Introduction to Software Testing

(*2nd edition*)

Chapter 3

Test Automation

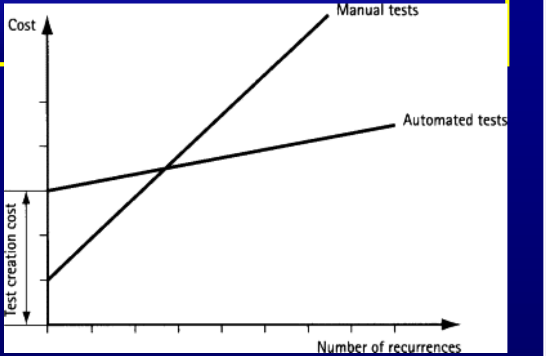
Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Updated February 2016

What is Test Automation?

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



- educes cost
- educes human error
- educes variance in test quality from different individuals
- gnificantly 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

- ainly 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)

test case is a multipart artifact with a definite structure

est case values

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

xpected 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

refix values

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

ostfix 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

est case

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

est set (vs. test suite)

A set of test cases

xecutable 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

- Unit can be used to test …
  - … an entire object
  - … part of an object – a method or some interacting methods
  - … interaction between several objects
- is primarily intended for unit and integration testing, not system testing
- ach test is embedded into one test method
- test class contains one or more test methods
- est 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
- et started at junit.org

Writing Tests for JUnit

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

How to Write A Test Case

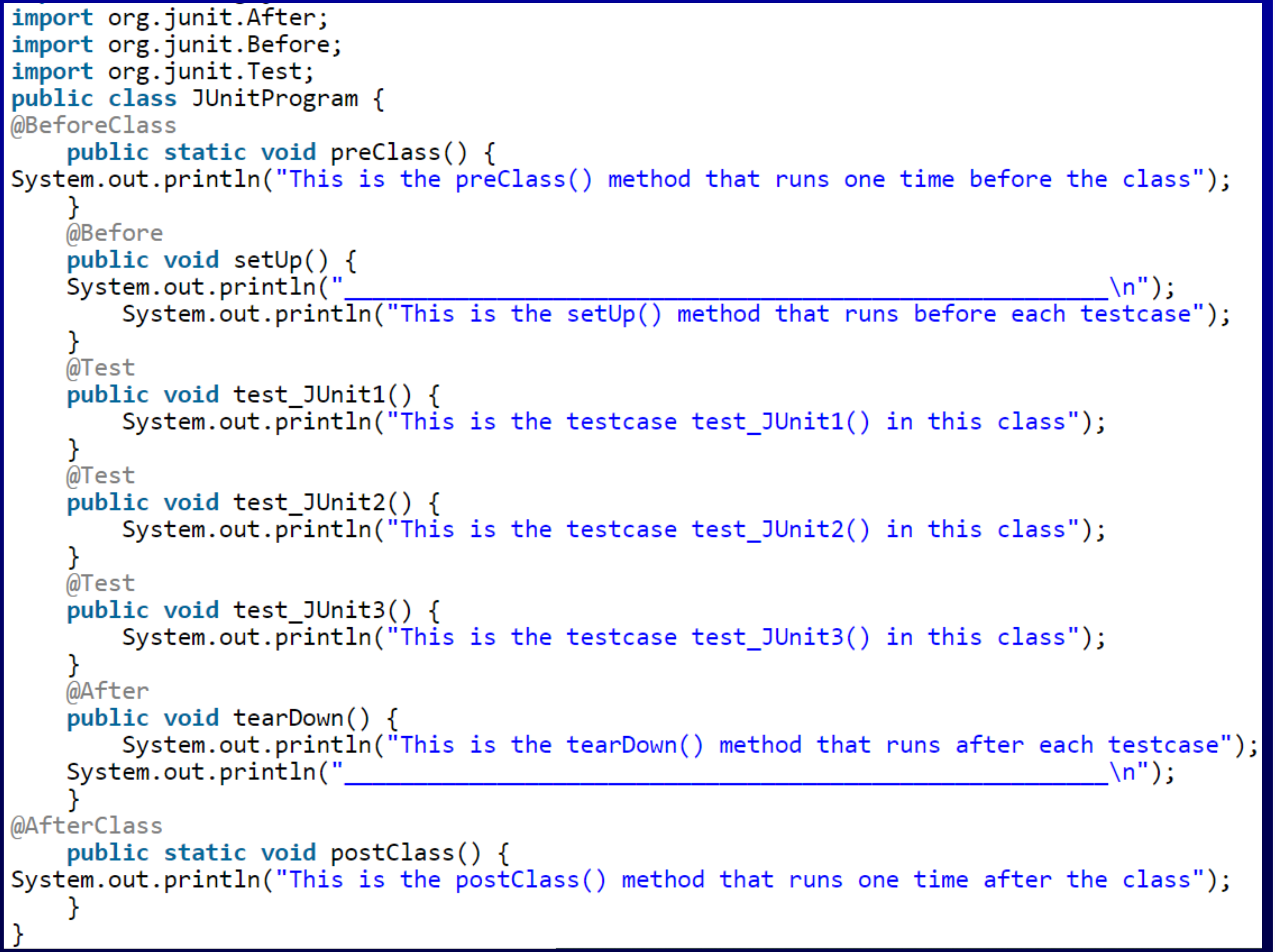
- ou may occasionally see old versions of JUnit tests
  – Major change in syntax and features in JUnit 4.0
  – Backwards compatible (JUnit 3.X tests still work)
- JUnit 3.X
  1. import junit.framework.*
  2. extend TestCase
  3. name the test methods with a prefix of ‘test’
  4. validate conditions using one of the several assert methods
- 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
- /e focus entirely on JUnit 4.X

