Project Automation: Unit Testing

CSCE 247 - Lecture 9 - 02/18/2019

Executing Tests

- How do you run test cases on the program?
  - You could run the code and check results by hand.
  - **Please don't do this.**
    - Humans are slow, expensive, and error-prone.
  - Test design requires effort and creativity.
  - Test execution should not.

Test Automation

- **Test Automation** is the development of software to separate repetitive tasks from the creative aspects of testing.
- Automation allows control over *how* and *when* tests are executed.
  - Control the environment and preconditions.
  - Automatic comparison of predicted and actual output.
  - Automatic hands-free reexecution of tests.

Testing Requires Writing Code

- Testing cannot wait for the system to be complete.
  - The component to be tested must be isolated from the rest of the system, instantiated, and *driven* using method invocations.
  - Untested dependencies must be *stubbed out* with reliable substitutions.
  - The deployment environment must be simulated by a controllable *harness*.

Test Scaffolding

Test scaffolding is a set of programs written to support test automation.

- Not part of the product
- Often temporary

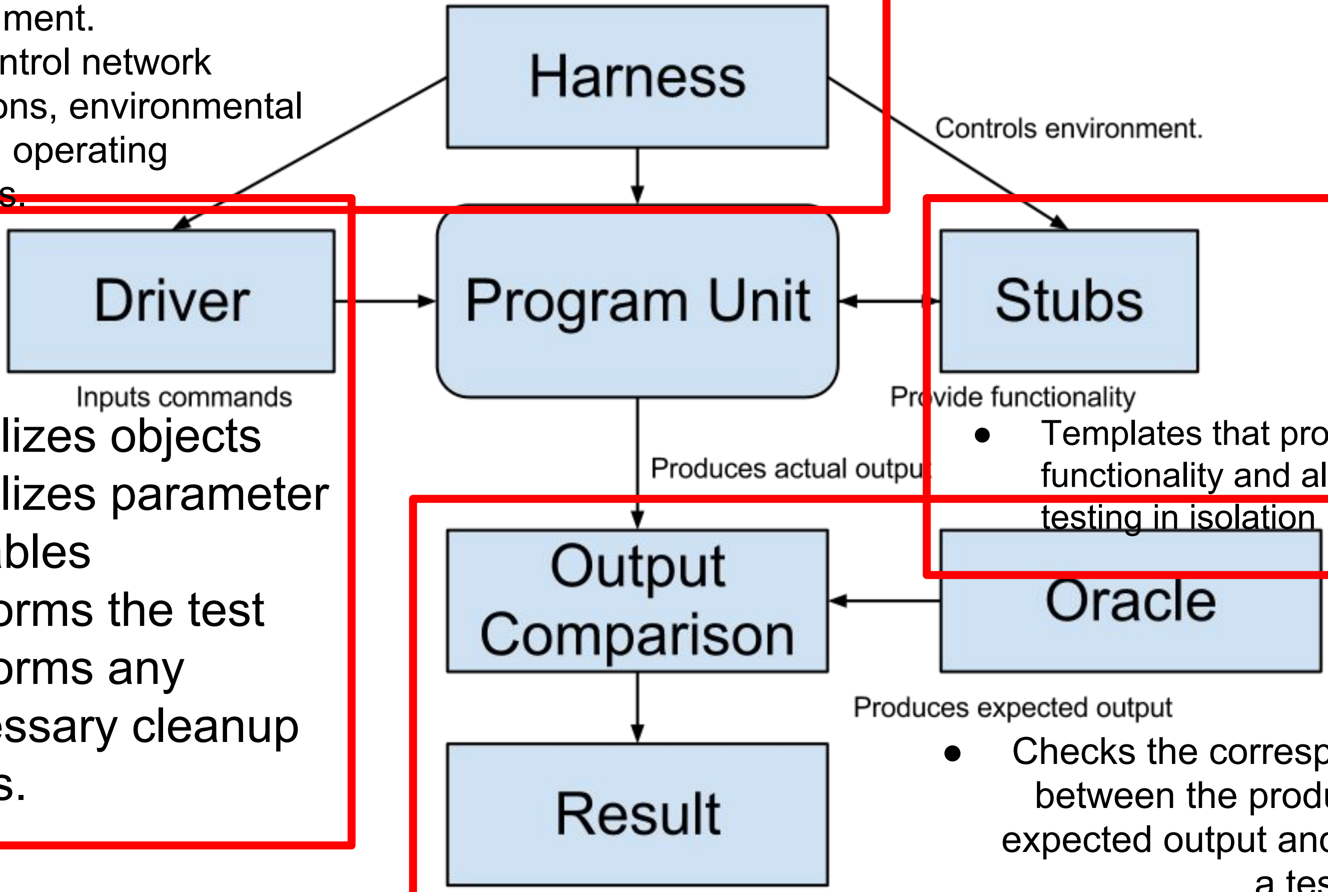
Allows for:

