# CS409
# Software Testing

**TAN, Shin Hwei**

陈馨慧

Southern University of Science and Technology

Slides adapted from Introduction to Software Testing, Edition 2 (Ch 1)

# Administrative Info

- Do not use wechat for communication!
- Use GitHub Classroom
  - Share the bugs that your find! (Remember to document when you find a bug through knowledge shared in GitHub discussion!)
  - Helps each other with tools installation
- Syllabus uploaded in Sakai
  - The first 3 weeks will be a recap of software testing
  - Then later on, we will learn more about systematic testing

# Grading

- Points
  - Assignment (40%)
  - Exams (20%)
  - Project (20%)
  - Project Final Presentation (20%)

- Assignments and Project is important part of the course!
- Don’t forget to submit all assignments and projects!

# Recap: Fault and Failure Example

- A patient gives a doctor a list of symptoms
  - Failures
- The doctor tries to diagnose the root cause, the ailment
  - Fault
- The doctor may look for anomalous internal conditions (high blood pressure, irregular heartbeat, bacteria in the blood stream)
  - Errors

# A Concrete Example

```
public static int numZero (int [ ] arr)
{  // Effects: If arr is null throw
NullPointerException
   // else return the number of occurrence
arr
   int count = 0;
   for (int i = 1; i <
   {
      if (arr [ i ] == 0)
      {
         count++;
      }
   }
   return count;
}
```

**Fault**: Should start searching at 0, not 1

**Test 1**
[ 2, 7, 0 ]
Expected: 1
Actual: 1

**Error**: i is 1, not 0, on the first iteration
**Failure**: none

**Test 2**
[ 0, 2, 7 ]
Expected: 1
Actual: 0

**Error**: i is 1, not 0
Error propagates to the variable count
**Failure**: count is 0 at the return statement

# Testing in the 21st Century

- More safety critical, real-time software
- Embedded software is ubiquitous … check your pockets
- Enterprise applications means bigger programs, more users
- Paradoxically, free software increases our expectations !
- Security is now all about software faults
  - Secure software is reliable software
- The web offers a new deployment platform
  - Very competitive and very available to more users
  - Web apps are distributed
  - Web apps must be highly reliable

*Industry desperately needs our inventions !*

# What Does This Mean?

**Software testing is getting more important**

**What are we trying to do when we test ?**
**What are our goals ?**

# Testing Goals Based on Test Process Maturity

- Level 0 : There's no difference between testing and debugging
- Level 1 : The purpose of testing is to show correctness
- Level 2 : The purpose of testing is to show that the software doesn't work
- Level 3 : The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4 : Testing is a mental discipline that helps all IT professionals develop higher quality software

# Level 0 Thinking

- Testing is the same as debugging
- Does not distinguish between incorrect behavior and mistakes in the program
- Does not help develop software that is reliable or safe

This is what we teach undergraduate CS majors

# Level 1 Thinking

- Purpose is to show correctness
- Correctness is impossible to achieve
- What do we know if no failures?
  - Good software or bad tests?
- Test engineers have no:
  - Strict goal
  - Real stopping rule
  - Formal test technique
  - Test managers are powerless

This is what hardware engineers often expect

# Level 2 Thinking

- Purpose is to show failures
- Looking for failures is a negative activity
- Puts testers and developers into an adversarial relationship
- What if there are no failures?

This describes most software companies.
How can we move to a *team approach* ??

# Level 3 Thinking

- Testing can only show the presence of failures
- Whenever we use software, we incur some risks
- Risk may be small and consequences unimportant
- Risk may be great and consequences catastrophic
- Testers and developers cooperate to reduce risk

This describes a few "enlightened" software companies

# Level 4 Thinking

**A mental discipline that increases quality**

- Testing is only one way to increase quality
- Test engineers can become technical leaders of the project
- Primary responsibility to measure and improve software qualit
- Their expertise should help the developers