JUnit Test Fixtures

- test fixture is the (fixed) state of (starting) the test
  - – Objects and variables that are used by more than one test
    - A fixed state of a set of objects used as a baseline for running tests
    - E.g., a database with a known set of data, a hard disk with a clean operating system, preparation of input data, creation of mock objects…
  - – Initializations (*prefix* values)
  - – Reset values (*postfix* values)
- ifferent tests can use the objects without sharing the state (i.e., tests are independent)
- Objects used in test fixtures should be declared as instance variables
- hey should be initialized in a @Before method
- an be deallocated or reset in an @After method

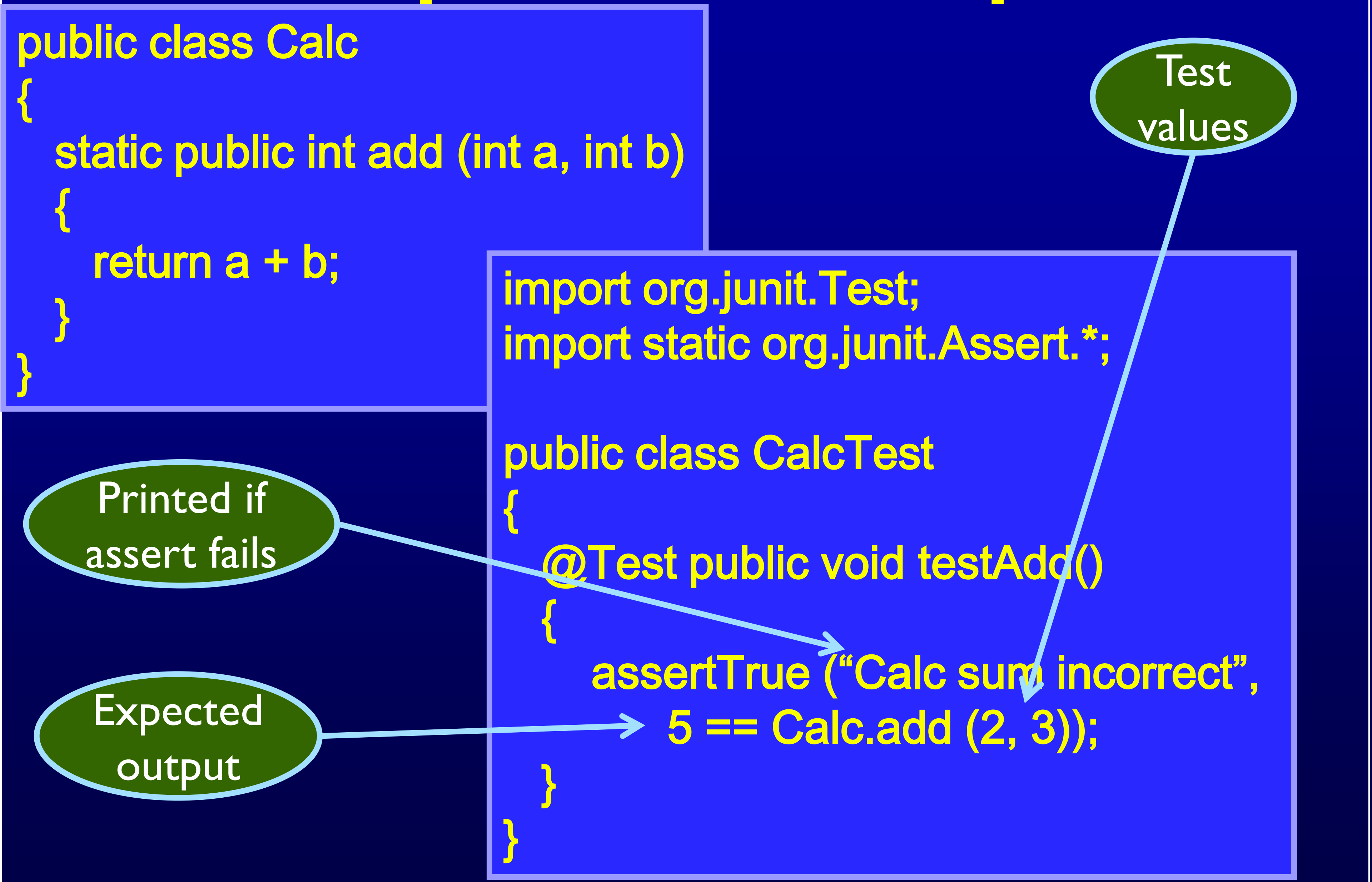
The Sequence of the Test Run

1. @BeforeClass
2. @Before
3. @Test
4. @After
5. @AfterClass



```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
public class JUnitProgram {
@BeforeClass
    public static void preClass() {
System.out.println("This is the preClass() method that runs one time before the class");
    }
    @Before
    public void setUp() {
    System.out.println("____________________________________________________\n");
        System.out.println("This is the setUp() method that runs before each testcase");
    }
    @Test
    public void test_JUnit1() {
        System.out.println("This is the testcase test_JUnit1() in this class");
    }
    @Test
    public void test_JUnit2() {
        System.out.println("This is the testcase test_JUnit2() in this class");
    }
    @Test
    public void test_JUnit3() {
        System.out.println("This is the testcase test_JUnit3() in this class");
    }
    @After
    public void tearDown() {
        System.out.println("This is the tearDown() method that runs after each testcase");
    System.out.println("____________________________________________________\n");
    }
@AfterClass
    public static void postClass() {