- Testing before all components complete.
- Testing independent components.
- Control over testing environment.

Test Scaffolding

- A **driver** is a substitute for a main or calling program.
  - Test cases are drivers.
- A **harness** is a substitute for all or part of the deployment environment.
- A **stub** (or **mock object**) is a substitute for system functionality that has not been completed.
- Support for recording and managing test execution.

Test Scaffolding



- Simulates the execution environment.
- Can control network conditions, environmental factors, operating systems.

- Initializes objects
- Initializes parameter variables
- Performs the test
- Performs any necessary cleanup steps.

- Templates that provide functionality and allow testing in isolation

- Checks the correspondence between the produced and expected output and renders a test verdict.

Writing an Executable Test Case

- Test Input
  - Any required input data.
- Expected Output (Test Oracle)
  - What *should* happen, i.e., values or exceptions.
- Initialization
  - Any steps that must be taken before test execution.
- Test Steps
  - Interactions with the system (such as method calls), and output comparisons.
- Tear Down
  - Any steps that must be taken after test execution to prepare for the next test.

Writing a Unit Test

JUnit is a Java-based toolkit for writing executable tests.

- Choose a target from the code base.
- Write a “testing class” containing a series of unit tests centered around testing that target.

```
public class Calculator {
  public int evaluate (String
                  expression) {
    int sum = 0;
    for (String summand:
                  expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Writing a Unit Test

```
public class Calculator {
  public int evaluate (String

    int sum
    for (St
              expression.split(
      sum += Integer.valueOf(su
    return sum;
  }
}
```

Each test is denoted with keyword **@test**.

Initialization

Test Steps

```
import static
org.junit.jupiter.api.Assertions.assert
Equals;
import org.j

public class CalculatorTest {
  @Test
  void testEvaluate_Valid_ShouldPass(){
    Calculator calculator =
         new Calculator();
    int sum =
         calculator.evaluate("1+2+3");
    assertEquals(6, sum);
     calculator = null;
  }
}
```

Convention - name the test class after the class it is testing or the functionality being tested.

Input

Oracle

Tear Down

Test Fixtures - Shared Initialization

@BeforeEach annotation defines a common test initialization method:

```
@BeforeEach
public void setUp() throws Exception
{
   this.registration = new Registration();
   this.registration.setUser(“ggay”);
}
```

Test Fixtures - Teardown Method

@AfterEach annotation defines a common test tear down method:

```
@AfterEach
public void tearDown() throws Exception
{
   this.registration.logout();
   this.registration = null;
}
```

More Test Fixtures

- @BeforeAll defines initialization to take place before any tests are run.
- @AfterAll defines tear down after all tests are done.

