Module 12: Contract Model in Testing, Unit Testing

# Overview and Objectives

## OVERVIEW

In this module, we look at software testing as a way to find bugs in software. Specifically, we will use the JUnit framework as an example to automate tests for Java programs. We will learn how to use JUnit for simple tests and JUnit theories for more complicated test automations.

## COURSE LEVEL OBJECTIVES (CLO)

Upon completion of this course, you should be able to:

1. Construct modern high quality software systems and reason about them.
2. Properly define software specifications and rep-invariants.
3. Leverage immutability to properly construct threat safe programs.
4. Explain object-oriented concepts such as information hiding, encapsulation, data and type abstraction, and polymorphism.
5. Properly use exception handling
6. Identify when it is appropriate to use inheritance and generics.

## MODULE LEVEL OBJECTIVES (MLO)

Upon completion of this module’s activities, you should be able to:

1. Explain the key concepts of testing (CLO 1, CLO 2)
2. Describe and compare terminologies and ideas of JUnit, JUnit Testing, and JUnit Theories (CLO 1, CLO 2)
3. Construct and use correct JUnit to automate tests for Java programs (CLO 1, CLO 2)

# Module Video (Wiley-Produced w/Dan Ramos) [3-5 minutes]

# Learning Materials [~100 pages, ~3.5 hours]

## TEXTBOOK READINGS

* None

# Learning Unit 1 – Intro to Testing (MLO 1) [~0.5 hour]

## Test Automation\*

* **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
  + Why we want it?
    - Reduces **cost**
    - Reduces **human error**
    - Reduces **variance** in test quality from different individuals
    - Significantly reduces the cost of **regression** testing
* **Software Testability**
  + 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
  + **how hard it is to find faults** in the software
  + 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

* 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
* 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**

# Learning Unit 2 – Intro to JUnit Testing (MLO 2) [~1 hour]

## What is JUnit?

* Open source Java testing framework used to write and run repeatable **automated tests**
* 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**
* **widely used** in industry
* can be used as **stand alone** Java programs (from the command line) or within an **IDE** such as Eclipse

### JUnit Tests

* 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

### Writing Tests for JUnit

* Need to use the methods of the junit.framework.assert class
* 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 **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

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

//CalcTest.java  
import org.junit.\*;  
import static org.junit.Assert.\*;  
  
public class CalcTest  
{  
 @Test public void testAdd()  
 {  
 assertTrue ("Calc sum incorrect", 5 == Calc.add (2, 3));  
 }  
}

### Testing the Min Class

//Min.java  
import java.util.\*;  
  
public class Min  
{  
 /\*\*  
 \* Returns the mininum element in a list  
 \* @param list Comparable list of elements to search  
 \* @return the minimum element in the list  
 \* @throws NullPointerException if list is null or  
 \* if any list elements are null  
 \* @throws ClassCastException if list elements are not mutually comparable  
 \* @throws IllegalArgumentException if list is empty  
 \*/  
 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.java  
import static org.junit.Assert.\*;  
import org.junit.\*;  
import java.util.\*;  
  
public class MinTest  
{  
 private List<String> list; // Test fixture  
  
 @Before // Set up - Called before every test method.  
 public void setUp()  
 {  
 list = new ArrayList<String>();  
 }  
  
 @After // Tear down - Called after every test method.  
 public void tearDown()  
 {  
 list = null; // redundant in this example!  
 }  
  
 @Test  
 public void testForNullList1()  
 {  
 list = null;  
 try {  
 Min.min (list);  
 } catch (NullPointerException e) {  
 return;  
 }  
 fail ("NullPointerException expected");  
 }  
  
 @Test (expected = NullPointerException.class)  
 public void testForNullList2()  
 {  
 list = null;  
 Min.min (list);   
 }  
  
 @Test (expected = NullPointerException.class)  
 public void testForNullElement()  
 {  
 list.add (null);  
 list.add ("cat");  
 Min.min (list);  
 }  
  
 @Test (expected = NullPointerException.class)  
 public void testForSoloNullElement()  
 {  
 list.add (null);  
 Min.min (list);  
 }  
  
 @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);  
 }  
  
 @Test (expected = IllegalArgumentException.class)  
 public void testEmptyList()  
 {  
 Min.min (list);  
 }  
  
 @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"));  
 }  
}

* **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
    1. single element
    2. two elements

# Learning Unit 3 – Intro to JUnit Theories (MLO 3) [~1 hour]

## **JUnit theories** are the JUnit implementation of "**property-based**" testing

* Property of Stack: for all stacks str == pop(push(str))
* Need to write tests to validate the above property
* Cannot have an infinite number of tests, we settle for n tests
* But why not leverage the fact that the n tests look the same (except for the input values)
* JUnit theories are relevant to this course because they show how the precondition/postcondition model applies beyond method contracts
  + **Postcondition**: for all stacks str == pop(push(str))
  + **Precondition**: stack is not null
* Example
* @Theory public void StackTheory(Stack s, String str)  
  //Precondition  
  assumeTrue(s != null)  
    