This is the way "traditional" engineering works

# Where Are You?

**Are you at level 0, 1, or 2 ?**

**Is your organization at work at level 0, 1, or 2 ?**
**Or 3?**

**We hope to teach you to become "change agents" in your workplace …**
**Advocates for level 4 thinking**

# Tactical Goals : Why Each Test ?

```
If you don't know why you're
conducting each test, it won't
be very helpful
```

- ✓ Written test objectives and requirements must be documented
- ✓ What are your planned coverage levels?
- ✓ How much testing is enough?
- ✓ Common objective – spend the budget…test until the ship-date …
  - Sometimes called the "date criterion"

# Here! Test This!

**Offutt's first "professional" job**

**A stack of computer printouts—and no documentation**

If you don't start planning for each test when the functional requirements are formed, you'll never know why you're conducting the test

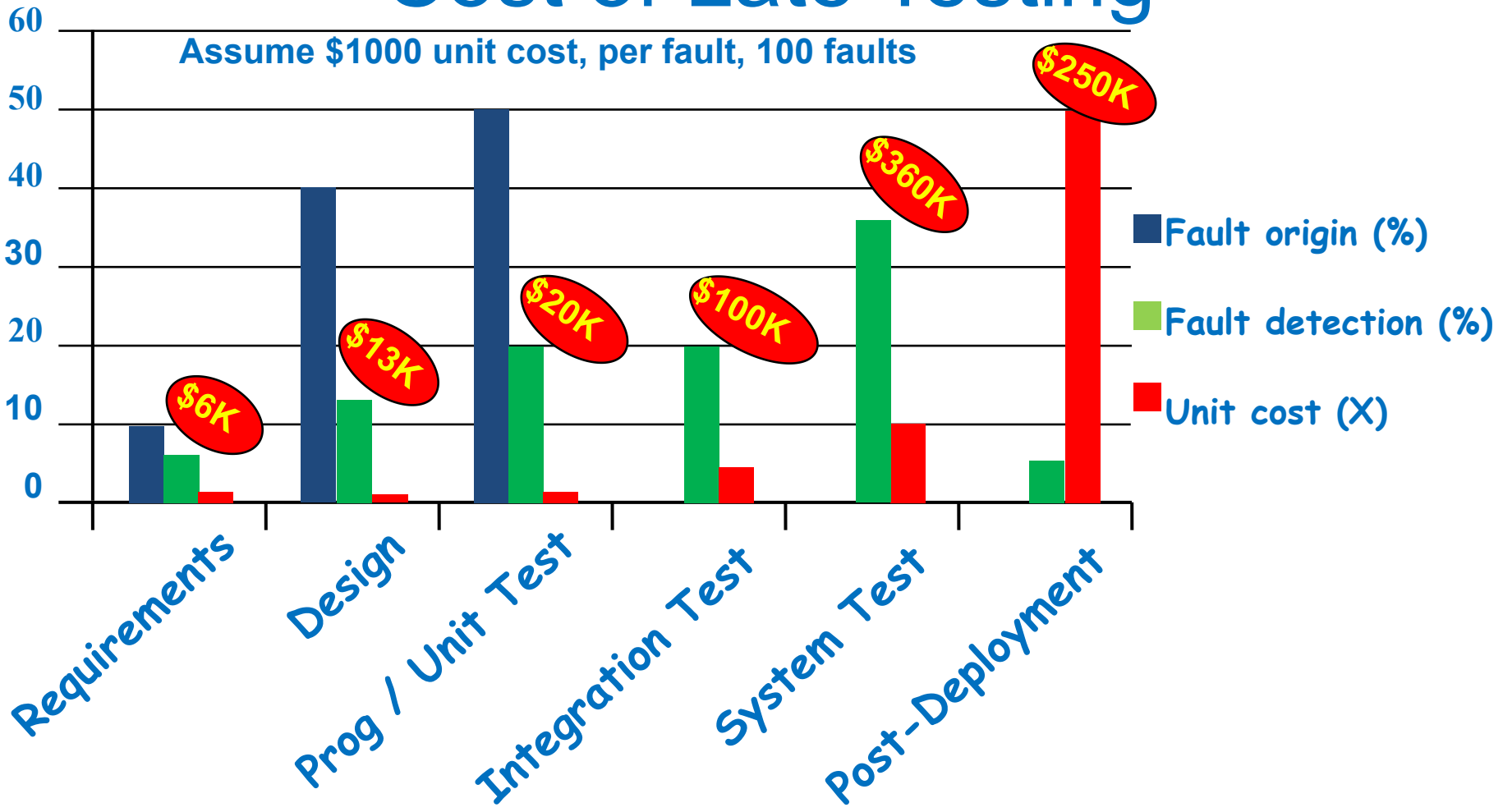
- 1980: "The software shall be easily maintainable"
- Threshold reliability requirements?
- What fact does each test try to verify?
- Requirements definition teams need testers!