```
@BeforeAll
  public static void setUpClass() {
     myManagedResource = new
          ManagedResource();
  }

  @AfterAll
  public static void tearDownClass()
throws IOException {
    myManagedResource.close();
    myManagedResource = null;
  }
```

Test Skeleton

@Test annotation defines a single test:

Type of scenario, and expectation on outcome.
I.e., testEvaluate_NullInput_ShouldThrowException()

```
@Test
public void test<MethodName>_<TestingContext>() {
   //Define Inputs
   try{ //Try to get output.
   }catch(Exception error){
      fail("Why did it fail?");
   }
   //Compare expected and actual values through
assertions or through if statements/fails
}
```

Assertions

Assertions are a "language" of testing - constraints that you place on the output.

- assertEquals, assertArrayEquals
- assertFalse, assertTrue
- assertNull, assertNotNull
- assertSame,assertNotSame

assertEquals

```
@Test
public void testAssertEquals() {
   assertEquals("failure - strings are not
equal", "text", "text");
}

@Test
public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays
not same", expected, actual);
}
```

- Compares two items for equality.
- For user-defined classes, relies on `.equals` method.
  - Compare field-by-field
  - `assertEquals(studentA.getName(), studentB.getName())` rather than `assertEquals(studentA, studentB)`
- assertArrayEquals compares arrays of items.

assertFalse, assertTrue

```
@Test
public void testAssertFalse() {
   assertFalse("failure - should be false",
(getGrade(studentA, “CSCE747”).equals(“A”));
}

@Test
public void testAssertTrue() {
     assertTrue("failure - should be true",
(getOwed(studentA) > 0));
}
```

- Take in a string and a boolean expression.
- Evaluates the expression and issues pass/fail based on outcome.
- Used to check conformance of solution to expected properties.

assertSame, assertNotSame

```
@Test
public void testAssertNotSame() {
   assertNotSame("should not be same Object",
studentA, new Object());
}

@Test
public void testAssertSame() {
    Student studentB = studentA;
   assertSame("should be same", studentA,
studentB);
}
```

- Checks whether two objects are clones.
- Are these variables aliases for the same object?
  - assertEquals uses .equals().
  - assertSame uses ==

assertNull, assertNotNull

```
@Test
public void testAssertNotNull() {
    assertNotNull("should not be null",
    new Object());
}

@Test
public void testAssertNull() {
    assertNull("should be null", null);
}
```

- Take in an object and checks whether it is null/not null.
- Can be used to help diagnose and void null pointer exceptions.

Grouping Assertions

```
@Test
void groupedAssertions() {
      Person person = Account.getHolder();
      assertAll("person",
          () -> assertEquals("John",
                      person.getFirstName()),
          () -> assertEquals("Doe",
                      person.getLastName())
      );
  }
```

- Grouped assertions are executed.
    - Failures are reported together.
    - Preferred way to compare fields of two data structures.

assertThat

```
@Test
public void testAssertThat{
○ assertThat("albumen", both(containsString("a")).and(containsString("b")));
○ assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
○ assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }),
              everyItem(containsString("n")));
○ assertThat("good", allOf(equalTo("good"), startsWith("good")));
○ assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
○ assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
○ assertThat(7, not(CombinableMatcher.<Integer>
              either(equalTo(3)).or(equalTo(4))));
}
```

either - pass if one of these properties is true.

Testing Exceptions

```
@Test
void exceptionTesting() {
    Throwable exception = assertThrows(
        IndexOutOfBoundsException.class,
        () -> {
            new ArrayList<Object>().get(0);
         });
         assertEquals("Index:0, Size:0",
             exception.getMessage());
}
```

- When testing error handling, we expect exceptions to be thrown.
  - **assertThrows** checks whether the code block throws the expected exception.
  - **assertEquals** can be used to check the contents of the stack trace.

Testing Performance

```
@Test
    void timeoutExceeded() {
            assertTimeout(
                ofMillis(10),
                () -> {
                    Order.process();
                });
    }
@Test
void timeoutNotExceededWithMethod() {
    String greeting = assertTimeout(
        ofMinutes(2),
        AssertionsDemo::greeting);
    assertEquals("Hello, World!", greeting);
}
```

- **assertTimeout** can be used to impose a time limit on an action.
    - Time limit stated using ofMilis(..), ofSeconds(..), ofMinutes(..)
    - Result of action can be captured as well, allowing checking of result correctness.

Activity - Unit Testing

You are testing the following method:

```
public double max(double a, double b);
```

Devise three executable test cases for this method in the JUnit notation. See the attached handout for a refresher on the notation.

Activity Solution

