VARIOUS OPEN SOURCE SOFTWARE TESTING TOOLS

- ***What are broken links on a Web page?***
- A ***broken link,*** often called a ***dead link,*** is any link on a web page that no longer works because there is an underlying issue with the link. When someone clicks on such a link, sometimes an error message is displayed like a page not found. There may not be any error message at all. These are essentially *invalid HTTP* requests and have ***4xx*** and ***5xx*** status code. Some common reasons for a broken link on a webpage can be:

- *The destination web page is down, moved, or no longer exists.*
- *A web page moved without adding a redirect link.*
- *The user entered an improper/misspell URL.*
- *The web page link removed from the website.*
- *With activated firewall settings, also the browser cannot access the destination web page at times.*

- ***A server generates HTTP Status codes*** in response to the request submitted by the client to the server. There are five types of responses to which we can segregate ***HTTP*** response status codes. The first digit of the status-code is the response type, and the last two digits have different interpretations associated with the status code. There are different ***HTTP*** status codes, and a few of them are as below:

- ***200*** *– Valid Link/success*
- ***301/302*** *- Page redirection temporary/permanent*
- ***404*** *– Page not found*
- ***400*** *– Bad request*
- ***401*** *– Unauthorized*
- ***500*** *– Internal Server Error*
- We will be using these *HTTP codes* in our tests to ensure that the link is valid or not.

- **How to identify broken links in Selenium WebDriver**
- Collect all the links present on a web page based on the <a> tag.
- Send HTTP request for each link.
- Verify the HTTP response code.
- Determine if the link is valid or broken based on the HTTP response code.
- Repeat the process for all links captured with the first step

Topics to be covered

- Introduction to JUNIT (TESTNG)
- Introduction to ECLEMMA
- Introduction to SELENIUM

Fact of testing

Testing does not guarantee the absence of defects

What is test case

- A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly

Types of testing

- Test case design techniques can be broadly split into two main categories
  - **Black box (functional)**
  - **White box (structural)**
  - **Gray box**

Testing tools

LOVELY
PROFESSIONAL
UNIVERSITY

Unit testing
with the help of JUnit

Unit Testing

LOVELY PROFESSIONAL UNIVERSITY

- Testing concepts
  - Unit testing
- Testing tools
  - JUnit,TestNG
- Practical use of tools
  - Examples
- How to create JUnit TestCase in Eclipse

JUnit

- JUnit is a framework for writing unit tests
  - A unit test is a test of a *single* class
    - A test case is a single test of a single method
    - A test suite is a collection of test cases
- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

- Que 1. A system designed to work out as per the policy by income tax department for deduction to be paid as per given slabs:

  An employee has Rs 150000 of salary tax free.
  The next Rs. 50000 is taxed at 10%.
  The next Rs 300000 after that is taxed at 22%.
  Any further amount is taxed at 40%.

  a) Write the equivalence class partitioning test cases for above statement.
  b) Write the boundary value analysis test cases.

JUnit..

- JUnit helps the programmer:
  - Define and execute tests and test suites
  - Formalize requirements and clarify architecture
  - Write and debug code

What JUnit does

- JUnit runs a suite of tests and reports results
- For *each* test in the test suite:
  - JUnit calls setUp()
    - This method should create any objects you may need for testing

What JUnit does…

- JUnit calls *one* test method
  - The test method may comprise multiple test cases; that is, it may make multiple calls to the method you are testing
  - In fact, since it's your code, the test method can do anything you want
  - The setUp() method ensures you *entered* the test method with a virgin set of objects; what you do with them is up to you
- JUnit calls tearDown()
  - This method should remove any objects you created

Creating a test class in JUnit

- Define a subclass of TestCase
- Override the setUp() method to initialize object(s) under test.
- Override the tearDown() method to release object(s) under test.
- Define one or more public testXXX() methods that exercise the object(s) under test and assert expected results.
- Define a static suite() factory method that creates a TestSuite containing all the testXXX() methods of the TestCase.
- Optionally define a main() method that runs the TestCase in batch mode.

Fixtures

- A fixture is just a some code you want run before every test
- You get a fixture by overriding the method
  - protected void setUp() { …}
- The general rule for running a test is:
  - protected void runTest() {
    setUp(); <run the test> tearDown();
    }
  - so we can override setUp and/or tearDown, and that code will be run prior to or after every test case

Implementing setUp() method

- Override setUp() to initialize the variables, and objects
- Since setUp() is your code, you can modify it any way you like (such as creating new objects in it)
- Reduces the duplication of code

Implementing the tearDown() method

- In most cases, the tearDown() method doesn't need to do anything
  - The next time you run setUp(), your objects will be replaced, and the old objects will be available for garbage collection
  - Like the finally clause in a try-catch-finally statement, tearDown() is where you would release system resources (such as streams)

