ECHLEON INSTITUTE OF TECHNOLOGY

SOFTWARE TESTING

UNIT 3

Testing object oriented software:

Object-oriented programming increases software reusability, extensibility, interoperability, and reliability. Software testing is necessary to realize these benefits by uncovering as many programming errors as possible at a minimum cost. A major challenge to the software engineering community remains how to reduce the cost while improving the quality of software testing. The requirements for testing object-oriented programs differ from those for testing conventional program

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-unit 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), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behavior 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 behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these
approaches.

Object – Oriented programming is centered on concepts like Object, Class, Message, Interfaces, Inheritance, Polymorphism etc., Traditional testing techniques can be adopted in Object Oriented environment by using the following techniques:

- Method testing
- Class testing
- Interaction testing
- System testing
- Acceptance testing

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 superclass, 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.

2. **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.
3. **Random Testing:**
   It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behavior of the categories
4. **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.
5. **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.

Object-oriented Testing in Software Testing

Overview

Initially, we were following the traditional method of testing which consists of procedures operating on data. But now, we are following the object-oriented testing method that focuses on the objects that are instances of classes. This change from traditional to object-oriented made us reconsider old strategies and software testing methods. There are many advantages of using the OO paradigm including reliability, interoperability, reusability, and extendibility.

Introduction

Different Levels of Object-Oriented Testing in Software Testing

Object Oriented testing can be performed at different levels to detect the issues. At the algorithmic level, a single module of every class should be tested. As discussed earlier, testing of classes is the main concern of the Object Oriented program. Every class gets tested as an individual entity at the class level. Generally, programmers who are creating the classes are involved in testing. Test cases for Object-Oriented Testing in Software Testing can be constructed based on the requirement specifications, programming language, and models.

Once class-level testing is done, Cluster level testing will be performed. Cluster-level testing is the integration of individual classes. The main purpose of doing integration testing is to verify the interconnection between classes and how well they perform interclass interactions. Thus, Cluster level testing can be viewed as integration testing of classes.

Once Cluster level testing is performed, system-level testing begins. At this level, integration between clusters can be taken care of. Also, at each level regression testing is a must after every new release.

Developing Test Cases in Object-oriented Testing

How to develop test cases in Object Oriented Testing in Software Testing?

Conventional methods can be used to design test cases in OO testing. However, these test cases can be redeveloped with some special features so that they can use for object-oriented environments. The following points should be considered while creating test cases for object-oriented environments.

- Which class is going to be tested should be mentioned properly within the test cases.
- What is the purpose of using particular test cases?
- What external pre-condition needs to be conducted while performing the test case?
- All the states should be specified for testing.

Object-Oriented Testing Levels /Techniques

Object Oriented Testing in Software Testing techniques are:

Fault-based testing: The main focus of fault-based testing is based on consumer specifications or code or both. Test cases are created in a way to identify all possible faults and flush them all. This technique finds all the defects that include incorrect specification and interface errors. In the traditional testing model, these types of errors can be detected through functional testing. While Object Oriented Testing in Software Testing will require scenario-based testing.

Scenario-based testing: This testing technique is useful to detect issues due to wrong specifications and improper interaction among the classes. Incorrect interactions lead to incorrect output which can cause the malfunctioning of some segments sometimes. Scenario-based testing focuses on how the end user will perform the task in a specific environment. These scenarios are more detailed and created concerning the user's requirements rather than product-specific.

Scenario-based testing combines all the classes included in created use cases and tests them all. The main purpose of this is to verify that all the methods of classes get tested at least once while performing Object Oriented Testing in Software Testing. However, it is difficult to test each object in a large system. Thus, the top-down or bottom-up integration approach gets followed.

The scenario-based testing technique has been considered the most effective method of the OO program.

Class Testing based on the method testing: This Object Oriented Testing in Software Testing can be considered the most simple and common approach. Each method of the class performs a proper cohesive function so that methods can be involved once during the testing.

To minimize the variety of operations, random sequence testing gets performed. It is less time-consuming and effective as well. Partition Testing: Inputs and outputs of the category get divided to minimize the number of test cases.

Challenges in Testing Object-oriented Programs

Issues with Object Oriented Testing in Software Testing

1. Dynamic testing of classes is not possible in an OO program because it allows instances of classes to be tested. Therefore, additional testing techniques are required to test the interconnection between classes.
2. In object-oriented programs, control flow can be monitored with message passing between objects. It changes from one object to another with intercommunication. To test these sequential flows different types of testing approaches will be required.
3. Inheritance of objects is an important part of the OO program. In a larger system, it is difficult to test the subclass and detect errors in one class.

4. The state associated with a particular object influences the methods, execution, and communication between classes. Therefore, the state plays a vital role in object-oriented testing in software testing.

Describe in detail the challenges of object oriented software testing.

The testing of software is an important means of assessing the software to determine its Quality. With the development of Fourth generation languages (4GL), which speeds up the implementation process, the proportion of time devoted to testing increased. As the amount of maintenance and upgrade of existing systems grow, significant amount of testing will also be needed to verify systems after changes are made. Most testing techniques were originally developed for the imperative programming paradigm, with relative less consideration to object-oriented features such as message passing, synchronization, dynamic binding, object instantiation, persistence, encapsulation, inheritance, and polymorphism. Objects may interact with one another with unforeseen combinations and invocations. The testing of concurrent object-oriented systems has become a most challenging task.

CHALLENGES OF OBJECT ORIENTED SOFTWARE TESTING

• Object-orientation has rapidly become accepted as the preferred paradigm for large scale system design.

• Object-oriented programs involve many unique features that are not present in their conventional counterparts.

• Classes provide an excellent structuring mechanism. They allow a system to be divided into well-defined units, which may then be implemented separately. • Classes support information hiding.

• Object-orientation encourages and supports software reuse. This may be achieved either through the simple reuse of a class in a library, or via inheritance, whereby a new class may be created as an extension of an existing one

• These might cause some types of faults that are difficult to detect using traditional testing techniques. To overcome these deficiencies, it is necessary to adopt an object-oriented testing technique that takes these features into account

TROUBLE MAKERS OF OBJECT ORIENTED SOFTWARE

• Following are trouble makers of OO Software

• **Encapsulation**

• A wrapping up of data and functions into a single unit is known as encapsulation. This restricts visibility of object states and also restricts observability of intermediate test results. Fault discovery is more difficult in this case.

• **Polymorphism**

• Polymorphism is one of the crucial features of OOP. It simply means one name multiple forms. Because of polymorphism, all possible bindings have to be tested. All potential execution paths and potential errors have to be tested.

Inheritance

• The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called the derived class or the subclass.

Inheritance results in invisible dependencies between super/sub-classes. Inheritance results in reduced code redundancy, which results in increased code dependencies. If the function is erroneous in the base class, it will be inherited in the derived class too. A subclass can't be tested without its super classes

State the differences from testing non- OO software

1. **Increasing modularization**
   - Decreasing module size
   - More inter-module dependencies (if methods depend on methods of other classes)
2. **Project is divided into OO packages**
   - Instead of function-oriented work packages
   - Functionality may depend on classes developed by co-workers
   - Dependencies require co-ordination
   - Co-ordination may result into misunderstanding
   - Misunderstanding results into errors
3. **Functionality –collaboration among objects**
   - Collaboration requires interfaces
   - Interfaces tend to be complex

   Interfaces require co-ordination
4. **General purpose classes**
   - Reuse beyond the current project
   - Higher degree of potential applications
   - Testing of all relevant states requires anticipation of user profile
5. **Program structure does not reflect program functionality**
   - New instrumentation technique to check functional test coverage
6. **Object methods communicate by common object attributes**
   - Object state produced by a former method may influence the behavior of the latter method
   - The method behavior is influenced by method parameters and object state
7. **Methods call often other methods of the same class**
   - Procedural coupling among methods

OO software is not only harder to test; there is even a rich set of potential errors. It exhibits a higher fault rate. Concise code results into higher fault density. It is also difficult to debug. On the other hand, the reused classes produce generally less faults.

• **Class testing for object-oriented software is the equivalent of unit testing for conventional software**

o Focuses on operations encapsulated by the class and the state behavior of the class

• **Drivers can be used**

o To test operations at the lowest level and for testing whole groups of classes

o To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface

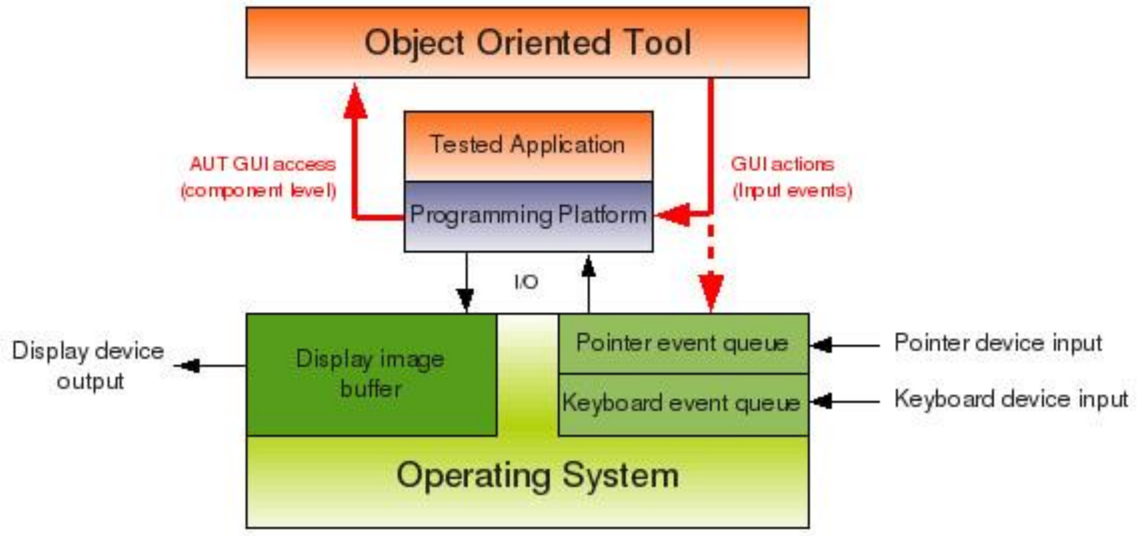
- **Stubs can be used**
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

Difference between Object-Oriented Testing and Conventional Testing

This article will provide you a detailed comparison between Object-oriented testing and Conventional Testing. Let's start with the brief introduction of object-oriented testing.

What is Object-Oriented Testing?

Object-oriented testing is a type of software testing that focuses on verifying the behaviour of individual objects or classes in an object-oriented system. The goal of object-oriented testing is to ensure that each object or class in the system performs its functions correctly and interacts properly with other objects or classes.



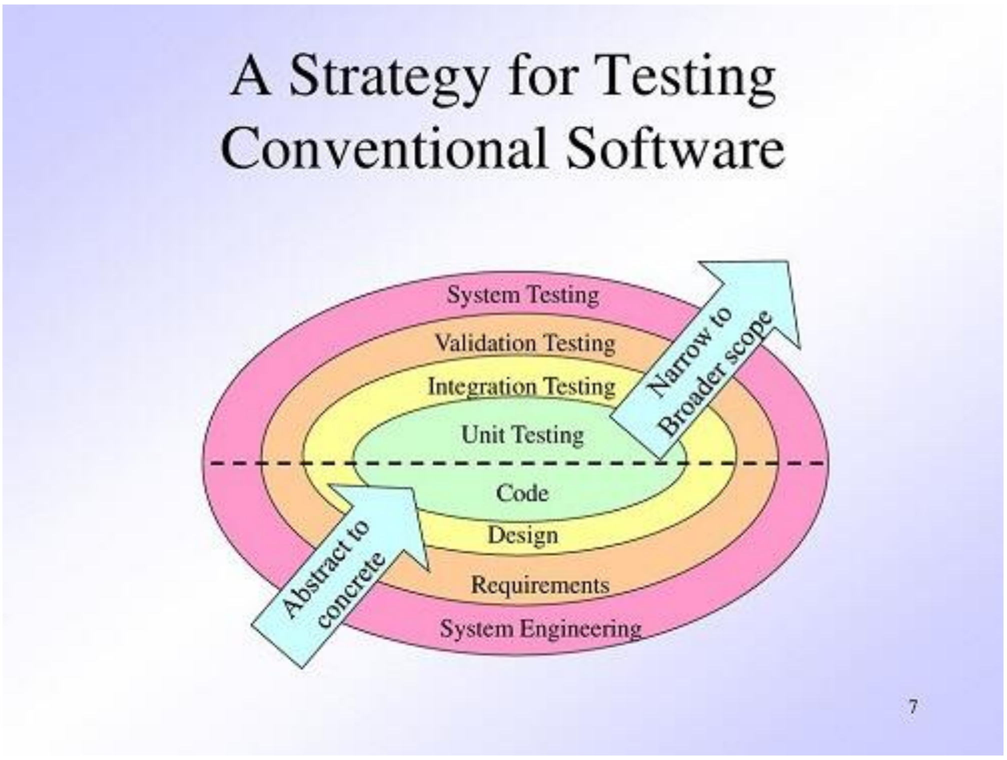
Object-oriented programming emphasises the use of objects and classes to organise and structure software, and object-oriented testing is built on these ideas. In object-oriented testing, the behaviour of an object or class is tested by creating test cases that simulate different scenarios or inputs that the object or class might encounter in the real world.

Unit testing, integration testing, and system testing are common testing phases in object-oriented testing. Unit testing focuses on testing individual objects or classes in isolation, while integration testing verifies that different objects or classes can work together as expected. System testing examines every component of the system, including all of its classes and objects.

Object-oriented testing can be challenging, as it requires a thorough understanding of the system's design and implementation, as well as the ability to create effective test cases that cover all possible scenarios. The system is made to be dependable, effective, and simple to manage, hence it is a crucial step in the software development process.

What is Conventional Testing?

Conventional testing, commonly referred to as traditional testing, is a type of software testing that focuses on comparing the functionality of a software system or application to a set of established standards or criteria. In conventional testing, the software is typically tested by executing a series of pre-written test cases that are designed to cover all of the specified requirements or features of the software.



Testing at several levels, such as unit testing, integration testing, system testing, and acceptance testing, is a common practise in conventional testing. Unit testing focuses on testing individual components or modules of the software in isolation, while integration testing verifies that different components or modules can work together as expected. In contrast to acceptance testing, which involves evaluating the programme with end users to make sure it satisfies their needs and requirements, system testing examines the entire software system as a whole, including its interfaces with other systems.

Conventional testing is often manual, meaning that human testers execute the test cases and evaluate the results. To speed up testing and lower the chance of human error, automated testing technologies are, however, being employed more and more in conventional testing.

As it offers an organised method of testing that helps to verify the software is dependable, functional, and fits the demands of its users, conventional testing is frequently employed in the creation of software. However, conventional testing has limitations, as it can be time-consuming, expensive, and may not detect all types of software defects or issues. As a result, newer testing approaches such as agile and DevOps are becoming increasingly popular, which focus on continuous testing, integration, and delivery of software.

Difference between Object-oriented testing and Conventional Testing

Here are 14 key differences between Object-oriented testing and Conventional Testing

S.No.	Object-Oriented Testing	Conventional Testing
1	This emphasises performing isolated testing on certain objects or classes	This emphasises testing a software system's functionality against predetermined criteria or requirements
2	It verifies the behaviour of each object or class in the system.	It verifies the behaviour of the entire software system.
3	Tests the interactions between objects or classes.	Tests the interactions between software components or modules.
4	It uses mock objects to simulate the behaviour of dependent objects or classes.	It does not use mock objects.
5	It can be more time-consuming than conventional testing.	It can be faster than object-oriented testing.
6	Requires a thorough understanding of the system's design and implementation.	Requires a thorough understanding of the system's requirements and specifications.
7	Involves testing at multiple levels, including unit testing, integration testing, and system testing.	Involves testing at multiple levels, including unit testing, integration testing, system testing, and acceptance testing.
8	Can be more complex than conventional testing.	Can be simpler than object-oriented testing.
9	Focuses on testing the individual behaviour of objects or classes.	Focuses on testing the overall behaviour of the software system.
10	Can detect defects or issues that may not be detected by conventional testing.	May not detect all types of software defects or issues.
11	Can be more effective in testing complex object-oriented systems.	May be less effective in testing complex object-oriented systems.
12	Involves creating test cases that simulate different scenarios or inputs that the object or class might encounter in the real world.	Involves creating test cases that cover all possible scenarios or requirements of the software.