```
@Test
  public void aLarger() {
    double a = 16.0;
    double b = 10.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue(“should be larger”, actual>b);
    assertEquals(expected, actual);
  }
@Test
  public void bLarger() {
    double a = 10.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertThat(“b should be larger”, b>a);
    assertEquals(expected, actual);
  }
```

```
@Test
  public void bothEqual() {
    double a = 16.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertEquals(a,b);
    assertEquals(expected, actual);
  }
@Test
  public void bothNegative() {
    double a = -2.0;
    double b = -1.0;
    double expected = -1.0;
    double actual = max(a,b);
    assertTrue(“should be negative”,actual<0);
    assertEquals(expected, actual);
  }
```

Best Practices

- Use assertions instead of print statements

```
public class StringUtil {
    public String concat(String a,String b) { return a + b;}
}
@Test
public void testStringUtil_Bad() {
    String result = stringUtil.concat("Hello ", "World");
    System.out.println("Result is "+result);
}

@Test
public void testStringUtil_Good() {
    String result = stringUtil.concat("Hello ", "World");
    assertEquals("Hello World", result);
}
```

- The first test will always pass (no assertions)
  - Developer would need to manually verify the output.

Best Practices

- Even if the code is non-deterministic, build tests that yield deterministic results.

```
public long calculateTime(){
    long time = 0;
    long before = System.currentTimeMillis();
    veryComplexFunction();
    long after = System.currentTimeMillis();
    time = after - before;
    return time;
}
```

- Each time this method is executed, the result will differ.
- Tests for this method should not specify the exact time returned, but properties of a “good” execution.
    - The time should be positive, not negative or 0.
    - Couple place a range on the output.

Best Practices

- Test negative scenarios and boundary cases, in addition to positive scenarios.
  - Can the system handle invalid data?
  - Example: Method expects a string of length 8, with only A-Z,a-z,0-9.
    - Try non-alphanumeric characters.
    - Try a blank value.
    - Try strings with length < 8, > 8
- Boundary cases test extreme values.
  - If method expects numeric value from 1 to 100, try 1 and 100.
    - Also, 0, negative, 100+ (negative scenarios).

Best Practices

- Test only one code unit at a time.
  - Capture each scenario in a separate test case.
  - Method with two parameters: separate one null, other null, both null, and “happy path” into different test cases.
  - Helps in isolating and fixing faults.
- Don’t use unnecessary assertions.
  - Unit tests are a specification on how behavior should work, not a list of observations.
  - Aim for each unit test method to perform exactly one assertion - at least ensure all assertions are related in purpose.

Best Practices

- Make each test independent of all others.
  - Use @BeforeEach and @AfterEach to set up state and clear state before the next test case.
- Create unit tests to target exceptions.
  - If an exception should be thrown based on certain input, make sure the exception is thrown.
