CSCD 212
Design Patterns
Lab 6
Test Driven Design (TDD)
Read the full document before you start doing anything!!!!!

Table 1 below, is a guide for our initial plan. NOTE: We are not using the LifeForm we created in the last lab. We will create a new simplistic LifeForm. The reason for this is we want to focus on Test Driven Design, and not on the inheritance and composition from LifeForm.

Environment Class Keep track of a small world of cells in which LifeForms can exist. These cells are arranged in a simple grid.	**Cell Class** Keeps track of a LifeForm, can be added on to keep track of other objects as well.	**LifeForm Class** An entity that resides in the world.
Instance Variables	**Instance Variables**	**Instance Variables**
Cell[][]theCells – A 2D array of individual cells	**LifeForm entity** – The LifeForm located in this Cell	**String name** – The name of the LifeForm **int currentLifePoints** – the amount of damage the LifeForm can withstand before dying
Methods	**Methods**	**Methods**
boolean addLifeForm(int row, int col, LifeForm entity) – Adds a LifeFrom to the Cell theCells[row][col]. Will not add the LifeForm if the row and col are invalid or if a LifeForm is already in that Cell. Returns true if successfully added, false otherwise. **LifeForm removeLifeForm (int row, int col)** – Removes the LifeForm at theCells[row][col]. Returns the LifeForm removed, null if no LifeForm in the Cell.	**boolean addLifeForm(LifeForm lf)** – Adds a LifeForm to this Cell. Will not add if LifeForm is already in the Cell. Returns true if added, false otherwise. **LifeForm removeLifeForm ()** – Removes the LifeForm in the Cell. Returns the LifeForm removed, null if no LifeForm in the Cell. **LifeForm getLifeForm()** – Returns the LifeForm in the Cell.	**String getName()** – Returns the name of the LifeForm. **int getCurrentLifePoints()** – Returns the currentLifePoints of the LifeForm.

<u>Setting Up Our Tests</u>

In Eclipse create a new project from the files cloned in your repository.

In the repository is the following

CSCD 212
Design Patterns

```
package cscd212tests.lab6;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class CSCD212Lab6LifeFormTests
{
	@Test
	void testInit()
	{
		LifeForm entity;
		entity = new LifeForm("Bob", 40);
		assertEquals("Bob", entity.getName());
		assertEquals(40, entity.getCurrentLifePoints());
	}

}
```

At this point the code won't compile because we haven't added the proper imports, and we haven't built the LifeForm class.

Notes About our Test

1) The import static statement at the top of the file is telling the compiler where to find the definition of specific JUnit classes that we need. They should give compiler errors.
2) We have exactly one test and it is defined by the method named testInit. It is good practice to name test methods as test<What you are testing>. (without the <>)
3) @Test precedes the declaration and is known as an annotation. It tells the compiler and the JVM the method is a JUnit test case.
4) Our first test case makes a modification to our code in the chart. We will now have to add a constructor.

Make the Test Compile

1) Create the simple LifeForm class. This class will reside in the package cscd212classes.lab6.
2) Your class should only contain base code for the tests.
    a. For this moment leave the constructor empty.
    b. You should have an empty constructor, and two get methods
    c. Don't forget the private data members in the class.
3) Run the tests. They should fail
4) In Eclipse select Help -> Eclipse Marketplace
5) In the search box type EclEmma and ensure it is installed
6) In Eclipse select the Play Button with the half green/half red line below it

7) Your output should be similar to the images below

LifeForm.java | CSCD212Lab6LifeFormTests.java

```
1  package cscd212classes.lab6;
2
3  public class LifeForm
4  {
5      private String name;
6      private int currentLifePoints;
7
8      public LifeForm(final String name, final int currentLifePoints)
9      {
10
11     }
12
13     public String getName()
14     {
15         return this.name;
16     }
17
18     public int getCurrentLifePoints()
19     {
20         return this.currentLifePoints;
21     }
22
23 }
24
```

```
1  package cscd212tests.lab6;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  import org.junit.jupiter.api.BeforeEach;
6  import org.junit.jupiter.api.Test;
7
8  import cscd212classes.lab6.LifeForm;
9
10 public class CSCD212Lab6LifeFormTests
11 {
12     @Test
13     void testInit()
14     {
15         LifeForm entity;
16         entity = new LifeForm("Bob", 40);
17         assertEquals("Bob", entity.getName());
18         assertEquals("Bob", entity.getName());
19     }
20 }
21
```

Examining the LifeForm class, the code in green indicates code that was executed when your test ran. Notice getName was executed and getCurrentLifePoints was not executed. This is because the assertEquals("Bob", entity.getName()); failed.

Make the Test Pass

1) Add the code to the constructor. Don't worry about preconditions at this point.
2) Rerun the tests

Questions

1) Does our test violate the Single Responsibility Principle?
2) If we write another test and copy the code, removing the test for name will this violate the Code Smell of duplicated code?