# Cost of Not Testing

Poor Program Managers might say:
"Testing is too expensive."

- Testing is the most time consuming and expensive part of software development
- Not testing is even more expensive
- If we have too little testing effort early, the cost of testing increases
- Planning for testing after development is prohibitively expensive

# Cost of Late Testing




**Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002**

# Summary:
# Why Do We Test Software ?

**A tester's goal is to eliminate faults as early as possible**

- **Improve quality**
- **Reduce cost**
- **Preserve customer satisfaction**

# Testing Levels Based on Software Activity

- Unit testing
- (Module testing)
- Integration testing
- System testing
- (Acceptance testing)
- Regression testing

- Names are not standardized
  - I don’t insist on names; we will follow the book

# 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 provide a program with the needed inputs, in terms of values, operations, and behaviors**

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

- Controllability

**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**

  - 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

**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**

Example of Test Automation Framework?

- 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

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 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**

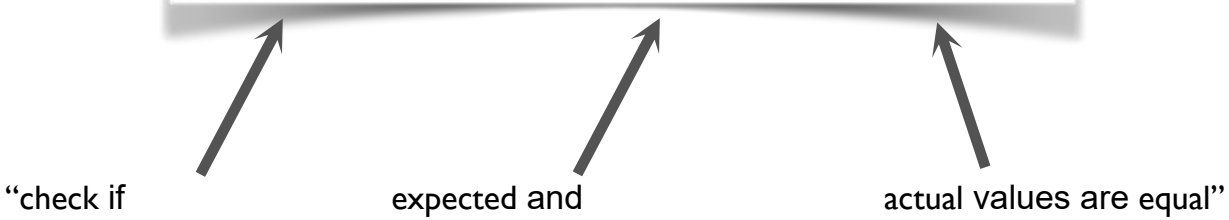
# Summary: Seven Tests for Min

- 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

# Assertions in JUnit

# assert:
## a family of methods to check conditions

```
assertEquals("", result);
```




```
@Test
 public void testWrapNull() {
    Wrapper wrapper = new Wrapper();
  String result = wrapper.wrap(null, 10);
    assertEquals("", result);
 }
```

# assert:

a family of methods to check conditions

```
assertEquals("", result);
```


If assertion fails (checked condition is not satisfied):

- current test is marked as 'failed'
- JUnit skips execution of the rest of the test method and proceeds executing remaining test methods

# assert

- for a boolean condition:

  *assertTrue("message for fail", condition);*

  *assertFalse("message", condition);*

- for object, int, long, and byte values, array:

  *assertEquals(expected_value, expression);*

- for float and double values:

  *assertEquals(expected, expression, error);*

- for objects references:

  *assertNull(obj_ref)*

  *assertNotNull(obj_ref)*

  *assertSame(obj_ref, obj_ref2)*

...

