Object-Oriented Software Testing
Use Case Based Testing

Use Case Based Testing

Step 1. Prioritizing the use cases using the requirement-use case traceability matrix.

Step 2. Generating test scenarios from the expanded use cases.

Step 3. Identifying test cases. (Abstract test cases)

Step 4. Generating test data (Concrete tests [using actual values]).

Step 5. Generating test scripts.

Prioritizing Use Cases

- Why prioritizing
  - We may not have needed resources or time.
  - Prioritizing ensures that high-priority use cases are tested.
- The requirement-use case traceability matrix
  - rows are requirements and columns are use cases
  - an entry is checked if the use case realizes the requirement
  - the bottom row sums up the priority of each use case.

Requirement-Use Case Traceability Matrix

	Priority Weight	UC1	UC2	UC3	UC4	UC5	UC6
R1	3	X	X				
R2	2					X	
R3	2	X					
R4	1		X	X			
R5	1				X		X
R6	1		X			X	
Score		5	5	1	1	3	1

Generating Use Case Scenarios

- A scenario is an instance of a use case.
- A scenario is a concrete example showing how a user would use the system.
- A use case has a primary scenario (the normal case) and a number of secondary scenarios (the rare or exceptional cases).
- This step should generate a sufficient set of scenarios for testing the use case.

Example: Register for Company Events

ID: UC5

Name: Register for Event

Actor: Event Participant

Precondition:

Participant has an account on the system and participant is not already registered for the event.

Primary Scenario: (see next slide)

Register for Event: Primary Scenario

user input

Actor: Participant	System: Web App.
1. TUCBW the participant clicks the "Register" button on homepage.	2. Systems ask the participant to enter email and password.
3. Participant enter email and password.	4. System checks that login is correct and displays list of events.
5. Participant selects an event.	6. System asks for event related info.
7. Participant enters event related info.	8. System verifies info and displays a confirmation.
9. TUCEW Participant acknowledges by clicking the OK button on the confirmation message dialog.	system final output

Identifying Secondary Scenarios/Exceptions

Base on user input:

User Input	Normal Case	Abnormal Cases
email	valid	invalid
password	valid	invalid
selected event	still open	• registration closed • duplicate registration
event related info	valid	invalid

Other exceptions:

- system is down
- user quit before completing registration

Identifying Test Cases (Abstract Tests)

- Identifying test cases from the scenarios using a test case matrix:
  - the rows list the test cases identified
  - the columns list the scenarios, the user input variables and system state variables, and the expected outcome for each scenario
  - each entry is either
    - V, indicating a valid variable value for the scenario
    - I, indicating an invalid variable value for the scenario
    - N/A, indicating a not applicable variable value for the scenario

Use Case Based Test Case Generation

Test Case ID	Scenario	Email ID	Password	Registered	Event Info	Event Open	Expected Result
TC1	Successful Registration	V	V	V	V	V	Display confirma-tion
TC2	User Not Found	I	NA	NA	NA	NA	Error msg
TC3	Invalid Info	V	V	NA	I	NA	Error msg
TC4	User Quits	V	V	NA	NA	NA	Back to login
TC5	System Unavailable	V	V	NA	NA	NA	Error msg
TC6	Registration Closed	V	V	NA	NA	I	Error msg
TC7	Duplicate Registration	V	V	I	NA	NA	Error msg

Identifying Test Data Values (Concrete Tests)

Test Case ID	Scenario	Email ID	Pass-Word	Registered	Event Info	Event Open	Expected Result
TC1	Successful Registration	lee@ca.com	Lee123	No	Yes	Yes	Display confirma-tion
TC2	User Not Found	unknow@ca.com	NA	NA	NA	NA	Error msg
TC3	Invalid Info	lee@ca.com	Lee123	NA	I	NA	Error msg
TC4	User Quits	lee@ca.com	Lee123	NA	NA	NA	Back to login
TC5	System Unavailable	lee@ca.com	Lee123	NA	NA	NA	Error msg
TC6	Registration Closed	lee@ca.com	Lee123	NA	NA	No	Error msg
TC7	Duplicate Registration	lee@ca.com	Lee123	Yes	NA	NA	Error msg