13	Requires the ability to create effective test cases that cover all possible scenarios.	Requires the ability to write test cases that cover all predetermined requirements or specifications.
14	May require the use of specialized testing tools and frameworks.	May not require specialized testing tools and frameworks.

What are the object oriented testing techniques? Explain the testing strategy in detail.

Object – Oriented programming is centered on concepts like Object, Class, Message, Interfaces, Inheritance, Polymorphism etc., Traditional testing techniques can be adopted in Object Oriented environment by using the following techniques:

- Method testing
- Class testing
- Interaction testing
- System testing
- Acceptance testing

Method Testing: Each individual method of the OO software has to be tested by the programmer. This testing ensures Statement Coverage to ensure that all statements have been traversed at least once, Decision Coverage to ensure all conditional executions and Path Coverage to ensure the execution the true and false part of the loop.

Class Testing: Class testing is performed on the smallest testable unit in the encapsulated class. Each operation as part of a class hierarchy has to be tested because its class hierarchy defines its context of use. New methods, inherited methods and redefined methods within the class have to be tested.

This testing is performed using the following approaches:

- Test each method (and constructor) within a class
- Test the state behavior (attributes) of the class between methods

Class testing is different from conventional testing in that Conventional testing focuses on input-process-output, whereas class testing focuses on each method. Test cases should be designed so that they are explicitly associated with the class and/or method to be tested. The purpose of the test should be clearly stated.

Each test case should contain:

- A list of messages and operations that will be exercised as a consequence of the test • A list of exceptions that may occur as the object is tested
- A list of external conditions for setup (i.e.,•changes in the environment external to the software that must exist in order to properly conduct the test)
- Supplementary information that will aid in understanding or implementing the test

Since object oriented software is rich in encapsulation, Inheritance and Polymorphism the following challenges are faced while performing class testing.

- It is difficult to obtain a snapshot of a class without building extra methods that display the classes' state.
- Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism). Other unaltered methods within the subclass may use the redefined method and need to be tested.
- Basis path, condition, data flow and loop tests can all apply to individual methods, but can't test interactions between methods

Integration Testing: Object Orientation does not have a hierarchical control structure so conventional top-down and bottom up integration tests have little meaning.

Integration testing can be applied in three different incremental strategies:

- Thread-based testing, which integrates classes required to respond to one input or event.
- Use-based testing, which integrates classes required by one use case.
- Cluster testing, which integrates classes required to demonstrate one collaboration.

System Testing: All rules and methods of traditional systems testing are also applicable to object-oriented systems. Various types of System Testing include:

Recovery testing: how well and quickly does the system recover from faults

Security testing: verify that protection mechanisms built into the system will protect from unauthorized access (hackers, disgruntled employees, fraudsters)

Stress testing: place abnormal load on the system

Performance testing: investigate the run-time performance within the context of an integrated system

Regression Testing: Regression testing is performed similar to traditional systems to make sure previous functionality still works after new functionality is added. Changing a class that has been tested implies that the unit tests should be rerun. Depending on what has changed, the test scenarios may have to be altered to support this test. In addition, the integration test should be redone for that suite of classes.

Describe in detail about class testing strategies

Class testing is performed on the smallest testable unit in the encapsulated class. Each operation as part of a class hierarchy has to be tested because its class hierarchy defines its context of use. New methods, inherited methods and redefined methods within the class have to be tested.

This testing is performed using the following approaches:

- Test each method (and constructor) within a class
- Test the state behavior (attributes) of the class between methods

Class testing is different from conventional testing in that Conventional testing focuses on input-process-output, whereas class testing focuses on each method. Test cases should be designed so that they are explicitly associated with the class and/or method to be tested.

The purpose of the test should be clearly stated. Each test case should contain:

- A list of messages and operations that will be exercised as a consequence of the test
- A list of exceptions that may occur as the object is tested
- A list of external conditions for setup (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
- Supplementary information that will aid in understanding or implementing the test

Since object oriented software is rich in encapsulation, Inheritance and Polymorphism the following challenges are faced while performing class testing.

- It is difficult to obtain a snapshot of a class without building extra methods that display the classes' state.
- Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism). Other unaltered methods within the subclass may use the redefined method and need to be tested.
- Basis path, condition, data flow and loop tests can all apply to individual methods, but can't test interactions between methods

Smallest testable unit is the encapsulated class

Test each operation as part of a class hierarchy because its class hierarchy defines its context of use

Approach:

– Test each method (and constructor) within a class

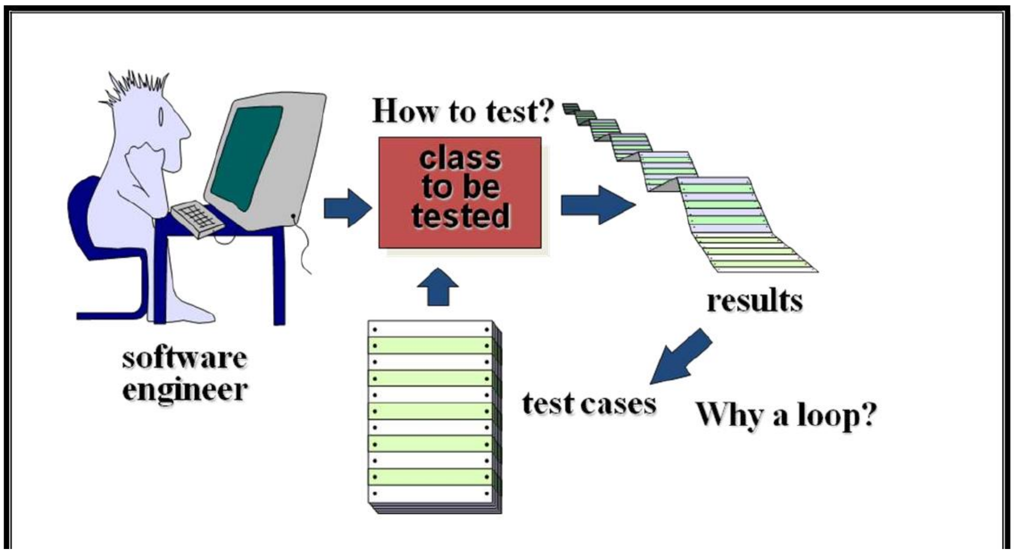
– Test the state behavior (attributes) of the class between methods

How is class testing different from conventional testing?

Conventional testing focuses on input-process-output, whereas class testing focuses on each method, then designing sequences of methods to exercise states of a class

But white-box testing can still be applied

Class Testing Process



Class Test Case Design

1. Identify each test case uniquely

- Associate test case explicitly with the class and/or method to be tested

2. State the purpose of the test

3. Each test case should contain:

a. A list of messages and operations that will be exercised as a consequence of the test b. A list of exceptions that may occur as the object is tested

c. A list of external conditions for setup (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)

d. Supplementary information that will aid in understanding or implementing the test

– Automated unit testing tools facilitate these requirements

Challenges of Class Testing

Encapsulation:

– Difficult to obtain a snapshot of a class without building extra methods which display the classes' state

Inheritance and polymorphism:

– Each new context of use (subclass) requires re-testing because a method may be implemented differently (polymorphism).

– Other unaltered methods within the subclass may use the redefined method and need to be tested

White box tests:

– Basis path, condition, data flow and loop tests can all apply to individual methods, but don't test interactions between methods

Random Class Testing

1. Identify methods applicable to a class

2. Define constraints on their use

– e.g. the class must always be initialized first

3. Identify a minimum test sequence

– an operation sequence that defines the minimum life history of the class

4**. Generate a variety of random (but valid) test sequences**

– this exercises more complex class instance life histories l

Example:

1. An account class in a banking application has open, setup, deposit, withdraw, balance, summarize and close methods

2. The account must be opened first and closed on completion

3. Open – setup – deposit – withdraw – close

4. Open – setup – deposit –* (deposit | withdraw | balance | summarize) – withdraw – close. Generate random test sequences using this template

CLASS MODALITY

Some classes are designed to accept any message in any state, but others limit sequence based on past messages or current content. Consequently, effective testing must recognize these differences and support focused generation of test cases. The concept of class modality provides a testable characterization of these differences in state-based behavior. Modality is the result of constraints on both content (domain) and message sequence.

If classes were all alike, it might be enough to devise tests based on either message sequence or state. Some classes are designed to accept any message in any state, but others limit sequence based on past messages or current content. Consequently, effective testing must recognize these differences and support focused generation of test cases. The concept of class modality provides a testable characterization of these differences in state-based behavior.

Modality is the result of constraints on both content (domain) and message sequence. Class behavior is an abstraction of the responses that objects of a class can make. A response is determined by the particular content of an object and a message, the prior sequence of messages accepted, or both. Class modality is therefore a general pattern of behavior.

Modality is determined by the kind of constraints on message sequence or instance variable content. Categorizing classes in terms of modality helps to focus on the most likely faults. The Free State model provides a testable definition of a class domain integrated with the sequential constraints on class behavior from which we can easily generate a test sequence

. Domain testing can be applied to classes and objects as integral wholes, instead of their composite parts. Invariant boundaries provide a systematic technique for selecting state-based test cases for classes with undifferentiated sequential behavior.

Four modalities are of interest for testing:

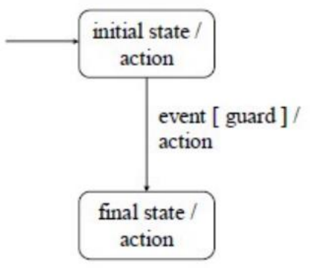
1. **A nonmodal class does not impose any constraints on the sequence of messages accepted**.
   For example, an object of a DateTime class will accept any interleaving of modifier/get messages. Classes that implement basic data types are often nonmodal.
2. **A unimodal class has constraints on the acceptable sequence of messages, despite content**.
   For example, an object of class TrafficSignal can accept the modifier message RedLightOn only after accepting a YellowLightOn message, GreenLightOn only after RedLightOn, and YellowLightOn only after GreenLightOn. Classes that provide application control are typically unimodal.
3. **A quasi-modal class imposes sequential constraints on message acceptance that change with the content of the object**
   . For example, objects of class Stack will reject push message if the stack is full, but accept it otherwise. Many container and collection classes are quasi-modal.
4. **A modal class places both message and domain constraints on the acceptable sequence of messages.**
   For example, an object of class Account will not accept a withdrawal message if the balance in an account is less than or equal to zero; a message to freeze the Account is accepted if the Account is not closed and not frozen. Classes that represent problem domain entities are often modal.

   Different kinds of state faults are therefore possible (or impossible) in each modality. The concept of modality allows us to focus on most probable faults.
   For example, nonmodal classes cannot have sequence faults because all sequences are allowed. These classes are purposely designed to respond to all methods despite prior messages or current instance values.
   A test suite constructed from the FREE class state model is simply a sequence of messages and resultant states with expected values. These test suites can reveal all control faults in a modal class.

. Explain in detail about state based testing.

STATE-BASED TESTING

State machine: implementation-independent specification (model) of the dynamic behavior of the system
§ state: abstract situation in the life cycle of a system entity (for instance, the contents of an object)
§ event: a particular input (for instance, a message or method call)
§ action: the result, output or operation that follows an event
§ transition: an allowable two-state sequence, that is, a change of state (”firing”) caused by an event
§ guard: predicate expression associated with an event, stating a Boolean restriction for a transition to fire



There are several types of state machines:

§ finite automaton (no guards or actions)

§ Mealy machine (no actions associated with states)

§ Moore machine (no actions associated with transitions)

§ statechart (hierarchical states: common superstates)

§ state transition diagram: graphic representation of a state machine

§ state transition table: tabular representation of a state machine

Example: Mealy model of a two-player video game

§ each player has a start button § the player who presses the start button first gets the first serve

§ the current player serves and a volley follows:

– if the server misses the ball, the server's opponent becomes the server

– if the server's opponent misses the ball, the server's score is incremented and the server gets to serve again

– if the server's opponent misses the ball and the server's score is at game point, the server is declared the winner (here: a score of 21 wins)

General properties of state machines

§ typically incomplete

– just the most important states, events and transitions are given

– usually just legal events are associated with transitions; illegal events (such as p1_Start from state Player 1 served) are left undefined

§ may be deterministic or nondeterministic

– deterministic: any state/event/guard triple fires a unique transition

– nondeterministic: the same state/event/guard triple may fire several transitions, and the firing transition may differ in different cases § may have several final states (or none: infinite computations)

§ may contain empty events (default transitions)

§ may be concurrent: the machine (state chart) can be in several different states at the same time

The role of state machines in software testing

§ Framework for model testing, where an executable model (state machine) is executed or simulated with event sequences as test cases, before starting the actual implementation phase

§ Support for testing the system implementation (program) against the system specification (state machine)

§ Support for automatic generation of test cases for the implementation

– there must be an explicit mapping between the elements of the state machine (states, events, actions, transitions, guards) and the elements of the implementation (e.g., classes, objects, attributes, messages, methods, expressions)

– the current state of the state machine underlying the implementation must be checkable, either by the runtime environment or by the implementation itself (built-in tests with, e.g., assertions and class invariants)

Validation of state machines

Checklist for analyzing that the state machine is complete and consistent enough for model or implementation testing:

-one state is designated as the initial state with outgoing transitions

– at least one state is designated as a final state with only incoming transitions; if not, the conditions for termination shall be made explicit

– there are no equivalent states (states for which all possible outbound event sequences result in identical action sequences)

– every state is reachable from the initial state

– at least one final state is reachable from all the other states

– every defined event and action appears in at least one transition (or state)

– except for the initial and final states, every state has at least one incoming transition and at least one outgoing transition

– for deterministic machines, the events accepted in a particular state are unique or differentiated by mutually exclusive guard expressions

– the state machine is completely specified: every state/event pair has at least one transition, resulting in a defined state; or there is an explicit specification of an error-handling or exception-handling mechanism for events that are implicitly rejected (with no specified transition)

– the entire range of truth values (true, false) must be covered by the guard expressions associated with the same event accepted in a particular state

– the evaluation of a guard expression does not produce any side effects in the implementation under test

– no action produces side effects that would corrupt or change the resultant state associated with that action

– a timeout interval (with a recovery mechanism) is specified for each state

– state, event and action names are unambiguous and meaningful in the context of the application

Control faults

When testing an implementation against a state machine, one shall study the following typical control faults (incorrect sequences of events, transitions, or actions):

– missing transition (nothing happens with an event)

– incorrect transition (the resultant state is incorrect)

– missing or incorrect event

– missing or incorrect action (wrong things happen as a result of a transition)

– extra, missing or corrupt state

– sneak path (an event is accepted when it should not be)

– trap door (the implementation accepts undefined events)

Test design strategies for state-based testing

Test cases for state machines and their implementations can be designed using the same notion of coverage as in white-box testing:

§ test case = sequence of input events

§ all-events coverage: each event of the state machine is included in the test suite (is part of at least one test case)

§ all-states coverage: each state of the state machine is exercised at least once during testing, by some test case in the test suite

§ all-actions coverage: each action is executed at least once

§ all-transitions: each transition is exercised at least once

– implies (subsumes) all-events coverage, all-states coverage, and all-actions coverage – ”minimum acceptable strategy for responsible testing of a state machine”

§ all n-transition sequences: every transition sequence generated by n events is exercised at least once

– all transitions = all 1-transition sequences

– all n-transition sequences implies (subsumes) all (n-1)-transition sequences

§ all round-trip paths: every sequence of transitions beginning and ending in the same state is exercised at least once

§ exhaustive: every path over the state machine is exercised at least once

– usually totally impossible or at least unpractical

MESSAGE SEQUENCE SPECIFICATION

Safe Sequence

• SafeSeq(C): A SafeSeq(C) defines a set of all sequences Si that can be derived from SeqSpec(C) of the class C.