CSCD 212
Design Patterns

Refactor CSCD212Lab6LifeFormTests

Since we violate Single Responsibility Principle, we need to break our tests into two different test cases. We don't want to violate our code smell of duplicate code; therefore, we are going to add a JUnit method that will create the object each time before our tests are executed.
The JUnit test is called

@BeforeEach
void setup() throws Exception
{
}

In our case we will to complete two task. First, add a private data member LifeForm entity at a class level. Second add entity = new LifeForm("Bob", 40); to the setUp test.

This test case will create a new object and then execute the appropriate tests.

Now you need to separate the two tests into two separate methods. You will need to rename the testInit method to be named testGetName and you will need to create another test named testGetCurrentLifePoints. Add the appropriate assertEquals to each test. NOTE: there will be a single line of code in each test.

Setting Up Cell using Tests

Since an Environment is made up of Cells, we need to create this class next. We created LifeForm first because it was the only class that did not rely on any other classes. As before we are going to use TDD to develop our code, using the order RED, GREEN, REFACTOR.

Start by creating a new class named CSCDLab6CellTests

The code should contain the following

```
1  package cscd212tests.lab6;
2
3  import static org.junit.jupiter.api.Assertions.*;
4
5  import org.junit.jupiter.api.Test;
6
7  import cscd212classes.lab6.*;
8
9  public class CSCD212Lab6CellTests
10 {
11
12     @Test
13     void testInit()
14     {
15         Cell cell = new Cell();
16         assertNull(cell.getLifeForm());
17     }
18 }
19
```

Notice we are using assertNull which checks the method call to ensure it returns null.

NOTE: TDD dictates RED, GREEN, REFACTOR, or fail, pass, refactor. In our case this means we will create testInit and put all our test cases in testInit. We will write our class code get our tests to pass, and then refactor our code to break up the test cases.

CSCD 212
Design Patterns

Create the Cell Class

Using the test case above create the Cell class to pass the test. You should have nothing else in the Cell class for the moment other than a single method. Why didn't we write the constructor?

At this point, everything should compile. Run the test and watch it pass. This is a rare time where the simplest code gets your test to pass. Let's see if we can get a test, or two, to fail by writing the method below.

```
@Test
public void testAddLifeForm()
{
    LifeForm bob  = new LifeForm("Bob", 40);
    LifeForm fred = new LifeForm("Fred", 40);
    Cell cell = new Cell();
    boolean success = cell.addLifeForm(bob);
    assertTrue(success);
    assertEquals(bob, cell.getLifeForm());
    success = cell.addLifeForm(fred);
    assertFalse(success);
    assertEquals(bob, cell.getLifeForm());
}
```

Hopefully you noticed that we added two additional ways to test the code using JUnit: assertTrue, assertFalse. For assert True/False when the test fails you get the error message:

java.lang.AssertionError: null

Before you run the tests, you need your code to compile, this means adding addLifeForm to your Cell class.

```
public boolean addLifeForm(final LifeForm entity)
{
    return false;
}
```

Compile and run your code. The new tests should fail.

```
@Test
public void testAddLifeForm()
{
    LifeForm bob  = new LifeForm("Bob", 40);
    LifeForm fred = new LifeForm("Fred", 40);
    Cell cell = new Cell();
    boolean success = cell.addLifeForm(bob);
    assertTrue(success);
    assertEquals(bob, cell.getLifeForm());
    success = cell.addLifeForm(fred);
    assertFalse(success);
    assertEquals(bob, cell.getLifeForm());
}
}
```

Now write the code to ensure your tests will pass. This means you will need to add a LifeForm entity as a private data member. You will also need to refactor getLifeForm, and finally you will need to refactor addLifeForm. What code should be added addLifeForm? Hint: look at the primitive design document above.

To Complete Group 1

1) Create a test named testRemoveLifeForm()
2) Add code to the test to test the method
3) Stub out the code in the Cell class

4) Run your test and take a screen capture showing your test from step 2, your stubbed out code from step 3, and your failing test. You can take 3 screenshots or take advantage of eclipse's split editor to put the files side by side. **Label this screenshot Group 1 Step 4.**
5) Refactor your Cell removeLifeForm code to pass the test.
6) Run your test and screen capture the test passed

Is a single test sufficient for testRemoveLifeForm()? The answer is no.

7) Add code to your test to test the other case. What other case you ask? There are two cases in this situation. The Cell is empty and you can't remove a LifeForm or the cell contains a LifeForm and you need to remove it.
8) Refactor your Cell class code for this method if you didn't handle both test cases.
9) Run your test and screen capture the tests passed.
10) Refactor CSCD212Lab6CellTests so the test methods enforce the Single Responsibility Principle and remove the code smell of repeating code.
11) Run your refactored tests and screen capture that all Cell tests passed. Screenshot should show all the methods in the test class involved in completing the requirements of step 10 as well as the passing tests. You can hit the minus sign to the left of the method to collapse the tests if needed. **Label this screenshot Group 1 Step 11.**

Test Driven Development Review

Even though this lab gives you much of the code you need, you have experienced the flow of TDD. At each step you made a test that required a small amount of new functionality, watched it fail, and then wrote the code to make it pass. This incremental approach to development has many benefits. At this point you should see that it helps to divide the problem into smaller, more manageable pieces so you don't have to think about the whole system at one time.