System.out.println("This is the postClass() method that runs one time after the class");
    }
}
```

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 (NullPointerException), CCE(ClassCastException) as needed
           T comp = itr.next();
           if (comp.compareTo (result) < 0)        // if comp < result
           {
               result = comp;
       }   }
       return result;
}
```

MinTest Class

tandard imports for all JUnit classes :

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

est 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>();
 }
```

ost 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 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

Parameterized Tests

- JUnit 4 has introduced a new feature called parameterized tests.
- Parameterized tests allow a developer to run the same test over and over again using different values.
- here are <u>five steps</u> that you need to follow to create a parameterized test
  - Annotate test class with @RunWith(Parameterized.class).
  - Create a public **static** method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
  - Create a public constructor that takes in what is equivalent to one "row" of test data.
  - Create an **instance variable** for each "column" of test data.
  - Create your test case(s) using the **instance variables** as the source of the test data.

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

JUnit Theories

- normal **test** captures the intended behavior in **one** particular scenario, given an input it expects a certain output.
- **theory** captures some aspect of the intended behavior in **possibly infinite numbers** of potential scenarios. This means whatever a theory asserts is expected to be true for **all data sets**.
- Theories are often used for finding bugs in boundary-value cases or mathematical theories.
- Theories are functionally similar to parameterized tests, but are expressively richer.