• A sequence in SafeSeq(C) is a valid sequence of messages accepted by any instance of the class C. SafeSeq(C) is the regular set (or the language) defined by SeqSpec(C).

Specification

• Valid message sequence for a class

• Sequence based on the functionality

• Regular definitions for specification

• Regular definition over the alphabet of methods set

Definition

The sequence in which each object invokes its method is important for the proper functioning of the object. The Method Sequence Specification (Mtss) of a class documents the correct causal order in which the methods of the class can be invoked. The Message Sequence Specification (Mgss) of a set of classes documents the causal order in which messages can be sent out to different instances of classes by a method.

Mtss & Mgss represents the causal relationship between the instance methods of a set of classes. It is not a complete specification of the class.

• Mgss is used to describe the causal order among the methods supported by different objects.

• The MgSS is specified for each of the method that sends out messages.

• The MgSS of a method identifies all the messages it sends out to other domain objects. It also identifies the causal order in which the messages are sent out. For the MgSS specification we use the same regular expression formalism used for MtSS. In addition to all the operators ' ', 'j', '+', and '*' we introduce a new operator '?' that indicates that a message prefixed to operator '?' is optional. In the following we describe MgSS of a method of a class.

Example Mgss

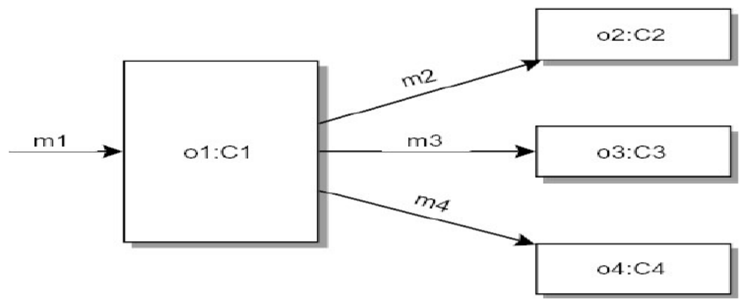


Fig MgSS

• The figure above contains four objects O1, O2, O3, and O4. The method m1 of O1 sends messages to O2, O3, and O4. The order in which messages m2, m3, and m4 are sent out to objects by the message m1 forms the MgSS of the method m1.

– If the messages are sent out in the order of m2, m3, and m4 then the MgSS of m1 is,

m1O1 ⎩(m2O2 · m3O3 · m4O4)

• If the messages are sent out as either m2 or m3 as first before m4 then, the MgSS of m1 is,

m1O1 ⎩((m2O2 | m3O3) m4O4)

Methods

 If a method sends out messages to objects one after the other, then all the messages are represented in the order in which they will be sent out.

o For example, if m1, m2, and m3 are sent out in a sequence then the corresponding MgSS will be m1 · m2 · m3

 If a method sends out a message optionally and other messages always, the optional message will be represented in MgSS using the operator '?'.

o For example, if m1 is sent out optionally and then m2 and m3 are sent out, then the MgSS will be m1? · m2 · m3. When messages are predicated on conditions, messages are sent out optionally.

 If a method sends out messages alternatively such as, a true condition results in one message and false condition results in another message, the MgSS will contain both the messages as alternative messages.

o For example, if m1 or m2 is sent out, but not both, then the MgSS corresponding to the above condition is, m1|m2.

 If a method sends out messages repeatedly such as messages in a loop, then the operator '?‘ or '+ is used. The operator '? is used when messages are sent out zero or more times, while the operator '+' is used when messages are sent out one or more times.

o For example, if a message m1 is sent out repeatedly one or more times, then followed by a message m2 then the corresponding MgSS is, (m1+) · m2.

Mtss and Mgss

 MgSS and MtSS together can identify the causal order between any two methods defined in two diferent classes.

 Since MgSS for a method that sends out no messages to other objects is null, MgSS is useful for those objects that behave as client objects to other objects

Interaction Diagram

 Interaction diagrams are used in many OO analysis and design techniques for representing the interaction between objects

 Interaction diagrams are used to describe how an use case can be realized through communicating objects

. The Interaction diagrams are controlled by the events owing between the blocks. The events are stimuli, i.e. messages sent from one object to the other object

. Example

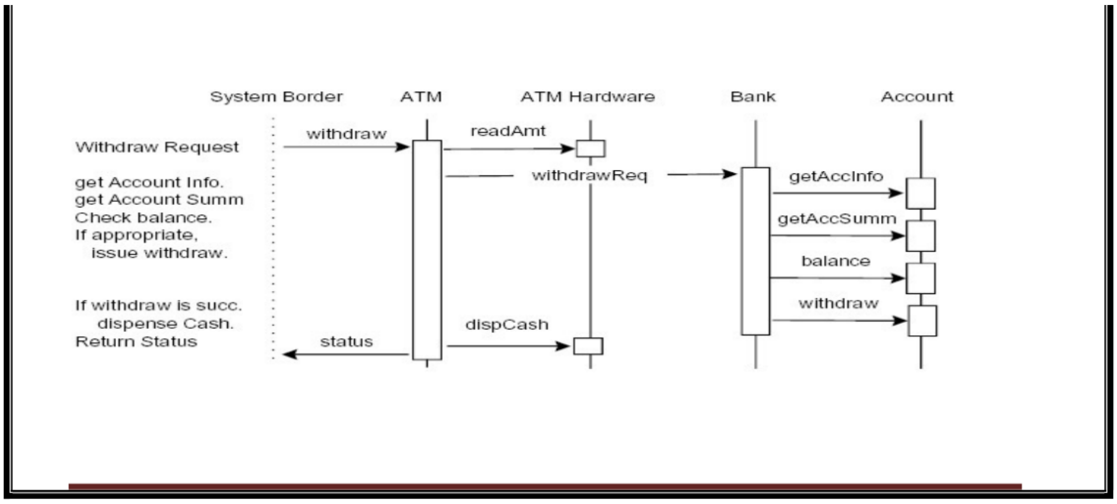


Fig Interaction diagram-Withdrawing of money

Inter Class Test Case Generation

- Methods usually use the services of other objects by sending messages to them. So, testing in OO is to generate test cases that check the correctness of these method-message interactions.
- The MgSS describes the causal order among the methods supported by different classes. The MgSS is specified for each of the method that sends out messages to class instances. So we use the MgSS of all the methods to test the method interactions.

We can use random techniques to generate test sequences from the MgSS. Similarly, we can partition the messages that are sent out from the method and then generate test sequences

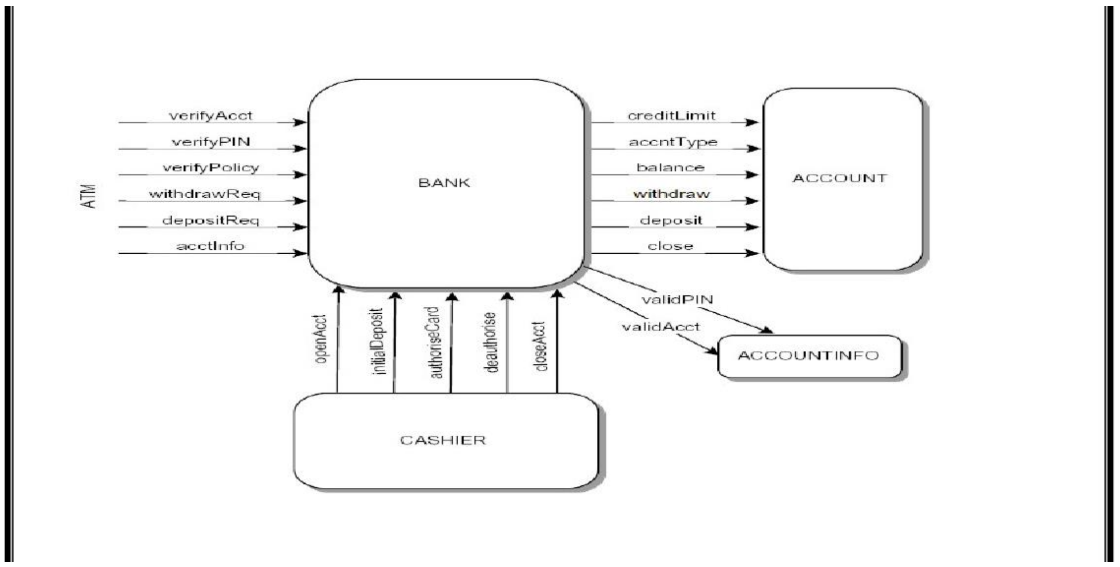


Fig Example

- **Method interactions between neighboring classes**

o Let a method m1 of class C1 send out messages m2, m3, m4, m5, and m6 to different instances of classes. Let the MgSS of the method m1 be,
m1C1 m2O2 · (m3O3 | m4O4 | m5O5)* · m6O6

o For example: The MgSS of the method withdraw() is given below:

♣ withdrawATM:getCashAmntATM_UI · verifyPolicyBank · withdrawReqBank · dispenseCashATM_UI

♣ In the above MgSS, the method withdraw() sends out messages to the classes Bank and ATM_UI. Since, the MgSS corresponds to one sequence that is possible, this forms a test sequence.

- **The MgSS of the methods verifyPolicy() and withdrawReq() of the class Bank is given below.**

o verifyPolicyBank ⎝ balanceAccount · creditLimitAccount

o withdrawReqBank ⎝ withdrawAccount

- A new message sequence that span the instances of Account call can be derived by substituting the above message sequences.

– withdrawATM ⎝ getCashAmntATM_UI · verifyPolicyBank · (balanceAccount · creditLimitAccount) · withdrawReqBank · (withdrawAccount) · dispenseCashATM_UI

– The above test sequence can be used as integration test as they span multiple classes. In fact, a test sequence can be expanded all together until each of the messages ends up at an object that does not send out any more messages.

..Negative Test Sequences from Mgss

- The MgSS of class denes a causal order in which a method of a class sends out messages.
- The MgSS of a method identies all possible valid sequences in which the messages are sent out.
- A message sequence that is not compliant with the MgSS of a method is thus a negative test case.

• We use regular expression formalism to represent the MgSS. The operators used are '' to represent an immediate sequence, 'j' to represent exclusive-or among sequence, '?' corresponds to a zero or more number of times repetition, and '+' corresponds to one or more number of times 82 repetition.

Three Techniques for Generating Negative Test Cases From Mgss

• **Message dropping**

o In this technique, a message that is a must in any message sequence is dropped.

For a given MgSS, if there are n messages that must appear in every message sequence, then one can generate n negative test sequences by dropping each message one at a time. Depending on the messages dropped, the class instances receiving the messages must take different corrective actions

o Example:

♣ withdrawATM ⸌ getCashAmntATM_UI · verifyPolicyBank · withdrawReqBank · dispenseCashATM_UI

♣ withdrawATM ⸌ verifyPolicyBank · withdrawReqBank · dispenseCashATM_UI

• **Message reversing**

o In this technique, a message that must be sent immediately after another message is reversed with each other to form a negative test sequence. If there are n such pairs of messages, then one can generate n negative test sequences by this technique. Depending on the messages that are reversed, it is useful if the class instances receiving the messages can identify the reversed order of the messages.

o Example:

♣ withdrawATM ⸌ verifyPolicyBank · getCashAmntATM_UI · withdrawReqBank · dispenseCashATM_UI

• **Repeating messages**

o In this technique, a message that must be sent out only once is used several times in the sequence to form a negative test case. Thus, if there are n methods that must be sent out only once, then one can generate at least n negative test sequences.

o Example: withdrawATM ⸌ getCashAmntATM_UI · verifyPolicyBank · verifyPolicyBank · withdrawReqBank · dispenseCashATM_UI

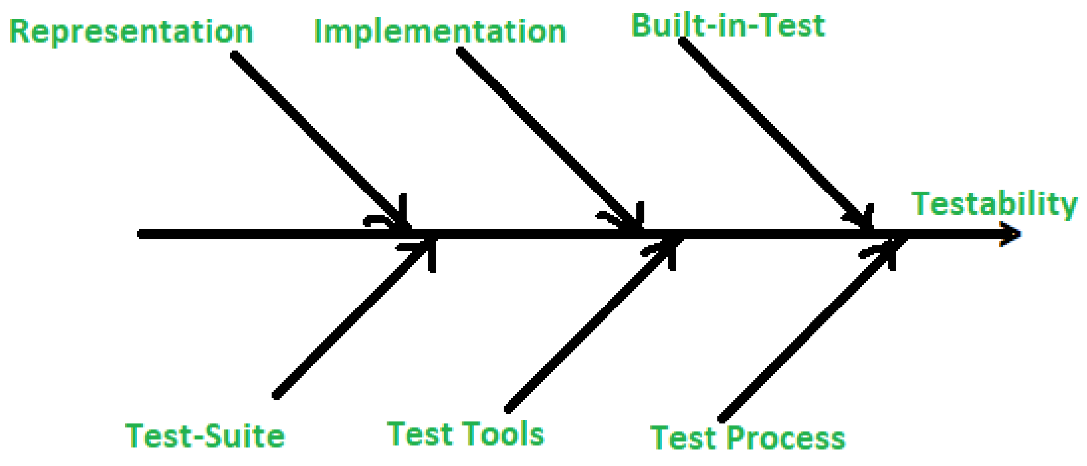
Testability and related issues:

Design for Testability

Design for testability (DFT) is a procedure that is used to set the development process for maximum effectiveness under either a resource-limited or reliability-driven scheme. A resource-limited process uses a testing approach to get the results that a pre-release reliability goal has been met. This process views testing as a way to delete as many rough edges from a system as time or money permits. The testability is significant either to reduce cost in a reliability-driven process. and also to increase reliability in a resource-limited process.
Software testability is a result of many factors some of which are given below :

1. The characteristics of the representation.
2. The characteristics of the implementation.
3. The Built-in test capabilities.
4. Test-suite.
5. Test support environment.
6. The software process in which testing is conducted.

Now, let's see the fishbone chart for considering testability relationships.



1. Representation :
The existence and usefulness of a representation in test development is a critical testability factor because of the following reasons:

- **a.** If you are testing without a representation is like experimenting with a prototype.
- **b.** The representation cannot decide that a test has been passed or failed without any explicit statement of the expected result.
- **c.** It may also force the production of a partial representation as part of the testing plan.

In representations, there are various approaches to develop object-oriented representations like object-oriented analysis (OOA) or object-oriented design (OOD).

2. Implementation :
An object-oriented program that complies with generally accepted principles of OOP poses the fewest obstacles to testing. Structural testability can be assessed by a few simple metrics. A metric may indicate testability, the scope of testing, or both. For example, with high coupling among classes, it is typically more difficult to control the class-under-test (CUT), thus reducing testability. The effect of all intrinsic testability metrics is the same:

- **a.** relatively high value = decreased testability.
- **b.** relatively low value = increased testability.

Scope metrics indicate that the number of tests is proportional to the value of the metric.

3. Built-in-Test :
It provides explicit separation of test and application functionality. The built-in test has some features that are given below:

- **a.** The assertions in built-n-test automate the basic checking and provide “set and forget” runtime checking of basic conditions for correct execution of the program.
- **b.** The Set or Reset helps in controllability.
- **c.** The reporters help in observability.
- **d.** A test suite is a collection of test cases and plans to use them and it defines the general contents of a test plan.
- **e.** The test tools require automation and without automation, there will be less testing, and more costs will be incurred to achieve a given reliability goal.

Testability refers to the extent of how easy or challenging it is to test a system or software artifact. It is often measured by how many tests can execute on a given system.
Software teams need full knowledge about software testing, what it does, and how it behaves. This understanding includes the behaviors and functionalities the software product should have to perform correctly. The more complex the system, the harder it is for testers to understand its behavior and thus find bugs or errors in the software.

As mentioned earlier, testability refers to the degree to which every module, requirement, subsystem, or software component of your architecture can be confirmed as satisfactory or not. In simple terms, the testability of the software artifacts can be a unit of source code, a design document, a test plan, a test procedure, or a system under test.
In this article, we will look into what actually is software testability, its importance and requirements in the industry, and how the work gets done.