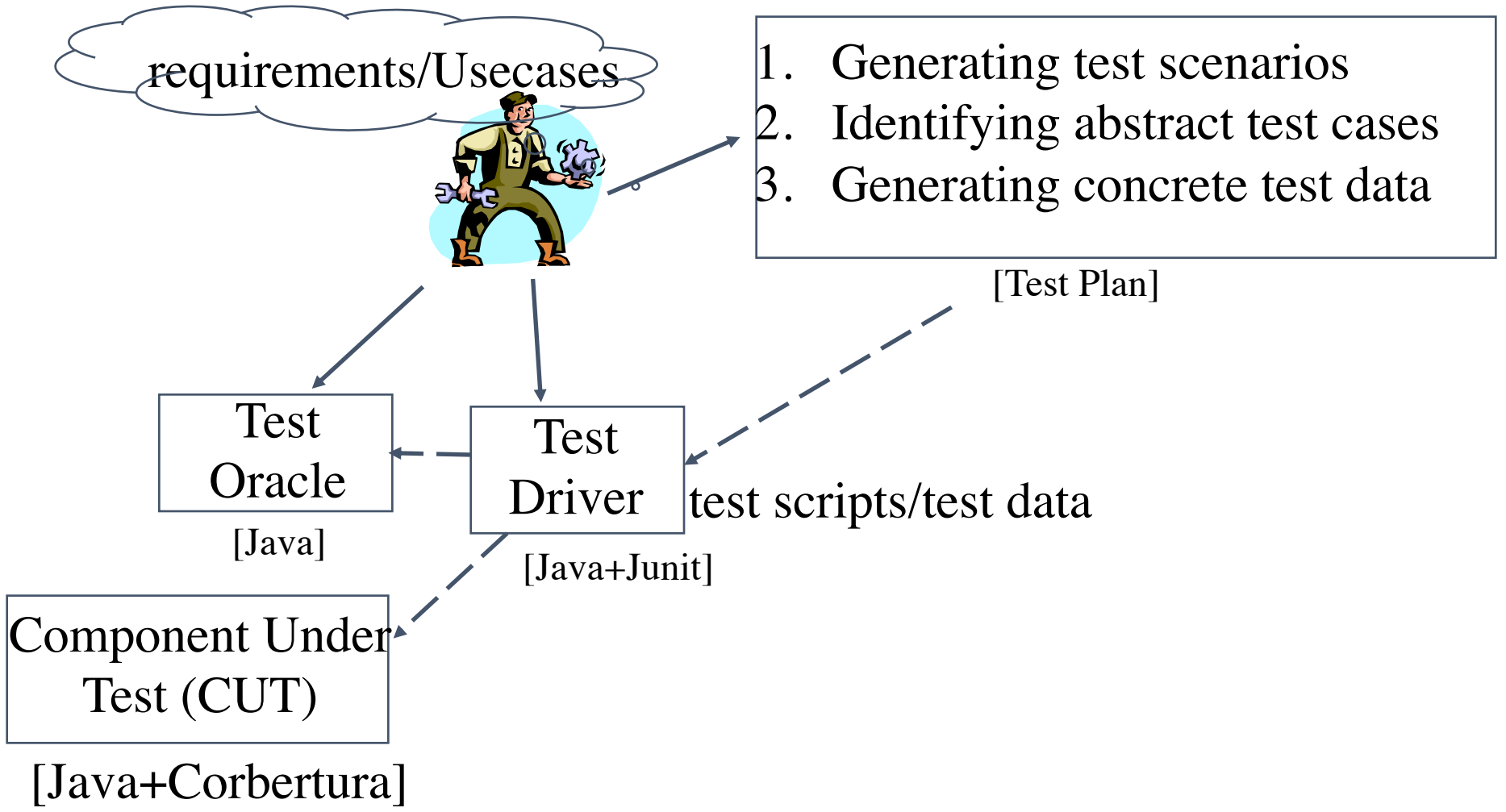
Class Exercise

- Identify input parameters from the expanded use case in the next slide.
- Specify the input values for the parameters.
- Generate use case based test cases.
- Discuss how would you design and implement the test cases in JUnit.

