Software Engineering (CSE 355)

Lecture 11

(Automation)

Principles of Software Testing

- There are seven principles of Software Testing.
  1. Testing shows presence of defects
  2. Exhaustive testing is impossible
  3. Early testing
  4. Defect clustering
  5. Pesticide paradox
  6. Testing is context dependent
  7. Absence of error – fallacy

Principles of S/W Testing Cont…

- **Testing Shows Presence of Defects:** Testing shows the presence of defects in the software. The goal of testing is to make the software fail. Sufficient testing reduces the presence of defects. In case testers are unable to find defects after repeated regression testing doesn't mean that the software is bug-free.

- **Exhaustive Testing is Impossible:** Testing all the functionalities using all valid and invalid inputs and preconditions is known as Exhaustive testing. If we keep on testing all possible test conditions then the software execution time and costs will rise. So instead of doing exhaustive testing, risks and priorities will be taken into consideration whilst doing testing and estimating testing efforts.

Principles of S/W Testing Cont...

- **Early Testing:** Defects detected in early phases of SDLC are less expensive to fix. So conducting early testing reduces the cost of fixing defects. It is cheaper to change the incorrect requirement compared to fixing the fully developed functionality which is not working as intended.

- **Defect Clustering:** Defect Clustering in software testing means that a small module or functionality contains most of the bugs or it has the most operational failures. As per the Pareto Principle (80-20 Rule), 80% of issues comes from 20% of modules and remaining 20% of issues from remaining 80% of modules. So we do emphasize testing on the 20% of modules where we face 80% of bugs.

Principles of S/W Testing Cont…

- **Pesticide Paradox:** The process of repeating the same test cases again and again, eventually, the same test cases will no longer find new bugs is called Pesticide Paradox in software testing. So to overcome this Pesticide Paradox, it is necessary to review the test cases regularly and add or update them to find more defects.

- **Testing is Context Dependent:** Testing approach depends on the context of the software we develop. We do test the software differently in different contexts. For example, online banking application requires a different approach of testing compared to an e-commerce site.

Principles of S/W Testing Cont…

- **Absence of Error – Fallacy:** The absence of error does not necessarily mean that the software is error free. 99% of bug-free software may still be unusable, if wrong requirements were incorporated into the software and the software is not addressing the business needs. The software which we built not only be a 99% bug-free software but also it must fulfill the business needs otherwise it will become an unusable software. The testing team must start with the hypothesis that there are errors in the software. Using test cases and other methods, the testing team desires to prove the hypothesis wrong.

Software Assessment

- Two types-
  - Static Analysis: based on factual system documents available such as system requirement study document, design document, and source code.
  - Dynamic Analysis: based on the behavioral and performance properties.

What is Test Automation?

The use of software to control the <u>execution</u> of tests, the <u>comparison</u> of actual outcomes with predicted outcomes, the <u>setting up</u> of test preconditions, and other test <u>control</u> and test <u>reporting</u> functions

- <u>Advantages</u>
  - Reduces cost
  - Reduces human error
  - Reduces variance in test quality from different individuals
  - Significantly reduces the cost of regression testing

Software Testability (3.1)

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met

- Plainly speaking – how hard it is to find faults in the software
- Testability is dominated by two practical problems
  - How to provide the test values to the software
  - How to observe the results of test execution

Observability and Controllability

- **Observability**

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

  – Software that affects hardware devices, databases, or remote files have low observability

- **Controllability**

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

  – Easy to control software with inputs from keyboards

  – Inputs from hardware sensors or distributed software is harder

- Data abstraction reduces controllability and observability

Components of a Test Case (3.2)

- A test case is a multipart artifact with a definite structure
- Test case values

The input values needed to complete an execution of the software under test

- Expected results

The result that will be produced by the test if the software behaves as expected

  – A *test oracle* uses expected results to decide whether a test passed or failed

Affecting Controllability and Observability

- Prefix values

Inputs necessary to put the software into the appropriate state to receive the test case values

- Postfix values

Any inputs that need to be sent to the software after the test case values are sent

1. *Verification Values* : Values needed to see the results of the test case values
2. *Exit Values* : Values or commands needed to terminate the program or otherwise return it to a stable state

Putting Tests Together

- Test case

The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test

- Test set

A set of test cases

- Executable test script

A test case that is prepared in a form to be executed automatically on the test software and produce a report

Test Automation Framework (3.3)

A set of assumptions, concepts, and tools that support test automation

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)*

JUnit Test Fixtures

- A test fixture is the <u>state</u> 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
pu
{
  /
}
```

```
public static <T extends Comparable<? super T>> T min (List<? extends T> list)
   {
      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 expected
}
```

This NullPointerException test uses the **fail** assertion

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 catches an easily overlooked special case

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

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

```java
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/

Summary

- The only way to make testing efficient as well as effective is to automate as much as possible
- Test frameworks provide very simple ways to automate our tests
- It is no "silver bullet" however … it does not solve the hard problem of testing :

What test values to use ?

- This is test design … the purpose of test criteria