Testability Significance

Software testing is necessary to determine if the product meets the customer’s needs and expectations. Software testing can be either manual or automated. Test automation is done by a software product that checks for errors, software bugs, or potential risks in the code.
Testability is one of the main aspects of software testing. Testability has two major dimensions: Test coverage and test reliability.

Test coverage refers to the number of parts that are tested in the product, while test reliability refers to how often tests are executed with a high degree of quality.
Software Testing is essential for any project in order to ensure that it meets customer expectations and standards.

Main Factors Of Software Testability

Software testing is the process of executing a program with the expectation of finding errors. It is an essential part of the software development process.
Software Testability is determined by how easily and effectively a product can be tested by its developers, operators, and users.

The main factors that determine better software testability are

The Complexity Of The Software

The ability to test functionality means that a component or application may be tested using techniques such as test-driven development or table-driven testing. To get better software testability the components should be designed to be decoupled, modular, and stable. Stubs or mocks are effective paths to achieve isolation.

The Size Of The Entire Team Developing It

The testing team members who may handle this operation can have 8-12 members for the workload.

The Size Of The Development System

The development of large software systems is a challenging and demanding task, and software testability is a problem in these systems. The cost of testing and reworking to ensure that systems are sufficiently reliable has made it very important to reduce the time and effort needed to test software.

The Number Of Stakeholders Involved In Defining Requirements For It

In a software development process, the stakeholders come with different perspectives and concerns who participate and define the requirements. These stakeholders include the customer, business analysts, software engineers, and test engineers.

Additional Testability Factors

The following are some factors that must be taken into account when testing software:

- The number of modules and software components.
- The degree to which the software is going to test in isolation.
- Time it will take to test the software.
- The importance of testing the software for a given use case or scenario.
- Whether there is an existing test plan for the software.

Software Testability Requirements

The higher the testability, the easier it is to find bugs in the system.

Module Capabilities

Module capabilities for software testability is a topic that has been discussed extensively in the past days. It is a great way to ensure that software modules are tested thoroughly and the testing activities are more efficient.

The module capabilities for software testability include:

- The ability of the module to be tested by itself.
- The ability of the module to be tested with other modules.
- The ability of the module to be tested with external entities such as hardware or other software modules.
- The ability of the module to be tested with its own data.

Observation Capabilities

The process of running a program or application with the goal of identifying errors is known as software testing. Testing methods involve Static analysis, dynamic analysis, and functional analysis.

Static Analysis:

Static analysis is a type of software testing that analyzes the code without actually running it. It is more like code inspection and is performed by programmers. It can find common errors that include coding logic issues, security vulnerabilities, and other defects in the code.

Dynamic Analysis:

Dynamic analysis is a type of software testing that executes the program or application while monitoring its execution with one or more tools in order to detect errors that can't be detected through static analysis alone. It can also find performance-related issues like bottlenecks and deadlocks.

Functional Analysis:

Functional analysis tests how well an application performs its intended function by executing it with various inputs and examining how it responds to each of the inputs provided.

Testing Support Capabilities

Testing can be done either by manual testing or automation testing means, with the former being more common in smaller organizations and the latter being more common in large ones.

Testing support capabilities improve software testability, which is a requirement for any organization that wants to reduce development costs and increase its speed of delivery. It helps them find out what needs to be improved before they start developing their product.

Defects Disclosure Capabilities

The following is a list of some defects that software testers should be aware of and be able to detect during the testability phase:

- Defects in code that can lead to crashes or other failures.
- Defects in code that can lead to security vulnerabilities.
- Defects in code that can lead to performance problems.
- Defects in code that can lead to data corruption or data loss.
- Defects in code that can lead to denial of service (DoS) attacks.
- Defects in code that can lead to information disclosure (privacy) violations.

Software Testability Measurements

Testability is a measure of how easy it is to test software. It is an essential factor in determining the quality and reliability of the software.

There are many different paths to measure the testability of the software. One way is through metrics such as code coverage, lines of code, or cyclomatic complexity.

Another way to measure software testability is by looking at the ease with which one can write tests. In this case, it can measure four different components:

- The number of lines of code that need to run in order to execute a given test.
- The number of lines of code that need to execute in order to set up a given test.
- The number of lines of code that need to execute in order for a number of test cases to fail.
- Time it takes for the programmers to know what needs testing and what doesn't.

How To Increase Testability

There are many ways to increase the testability of your code. Below are some tips

- **Use a unit testing framework:** A unit test framework provides a set of tools and conventions that make it easy to write and run tests. By using a unit test framework, you can more easily write and maintain tests. **QA Touch** is one of the finest test case management tools that help QA teams to develop a quality project on time.
- **Write small, focused modules:** Smaller modules are easier to test in isolation. Tightly coupled modules are only possible to test one module without affecting the others.
- **Use dependency injection:** Dependency injection is a technique for decoupling modules. By injecting dependencies into modules, you can more easily test them in isolation.
- **Write self-contained tests:** Self-contained tests are easier to set up and run. They don't rely on external resources they can be run anywhere and at any time.
- **Use mock objects:** Mock objects are simulated versions of real objects. During tests, they can act as a stand-in for real objects. This enables you to test the behavior of your code without depending on the mock object's actual implementation.

Key Benefits Of Software Testability

There are many benefits to software testability, but some of the key benefits include:

- **Increased Quality:** By making your software more testable, you can increase the overall quality of your software. It is because testability leads to better design and fewer software bugs.
- **Faster Testing:** When your software is more testable, you can test it faster. It means you can get feedback on your software sooner and iterate more quickly.
- **Reduced Costs:** By making your software more testable, you can reduce the costs associated with testing because testing is typically faster and effortless when the code is more testable.

- **Better Understanding of Requirements:** When your software is more testable, it can help you to understand the requirements for your software. It is because you can test different parts of the code to see how they work together.
- **Greater Flexibility:** When your software is more testable, it provides greater flexibility. It means that you can change the code without breaking existing tests. Additionally, you can add new tests as needed to cover the new functionality.

Software Testability — Final Points

Testability is a critical success factor for any software development team. Having the knowledge of the factors that impact testability, and taking steps to address them early in the development process can help software companies improve the overall quality of their software products and avoid many of the problems associated with testing activities.

FAQ

What Is Software Testability?

Software testability is a measure of how easy it is to test a software system or component. Testability can be affected by factors for example structure of the code, the size and complexity of the system, and the tools and techniques available.

Why Is Software Testability Important?

Software testability is essential for quality assurance because it can significantly impact the availability, cost, and effectiveness of testing. If a system is difficult to test, it may require more time and resources to achieve adequate coverage. Additionally, defects may be more expensive to fix if they are found late in the development process, they may be more expensive to fix.

How Can I Improve The Testability Of My Software?

There are many ways to improve the testability of your software. Some common approaches include refactoring code to improve readability and modularity, using design patterns to simplify the test process, and providing adequate documentation. Additionally, automated testing tools can help to speed up the testing process and reduce costs

What is Testability?

Testability is a metric that defines how easily, effectively, and efficiently an application can be tested by QA teams.

This may sound a little vague. Isn't all software technically "easy to test"? You just use it to initiate the functions already built into it and check if it performs them all accurately.

However, testing efficacy is heavily dependent on the software's underlying architecture. QA professionals need complete knowledge of the application-under-test in order to design and execute requisite tests. They have to understand the behavior and features the app is expected to display and accomplish at all times so that they know what counts as "passing" or "failing" a test.

This is easier said than done when it comes to complex systems. It takes more time and effort to understand the technical schema, deciding upon the right tests, design said tests, run them, and identify bugs, and run debugging activities. The more complex a system, the less its testability.

Basically, testability is a measure of how easy or difficult it is to confirm the success/failure of every software module, subsystem, component, and requirement in the application ecosystem.

Of course, certain software systems (such as, for eg., the computers used to map the skies in astrophysics labs) will have to be more complex if they are to work. But, as far as possible, it is advisable to design source code for high testability.

Why does software testability matter?

Let's take an example.

In a certain project, devs are looking for the root cause of a certain bug by looking through test logs. However, while there are detailed logs for certain modules, others do not. This is because different testers are working on different modules – one maintains logs for everything and the other only logs in the event of a serious malfunction.

But, when put together, the devs can't distinguish between the detailed and non-detailed logs, so it's harder to find the source of the bug. This is the definition of a software product with low testability.

The solution is to have precise, consistent blogs for all modules, whether or not they trigger bugs. This consistency is what will make the software easier to test, and therefore, more testable.

The more testable a software, the more successfully testers will be able to scan through and identify the maximum number of bugs. Tests are easier to create and execute. Bugs are found faster and are also easier to resolve. Testers don't have to spend as much time and effort, and the product hits the market much faster.

On the other hand, if testability is low, tests are harder to design and take longer to execute. If faced with a tight deadline, the manager might have to sacrifice some tests and push through a buggy product.

This is why 'software testability' or 'testability in software testing' matters.

Factors of Testability in Software

Observability: The ability to detect each software module and components' response to user inputs. It also involves monitoring the changes the inputs implement in the system's internal processes. Testable software makes this process as simple as possible, since observing these responses is the basis of tests.

Controllability: The ability to control every single software module in isolation. The more controllable an app, the more testable it is. Controlling every module makes it easier to automate tests pertaining to each specific module.

Simplicity: The measure of how much effort devs and QAs need to test an app. This is decided after evaluating the functional, structural, and code-level simplicity. The higher your software simplicity, the more testable (and debugable) it is.

Stability: The measure of how many or few changes a certain app will require, once it has been put through the relevant tests. A high-stability software will require far fewer changes than it's low-stability counterpart. Software stability is also required before QAs can start running automated tests. Needless to say, the high the software stability, the more testable it is.

Know more about automated web application testing

Availability: The measure of how available all objects and entities needed for testing (bugs, source code, software components) are at any stage of development. High testability is a direct result of high availability.

Requirements of Software Testability

Use the attributes mentioned & described below to create a more testable software system. By incorporating these requirements into the configuration(documents, programs, data points), you stand a higher chance of ensuring high testability. Basically, do the following and your software will be easier to test.

Module capabilities

Each software module is, in the ideal, best-practice-driven scenarios, tested separately. Test cases should be designed for each module, and also designed to gauge the quality and consequence of interaction between the modules.
Module capabilities include checking for the following:

- Can each module be tested in isolation?
- Can each module be tested with every other relevant module?
- Can every module be tested (if needed) with third-party hardware and software modules?
- Can every module be tested with its own data?

If the answer to the above questions is yes, you have high-testability software on your hands.

Test support capabilities

During active tests, the entry point to test drivers and root must be saved for every tester working on the system, every test interface, and test scenario. This is so that, during increment-level testing, you don't have trouble gauging the accuracy of the testing root and driver.

Defect disclosure capabilities

System errors should be minimal so that they do not show up as blockers to larger testing. Testers must be aware of all defects that can cause system anomalies (for eg., certain defects lead to performance problems while other causes security vulnerabilities and can lead to DoS attacks). Understanding and disclosing as many defects as possible is the very cornerstone of high software testability.
Requirement documents must also insist upon the following parameters for high testability:

- Every single requirement should be precise, brief, and complete.
- Each requirement should be unambiguous – it's meaning should be the same for every dev and tester who sees it.
- Every requirement should be in no contradiction with any other requirement.
- Every requirement should be ranked on the basis of priority.
- Every requirement should be domain-based. This minimized problems if requirements do need to be changed, whether during ideation, software development, and/or testing.

Observation capabilities

The software should have some mechanisms (or the team should use the right tools) to monitor user inputs, output and factors influencing said output. Examples of such capabilities would be static analysis, dynamic analysis, and functional analysis.
Types of Testability in Software

Object-oriented program testability

Object-oriented software is tested at the levels of Unit, Integration, and System verification. Of all three, it is easiest to access unit tests to improve testability. This is because unit tests are the very beginning of any test cycle, and any changes for more testability, implemented at this level, will positively affect the entire cycle down the line.

Domain-based testability

Any software created with the mechanics of domain-driven development will be easy to test and change. The key to making domain-based software more testable is to establish high levels of observability and controllability.

Module-based testability

To make module-based software highly testable, devs need to account for three stages:

- Normalize program: Normalize the program via semantic & system tools so that it is more equipped to absorb and work with initiatives driving high testability.
- Identify testability components: Detect the testable components based on your normalized data pathways and workflows.
- Measure program testability: Evaluate and gauge program testability on the basis of the testing criteria required by the aforementioned data stream.

How to Measure Software Testability

Fundamentally, measuring software testability means finding and specifying the software components that are of questionable quality (at this stage), and distinguishing them from components that have less apparent defects. Low-quality components will be harder to test, so they should be prioritized beneath the low-defect, easier-to-test components.
However, to determine which component holds what testability, your team needs to look closely at the following metrics:
Depth Of Inheritance Tree;
Fan Out (FOUT);
Lack Of Cohesion Of Methods (LCOM);

Lines Of Code per Class (LOCC);
Response For Class (RFC);
Weighted Methods per Class (WMC)
All these metrics determine, at the core, which components are more challenging to test, and vice-versa. This is something testers need to know at the very beginning of testing, even before they start creating scripts so that they can plan test scenarios, test cases and request equipment (specific test environments) more efficiently.
The metrics assess how testable the application is and will be through it’s entire lifecycle. Each metric is related to one or more of the testability factors detailed previously in this article.

Benefits of software testability

- **Facilitates earlier detection of bugs/anomalies**: Since high testability enables more voluminous and comprehensive testing right from the beginning, QAs end up identifying more bugs at the early stages of a test cycle.

  This is great because bugs detected earlier are easier to remove. They are not as inextricably entwined into the larger system as they would be, if they were later in the cycle.
- **Makes life easier for testers**: This is a no-brainer, right? High testability makes software easier to test, which means testers do not have to spend as much time & work to create the right tests, find bugs and report them to devs.
- **Makes it easier to evaluate automation needs**: Software testability levels depend heavily on controllability. The level of controllability inherent in a software system is directly related to how much automation it can take. In other words, software testability helps evaluate the level of test automation required for a certain project.

Automate your tests for web, mobile, desktop applications and APIs, 5x faster, with Testsigma

Improving Software Testability

- **Name elements correctly & obviously**: When devs label each element in line with logic and uniqueness, it is much easier, from an admin point of view, to run tests. However, this isn’t always possible, especially in large-scale projects.

When multiple teams of devs, engineers and testers are working on a single project, they aren't always aware of the naming convention used by other teams. Im such cases, unique naming is often a casualty.

- **Testing in the appropriate environment**: Testing is infinitely easier if the test environments mimic the production environment as closely as possible. It is advisable to run tests on real browsers, devices, and OSes that your target audience is most likely to use.

  Testsigma is a unified, fully customizable software testing platform that works out of the box. It is designed to help automate and execute end-to-end tests 5X faster for web, mobile apps, & APIs. You can use Testsigma to create test scripts in plain English scripts – scripts that self-heal and require low or no maintenance.

You can run tests in your local browser/device or run across 800+ browsers and 2000+ devices on our cloud-hosted test lab. You can also view step-wise results for each test and analyze real-time reports & dashboards at the individual, test suite & device levels. Moreover, Testsigma's intelligent AI automatically fixes broken scripts, heals dynamically changing elements, and suggests fixes for test failures.

- **Logging mechanisms**: Tests are most effectively streamlined if the test software automatically logs the state of the application before and after every test. The logs should also track every single test step, so that devs can go back and check at which step a bug occurred. This makes it easier to identify the cause of the bug.
- **A stable, consistent UI design**: Consistent design makes it easy to predict how software components & modules will behave, which in turn, makes it easiest to create tests that provide sufficient test & code coverage.
- **Better observability**: Once again, you need a tool like Testsigma to achieve this. Improved observability lets testers look closely at the software output in response to every single input.

ISSUES IN TESTABILITY

1. **Lack of Modularity:**
   - **Problem:** When the software is not designed with modular components, it becomes challenging to isolate and test individual units or functionalities independently.
   - **Solution:** Emphasize modular design principles to create independent and testable components.
2. **Inadequate Documentation:**

- **Problem:** Lack of clear and up-to-date documentation makes it difficult for testers to understand the software's functionality and expected behavior.
- **Solution:** Maintain comprehensive and accurate documentation, including requirements, design specifications, and test cases.