Insert Coin Expanded Use Case

Actor: Customer	System: Vending Machine
1. TUCBW customer inserts a nickel, a dime, or a quarter into the coin slot.	2. System displays the amount inserted.
3. Customer repeats step 1 above.	4. System displays the total amount inserted.
5. TUCEW customer sees the total amount is sufficient or presses the return-coin button.	

Automating Testing and Implementing



Test Driver

A test driver is a class, or a function that

- It creates and initializes an instance of the component under test (CUT).
- It initializes variables and parameters.
- It invokes the CUT.
- It checks the actual outcome for correctness.

Test Stub

A test stub is a class, or a program that

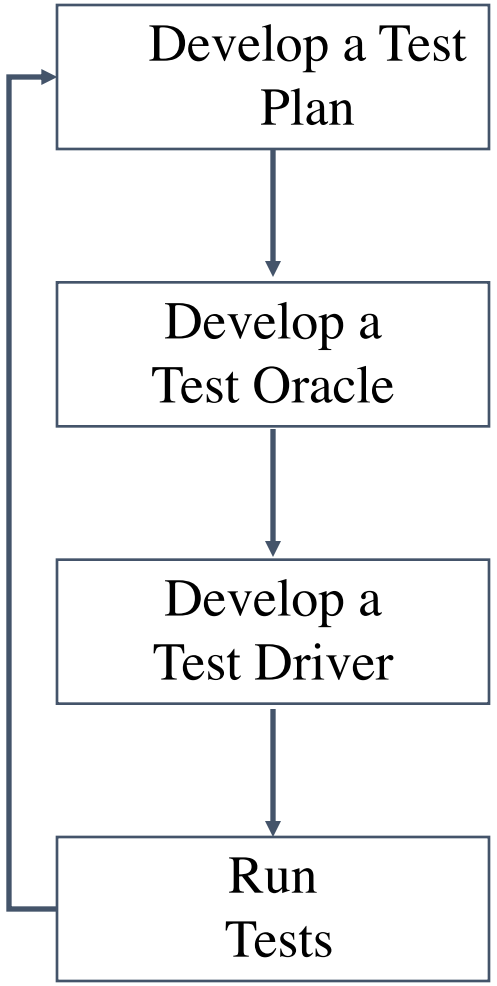
- It is a substitute for an untested, or unimplemented class or component that is invoked by the CUT during testing.
- It is simpler and implements only the functionality needed to test the CUT.
- Most test stubs are “throw-away” software.

Test Oracle

A test oracle is a class that simulates the behavior of the CUT.

- It is used to check the outcome of the CUT.
- It simplifies the test driver.
- It has the same member functions as the CUT.
- Its implementation is much simpler, and need not be as efficient as the CUT.

What a Tester Needs to Do



A test oracle simulates the behavior of the class under test (CUT). It is much simpler than the CUT. It is used to check the result.

The test driver calls the CUT, the oracle and checks the results. It is called when a node or arc is traversed.

The tests are automatically executed by using JUnit.

What is JUnit?