Creating a JUnit Theory

- he class should be annotated with @RunWith(Theories.class) and have:
  - A data method that generates and returns test data
    - By annotating a static member variable with @DataPoint
    - By annotating a static member variable with @DataPoints
  - A theory by annotating a test method with the @Theory annotation

JUnit Theory Annotations

[redacted]heories come up with many annotations and a class runner.

- @Theory same like @Test, this annotation identifies a theory test.
- @DataPoint annotation identifies a single set of test data. This annotation is similar to @Parameter. It can be annotated by either a static variable or a method.
- @DataPoints annotation identifies multiple sets of test data. This annotation is similar to @Parameters and is generally used for an array. It can be annotated by either a static variable or a method.
- @ParametersSuppliedBy annotation provides the parameters to the test cases.
- Theories is a JUnit runner for running theory test classes.
- ParameterSupplier is able to provide parameters that we can supply to the test case.

Passing Data Via @DataPoint

- contrast to a normal test, theories can have arguments.
- he data that is passed to these arguments come from a static member variable annotated by either @DataPoint or @DataPoints.
- Vhen multiple @DataPoint annotations are defined in a test, the theories apply to **all** possible type complient combinations of data points for the test arguments.

Tests with Parameters: JUnit Theories

- nit tests can have actual parameters
  - So far, we've only seen parameterless test methods
- ontract 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?

nswer:

- 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"))
};
```

```
Set, string: [bat, ant], ant
Set, string: [bat, ant], bat
Set, string: [bat, elk, cat, dog], bat
Set, string: [bat, elk, cat, dog], cat
```

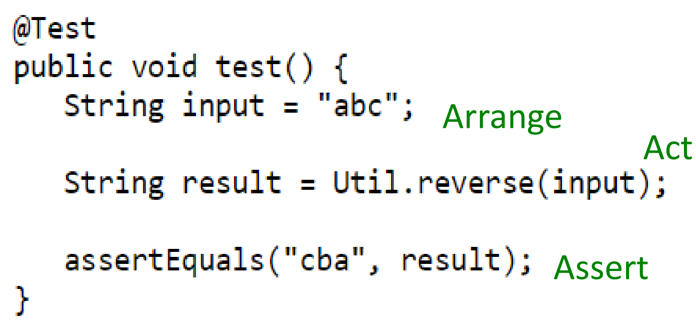
Nine combinations of animalSets[i].contains (animals[j]) is false for five combinations

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
}
```