3. **Complex Dependencies:**
   - **Problem:** Software with complex dependencies between modules or external systems can be challenging to test as changes in one area may impact others.
   - **Solution:** Minimize dependencies and decouple components where possible. Use mock objects or stubs to simulate external dependencies during testing.
4. **Poor Code Quality:**
   - **Problem:** Code that is poorly structured, not adhering to coding standards, or lacks readability can be challenging to test effectively.
   - **Solution:** Encourage and enforce coding standards, conduct code reviews, and emphasize clean code practices to improve testability.
5. **Limited Accessibility to Code:**
   - **Problem:** Testers may face difficulties testing software if they don't have access to the source code or if the code is obfuscated.
   - **Solution:** Ensure that testers have access to the source code and promote transparency in the development process.
6. **Insufficient Logging and Monitoring:**
   - **Problem:** Inadequate logging and monitoring make it difficult to trace and diagnose issues during testing.
   - **Solution:** Implement robust logging mechanisms and monitoring tools to capture relevant information about the software's behavior.
7. **Inconsistent Test Data:**
   - **Problem:** Testers may struggle if there is a lack of consistent and realistic test data, leading to incomplete test coverage.
   - **Solution:** Develop a comprehensive set of test data that covers various scenarios, and ensure that it is consistently maintained and updated.
8. **Unrealistic Test Environments:**
   - **Problem:** Testing in environments that do not accurately represent the production environment can lead to issues not being identified until later stages.
   - **Solution:** Use realistic test environments that mirror the production environment as closely as possible to uncover environment-specific issues early in the testing process.
9. **Ineffective Test Automation:**
   - **Problem:** Poorly designed or maintained automated tests can result in false positives, false negatives, and reduced test effectiveness.
   - **Solution:** Regularly review and update automated test scripts, and prioritize test automation efforts based on critical functionalities.
10. **Late Involvement of Testers:**
    - **Problem:** If testers are brought into the development process too late, it can limit their ability to influence testability considerations and identify potential issues early.
    - **Solution:** Involve testers early in the development lifecycle, promote collaboration between developers and testers, and conduct regular reviews of design and code for testability concerns.

What is Test Observability?

Test Observability signifies a specialized aspect of software testing that provides insights into genuine test failures by filtering out the distracting noise caused by unstable and consistently failing tests. This method aids in pinpointing the underlying reasons for these test failures. You can expedite the run verification process by employing Test Observability, reducing it to mere minutes. It also furnishes you with a wealth of historical data, opening a window to the comprehensive health of your test suites.

This data, in turn, illuminates key issues that may be jeopardizing your test stability, empowering you with the knowledge to streamline and enhance your testing procedures.

Principles of Observability

The core principles of Test Observability pivot around three crucial components: logging, metrics, and traces. Understanding these elements allows you to unravel the intricacies of your system's behavior, equipping you to detect and address issues effectively and efficiently.

Difference between Observability and Controllability in Software Testing

Observability and controllability are two key components in the software testing landscape. While they may seem similar, they serve distinct functions, and when used in tandem, they can significantly enhance the effectiveness of your software testing.

Observability	**Controllability**
Observability refers to the degree to which you can infer the system's internal states based on its external outputs. In other words, it helps us 'observe' what's happening inside the system through its output.	Controllability pertains to controlling the system's internal states from the outside. It allows us to test different variables to see how the system responds.

For example, by observing a software’s response time under different workloads (output), we can infer its performance characteristics (internal state).	For instance, we may manipulate a system’s input to create different test scenarios. We can control the system’s behavior and test its response by varying user credentials in a login feature.

Understanding these principles and their interplay is crucial for an effective and efficient software testing process. They are complementary and, when integrated correctly, can result in more reliable software and faster troubleshooting.

Examples of Test Observability
Test Observability in action can be best understood with a few practical examples:

- **Debugging a Login Issue:** Let’s assume that an online application has been receiving complaints about users being unable to log in. Observability tools would report the failed login attempts and provide detailed logs and traces for each failure. These could include details about the user’s input, server responses, error messages, and even the sequence of microservices invoked during the process. This rich data set can provide valuable clues about whether the issue lies in the user validation service, database, or elsewhere.
- **Performance Testing an E-commerce Site:** Consider an e-commerce website preparing for a significant sales event. Testers can simulate high traffic through observability and observe how the system behaves under stress. They can monitor metrics like page load times, database response times, server utilization, etc. If the system performance degrades, traces can help identify the bottlenecks in the workflow.

Benefits of Test Observability in Software Testing: Why Should You Do It?
The role of Test Observability in the realm of software testing is nothing short of transformative. Below are some compelling reasons to consider integrating Test Observability into your testing strategy:

1. **Improved Fault Detection:** With its in-depth data collection, Test Observability enables teams to diagnose and pinpoint the root cause of issues more efficiently and accurately, reducing system downtime.
2. **Enhanced System Understanding:** Observability allows a detailed examination of the internal states of your software system through its external outputs. This increases understanding of your system's health, performance, and behavior, promoting informed decision-making and proactive problem-solving.
3. **Faster Issue Resolution:** The deep insight provided by observability tools can significantly reduce Mean Time to Resolution (MTTR). Understanding what's happening within a system in real-time can help resolve issues faster, enhancing overall system reliability and performance.

Where Logs Meet Observability

Here are the four benefits of logs and the reason why they're essential to improving observability into your services.

1. Logs Provide a Detailed Path of Action and Enable Reproducibility

Logs provide a developer with a detailed path of all the actions the user performed that led up to whatever went wrong. You can use this path to reproduce exactly what the user experienced. It's vital that developers are able to do this so they can gain a deeper understanding of the issue.

Once developers have determined the exact location of the bug, a debugger or other problem-resolution tool can be used to find the bug in the code itself.

2. Logs Provide Detailed Information

Logs produce detailed information about what happens in your application. In order words, they provide context. Logs often include meta information such as a timestamp or user request ID that enriches the data.

Additionally, when you or a user encounters an error, you'll usually be able to find a stack trace in your logs. Wikipedia defines a stack trace as "a report of the active stack frames at a certain point in time during the execution of a program." In other words, a stack trace produces a detailed path of all the code that was executed. This gives developers amazing insight into where to look for a possible solution.

3. Logs Provide Better Insights

Lastly, logs can give you better insights than regular monitoring metrics can. You can track the number of incidents, the number of errors, and the number of failed requests. This kind of data gives you much better visibility into the well-being of your application.

For example, you might notice a sudden increase in the error rate. This might signal that you have a problem, and it allows you to detect problems proactively through the observability of a system.

Next, let's look at the subtle difference between observability and controllability.

Observability vs. Controllability

Now let's examine the difference between observability and controllability. Where observability is only concerned with understanding the internals of a system by analyzing the outputs, controllability is concerned with examining outputs for given inputs. This means that software testing helps with improving the controllability of a software system.

Therefore, when we know that a particular input generates a certain expected output, we can improve the control we have over the system since we can be sure this aspect of the system works as intended. Controllability is actually one of the main pillars of writing testable software.

To summarize, controllability is concerned with both the input and the output of a system, whereas observability tries to illuminate the internal state by solely looking at the output.

Built-in Test – Design by Contract - Precondition, Post condition and Invariant

"Design by Contract" (DbC) is a software development approach that emphasizes the importance of specifying the behavior of software components through contracts. Contracts consist of three main parts: preconditions, postconditions, and invariants. These elements play a crucial role in built-in testing, which is a method of embedding tests directly into the software design and code to ensure correctness and reliability.

Design by contract (**DbC**), also known as **contract programming**, **programming by contract** and **design-by-contract programming**, is an approach for designing software.

It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

The DbC approach assumes all *client components* that invoke an operation on a *server component* will meet the preconditions specified as required for that operation.

Where this assumption is considered too risky (as in multi-channel or distributed computing), the inverse approach is taken, meaning that the *server component* tests that all relevant preconditions hold true (before, or while, processing the *client component*'s request) and replies with a suitable error message if not.

1. **Preconditions:**
   - **Definition:** Preconditions are the conditions or assumptions that must be true before a function or method is executed.
   - **Role in Testing:** Preconditions serve as a form of input validation. They define the acceptable state of the system before a particular operation is performed. If a precondition is violated, it indicates a bug or error in the calling code or the function itself.
   - **Example:** Before withdrawing money from a bank account, a precondition might be that the account balance must be sufficient to cover the withdrawal amount.
2. **Postconditions:**
   - **Definition:** Postconditions are the conditions that must be true after the execution of a function or method, assuming the preconditions were met.
   - **Role in Testing:** Postconditions help ensure that the function or method has produced the expected results and has left the system in a valid state. Violations of postconditions can highlight errors or bugs in the implementation.
   - **Example:** After depositing money into a bank account, a postcondition might be that the account balance has increased by the deposited amount.

3. **Invariants:**
    - **Definition:** Invariants are conditions that must remain true throughout the execution of a function or method, assuming the preconditions were met initially.
    - **Role in Testing:** Invariants help ensure the consistency and integrity of the system during and after the execution of a function or method. They represent properties that should not be violated by the operation.
    - **Example:** In a list data structure, an invariant might be that the length of the list remains the same after a particular operation unless explicitly modified.

Built-in Testing:

- **Definition:** Built-in testing involves incorporating tests directly into the codebase to automatically check the compliance of functions and methods with their contracts.
- **Role in Testing:** By embedding tests within the code, developers can catch errors early in the development process, leading to more robust and reliable software.
- **Example:** Unit tests can be created alongside the development of a function, checking preconditions, postconditions, and invariants to ensure the correct behavior of the function.

Design by Contract

It is an approach to designing robust yet simple software. It provides methodological guidelines to achieve these goals without resorting to defensive programming. Instead, Design by Contract builds class invariants and pre/post validation of methods arguments and return values into the code itself.

Design by Contract received a lot of attention when it was initially released, but it has never become a significant approach in most languages (the most obvious exception is Eiffel, which was built around Design by Contract).

Several libraries have been introduced for using DbC in Java, but none ever gained mass acceptance. However, a new library called Contracts for Java has been introduced by Google. With Google's weight behind it, perhaps this implementation will finally bring Design by Contract to the attention of the general Java community.

DESIGN BY CONTRACT

Design by Contract (DbC) was introduced in the mid 1980s in the Eiffel language by Bertrand Meyer, a French computer scientist. The goal of DbC is to enable programmers to "build software specification into the software source code and make it self-checking at runtime." This is achieved through the introduction of "contracts" — executable code contained within the source that specifies obligations for classes, methods, and their callers.

The primary advantage of DbC is that contracts and obligations are made explicit instead of implicit. For example, many methods have an implicit requirement that an argument be non-null (though there are many tools that add support for this through annotations, none are standard yet). With contracts, these requirements are fully documented and enforced in the code itself.

DbC generally provides three types of validations:

Preconditions
: Preconditions specify conditions that must be true before a method can execute. These conditions can be imposed upon the arguments passed into the method or upon the state of the called class itself. Preconditions impose obligations on the client of the method; if the conditions are not met, it is a bug in the client.

Postconditions
: Postconditions specify conditions that must be true when a method is finished executing. These conditions can involve the current class state, the class state as it was before the method was called, the method arguments, and the method return value. Postconditions impose obligations on the method; if the conditions are not met, it is a bug in the method.

Invariants
: Invariants specify conditions that must be true of a class whenever it is accessible to a client (specifically when the class has finished loading and whenever its methods are called).

Additionally, Eiffel provides more types of validation than these standard types, such as invariants and variants for loops. We will see later that Contracts for Java does not share this feature. However, Contracts for Java instead adds validations on thrown exceptions.

Some of these features can be implemented in Java with simple assertions, but contracts are more powerful for several reasons:

- They are fully integrated to the type system and the inheritance mechanism, and thus provide the necessary semantics for subtyping and subclassing.

- Postconditions keep a reference to the "old" value as it existed before the method ran, which can be useful for comparison; assertions do not support this without extra code.

- Class invariants are automatically added to every public method, greatly increasing robustness without adding duplicate code.

- Validations can be added to check the condition of exceptions that get thrown.

Inheritance in Object-Oriented Programming (OOP):

Inheritance is a key principle in OOP that allows a new class to inherit the properties and behaviors of an existing class. It promotes code reuse, modularity, and the organization of code into a hierarchy. Inheritance typically involves a superclass (or base class) and one or more subclasses (or derived classes). The subclasses inherit attributes and behaviors from the superclass.

2. Impact on Software Testing:

The impact of inheritance in software testing lies in how it affects test design, test coverage, and the identification of potential issues.

Test Design and Coverage:

1. **Testing Inherited Functionality:**
    - Test cases need to cover both the inherited and specific functionalities of subclasses. Verifying that inherited methods and attributes work as intended is crucial to ensuring the correct behavior of the entire class hierarchy.
2. **Polymorphism Testing:**
    - Inheritance often involves polymorphism, where objects of different classes can be treated as objects of a common base class. Testing polymorphic behavior ensures that the right methods are called at runtime based on the actual type of the object.

Potential Issues and Challenges:

1. **Inherited Bugs:**
    - Bugs in the inherited methods or attributes of the superclass can propagate to the subclasses. Regression testing is essential to ensure that changes in the superclass do not introduce issues in the subclasses.
2. **Overridden Methods:**
    - When a subclass overrides a method from the superclass, it introduces the potential for errors. Testing should verify that overridden methods in subclasses behave correctly and do not introduce unexpected side effects.
3. **Dependency on Superclass Changes:**
    - Changes in the superclass might impact the behavior of subclasses. It's essential to test how modifications in the base class affect the derived classes and ensure backward compatibility.
4. **Understanding Class Hierarchy:**
    - Testers need a good understanding of the class hierarchy to design effective test cases. Lack of understanding may lead to insufficient coverage or overlooking potential issues.

In conclusion, while inheritance in OOP promotes code organization and reuse, it introduces considerations in software testing. Thorough testing should cover both the inherited and specific functionalities of subclasses, with a focus on potential issues related to overridden methods, polymorphism, and the impact of changes in the superclass. Regression testing becomes critical to catch any unintended consequences of modifications in the inheritance hierarchy.

Certainly, let's dive deeper into the impact of inheritance on software testing by exploring additional considerations, challenges, and best practices.

1. Testing Inherited Functionality:

a. *Method Overriding:*

- Inheritance often involves method overriding, where a subclass provides a specific implementation for a method already defined in the superclass. Testing should ensure that the overridden methods in the subclass behave as expected and fulfill the intended purpose.

b. *Attributes Inheritance:*

- Inherited attributes, such as fields or properties, should be tested to ensure they are accessible and have the correct values in the subclass instances.

2. Polymorphism Testing:

a. *Runtime Behavior:*

- Polymorphism allows objects of different types to be treated as objects of a common base type. Testing should verify that, at runtime, the correct methods are invoked based on the actual type of the object.

3. Potential Issues and Challenges:

a. *Inheritance Depth:*

- Deep inheritance hierarchies may lead to increased complexity and potential issues. Testing should consider scenarios where multiple levels of inheritance exist.

b. *Dependencies Between Classes:*

- Understanding and testing dependencies between classes are crucial. Changes in the superclass might have unintended consequences on subclasses, and vice versa.

4. Best Practices for Testing Inheritance:

a. *Comprehensive Test Coverage:*

- Design test cases that cover all aspects of inheritance, including method overriding, attribute inheritance, and polymorphic behavior. Aim for comprehensive test coverage to catch potential issues.

b. *Regression Testing:*

- As the codebase evolves, conduct regression testing to ensure that changes in one part of the code do not adversely impact inherited functionality.

c. *Mocking and Stubbing:*

- Use mocking and stubbing techniques to isolate the testing of specific classes. This is particularly useful when testing classes with dependencies on external services or complex behavior.

d. *Test Automation:*

- Leverage test automation to repeatedly execute test suites, especially when dealing with large and complex inheritance hierarchies. Automated tests can provide rapid feedback on the impact of changes.

e. *Documentation:*

- Maintain clear documentation of the class hierarchy, including the relationships between classes and any design decisions related to inheritance. This documentation aids both developers and testers in understanding the system.