When you are coding, always remember the mantra: FUNCTIONALITY, FAIL, REFACTOR, PASS, REFACTOR.

Test Suites

At this point you have multiple test classes and it would be nice to be run all the tests at once. To run all the tests in separate classes at one time we need to create a test suite. The code for creating a test suite is below.

```
1  package cscd212tests.lab6;
2
3  import org.junit.platform.runner.JUnitPlatform;
4  import org.junit.platform.suite.api.SelectPackages;
5  import org.junit.runner.RunWith;
6
7  import org.junit.platform.suite.api.*;
8
9  @RunWith(JUnitPlatform.class)
10 @SelectPackages("cscd212tests.lab6")
11 @SelectClasses({CSCD212Lab6CellTests.class, CSCD212Lab6LifeFormTests.class})
12 public class CSCD212Lab6Tests{}
13
```

When you use JUnit to run this class, it will run all the tests. Notice the class is marked red because it does nothing. The test are run using @RunWith and @Select

Creating the Environment

CSCD 212
Design Patterns

Now it is time to apply what you have learned to the building of the Environment class. An environment is a grid of interconnected Cells For example, we could create a simple Environment that has two rows and three columns

```
Cell [][] cells = new Cell[2][3];
```

Next you will need to go through the cells and create actual Cell objects.

```
cells[0][0] = new Cell();
```

Of course, you will need to create a total of six Cells to fill the two-dimensional array.

Next you will need to add a LifeForm to a Cell.

```
LifeForm bob = new LifeForm("Bob", 40);
cells[1][0].addLifeForm(bob);
```

To Complete Group 2

1) Create CSCD212Lab6EnvironmentTests
2) Create a test to ensure you can create a basic Environment that consists of just one Cell
3) Run your test and screen capture its failure. Be sure to include the test in the screenshot. **Label this screenshot Group 2 Step 3.**
4) Refactor
5) Run your test and screen capture its success. Be sure to include the test and the Environment code in the screenshot. **Label this screenshot Group 2 Step 5.**
6) Systematically create tests, screen capture failure, refactor screen capture success for the Environment methods in the chart. You test methods will be named testMethodName for what you are testing. **Label the group of screenshots Group 2 Step 6** and label each individual screenshot in a way that makes it easy to tell what it is.
7) Remember to ensure you follow the Single Responsibility Principle and you don't violate the code smell of repeating code.

Keeping the Internals Secure

An important note as you try to get your tests to pass, has to do with testing the Cell exists and is empty. When building a class like Environment it is a wise programming practice to not allow direct access to non-primitive data (e.g Cell). This is because not doing so grants other classes the ability to make direct changes to an instance variable without having to go through a method of that instance variable's class. While it is easy to write a method that would return the Cell at a specific location to get the test to pass (e.g. getCell(row, col)), it would not be wise to do this.

Instead of returning the Cell directly, you will need to create a new method called getLifeForm that will return the LifeForm at that Cell location. You won't be able to directly test that the Cell is there, but you can at least test for its presence indirectly (we can't get a LifeForm from a Cell that doesn't exist). Create a test to use getLifeForm.

CSCD 212
Design Patterns

Unfortunately, there are times when even indirect tests are not feasible and during those times as a Test-Driven Developer you have to choose whether to expose more of the class you want (usually using protected) or simply not testing that part of the class.

To Complete Group 3

1) Create the getLifeForm test, add the code to Environment, run the test capture the fail. Show all 3 parts in the screenshot and label Group 3 step 1.
2) Refactor Environment to pass test, run the test and capture success. Show refactor and test result in the screenshot and label Group 3 step 2.
3) Create an Environment instance that is 2 rows and 3 columns
4) Create a LifeForm instance to store in the Environment
5) Create a second LifeForm instance to store in the Environment
6) Add the first LifeForm to row 1 column 2 of the Environment
7) Check the Environment to ensure it is holding the LifeForm
8) Attempt to add the second LifeForm to row 1 column 2
9) Remove the LifeForm from row 1 column 2
10) Check to ensure the LifeForm is removed from the Environment
11) Add the second LifeForm to row 1 column 2
12) Check to ensure the LifeForm is in the Environment
13) Test boundary cases for LifeForm placement. You have to figure out what these are
14) Add the Environment test class to your test suite
15) Run your test suite

Of course, you will follow the mantra and screen capture test, fail, refactor, pass.

Submit your final code via pushing to your GitHub repository.

You PDF containing all your screen captures, appropriately labeled with **group number and description** will be named your last name first letter of your first name lab6tests.pdf (Example: steinerslab6tests.pdf). This file will reside in your repository at the same level as cscd212tests package.