- Open source Java testing framework used to write and run repeatable automated tests
- JUnit is open source (junit.org)
- A structure for writing test drivers
- JUnit features include:
  - Assertions for testing expected results
  - Test features for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners
- JUnit is widely used in industry
- JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse

JUnit Tests

- JUnit can be used to test …
  - … an entire object
  - … part of an object – a method or some interacting methods
  - … interaction between several objects
- It is primarily intended for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests
- Get started at junit.org

Writing Tests for JUnit

- Need to use the methods of the junit.framework.assert class
  - javadoc gives a complete description of its capabilities
- Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded
- The test runner uses the result to report to the user (in command line mode) or update the display (in an IDE)
- All of the methods return void
- A few representative methods of junit.framework.assert
  - *assertTrue (boolean)*
  - *assertTrue (String, boolean)*
  - *fail (String*)

How to Write A Test Case

- You may occasionally see old versions of JUnit tests
  - Major change in syntax and features in JUnit 4.0
  - Backwards compatible (JUnit 3.X tests still work)
- In JUnit 3.X
  1. import junit.framework.*
  2. extend TestCase
  3. name the test methods with a prefix of 'test'
  4. validate conditions using one of the several assert methods
- In JUnit 4.0 and later:
  - Do not extend from Junit.framework.TestCase
  - Do not prefix the test method with "test"
  - Use one of the assert methods
  - Run the test using JUnit4TestAdapter
  - @NAME syntax introduced
- We focus entirely on JUnit 4.X

JUnit Test Fixtures

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (*prefix* values)
  - Reset values (*postfix* values)
- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables
- They should be initialized in a @Before method
- Can be deallocated or reset in an @After method

Simple JUnit Example

```
public class Calc
{
   static public int add (int a, int b)
   {
      return a + b;
   }
}
```

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalcTest
{
   @Test public void testAdd()
   {
      assertTrue ("Calc sum incorrect",
         5 == Calc.add (2, 3));
   }
}
```

Test values

Printed if assert fails

Expected output

Testing the Min Class

```
im  public static <T extends Comparable<? super T>> T min (List<? extends T> list)
        {
pu      if (list.size() == 0)
{       {
  /*       throw new IllegalArgumentException ("Min.min");
        }
        Iterator<? extends T> itr = list.iterator();
        T result = itr.next();

        if (result == null) throw new NullPointerException ("Min.min");

        while (itr.hasNext())
        {   // throws NPE, CCE as needed
            T comp = itr.next();
            if (comp.compareTo (result) < 0)
            {
                result = comp;
        }   }
        return result;
}
```

MinTest Class

- Standard imports for all JUnit classes :

```
import static org.junit.Assert.*;
import org.junit.*;
import java.util.*;
```

- Test fixture and pre-test setup method (prefix) :

```
private List<String> list;   // Test fixture

// Set up - Called before every test method.
@Before
 public void setUp()
 {
     list = new ArrayList<String>();
 }
```

- Post test teardown method (postfix) :

```
// Tear down - Called after every test method.
@After
public void tearDown()
{
   list = null;   // redundant in this example
}
```

Min Test Cases: NullPointerException

```
@Test public void testForNullList()
{
   list = null;
   try {
       Min.min (list);
   } catch (NullPointerException e) {
       return;
   }
   fail ("NullPointerException expecte
}
```

This NullPointerException test decorates the @Test annotation with the class of the exception

```
@Test (expected = NullPointerException.class)
public void testForNullElement()
{
    list.add (null);
    list.add ("cat");
    Min.min (list);
}
```

This NullPointerException test uses the fail assertion

```
@Test (expected = NullPointerException.class)
public void testForSoloNullElement()
{
    list.add (null);
    Min.min (list);
}
```

This NullPointerException test catches an easily overlooked special case

More Exception Test Cases for Min

```
@Test (expected = ClassCastException.class)
@SuppressWarnings ("unchecked")
public void testMutuallyIncomparable()
{
   List list = new ArrayList();
   list.add ("cat");
   list.add ("dog");
   list.add (1);
   Min.min (list);
}
```

Note that Java generics don't prevent clients from using raw types!