In summary, the impact of inheritance on software testing involves thorough validation of inherited functionality, consideration of polymorphic behavior, and addressing potential challenges related to inheritance depth and dependencies between classes. Adopting best practices, including comprehensive test coverage, regression testing, and test automation, contributes to the effective testing of code with inheritance.

What is Regression Testing?

Regression testing is a black box testing techniques. It is used to authenticate a code change in the software does not impact the existing functionality of the product. Regression testing is making sure that the product works fine with new functionality, bug fixes, or any change in the existing feature.

Regression testing is a type of software testing. Test cases are re-executed to check the previous functionality of the application is working fine, and the new changes have not produced any bugs.

Regression testing can be performed on a new build when there is a significant change in the original functionality. It ensures that the code still works even when the changes are occurring. Regression means Re-test those parts of the application, which are unchanged.

Regression tests are also known as the Verification Method. Test cases are often automated. Test cases are required to execute many times and running the same test case again and again manually, is time-consuming and tedious too.

Example of Regression testing

Here we are going to take a case to define the regression testing efficiently:

Consider a product Y, in which one of the functionality is to trigger confirmation, acceptance, and dispatched emails. It also needs to be tested to ensure that the change in the code not affected them. Regressing testing does not depend on any programming language like Java, C++, C#, etc. This method is used to test the product for modifications or any updates done. It ensures that any change in a product does not affect the existing module of the product. Verify that the bugs fixed and the newly added features not created any problem in the previous working version of the Software.

When can we perform Regression Testing?

We do regression testing whenever the production code is modified.

We can perform regression testing in the following scenario, these are:

1. When new functionality added to the application.

Example:

A website has a login functionality which allows users to log in only with Email. Now providing a new feature to do login using Facebook.

2. When there is a Change Requirement.

Example:

Remember password removed from the login page which is applicable previously.

3. When the defect fixed

Example:

Assume login button is not working in a login page and a tester reports a bug stating that the login button is broken. Once the bug fixed by developers, tester tests it to make sure Login Button is working as per the expected result. Simultaneously, tester tests other functionality which is related to the login button.

4. When there is a performance issue fix

Example:

Loading of a home page takes 5 seconds, reducing the load time to 2 seconds.

5. When there is an environment change

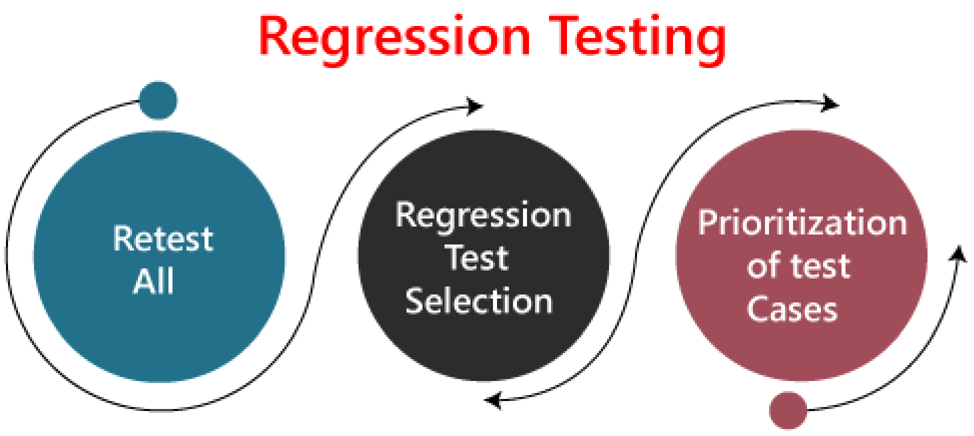
Example:

When we update the database from MySql to Oracle.

How to perform Regression Testing?

The need for regression testing comes when software maintenance includes enhancements, error corrections, optimization, and deletion of existing features. These modifications may affect system functionality. Regression Testing becomes necessary in this case.

Regression testing can be performed using the following techniques:



1. Re-test All:

Re-Test is one of the approaches to do regression testing. In this approach, all the test case suits should be re-executed. Here we can define re-test as when a test fails, and we determine the cause of the failure is a software fault. The fault is reported, we can expect a new version of the software in which defect fixed. In this case, we will need to execute the test again to confirm that the fault fixed. This is known as re-testing. Some will refer to this as confirmation testing.

The re-test is very expensive, as it requires enormous time and resources.

2. Regression test Selection:

- In this technique, a selected test-case suit will execute rather than an entire test-case suit.
- The selected test case suits divided in two cases
    1. Reusable Test cases.
    2. Obsolete Test cases.
- Reusable test cases can use in succeeding regression cycle.
- Obsolete test cases can't use in succeeding regression cycle.

3. Prioritization of test cases:

Prioritize the test case depending on business impact, critical and frequently functionality used. Selection of test cases will reduce the regression test suite.

What are the Regression Testing tools?

Regression Testing is a vital part of the QA process; while performing the regression we may face the below challenges:

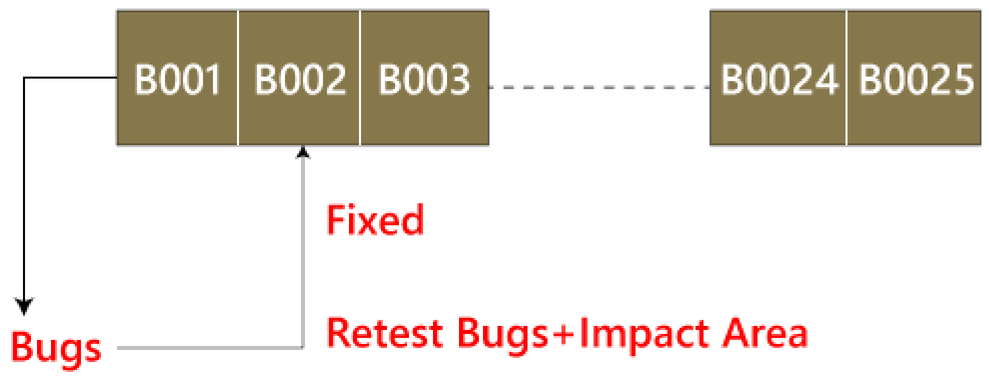
- **Time Consuming**
  Regression Testing consumes a lot of time to complete. Regression testing involves existing tests again, so testers are not excited to re-run the test.
- **Complex**
  Regression Testing is complex as well when there is a need to update any product; lists of the test are also increasing.
- **Communicating business rule**
  Regression Testing ensures the existing product features are still in working order. Communication about regression testing with a non-technical leader can be a difficult task. The executive wants to see the product move forward and making a considerable time investment in regression testing to ensure existing functionality working can be hard.
- **Identify Impact Area**
- **Test Cases Increases Release by Release**
- **Less Resources**
- **No Accuracy**
- **Repetitive Task**
- **Monotonous Job**

Regression Testing Process

The regression testing process can be performed across the **builds** and the **releases**.

Regression testing across the builds

Whenever the bug fixed, we retest the Bug, and if there is any dependent module, we go for a Regression Testing.



For example, How we perform the regression testing if we have different builds as **Build 1, Build 2, and Build 3**, which having different scenarios.

Build1

- Firstly the client will provide the business needs.
- Then the development team starts developing the features.
- After that, the testing team will start writing the test cases; for example, they write 900 test cases for the release#1 of the product.
- And then, they will start implementing the test cases.
- Once the product is released, the customer performs one round of acceptance testing.
- And in the end, the product is moved to the production server.

Build2

- Now, the customer asks for 3-4 extra (new) features to be added and also provides the requirements for the new features.
- The development team starts developing new features.
- After that, the testing team will start writing the test case for the new features, and they write about 150 new test cases. Therefore, the total number of the test case written is 1050 for both the releases.
- Now the testing team starts testing the new features using 150 new test cases.
- Once it is done, they will begin testing the old features with the help of 900 test cases to verify that adding the new feature has damaged the old features or not.
- Here, testing the old features is known as **Regression Testing**.
- Once all the features (New and Old) have been tested, the product is handed over to the customer, and then the customer will do the acceptance testing.

- Once the acceptance testing is done, the product is moved to the production server.

Build3

- After the second release, the customer wants to remove one of the features like Sales.
- Then he/she will delete all the test cases which are belonging to the sales module (about 120 test cases).
- And then, test the other feature for verifying that if all the other features are working fine after removing the sales module test cases, and this process is done under the regression testing.

Note:

- Testing the stable features to ensure that it is broken because of the changes. Here changes imply that the **modification, addition, bug fixing, or the deletion**.
- Re-execution of the same test cases in the different builds or releases is to ensure that changes (modification, addition, bug fixing, or the deletion) are not introducing bugs in stable features.

Regression Testing Across the Release

The regression testing process starts whenever there is a new Release for same project because the new feature may affect the old elements in the previous releases.

To understand the regression testing process, we will follow the below steps:

Step1

There is no regression testing in **Release#1** because there is no modification happen in the Release#1 as the release is new itself.

Step2

The concept of Regression testing starts from **Release#2** when the customer gives some **new requirements**.

Step3

After getting the new requirements (modifying features) first, they (the developers and test engineers) will understand the needs before going to the **impact analysis**.

Step4

After understanding the new requirements, we will perform one round of **impact analysis** to avoid the major risk, but here the question arises who will do the Impact analysis?

Step5

The impact analysis is done by the **customer** based on their **business knowledge**, the **developer** based on their **coding knowledge**, and most importantly, it is done by the **test engineer** because they have the **product knowledge**.

Step6

Once we are done with the **impact area**, then the developer will prepare the **impact area (document)**, and the **customer** will also prepare the **impact area document** so that we can achieve the **maximum coverage of impact analysis**.

Step7

After completing the impact analysis, the developer, the customer, and the test engineer will send the **Reports#** of the impact area documents to the **Test Lead**. And in the meantime, the test engineer and the developer are busy working on the new test case.

Step8

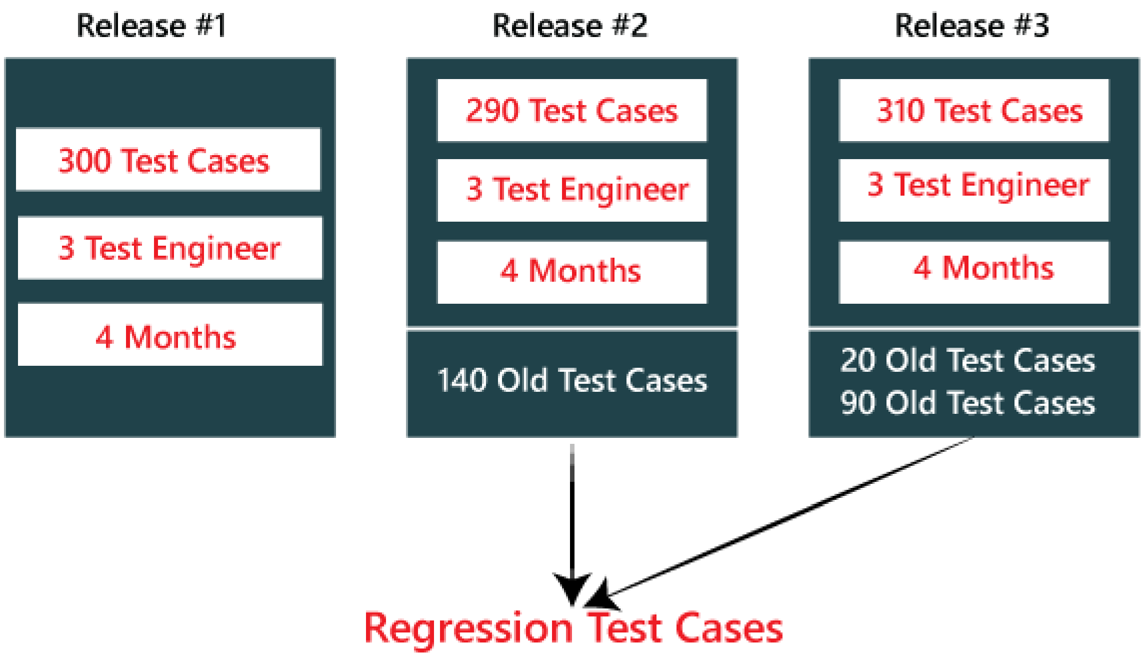
Once the Test lead gets the Reports#, he/she will **consolidate** the reports and stored in the **test case requirement repository** for the release#1.

Step9

After that, the Test Lead will take the help of RTM and pick the necessary **regression test case** from the **test case repository**, and those files will be placed in the **Regression Test Suite**.

Note:

- The test lead will store the regression test case in the regression test suite for no further confusion.
- **Regression test suite**: Here, we will save all the impact area test documents.
- **Regression Test Cases**: These are the test cases of the old releases text document which need to be re-executed as we can see in the below image:

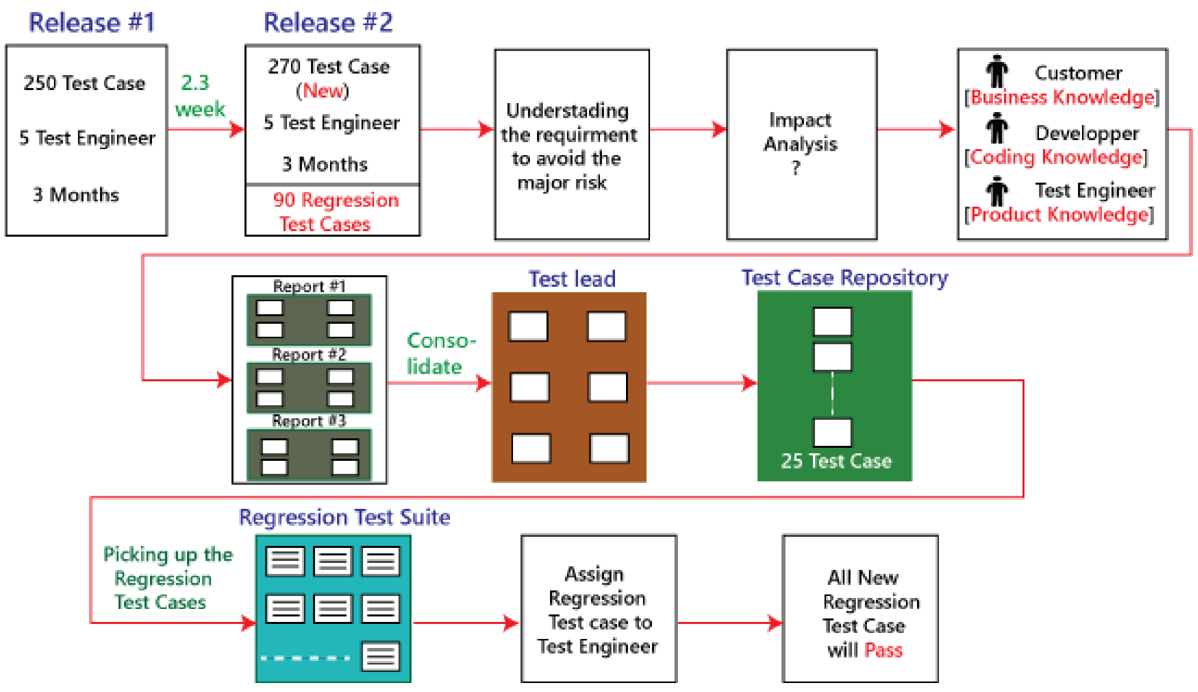


Step10

After that, when the test engineer has done working on the new test cases, the test lead will **assign the regression test case** to the test engineer.