# assert: example

```
@Test
public void testPush() {
      Stack aStack = new Stack();
        assertTrue("Stack should be empty!",
                                  aStack.isEmpty());
        aStack.push(10);
        assertFalse("Stack should not be empty!",
                                  aStack.isEmpty());
        aStack.push(4);
        assertEquals(4, aStack.pop());
        assertEquals(10, aStack.pop());
}
```

# assert: better example

```
@Test
public void testStackEmpty() {
    Stack aStack = new Stack();
    assertTrue("Stack should be empty!", aStack.isEmpty());
    aStack.push(10);
    assertFalse("Stack should not be empty!", aStack.isEmpty());
}

@Test
public void testStackOperations() {
    Stack aStack = new Stack();
    aStack.push(10);
    aStack.push(-4);
    assertEquals(-4, aStack.pop());
    assertEquals(10, aStack.pop());
}
```

Separate *@Test* methods for testing individual scenarios and methods

# Hamcrest Matcher

# What is Hamcrest?

- http://hamcrest.org/
- http://code.google.com/p/hamcrest/wiki/Tutorial
- http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- http://hamcrest.org/JavaHamcrest/javadoc/1.3

# Hamcrest

## Matchers that can be combined to create flexible expressions of intent

Born in Java, Hamcrest now has implementations in a number of languages.

- Java
- Python
- Ruby
- Objective C
- PHP
- Erlang

http://code.google.com/p/hamcrest/wiki/Tutorial

# Hamcrest

- Hamcrest is a framework for writing matcher objects allowing 'match' rules to be defined declaratively.
- There are a number of situations where matchers are invaluble, such as UI validation, or data filtering,
- but it is in the area of writing flexible tests that matchers are most commonly used.

**http://code.google.com/p/hamcrest/wiki/Tutorial**

# My first Hamcrest test

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

import junit.framework.TestCase;

public class BiscuitTest extends TestCase {
  public void testEquals() {
    Biscuit theBiscuit = new Biscuit("Ginger");
    Biscuit myBiscuit = new Biscuit("Ginger");
    assertThat(theBiscuit, equalTo(myBiscuit));
  }
}
```

// Note JUnit3

**http://code.google.com/p/hamcrest/wiki/Tutorial**

- If you have more than one assertion in your test you can include an identifier for the tested value in the assertion:

```
assertThat("chocolate chips",
theBiscuit.getChocolateChipCount(), equalTo(10));

assertThat("hazelnuts",
theBiscuit.getHazelnutCount(), equalTo(3));
```

**http://code.google.com/p/hamcrest/wiki/Tutorial**

# Hamcrest Common Matchers

- A tour of Hamcrest comes with a library of useful matchers. Here are some of the most important ones.
- Core
  - anything - always matches, useful if you don't care what the object under test is
  - describedAs - decorator to adding custom failure description
  - is - decorator to improve readability - see "Sugar", below
- Logical
  - allOf - matches if all matchers match, short circuits (like Java &&)
  - anyOf - matches if any matchers match, short circuits (like Java ||)
  - not - matches if the wrapped matcher doesn't match and vice versa

http://code.google.com/p/hamcrest/wiki/Tutorial

- Object
  - equalTo - test object equality using Object.equals
  - hasToString - test Object.toString
  - instanceOf, isCompatibleType - test type
  - notNullValue, nullValue - test for null
  - sameInstance - test object identity
- Beans
  - hasProperty - test JavaBeans properties

**http://code.google.com/p/hamcrest/wiki/Tutorial**

- Collections
  - array - test an array's elements against an array of matchers
  - hasEntry, hasKey, hasValue - test a map contains an entry, key or value
  - hasItem, hasItems - test a collection contains elements
  - hasItemInArray - test an array contains an element
- Number
  - closeTo - test floating point values are close to a given value
  - greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo - test ordering

**http://code.google.com/p/hamcrest/wiki/Tutorial**

- Text
  - equalToIgnoringCase - test string equality ignoring case
  - equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
  - containsString, endsWith, startsWith - test string matching

**http://code.google.com/p/hamcrest/wiki/Tutorial**

# Syntactic Sugar

- Hamcrest strives to make your tests as readable as possible.
- For example, the “is” matcher is a wrapper that doesn't add any extra behavior to the underlying matcher., but increases readability.
- The following assertions are all equivalent:
  - assertThat(theBiscuit, equalTo(myBiscuit));
  - assertThat(theBiscuit, is(equalTo(myBiscuit)));
  - assertThat(theBiscuit, is(myBiscuit));
- The last form is allowed since is(T value) is overloaded to return is(equalTo(value)).

**http://code.google.com/p/hamcrest/wiki/Tutorial**

# Testing Session

