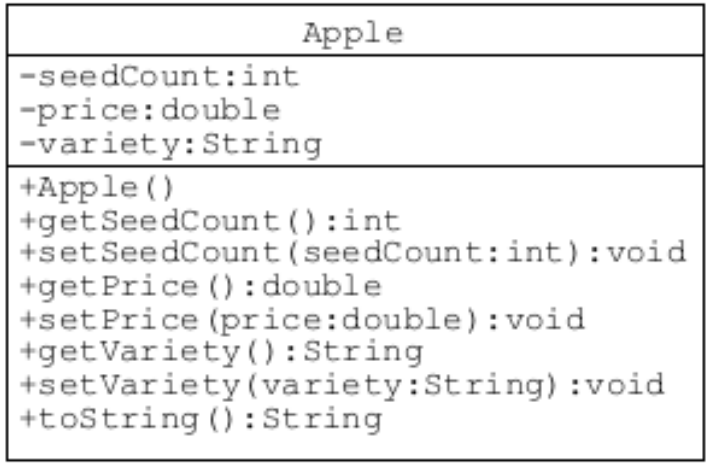
CST8284 Unit Testing Part 1 Overview

Class Apple as used in this handout and the UMLet script (UMLet.org diagram software)




```
Apple
--
-seedCount:int
-price:double
-variety:String
--
+Apple()
+getSeedCount():int
+setSeedCount(seedCount:int):void
+getPrice():double
+setPrice(price:double):void
+getVariety():String
+setVariety(variety:String):void
+toString():String
```

Class Apple Test Plan as seen in an Intro Programming Course

Input	Expected Output	Actual Output	Description
5 0.25 Red Delicious	Seed Count: 5, Price: 0.25, Red Delicious.	Seed Count: 5, Price: 0.25, Red Delicious.	Matches
Tuna Fish 0.25 Red Delicious	Crash? Space Monkeys? Temporal Tuna Fish Paradox?	InputMismatchException	Program crash.

- This test plan tests the input and output of a program over-all from start to finish, for a simple program this is okay.
- Larger more complex software will test first at the most detailed level, for example examine a single class especially those with fewer dependencies first, rather than at a program level as seen here (above). We are assuming in the table above at least one Apple was instantiated with a constructor, the set methods were called, and then method toString() to get the output.
- Unit testing tests at a Unit level, here a Unit would be a class.
- To more formally test a class, the constructor(s), the get, set methods, the worker methods a different table can be used (next page).
- This type of table also lends itself for use with test automation software like JUnit testing.

Improved Test Plan Table (This Course / Future Courses)

Test ID	Member	Test Description	Pre-Condition	Test methodology	Post-Condition expected results	Post-Condition actual results
4	getSeedCount	Check that appropriate value is returned	Apple object has been created, seedCount was changed to 25 using setSeedCount	JUnit test case	Returned value should be 25	Matches

Test ID
- This is a unique number that can be used in a programmer comment as well as test report output so a reviewer can cross check against the test plan as seen above. We need to do our best to use unique identifiers where our test table is only in MS Word.

Member
- What class member are we testing; anything that is accessible outside the class can be tested (public members typically)

Test Description
- We know the class member we are testing; this is a brief description of what we are testing.

Pre-Condition:
- This documents if there are any special object states to consider for testing, starting values, availability of external resources etc.

Test methodology
- This is a note on how the test will be conducted. Examples include stepping through code with a debugger (white box testing), JUnit test case (black box testing), program output, program logging output, and more.
    - white box testing means inner workings of a class can be seen “working in the light (white)”
    - black box testing means inner workings of a class are not known i.e. “working in the dark (black)”

Post-Condition expected results
- This can document the state of the object under test after the member under test has been manipulated
- This plans for the @Test method, which will use an assert method call to conduct the test. Either an error message is reported or simply a test-passed message.

Post-Condition expected results
- This is where we document what the test result was; if a test fails, the report would need to be brought to the attention of the programmers working on the project so it can be fixed.

Other Considerations:
- Each of the rows in the table should be independent of other rows.
- Although Test ID values increase incrementally, the tests themselves could be conducted in any order.