Step11

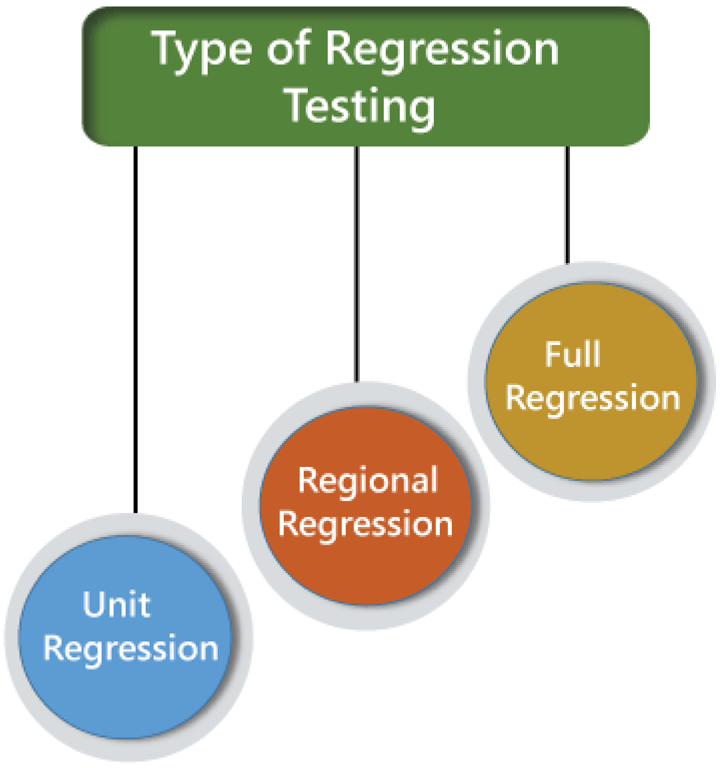
When all the regression test cases and the new features are **stable and pass**, then check the **impact area using the test case** until it is durable for old features plus the new features, and then it will be handed over to the customer.


Release #1
250 Test Case
5 Test Engineer
3 Months
2.3 week
Release #2
270 Test Case
(New)
5 Test Engineer
3 Months
90 Regression Test Cases
Understading the requirment to avoid the major risk
Impact Analysis ?
Customer [Business Knowledge]
Developper [Coding Knowledge]
Test Engineer [Product Knowledge]
Report #1
Report #2
Report #3
Conso-lidate
Test lead
Test Case Repository
25 Test Case
Regression Test Suite
Picking up the Regression Test Cases
Assign Regression Test case to Test Engineer
All New Regression Test Case will Pass

Types of Regression Testing

The different types of Regression Testing are as follows:

1. Unit Regression Testing [URT]
2. Regional Regression Testing[RRT]
3. Full or Complete Regression Testing [FRT]

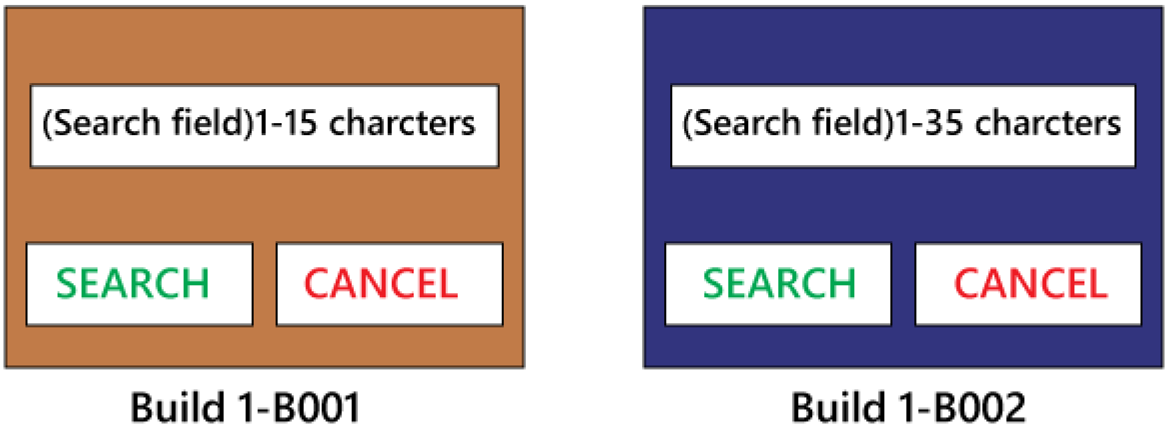


1) Unit Regression Testing [URT]

In this, we are going to test only the changed unit, not the impact area, because it may affect the components of the same module.

Example1

In the below application, and in the first build, the developer develops the **Search** button that accepts **1-15 characters**. Then the test engineer tests the Search button with the help of the **test case design technique**.

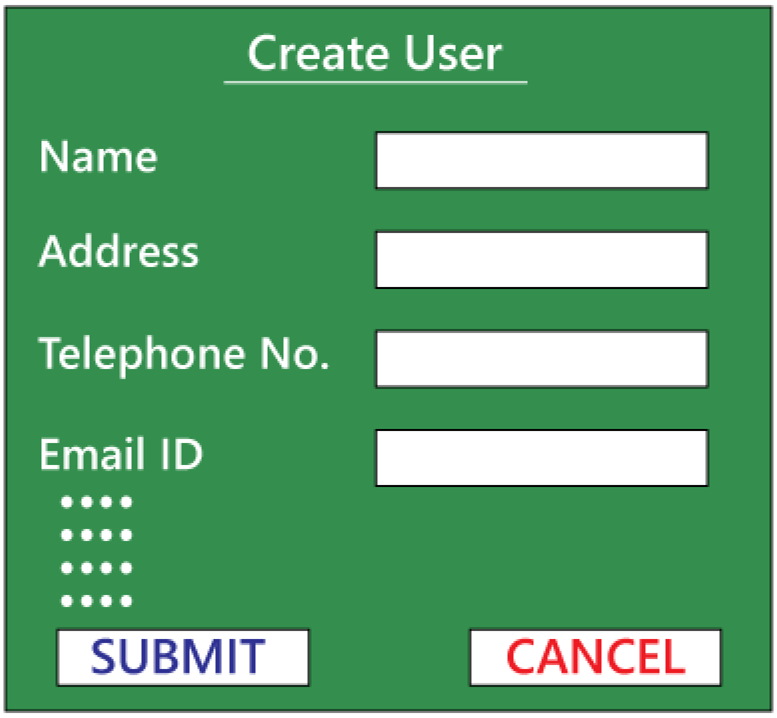


Now, the client does some modification in the requirement and also requests that the **Search button** can accept the **1-35 characters**. The test engineer will test only the Search button to verify that it takes 1-35 characters and does not check any further feature of the first build.

Example2

Here, we have **Build B001**, and a defect is identified, and the report is delivered to the developer. The developer will fix the bug and sends along with some new features which are developed in the second **Build B002**. After that, the test engineer will test only after the defect is fixed.

- The test engineer will identify that clicking on the **Submit** button goes to the blank page.
- And it is a defect, and it is sent to the developer for fixing it.
- When the new build comes along with the bug fixes, the test engineer will test only the Submit button.
- And here, we are not going to check other features of the first build and move to test the new features and sent in the second build.
- We are sure that fixing the **Submit**button is not going to affect the other features, so we test only the fixed bug.



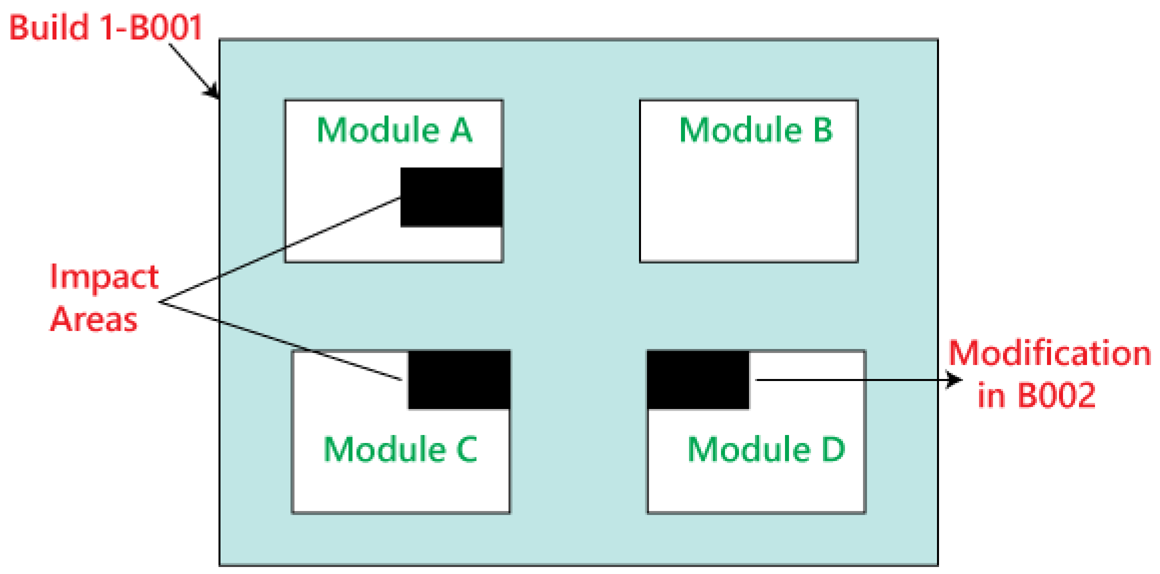
Therefore, we can say that by testing only the changed feature is called the **Unit Regression Testing**.

2) Regional Regression testing [RRT]

In this, we are going to test the modification along with the impact area or regions, are called the **Regional Regression testing**. Here, we are testing the impact area because if there are dependable modules, it will affect the other modules also.

For example:

In the below image as we can see that we have four different modules, such as **Module A, Module B, Module C, and Module D**, which are provided by the developers for the testing during the first build. Now, the test engineer will identify the bugs in **Module D**. The bug report is sent to the developers, and the development team fixes those defects and sends the second build.



In the second build, the previous defects are fixed. Now the test engineer understands that the bug fixing in Module D has impacted some features in **Module A and Module C**. Hence, the test engineer first tests the Module D where the bug has been fixed and then checks the impact areas in **Module A and Module C**. Therefore, this testing is known as **Regional regression testing.**

While performing the regional regression testing, we may face the below problem:

Problem:

In the first build, the client sends some modification in requirement and also wants to add new features in the product. The needs are sent to both the teams, i.e., development and testing.

After getting the requirements, the development team starts doing the modification and also develops the new features based on the needs.

Now, the test lead sends mail to the clients and asks them that all are the impact areas that will be affected after the necessary modification have been done. Therefore, the customer will get an idea, which all features are needed to be tested again. And he/she will also send a mail to the development team to know which all areas in the application will be affected as a result of the changes and additions of new features.

And similarly, the customer sends a mail to the testing team for a list of impact areas. Hence, the test lead will collect the impact list from the client, development team, and the testing team as well.

This **Impact list** is sent to all the test engineers who look at the list and check if their features are modified and if yes, then they do **regional regression testing**. The impact areas and modified areas are all tested by the respective engineers. Every test engineer tests only their features that could have been affected as a result of the modification.

The problem with this above approach is that the test lead may not get the whole idea of the impact areas because the development team and the client may not have so much time to revert his/her mails.

Solution

To resolve the above problem, we will follow the below process:

When a new build comes along with the latest features and bug fixes, the testing team will arrange the meeting where they will talk about if their features are affecting because of the above modification. Therefore, they will do one round of **Impact Analysis** and generate the **Impact List**. In this particular list, the test engineer tries to enclose the maximum probable impact areas, which also decreases the chance of getting the defects.

When a new build comes, the testing team will follow the below procedure:

- They will do smoke testing to check the basic functionality of an application.
- Then they will test new features.
- After that, they will check the changed features.
- Once they are done with checking the changed features, the test engineer will re-test the bugs.
- And then they will check the impact area by performing the regional regression testing.

Disadvantage of using Unit and Regional Regression testing

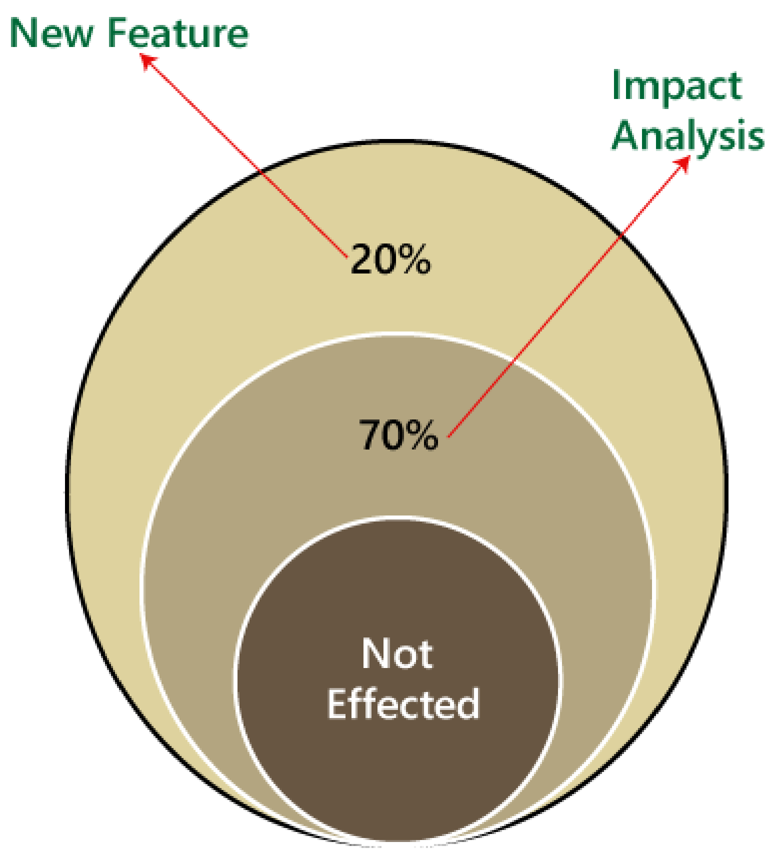
Following are some of the drawbacks of using unit and Regional regression testing:

- We may miss some impact area.
- It is possible that we may identify the wrong impact area.

3) Full Regression testing [FRT]

During the second and the third release of the product, the client asks for adding 3-4 new features, and also some defects need to be fixed from the previous release. Then the testing team will do the Impact Analysis and identify that the above modification will lead us to test the entire product.

Therefore, we can say that testing the **modified features** and **all the remaining (old) features** is called the **Full Regression testing**.



When we perform Full Regression testing?

We will perform the FRT when we have the following conditions:

- When the modification is happening in the source file of the product. **For example**, JVM is the root file of the JAVA application, and if any change is going to happen in JVM, then the entire JAVA program will be tested.
- When we have to perform n-number of changes.

Note:

The regional regression testing is the ideal approach of regression testing, but the issue is, we may miss lots of defects while performing the Regional Regression testing.

And here we are going to solve this issue with the help of the following approach:

- When the application is given for the testing, the test engineer will test the first 10-14 cycle, and will do the **RRT**.
- Then for the 15th cycle, we do FRT. And again, for the next 10-15 cycle, we do **Regional regression testing**, and for the 31th cycle, we do the **full regression testing**, and we will continue like this.

- But for the last ten cycle of the release, we will perform only **complete regression testing**.

Therefore, if we follow the above approach, we can get more defects.

The drawback of doing regression testing manually repeatedly:

- Productivity will decrease.
- It is a difficult job to do.
- There is no consistency in test execution.
- And the test execution time is also increased.

Hence, we will go for the automation to get over with these issues; when we have n-number of the regression test cycle, we will go for the **automation regression testing process**.

Automated Regression Testing Process

Generally we go for the automation whenever there are multiple releases or multiple regression cycle or there is the repetitive task.