```
@Test (expected = IllegalArgumentException.class)
public void testEmptyList()
{
      Min.min (list);
}
```

Special case:Testing for the empty list

Remaining Test Cases for Min

```
@Test
public void testSingleElement()
{
    list.add ("cat");
    Object obj = Min.min (list);
    assertTrue ("Single Element List", obj.equals ("cat"));
}

@Test
 public void testDoubleElement()
 {
    list.add ("dog");
    list.add ("cat");
    Object obj = Min.min (list);
    assertTrue ("Double Element List", obj.equals ("cat"));
 }
```

Finally! A couple of "Happy Path" tests

Summary: Seven Tests for Min

- Five tests with exceptions
  1. null list
  2. null element with multiple elements
  3. null single element
  4. incomparable types
  5. empty elements
- Two without exceptions
  6. single element
  7. two elements

Data-Driven Tests

- Problem : Testing a function multiple times with similar values
  - How to avoid test code bloat?
- Simple example : Adding two numbers
  - Adding a given pair of numbers is just like adding any other pair
  - You really only want to write one test
- Data-driven unit tests call a constructor for each collection of test values
  - Same tests are then run on each set of data values
  - Collection of data values defined by method tagged with @Parameters annotation

Example JUnit Data-Driven Unit Test

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith (Parameterized.class)
public class DataDrivenCalcTest
{  public int a, b, sum;

   public DataDrivenCalcTest (int v1, int v2, int expected)
   { this.a = v1; this.b = v2; this.sum = expected; }

  @Parameters public static Collection<Object[]> parameters()
  { return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}}); }

  @Test public void additionTest()
  { assertTrue ("Addition Test", sum == Calc.add (a, b)); }
}
```

Constructor is called for each triple of values

Test 1
Test values: 1, 1
Expected: 2

Test 2
Test values: 2, 3
Expected: 5

Test method

Tests with Parameters: JUnit Theories

- Unit tests can have actual parameters
  - So far, we've only seen parameterless test methods
- Contract model: Assume, Act, Assert
  - *Assumptions* (preconditions) limit values appropriately
  - *Action* performs activity under scrutiny
  - *Assertions* (postconditions) check result

```
@Theory public void removeThenAddDoesNotChangeSet (
            Set<String> someSet, String str)  {                    // Parameters!
        assumeTrue (someSet != null)                                // Assume
        assumeTrue (someSet.contains (str)) ;                       // Assume
        Set<String> copy = new HashSet<String>(someSet);  // Act
        copy.remove (str);
        copy.add (str);
        assertTrue (someSet.equals (copy));                         // Assert
}
```

Question: Where Do The Data Values Come From?

- Answer:
  - All combinations of values from @DataPoints annotations where assume clause is true
  - Four (of nine) combinations in this particular case
  - Note: @DataPoints format is an array

```
@DataPoints
public static String[] animals = {"ant", "bat", "cat"};


@DataPoints
public static Set[] animalSets = {
   new HashSet (Arrays.asList ("ant", "bat")),
   new HashSet (Arrays.asList (“bat", “cat", “dog“, “elk")),
   new HashSet (Arrays.asList (“Snap”, “Crackle”, “Pop"))
};
```

Nine combinations of animalSets[i].contains (animals[j]) is false for five combinations

JUnit Theories Need BoilerPlate

```
import org.junit.*;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;
import static org.junit.Assume.*;

import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;

import java.util.*;

@RunWith (Theories.class)
public class SetTheoryTest
{
   …  // See Earlier Slides
}
```

Running from a Command Line

- This is all we need to run JUnit in an IDE (like Eclipse)
- We need a main() for command line execution …

AllTests

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import junit.framework.JUnit4TestAdapter;

// This section declares all of the test classes in the program.
@RunWith (Suite.class)
@Suite.SuiteClasses ({ StackTest.class })  // Add test classes here.

public class AllTests
{
    // Execution begins in main(). This test class executes a
    // test runner that tells the tester if any fail.
    public static void main (String[] args)
    {
       junit.textui.TestRunner.run (suite());
    }