- Name test cases clearly and consistently.
  - Name tests after what they do and test.
  - Name should encode operation, scenario, and expectation:
    - TestCreateEmployee_NullId_ShouldThrowException
    - TestCreateEmployee_NegativeId_ShouldThrowException
    - TestCreateEmployee_DuplicateId_ShouldThrowException
    - TestCreateEmployee_ValidId_ShouldPass

Scaffolding

- Stubs and drivers are code written as replacements other parts of the system.
  - May be required if pieces of the system do not exist.
- Scaffolding allows greater control over test execution and greater observability to judge test results.
  - Ability to simulate dependencies and test components in isolation.
  - Ability to set up specialized testing scenarios.
  - Ability to replace part of the program with a version more suited to testing.

Replacing Interfaces

- Scaffolding can be complex - can replace any portion of the system.
- If an interface does not allow control or observability - write scaffolding to replace it.
  - Allow inspection of previously-private variables.
  - Replace a GUI with a machine-usable interface.
  - May be useful after testing.
    - Expose a command-line interface for scripting.

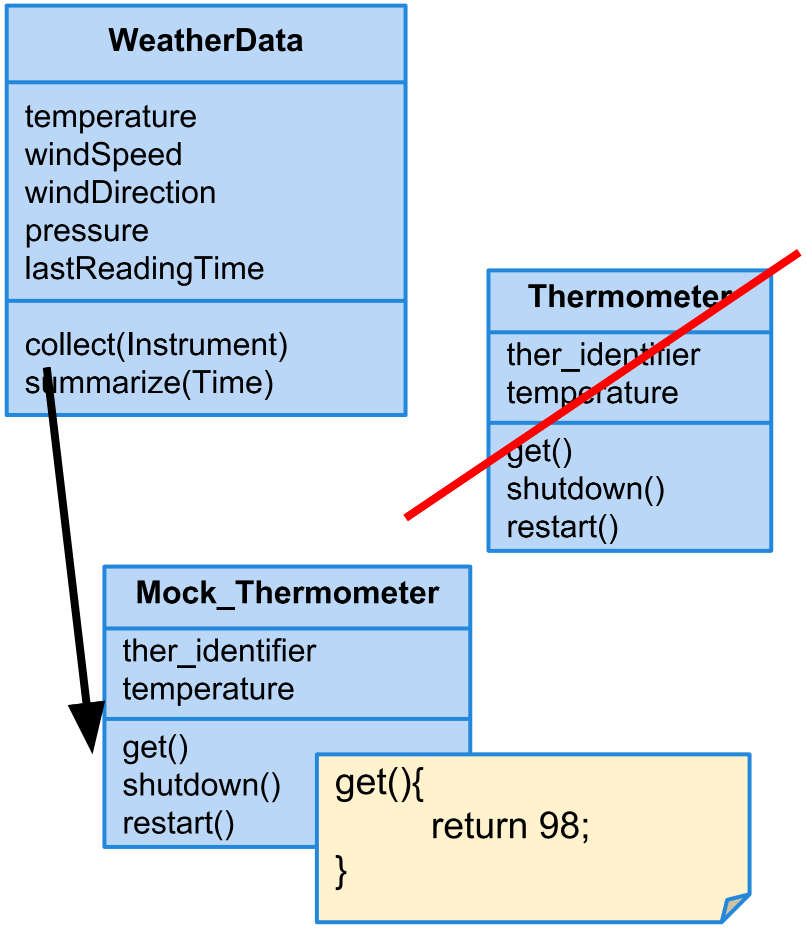
Generic vs Specific Scaffolding

- Simplest driver - one that runs a single specific test case.
- More complex:
  - Common scaffolding for a set of similar tests cases,
  - Scaffolding that can run multiple test suites for the same software (i.e., load a spreadsheet of inputs and run then).
  - Scaffolding that can vary a number of parameters (product family, OS, language).
- Balance of quality, scope, and cost.

Object Mocking

Components may depend on other, unfinished (or untested) components. You can **mock** those components.

- Mock objects have the same interface as the real component, but are hand-created to simulate the real component.
- Can also be used to simulate abnormal operation or rare events.



Mocking Example (Mockito)

- Declare a mock object:
  `LinkedList mList = mock(LinkedList.class);`
- Specify method behavior:
  `when(mList.get(0)).thenReturn(“first”);`
  - Returns “first”: `mList.get(0);`
  - Returns null: `mList.get(99);`
    - Because behavior for “99” is not specified.

  `when(mList.get(anyInt()).thenReturn(“element”);`
  - `mList.get(0)`, `mList.get(99)` both return “element”, as all input are specified.

Mocking Within a Test

```
@test
public void temperatureTest(){
    Thermometer mockTherm =
                mock(Thermometer.class);
    when(mockTherm.get()).thenReturn(98);
    WeatherData wData = new WeatherData();
    wData.collect(mockTherm);
    assertEquals(98,wData.temperature);
}
```

We Have Learned

- Test automation can be used to lower the cost and improve the quality of testing.
- Automation involves creating drivers, harnesses, stubs, and oracles.
- Test cases are often written in unit testing frameworks, as executable pieces of code.
    - Assertions allow deep examination of program output for failures.

Next Time

- Build Systems and Continuous Integration
    - Ensuring your system can be compiled, tested, and deployed on command.

- Assignment 2
    - Due March 3
    - Any questions?