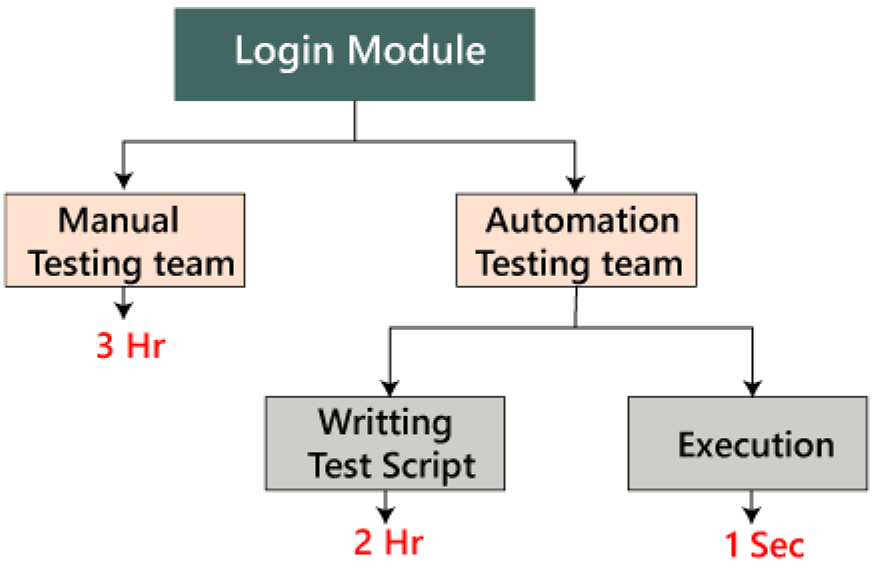
The automation regression testing process can be done in the following steps:

Note1:

The process of testing the application by using some tools is known as automation testing.

Suppose if we take one sample example of a **Login module**, then how we can perform the regression testing.

Here, the Login can be done in two ways, which are as follows:



Manually: In this, we will perform regression only one and twice.

Automation: In this, we will do the automation multiple times as we have to write the test scripts and do the execution.

Note2: In real-time, if we have faced some issues such as:

Issues	Handle by
New features	Manual test engineer
Regressing testing features	Automation test engineer
Remaining (110 feature + Release#1)	Manual test engineer

Step1

When the new release starts, we don't go for the automation because there is no concept of regression testing and regression test case as we understood this in the above process.

Step2

When the new release and the enhancement starts, we have two teams, i.e., manual team and the automation team.

Step3

The manual team will go through the requirements and also identify the impact area and hand over the **requirement test suite** to the automation team.

Step4

Now, the manual team starts working on the new features, and the automation team will start developing the test script and also start automating the test case, which means that the regression test cases will be converted into the test script.

Step5

Before they (automation team) start automating the test case, they will also analyze which all cases can be automated or not.

Step6

Based on the analysis, they will start the automation i.e., converting every regression test cases into the test script.

Step7

During this process, they will take help of the **Regression cases** because they don't have product knowledge as well as the **tool** and the **application**.

Step8

Once the test script is ready, they will start the execution of these scripts on the new application [old feature]. Since, the test script is written with the help of the regression feature or the old feature.

Step9

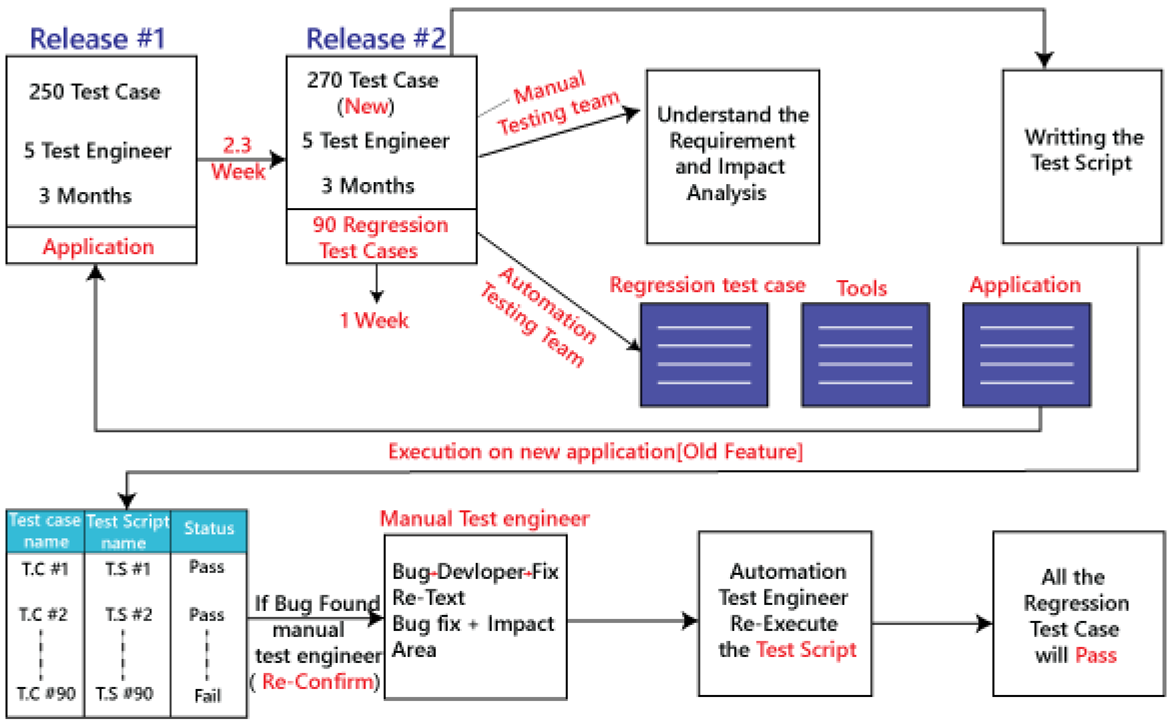
Once the execution is completed, we get a different status like **Pass/fail**.

Step10

If the status is failed, which means it needs to be re-confirmed manually, and if the Bug exists, then it will report to the concerned developer. When the developer fixes that bug, the Bug needs to be re-tested along with the Impact area by the manual test engineer, and also the script needs to be re-executed by the automation test engineer.

Step11

This process goes on until all the new features, and the regression feature will be passed.



Benefits of doing regression testing by the automation testing:

- **Accuracy** always exists because the task is done by the tools and tools never get bored or tired.
- The test script can be re-used across multiple releases.
- **Batch execution** is possible using the automation i.e.; all the written test scripts can be executed parallel or simultaneously.
- Even though the number of regression test case increase release per release, and we don't have to increase the automation resource since some regression case are already automated from the previous release.
- It is a **time-saving process** because the execution is always faster than the manual method.

How to select test cases for regression testing?

It was found from industry inspection. The several defects reported by the customer were due to last-minute bug fixes. These creating side effects and hence selecting the Test Case for regression testing is an art, not an easy task.

Regression test can be done by:

- A test case which has frequent defects
- Functionalities which are more visible to users.
- Test cases verify the core features of the product.
- All integration test cases
- All complex test cases
- Boundary value test cases
- A sample of successful test cases
- Failure of test cases

Regression Testing Tools

If Software undergoes frequent changes, regression testing costs also increase. In those cases, manual execution of test cases increases test execution time as well as costs. In that case, automation testing is the best choice. The duration of automation depends on the number of test cases that remain reusable for successive regression cycles.

Following are the essential tools used for regression testing:

Selenium

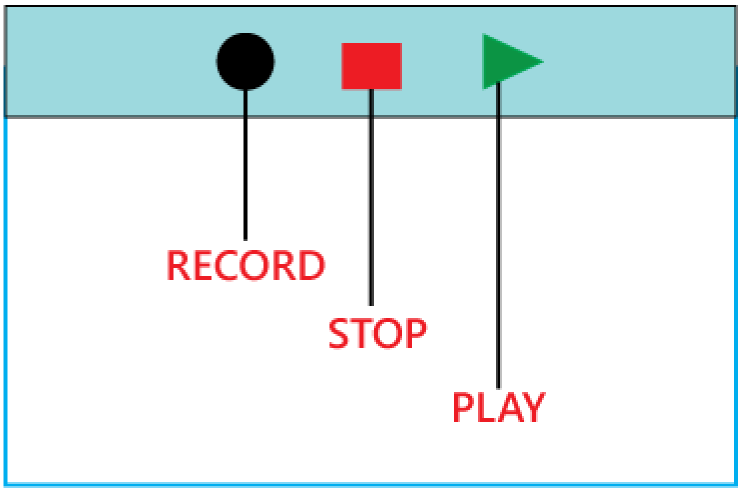
Selenium is an open-source tool. This tool used for automated testing of a web application. For browser-based regression testing, selenium used. Selenium used for UI level regression test for web-based application.

Ranorex Studio

All in one regression test automation for desktop, web, and mobile apps with built-in Selenium Web Driver. Ranorex Studio includes full IDE plus tools for codeless automation.

Quick Test Professional (QTP)

QTP is an automated testing tool used for Regression and Functional Testing. It is a Data-Driven, keyword-based tool. It used VBScript language for automation. If we open the QTP tool, we see the three buttons which are **Record, Play and Stop**. These buttons help to record every click and action performed on the computer system. It records the actions and play it back.



Rational Functional Tester (RTF)

Rational functional tester is a Java tool used to automate the test cases of software applications. RTF used for automating regression test cases, and it also integrates with the rational functional tester.

Regression Testing and Configuration Management

Configuration Management in the regression testing becomes imperative in Agile Environments, where a code is continuously modified. To ensure a valid regression test, we must follow the steps:

- Changes are not allowed in the code during the regression testing phase.
- A regression test case must be unaffected developer changes.
- The database used for regression testing must be isolated; changes are not allowed in the database.

Differences between Retesting and Regression Testing

Re-testing Testing means testing the functionality or bug again to ensure the code fixed. If not set, defects need not be re-opened. If fixed, the defect closed.

Re-testing is a type of testing which performed to check the test-cases that were unsuccessful in the final execution are successfully pass after the defects repaired.

Regression Testing means testing the software application when it undergoes a code change to ensure that new code has not affected other parts of the Software.

Regression testing is a type of testing executed to check whether a code has not changed the existing functionality of the application.

Differences between the Re-testing and Regression Testing are as follows:

Re-testing	Regression Testing
Re-testing is performed to ensure that the test cases that are failed in the final execution are passing after the defects fixed.	Regression Testing is done to confirm whether the code change has not affected the existing features.
Re-Testing works on defect fixes.	The purpose of regression testing is to ensure that the code changes adversely not affect the existing functionality.
Defect verification is the part of the Retesting.	Regression testing does not include defect verification
The priority of Retesting is higher than Regression Testing, so it is done before the Regression Testing.	Based on the project type and availability of resources, regression testing can be parallel to Retesting.
Re-Test is a planned Testing.	Regression testing is a generic Testing.

We cannot automate the test-cases for Retesting.	We can do automation for regression testing; manual testing could be expensive and time-consuming.
Re-testing is for failed test-cases.	Regression testing is for passed Test-cases.
Re-testing make sure that the original fault is corrected.	Regression testing checks for unexpected side effect.
Retesting executes defects with the same data and the same environment with different input with a new build.	Regression testing is when there is a modification or changes become mandatory in an existing project.
Re-testing cannot do before start testing.	Regression testing can obtain test cases from the functional specification, user tutorials and manuals, and defects reports in regards to the corrected problem.

Advantages of Regression Testing

Advantages of Regression Testing are:

- Regression Testing increases the product's quality.
- It ensures that any bug fix or changes do not impact the existing functionality of the product.
- Automation tools can be used for regression testing.
- It makes sure the issues fixed do not occur again.

Disadvantages of Regression Testing

There are several advantages of Regression Testing though there are disadvantages as well.

- Regression Testing should be done for small changes in the code because even a slight change in the code can create issues in the existing functionality.
- If in case automation is not used in the project for testing, it will time consuming and tedious task to execute the test again and again.

CHALLENGES :

Regression testing is a critical aspect of the software development lifecycle that involves retesting a software application to ensure that new changes or updates do not negatively impact existing functionality. While regression testing is essential for maintaining the integrity of the software, it comes with its own set of challenges when applied in real-world scenarios. Here are some common challenges associated with regression testing:

1. Test Case Maintenance:

Challenge:

- **Description:** As the software evolves, new features are added, and existing features are modified. This requires continuous updates to the test cases to reflect these changes accurately.

Explanation:

- As developers introduce changes or enhancements to the codebase, the corresponding test cases must be updated to accommodate these modifications. Failure to keep the test cases aligned with the evolving software can lead to inaccurate test results and missed defects.

Mitigation:

- Establish a robust test case management system.
- Use version control systems to track changes in both code and test cases.
- Regularly review and update test cases in tandem with code changes.

2. Time and Resource Constraints:

Challenge:

- **Description:** Regression testing can be time-consuming, especially for large and complex software systems. Allocating sufficient time and resources for thorough regression testing becomes a challenge, particularly in agile development environments with frequent releases.

Explanation:

- The need for quick releases and continuous delivery can limit the time available for comprehensive regression testing. Rapid development cycles may result in incomplete testing, increasing the risk of overlooking critical defects.

Mitigation:

- Prioritize test cases based on critical functionalities.
- Implement automated regression testing to expedite the testing process.
- Use parallel testing to distribute test execution across multiple environments.

3. Test Data Management:

Challenge:

- **Description:** Managing test data for a variety of test cases can be complex. Changes in the software may require updates to test data, and maintaining a realistic and diverse set of data adds to the complexity.

Explanation:

- Test data needs to cover various scenarios and edge cases to ensure thorough testing. However, managing and maintaining this diverse set of data, especially when dealing with large datasets, can be challenging.

Mitigation:

- Develop a strategy for creating and managing test data.
- Use data generation tools to automate the creation of diverse datasets.
- Refresh test environments with relevant data regularly.

4. Versioning and Configuration Management:

Challenge:

- **Description:** In projects with multiple branches, managing different versions and configurations can be challenging. Ensuring that the correct set of tests is executed for a particular version or configuration is crucial.

Explanation:

- Different branches of the codebase may have unique features or configurations. Ensuring that the appropriate set of regression tests is executed for each version or configuration is essential to avoid false negatives or positives.

Mitigation:

- Implement a versioning and configuration management system.
- Tag releases and associate the corresponding regression test suite with each version.
- Use continuous integration tools to automate version-specific test execution.

5. Automated Test Script Maintenance:

Challenge:

- **Description:** When automated tests are used for regression testing, maintaining the test scripts becomes a challenge. Any changes in the application may require corresponding updates in the test scripts to ensure they remain accurate and effective.

Explanation:

- As the application evolves, UI elements, workflows, or underlying code structures may change, rendering existing automated scripts obsolete. The effort required to update and maintain automated scripts can be significant.

Mitigation:

- Design modular and maintainable automated test scripts.
- Use automation frameworks that support easy script maintenance.
- Regularly review and update automated scripts to align with application changes.

6. Prioritization of Test Cases:

Challenge:

- **Description:** Not all test cases can be executed in every regression cycle. Prioritizing test cases based on critical functionalities and areas impacted by recent changes is essential.

Explanation:

- In a time-constrained environment, it may not be feasible to execute the entire regression test suite for every release. Deciding which test cases to prioritize becomes crucial to achieve a balance between thorough testing and rapid releases.

Mitigation:

- Employ risk-based testing to prioritize test cases based on potential impact.
- Focus on high-risk areas and critical functionalities during each regression cycle.
- Create a priority matrix to guide test case selection.

7. Dependency Management:

Challenge:

- **Description:** Identifying and managing dependencies between different modules or components is crucial. Changes in one part of the software may have ripple effects on other areas, and regression testing must account for these dependencies.

Explanation:

- Interconnected modules or components may have dependencies that are not immediately apparent. Changes in one module may impact the behavior of dependent modules, requiring careful consideration during regression testing.

Mitigation:

- Document and analyze dependencies between different modules.
- Develop a comprehensive understanding of the application architecture.
- Conduct impact analysis before making changes to identify potential dependencies.

8. Parallel Execution and Scalability:

Challenge:

- **Description:** To expedite regression testing, running test cases in parallel is often necessary. However, ensuring scalability and efficient resource utilization in a parallel testing environment can be challenging.

Explanation:

- Scaling up regression testing by running multiple test cases simultaneously can strain resources, leading to bottlenecks and increased test execution times. Efficient distribution and coordination of parallel test execution are critical.

Mitigation:

- Use parallel testing wisely, considering available resources.
- Optimize test execution by distributing tests based on dependencies and resource availability.
- Implement a scalable testing infrastructure to accommodate growing test suites.

Addressing these challenges requires a strategic and well-coordinated approach to regression testing. A combination of test automation, effective test case management, and collaboration among development and testing teams is essential for successful regression testing in real-world software development scenarios.