  //Postcondition  
  ... str == s.pop(s.push(str))

## 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"};  
  // Nine combinations of animalSets[i].contains (animals[j])  
  // is false for five combinations  
   @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"))  
   };

## 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  
{  
 ...  
}

* Then 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);  
 }  
}

## 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**

## Instructor Screencast: TITLE

# In Class 1 – (MLO 1, 2, 3) [~.5 hour]

JUnit Theories

@RunWith(Theories.class)  
 public class MyJunitTheories {  
  
 @DataPoints  
 public static int[] dataPoints() {  
 return new int[]{  
 71, 82, 53, -1  
 };  
 }  
  
 //For a and b where a,b > 0  
 //(a+b)^2 = a^2+b^2+2ab   
 //a+b > a and a+b > b  
  
 @Theory  
 public void squareTheory(Integer a, Integer b){  
  
 //Below assume statement ensures that we are testing only positive numbers  
 Assume.assumeTrue(a > 0 && b > 0);  
  
 System.out.println("Running with Data points - " + a + " , "+ b);  
  
 Double leftSide = Math.pow(a+b,2);  
 Double rightSide = Double.valueOf(a \* a + b \* b + 2 \* a \* b);  
  
 assertEquals(leftSide,rightSide);  
  
 assertTrue(a + b > a);  
 assertTrue(a + b > b);  
 }  
}

* 16 tests in total
* -1,-1  
  53,53  
  82,82  
  71,71  
  71,82  
  82,71  
  82,53  
  53,82  
  53,-1  
  -1,53  
  71,53  
  53,71  
  71,-1  
  -1,71  
  82,-1  
  -1,82
* 7 fail preconditions
* -1,-1  
  53,-1  
  ...
* 7 pass the tests (fail preconds) + the other 9 = 16
* 53,71 .. pass
* pair of points over 5 data points

5\*\*2 = 25 (Cartesian product)

* k over n

n\*\*k n=5,k=2 5x5 = 25 n=5,k=3 5x5x5 = 1255 n=100,k=2 100x100 = 10,000

# In class 2 (MLO 1, 2, 3) [.5 hours]

This is a JUnit theory exercise.

1. Write a JUnit theory that captures the symmetry property of the equals() method.
   1. Create @DataPoints from Bloch's Point, ColorPoint classes. So that we're all on the same page, create 1 null reference, 1 Point object and 2 ColorPoint objects.
   2. Given this set of data points:
      * How many combinations are considered by the theory?
      * How many combinations make it past the preconditions of the theory?
      * How many combinations make it to the postcondition of the theory?
   3. What happens to this theory and the accompanying data points when favoring composition over inheritance?
2. Repeat the exercise for the transitive property for equals().
3. Recall the equals() and hashCode() discussion in Bloch. Write a JUnit theory that encodes the consistency property between equals() and hashCode().
4. Build a toy example that violates the theory. Fix the toy example so that the theory is no longer violated.
5. Consider the Comparable interface: what properties should be checked with theories?

**SOLUTION:**

@DataPoints  
 public static Point[] points = {null, //a  
 new Point(2,2), //b  
 new ColorPoint(2,2,COLOR.BLACK), //c  
 new ColorPoint(2,2 ,COLOR.RED)}; //d  
  
 @Theory  
 public void testEquals(Object a, Object b) { // can also use Point, but Object is more general and just  
 // symmetry  
 // \*if\* (a==b) then b==a AND  
  
 // if (b==a) then a==b  
 // a==b iff b==a  
  
 // NOT correct : too strong  
 // a==b AND b==a  
 // a.equals(b) == FALSE => b.equals(a) == FALSE  
  
 assumeTrue(a!= null && b != null);  
  
 if(a.equals(b)){assertTrue(b.equals(a));}  
 if(b.equals(a)){assertTrue(a.equals(b));}  
  
 //or , a shorter version  
 assertTrue(a.equals(b) == b.equals(a));  
  
 //# of tests = 4\*4 = 16  
 //# of tests passing the preconds: 9  
 // fail preconds(null,null; \*,null, null,\*)  
  
 //passing tests  
 // a,a  
 // a,\*  
 // \*,a  
 // b,b  
 // c,c  
 // d,d  
 // c,d because c.equals(d) == FALSE == d.equals(c) == FALSE   
 // d,c because d.equals(c) == FALSE == c.equals(d) == FALSE  
  
 //failing tests  
 // b,c b.equals(c) == T  
 // c.equals(b) == F  
 // c,b ....  
 }  
  