Arrange-Act-Assert

- pattern for arranging and formatting code in UnitTest methods (i.e., structure test cases)
  - Similar to **Given-When-Then** in BDD (Behavior-Driven Design)
- ach method should group these functional sections, separated by blank lines:
  - Arrange all necessary preconditions and inputs.
  - Act on the object or method under test.
  - Assert that the expected results have occurred.



```
@Test
public void test() {
    String input = "abc";

    String result = Util.reverse(input);

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

- enefits
  - Clearly separates what is being tested from the setup and verification steps.
  - Clarifies and focuses attention on a historically successful and generally necessary set of test steps.
  - Makes some TestSmells more obvious:
    - Assertions intermixed with "Act" code.
    - Test methods that try to test too many different things at once.

Running from a Command Line

- his is all we need to run JUnit in an IDE (like Eclipse)

- e 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

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

JUnit Resources

- ome 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)
- Unit: Download, Documentation
  - http://www.junit.org/

Test Doubles (3.4)

- ctors use doubles to replace them during certain scenes
  - Dangerous or athletic scenes
  - Skills the actor doesn't have, like dancing or singing
  - Partial nudity

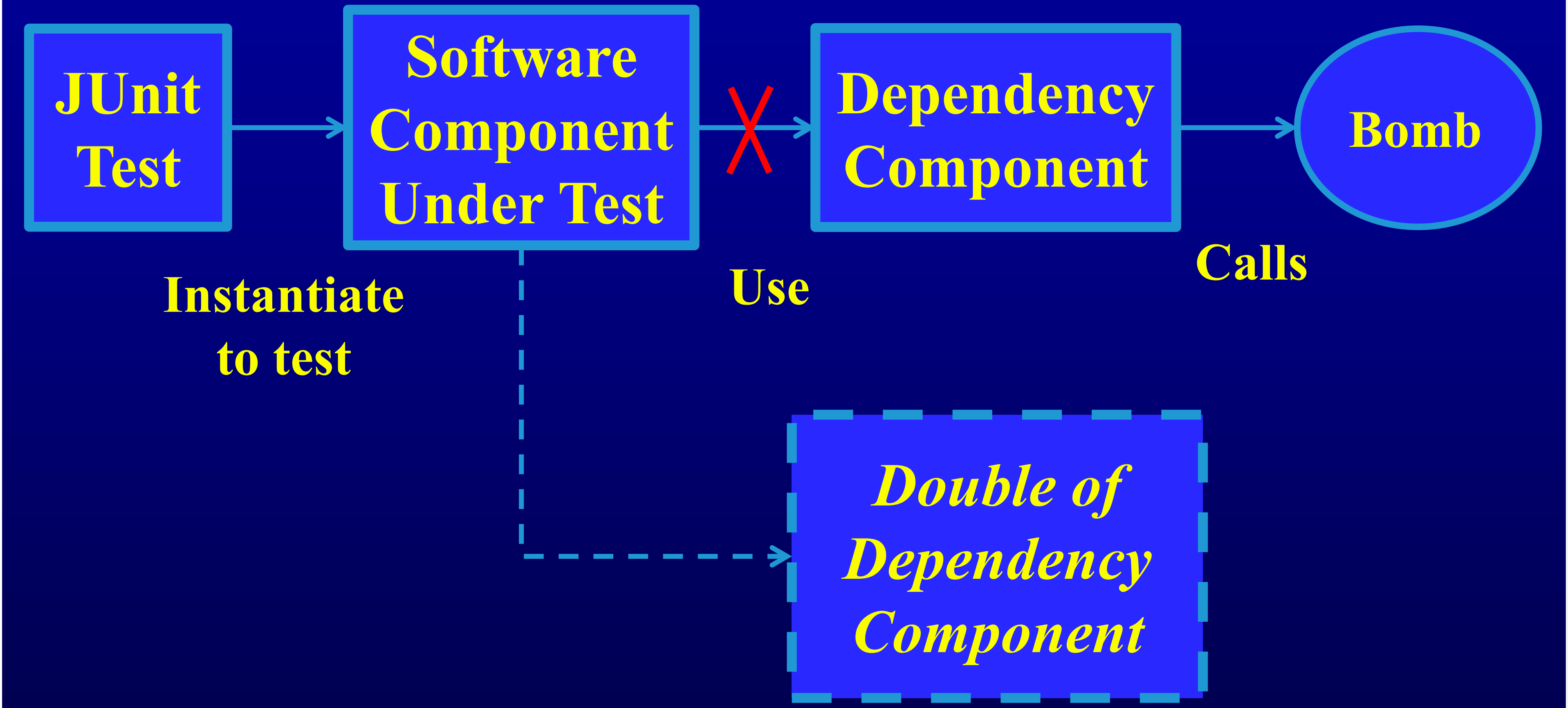
- **Test doubles** replace software components that cannot be used during testing

Reasons for Test Doubles

- omponent has not been written
- he real component does something destructive that we want to avoid during testing (unrecoverable actions)
- he real component interacts with an unreliable resource
- he real component runs very slowly
- he real component creates a test cycle
  - *A* depends on *B*, *B* depends on *C*, *C* depends on *A*

A **test double** is a software component that implements partial functionality to be used during testing

Test Double Illustration


JUnit Test
Software Component Under Test
Dependency Component
Bomb
Instantiate to test
Use
Calls
Double of Dependency Component

Types of Test Doubles

1. Dummy : Used to fill parameter lists
2. Fake : A working implementation that takes shortcuts
   - For example, an in-memory database
3. Stub : Hard-coded return values for the tests
4. Mock : Objects preprogrammed with preliminary specifications

Summary

- he 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
- 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