```
import java.io.*;

class Trityp
{...
public static void main (String[] argv)
{  // Driver program for trityp
   int A, B, C;
   int T;


   System.out.println (instructions);
   System.out.println ("Enter side 1: ");
   A = getN();
   System.out.println ("Enter side 2: ");
   B = getN();
   System.out.println ("Enter side 3: ");
   C = getN();
   T = Triang (A, B, C);
   System.out.println ("Res
triTypes[T]);
}
```

**Black box/White box?**

- **White box because we could see the code**

**What are the test inputs?**

- **Test inputs from program arguments**

**What are the test outputs?**

- **Test outputs from System.out**

```
// ===================================
// Read (or choose) an integer
private static int getN ()
{
   int inputInt = 1;
   BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
   String inStr;

   try
   {
      inStr = in.readLine ();
      inputInt = Integer.parseInt(inStr);
   }
   catch (IOException e)
   {  // JDK requires the IOException to be caught.
      System.out.println ("Could not read input, choosing 1.");
   }
   catch (NumberFormatException e)
   {
      System.out.println ("Entry must be a number, choosing 1.");
   }
     return (inputInt);
}}
```

**What are the test inputs?**

- **Test input from System.in**

# Which example has better tests?

**Example 1 is better!**

- **Each test should be independent of each other**

## Example 1

```
@Test
public void popTest() {
 MyStack s = new MyStack ();
  s.push (314);
  assertEquals (314, s.pop ());
}
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 s.push (2);
 assertEquals (1, s.size ());
}
```

## Example 2

```
MyStack s = new MyStack ();
@Test
public void popTest() {
 s.push (314);
 assertEquals (314, s.pop ());
}

@Test
public void sizeTest() {
 assertEquals (1, s.size ());
}
```

# Which example has better tests?

**Example 2 is better!**

- **Any given behaviour should be specified in one and only one test.**

## Example 1

```
@Test
public void sizeTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
  s.push (2);
  assertEquals (1, s.size ());
}
```

Multiple assertions are bad because after one assertion fail, execution stops

## Example 2

```
@Test
public void emptyTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
}

@Test
public void sizeTest() {
  MyStack s = new MyStack ();
  s.push (2);
  assertEquals (1, s.size ());
```

# Which tests is correct?

**Example 2 is correct!**

- Correct method signature should be assertEquals(expected,actual)

## Example 1

```
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 assertEquals (s.size (),0);
}
```

## Example 2

```
@Test
public void emptyTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
}
```

# Which tests is correct?

**Example 1 is correct!**

- Use .equals() to compare strings

## Example 1

```
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 s.push("Hello")
 assertEquals ("Hello",
s.pop());
}
```

## Example 2

```
@Test
public void emptyTest() {
 MyStack s = new MyStack ();
 assertTrue (s.pop()=="Hello");
}
```

# How to test print to console (System.out)?

```
@Test
public void printTest() {
 //Step1: Prepare to redirect output
 OutputStream os = new ByteArrayOutStream();
 PrintStream ps = new PrintStream(os);
 System.setOut(ps);

 //Step2: need System.getProperty("line.separator") to
 //properly test for the next line
HelloWorld.printHello();
assertEquals("Hello World" +
                System.getProperty("line.separator"),
                      os.toString());

 //Step3: Restore normal output
 PrintStream originalOut = System.out;
 System.setOut(originalOut);
}
```

# How to test main?

```
class App
{...
public st
{
        S
        A
        t
        }
        }
}
 protecte
}
```

```
@Test
public void mainTest() {

  //Step1: Prepare to redirect input
  String[] args = null;
  final InputStream original = System.in;
  final FileInputStream fips = new FileInputStream(new
File("[path_to_file]"));
  System.setIn(fips);

//Step2: construct test inputs
 String [] args = { "one", "two", "three" };
 Application.main(args);


 //Step3: Restore normal input
 System.setIn(original);
}
```