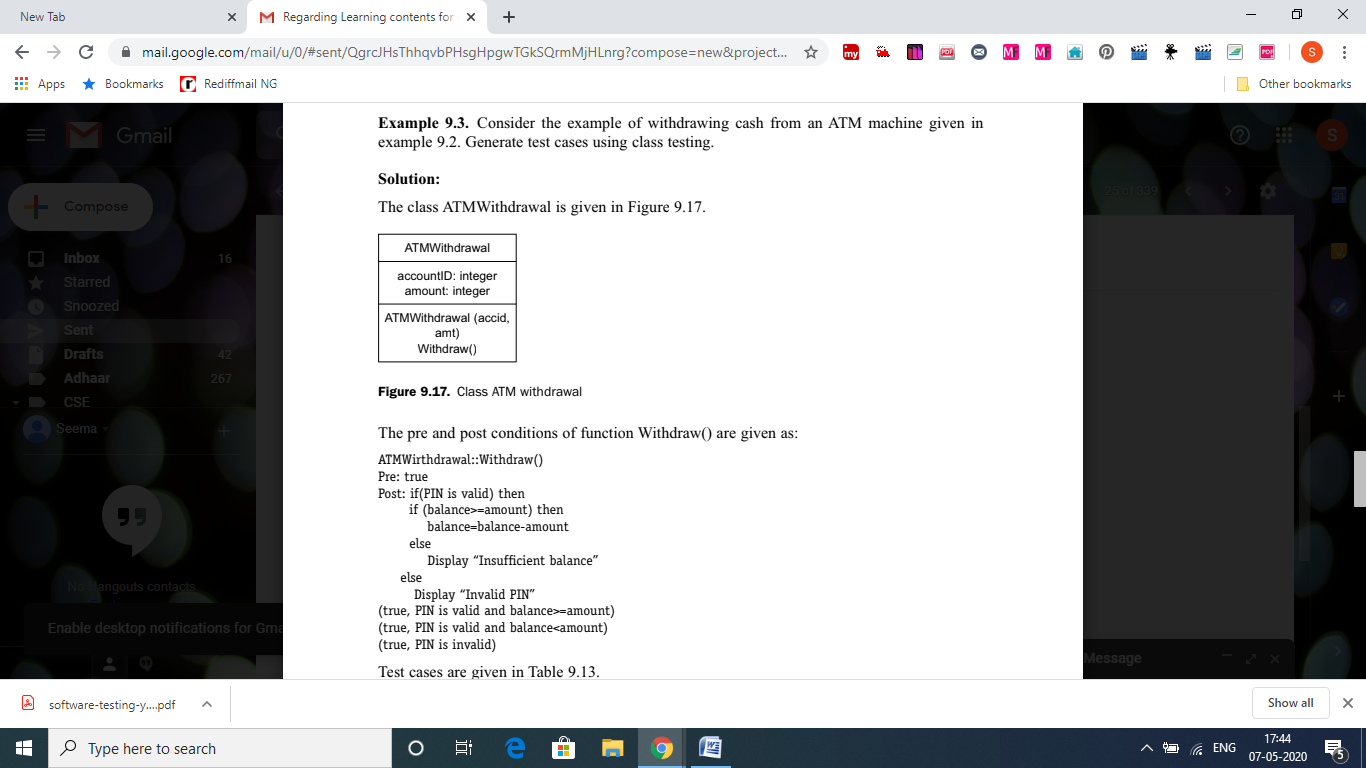
**Generating Test Cases**

One of the methods of generating test cases is from pre and post conditions specified in the use cases. As discussed in chapter 6, use cases help us to generate very effective test cases. The pre and post conditions of every method may be used to generate test cases for class testing. Every method of a class has a pre-condition that needs to be satisfied before the execution. Similarly, every method of a class has a post-condition that is the resultant state after the execution of the method. Consider a class ‘stack’ given in Figure 9.16 with two attributes (x and top) and three methods (stack(), push(x), pop()).

We Consider the example of withdrawing cash from an ATM machine . Generate test cases using class testing.

Solution:

The class ATM Withdrawal is given in following Figure.



The pre and post conditions of function Withdraw() are given as:

**ATM Withdrawal::Withdraw()**

**Pre: true**

**Post: if(PIN is valid) then**

**if (balance>=amount) then**

**balance=balance-amount**

**else**

**Display “Insufficient balance”**

**else**

**Display “Invalid PIN”**

**(true, PIN is valid and balance>=amount)**

**(true, PIN is valid and balance<amount)**

**(true, PIN is invalid)**

Test cases are given in following Table