    // The suite() method helps when using JUnit 3 Test Runners or Ant.
    public static junit.framework.Test suite()
    {
       return new JUnit4TestAdapter (AllTests.class);
    }
}
```

How to Run Tests

- JUnit provides test drivers
  - Character-based test driver runs from the command line
  - GUI-based test driver-*junit.swingui.TestRunner*
    - Allows programmer to specify the test class to run
    - Creates a "Run" button

- If a test fails, JUnit gives the location of the failure and any exceptions that were thrown

JUnit Resources

- Some JUnit tutorials
  - http://open.ncsu.edu/se/tutorials/junit/
    (Laurie Williams, Dright Ho, and Sarah Smith)
  - http://www.laliluna.de/eclipse-junit-testing-tutorial.html
    (Sascha Wolski and Sebastian Hennebrueder)
  - http://www.diasparsoftware.com/template.php?content=jUnitStarterGuide
    (Diaspar software)
  - http://www.clarkware.com/articles/JUnitPrimer.html
    (Clarkware consulting)
- JUnit: Download, Documentation
  - http://www.junit.org/

Software Testing
- Web Testing with Selenium Web Driver

What Is Selenium Web Driver ?

- **Selenium WebDriver** is a collection of open source APIs
- Is an free, open source software (https://www.seleniumhq.org/ .
- Used to test web applications
- Supported Browsers: Firefox, IE, Chrome, Safari, Opera, Android, iPhone
- Supported O/S: Windows, OS/X, Linux, Solaris
- Supported Languages: Java, C#, Ruby, Python, JavaScript
- Easy, fast automation development

What Selenium Web Driver Can Do

- This tool is used to automate web application testing to verify that it works as expected.
- Simulates user actions on web browsers
- It provides methods for locating UI elements
- Comparing the expected results with actual application behavior at the user end

How to download and install selenium web driver

1. Go to https://www.seleniumhq.org/
2. Download tab
3. Selenium Client & WebDriver Language Bindings (Which we need the ***java*** programming language)
   - Download the selenium web driver library and use it along with a programming language to create a test automation test scripts
4. Click on download and **unzip** the zip file (**selenium-java-x.y.z.zip**)
   - Note that x,y,z are version numbers, depends on the current version of your web driver
5. Go to the java project under test, include the web driver to the project.
   a. Go to project properties
   b. Go to Java build path
   c. Add external jars in the library tab
   d. Select the all the jar files from the **selenium-java-x.y.z** directory (from step 4)
   e. Select the all the jar files from the **selenium-java-x.y.z/lib** directory as well
   f. Click apply and ok. (Something around 12 jar files should be added to your project library)
6. Go to the http://chromedriver.chromium.org/downloads and download the latest version of the chromedriver executable file.
   - Locate it inside the project workspace folder

Example test plan

Test the Wikipedia page search functionality

1. Open a Wikipedia page
2. Select to English language
3. Search for keyword “software”
4. Observe the software page
5. Close the page

Example junit test in eclipse

```
import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class WikipediaTest {

	WebDriver driver;

	@Before
	public void openWikipediaEnglishPage() throws InterruptedException{
		System.setProperty("webdriver.chrome.driver","chromedriver");
		driver = new ChromeDriver();
		driver.get("https://www.wikipedia.org");
		Assert.assertEquals("Wikipedia", driver.getTitle());
	}
	@Test
	public void testEnglishLink() throws InterruptedException{
		WebElement link = driver.findElement(By.id("js-link-box-en"));
		link.click();
		Thread.sleep(5000);
		Assert.assertEquals("Wikipedia, the free encyclopedia",
driver.getTitle());
	}
	@Test
	public void testSearchBox() throws InterruptedException{
		WebElement searchBox = driver.findElement(By.id("searchInput"));;
		searchBox.sendKeys("Software testing");
		Thread.sleep(5000);
		searchBox.submit();
		Thread.sleep(5000);
		Assert.assertEquals("Software testing - Wikipedia",
driver.getTitle());
	}
	@After
	public void closePage(){
		driver.quit();
	}
}
```