The structure of a test method

- A test method doesn’t return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an AssertionFailedError
- The JUnit framework catches the error and deals with it; you don’t have to do anything

assert*X* methods

- static void assertTrue(boolean ***test***)
- static void assertFalse(boolean ***test***)
- assertEquals(***expected***, ***actual***)
  - This method is heavily overloaded: *arg1* and *arg2* must be both objects *or* both of the same primitive type
  - For objects, uses your equals method, ***if*** you have defined it properly, as **public boolean equals(Object o)** --otherwise it uses ==.
- assertSame(Object ***expected***, Object ***actual***)
  - Asserts that two objects refer to the same object (using ==)
- assertNotSame(Object ***expected***, Object ***actual***)
- assertNull(Object ***object***)

assert*X* methods

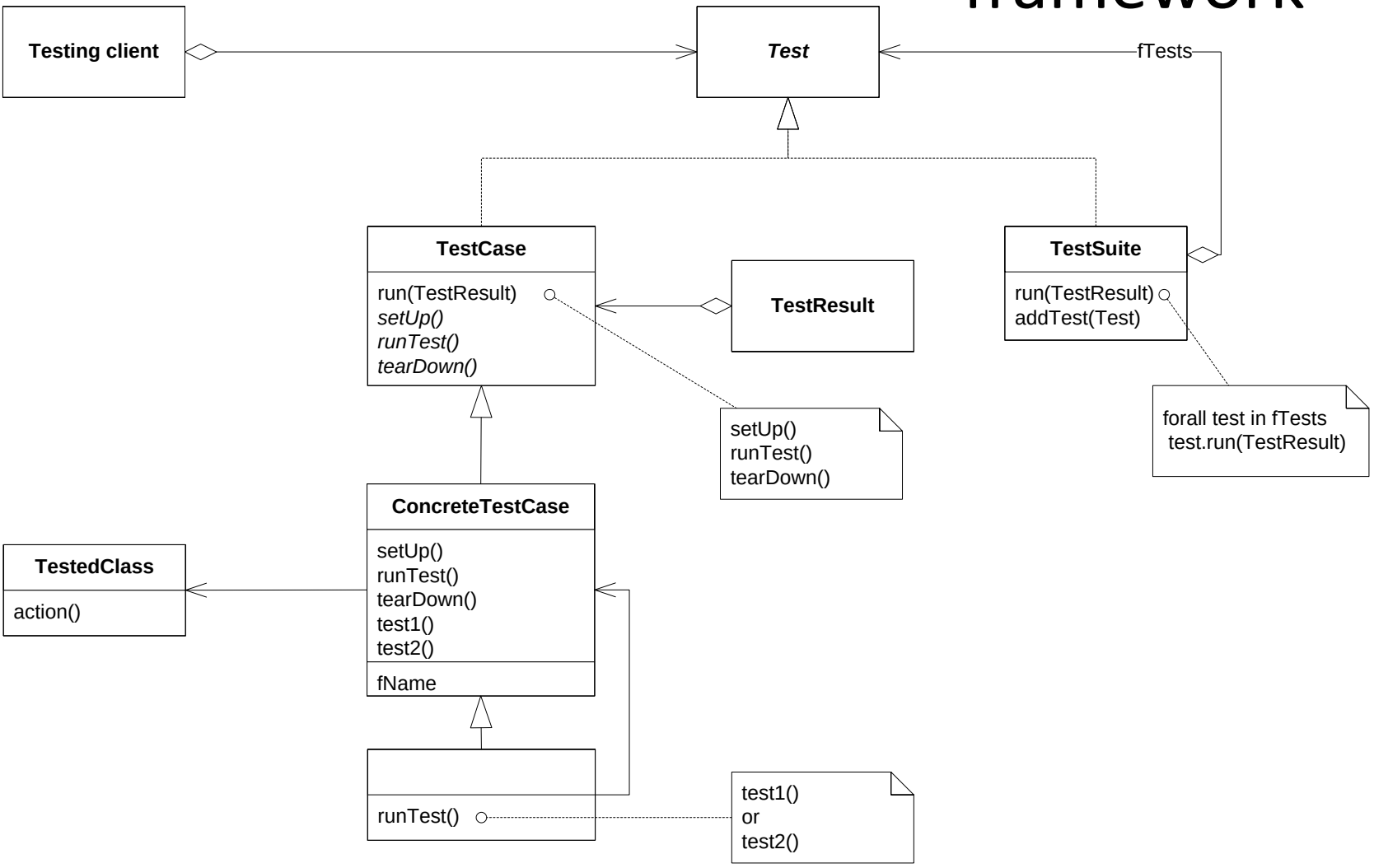
- assertNotNull(Object ***object***)
- fail()
  - Causes the test to fail and throw an AssertionFailedError
  - Useful as a result of a complex test, when the other assert methods aren't quite what you want .

- All the above may take an optional String message as the first argument, for example,
  static void assertTrue(String ***message***, boolean ***test***)

Organize The Tests

- Create test cases in the same package as the code under test
- For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package
- Define similar TestSuite classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application
- Make sure your build process includes the compilation of all tests

JUnit framework



Example: Counter class

- For the sake of example, we will create and test a trivial "counter" class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value

Example: Counter class

- We write the test methods before we write the code
  - This has the advantages described earlier
  - Depending on the JUnit tool we use, we *may* have to create the class first, and we *may* have to populate it with stubs (methods with empty bodies)
- Don't be alarmed if, in this simple example, the JUnit tests are more code than the class itself

JUnit tests for Counter


```
public class CounterTest extends junit.framework.TestCase {
   Counter counter1;

   public CounterTest() { }   // default constructor

   protected void setUp() {   // creates a (simple) test fixture
      counter1 = new Counter();
   }

   protected void tearDown() { } // no resources to release
```