Full Test Plan Class Apple

Test ID	Member	Test Description	Pre-Condition	Test methodology	Post-Condition expected results	Post-Condition actual results
1	Constructor	Check that constructor sets seed count to 10 as a starting value	Apple object has been created using no-parameter constructor	JUnit test case	Field seedCount should have value of 10	Matches
2	Constructor	Check that constructor sets price to 0.25D as a starting value	Apple object has been created using no-parameter constructor	JUnit test case	Field price should have value of 0.25D	Matches
3	Constructor	Check that constructor sets variety to “Red Delicious” as a starting value	Apple object has been created using no-parameter constructor	JUnit test case	Field variety should have value of “Red Delicious”	Matches
4	getSeedCount	Check that appropriate value is returned	Apple object has been created, seedCount was changed to 25 using setSeedCount	JUnit test case	Returned value should be 25	Matches
5	setSeedCount	Check that argument passed is assigned to field correctly	Apple object has been created, setSeedCount was changed to 25 using setSeedCount	JUnit test case	Field should be changed to 25	Matches
6	getPrice	Check that appropriate value is returned	Apple object has been created, price was changed to 0.50D using setPrice	JUnit test case	Returned value should be 0.50D	Matches
7	setPrice	Check that argument passed is assigned to field correctly	Apple object has been created, price was changed to 0.050D using setPrice	JUnit test case	Field should be changed to 0.50D	Matches
8	getVariety	Check that appropriate value is returned	Apple object has been created, variety was changed to “Grannysmith” using setVariety	JUnit test case	Returned value should be “Grannysmith”	Matches
9	setVariety	Check that argument passed is assigned to field correctly	Apple object has been created, variety was changed to “Grannysmith” using setVariety	JUnit test case	Field should be changed to “Grannysmith”	Matches
10	toString	Check that generated string has correct format and values	Apple object has been created, toString is called and tested against default values	JUnit test case	Returned value should be "Seed Count:10, Price:0.25, Red Delicious"	Matches

Sample JUnit Tests Matching Test Plan Seen Above

- Note:
  - Javadoc comments should be used on each method, conveying the information in the paper test plan above.
  - Each @Test method only has one assert statement!
    - The first assert that fails ends the test method execution, so if you attempt to 'save-time' by creating one test method that has many assert statements all you are doing is preventing all of your tests from running if one assert fails.
  - The testing framework chooses the order the tests are run in
    - Do not assume that test id 1 runs 1st, test id 2 runs 2nd or that test methods are run from top to bottom of the file.
    - Each @Test should be self-contained and not depend on leftover object states from other @Test methods.

```
package applefarmtest; // should be used to keep test code away from the rest of the project code
import static org.junit.Assert.*;
import org.junit.Test;
import applefarm.Apple;

public class AppleTest {
    /* Note: test id's match a Word document provided as part of this in class activity
     * The tests themselves will not necessarily be run in the same order as the id values,
     * or even in their order of appearance from top-down within this file.
     * Stan
     */

    @Test // id = 1
    public void testAppleConstuctorDefaultSeedCount(){
        Apple apple = new Apple();
        final int SEED_COUNT = 10;
        int result = apple.getSeedCount();

        assertEquals(
                "Apple constuctor did not set correct seed count",
                SEED_COUNT, result);

        // apple = null;
    }
```

```java
@Test // id = 2
public void testAppleConstuctorDefaultPrice(){
    Apple apple = new Apple();
    final double PRICE = 0.25;
    final double DELTA = 1.0E-3; // +/-0.001
    double result = apple.getPrice();

    assertEquals(
            "Apple constuctor did not set correct Price",
            PRICE, result, DELTA);

    // apple = null;
}

@Test // id = 3
public void testAppleConstuctorDefaultVariety(){
    Apple apple = new Apple();
    final String VARIETY = "Red Delicious";
    String result = apple.getVariety();

    assertEquals(
            "Apple constructor did not set correct variety",
            VARIETY, result);

    // apple = null;
}
```

```java
@Test // id = 4, 5
public void testAppleGetSetSeedCount(){
    Apple apple = new Apple();
    final int NEW_SEED_COUNT = 25;
    int result = 0;

    apple.setSeedCount(NEW_SEED_COUNT);
    result = apple.getSeedCount();

    assertEquals("Get Set SeedCount not correct",
            NEW_SEED_COUNT, result);
    // apple = null;
}

@Test // id = 6, 7
public void testAppleGetSetPrice(){
    Apple apple = new Apple();
    final double NEW_PRICE = 0.50;
    final double DELTA = 0.001;
    double result = 0;

    apple.setPrice(NEW_PRICE);
    result = apple.getPrice();

    assertEquals("Get Set Price not correct",
            NEW_PRICE, result, DELTA);
    // apple = null;
}
```

```
    @Test // id = 8, 9
    public void testAppleGetSetVariety(){
        Apple apple = new Apple();
        final String NEW_VARIETY = "Grannysmith";
        String result = null;

        apple.setVariety(NEW_VARIETY);
        result = apple.getVariety();

        assertEquals("Get Set Price not correct",
                NEW_VARIETY, result);
        // apple = null;
    }

    @Test // id = 10
    public void testAppleToString(){
        Apple apple = new Apple();
        final String EXPECTED =
                "Seed Count:10, Price:0.25, Red Delicious";
        String actual = apple.toString();
        assertEquals("toString did not return expected value",
                EXPECTED, actual);
        // apple = null;
    }

}
```