@Theory  
public void equalsTransitivity(Object x, Object y, Object z){  
 //(x.equals(y) and y.equals(z)) => x.equals(z)  
 //if (x.equals(y) and y.equals(z)) then x.equals(z)  
  
 assumeTrue(x!=null && y!=null && z!= null);  
 if(x.equals(y) && y.equals(z)){  
 assertTrue(x.equals(z))  
 }  
  
 assumeEquals(x,y);  
 assumeEquals(y,z);  
 assertTrue(x,z);  
}  
  
  
@Theory  
public void equalsHashCode(Object x, Object y){  
 //if(x==y) then x.hash == y.hash  
  
 assumeTrue(x != null);  
 assumeTrue(y != null);  
  
 assumeTrue(x.equals(y));  
 assertTrue(x.hashCode() == y.hashCode());  
  
}

# Assignment – (MLO 1, 2, 3) [~2 hours]

## Purpose

Practing using testing tools and also review ways to identify security vulnerabilities from previous modules.

## Instructions

You have a choice of possible assignments:

1. Consider one of the copyOf() methods in the Java [Arrays](https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html) utility class. Bloch uses this method in his Stack example. Code a corresponding method in C++, changing the argument list as necessary. Provide a specification for the C++ code by translating the JavaDoc and adding preconditions as necessary. Explain what this exercise demonstrates about C++ type safety.
2. For most of the semester, we have focused on design considerations for constructing software that does something we want it to do. For this last assignment, I would like students to appreciate just how vulnerable software is to malicious parties intent on attacking their software.

* There are two attacks documented in Bloch's Item 88: *Write readObject() methods defensively*. One is called BogusPeriod, and the other is called MutablePeriod. Implement either (your choice) of these attacks (basically involves typing in code from Bloch) and verify that the attack takes place.

1. A different source of security vulnerabilities in Java also involve serialization. Bloch (and others) recommend "cross-platform structured data representations" (e.g. JSON or Protocol Buffers) as safe alternatives. Develop a simple serialization example in Java and convert it into a safe alternative (probably, JSON is easier to use, since it is text-based). To make the example more interesting, use some objects types that are not directly supported.
2. Find some existing (Java) code that uses the "int enum pattern" and refactor it to use Java Enums instead. Identify any type-safety issue you uncover in the existing code. To make the exercise interesting, extend your enums beyond simple named-constants in one of the ways discussed by Bloch in Item 34.
3. Where appropriate, code up, as JUnit theories, constraints for classes that implement the Java Comparable interface. Note that there is significant overlap with the in-class exercise. Note also that the Comparable interface is generic; hence, you should use generics in your JUnit test class.
4. Gain experience with one of the property-based testing tools. I suggest a Java-based one (such as [jqwik](https://jqwik.net)). One way to do this is work through one of the articles linked on the jqwik site.

### Grading Criteria

In each case, the deliverable is a story. Write a brief report, and include enough evidence (output, screen shots, etc.) that the GTA can figure out that you actually completed the assignment.

## Deliverable

* Submit a .java file for your implementation.

## Due Date

Your assignment is due by Sunday 11:59 PM, ET.

# Module 1 Quiz (MLO 1, 2, 3) [~.5 hour]

## Purpose

Quizzes in this course give you an opportunity to demonstrate your knowledge of the subject material.

## Instructions

Consider the following incomplete JUnit theory about the consistency of compareTo() (from the Comparable interface) and equals().

@Theory public void compareToConsistentWithEquals( ... ) {  
 assumeTrue (...); // Assume none of the parameters are null (i.e. no NPE)  
 assumeTrue (...); // Assume parameters are mutually comparable (i.e. no CCE)  
  
 assertTrue (...); // See question 3  
 }

* How many parameters should this theory have?
* **Solution**: Answer: 2
* What type should each of the paramters have?
* **Solution**: Answer: Comparable Grading note: Technically, this should be type E, where there is a constraint in the class type: public class SomeJUnitTestClass <E extends Comparable<E>> { Anything that shows that you know that the parameters need to be comparable is fine.
* What is an appropriate assertion? Note: assume that the assumeTrue(...) statements are correctly implemented.
* **Solution**: assertTrue (x.equals(y) == (x.compareTo(y) == 0)); // Assert There are other ways of stating this, of course. The key point is that this is an "iff" relationship.
* Suppose you had the following DataPoints. How many times does JUnit evaluate the assertTrue() statement in this theory?
* @DataPoints  
   public static String[] stuff = { "cat", "cat", "dog"};
* **Solution**: 3\*3 = 9 11,22,33,12,13,23,21,31,32

The quiz is 30 minutes in length. The quiz is closed-book.

## Deliverable

Use the link above to take the quiz.

## Due Date

Your quiz submission is due by Sunday 11:59 PM, ET.