A Practice in Software Testing with OpenHopla

Jake Collins, Zhou Fang, Michael Heffernan

CS 490 Software Testing Spring 2012

Instructor: Aditya Mathur

Project Sponsor: John Spurgeon at Intel

Table of Contents

1: Introduction 3

2: Lessons Learned 3

3: OpenHopla Code Kata #1 (**SpartanUnit**) 3
3.1 Overview 3
3.2 Comparison with nUnit 3
3.3 Comparison with **PyUnit** 4
3.4 Suggestions for Improvement of **SpartanUnit** 5

4: OpenHopla Code Kata #2 (TriangleGuru) 5
4.1 Testing Preparation 5
4.2 Test Design 9
4.3 Test Results 9
4.4 Suggestions for Improvement 10

5: OpenHopla Code Kata #2 (TriangleGuru) Part 2 9
5.5 Suggestions for Improvement 9
5.5.1 Types of the triangles 9
5.5.2 Conditions to classify the triangles 10

6: OpenHopla Code Kata #4 (DysonNumbers) 11
6.1 Test Preparation 11
6.2 Test Design 12
6.3 Test Results 12
6.4 Suggestions for Improvement 13

7: Acknowledgements 13

Appendix A1 14
Appendix A2 16
Appendix A3 17
Appendix A4 18

1: Introduction

The primary purpose of this project was to gain a deeper understanding of Software Testing through Code Kata and **SpartanUnit** using techniques taught in class. We also wanted to evaluate **SpartanUnit** as a unit-testing framework. A Code Kata is a formal exercise meant to reveal more information about the code in **SpartanUnit**. **SpartanUnit** is a **xUnit** test framework written in C#.

We used the Code Kata's in OpenHopla to test **SpartanUnit**. In addition we compared **SpartanUnit** with traditional x**Unit** frameworks such as **nUnit** and **PyUnit**. **nUnit** is the de-facto standard for testing C# code and **PyUnit** is the **xUnit** implementation for the Python scripting language. As will be explained, **SpartanUnit** takes a completely different direction than nUnit and **PyUnit** but the comparison will still reveal some interesting similarities and differences.

2: Lessons Learned

- **OpenHopla** and **SpartanUnit** make great tools to use in almost any testing situation.
- **SpartanUnit** uses a very simple implementation that would not take long to understand completely for any programmer familiar with object-oriented programming.
- No test framework could ever be the end all solution.
- It is up to the coder to decide which framework he/she wants to use.
- **SpartanUnit** seems to be one of the most versatile test frameworks out there, but still needs work to be more programmer friendly and attractive to multiple groups.

3: Code Kata #1 (SpartanUnit)

3.1 Overview

Code Kata #1 is an exercise to familiarize a programmer with **SpartanUnit**. We chose to implement **SpartanUnit** and compare it with other **xUnit** frameworks to maximize our understanding of the purpose and place of **SpartanUnit**.

3.2 Comparison with nUnit

A comparison of **nUnit** and **SpartanUnit** seems to come down to these points: Features and Ease of Use (and these are not mutually exclusive). Upon looking at the source code for **nUnit**, it is clear from the hundreds of class files

that **nUnit** is a unit-testing framework with many features To name a few, **nUnit** has the capability to watch open files for changes, change directories while remembering previous locations, be run either by command line or GUI, networking interaction, etc. These features can be very useful for the experienced user, allowing very in depth fine tune unit testing. The keyword there, however, is 'experienced'. To this college student, it seems **nUnit** could take a significant amount of time to learn. In addition, without the aid of tutorials and online forums, trying to understand how to use it would be a daunting task of swimming through a sea of source files and examples.

SpartanUnit, in contrast, contains only the features necessary for a unit-testing framework. This allows programmers to quickly pick up on the essence of unit testing and create useful tests for their current task. Additionally, this succinct framework style lends itself to be intuitively understood simply from the source.

*3.3 Comparison with **PyUnit***

Upon comparing **SpartantUnit** and **PyUnit** we have found some definite benefits and drawbacks of both implementations. Below is a comparison of the pros and cons of **SpartanUnit** over **PyUnit**.

Pros	Cons
• More versatile for multiple languages • More versatile interaction with the test	• Harder to interface with results • Much more code to implement

SpartanUnit will be easy to port to about any object oriented programming language since it is a barebones framework using simple object oriented concepts. Since the tests are detached in a way from the user interaction it is possible to present the results in a number of different ways to the user or even another program. For example, the tests we ran used a dialog box with a "Run Tests" button that presented which tests passed and which failed. The drawbacks of this interface were clear. If one test failed it would cancel all the tests after the failed test. If a test failed it could not give any detailed information of which part of the test failed or why. But since **SpartanUnit** is so versatile another interface could be written to run it via command line or a better GUI.

PyUnit runs strictly in the command line and immediately informs the user why the test failed and even gives you a summary of differences when it

compares equality of objects. **PyUnit** also requires much less code to get a test up and running, but as a consequence the tests must be run command line. **PyUnit** also offers more assertions than **SpartanUnit**. The **assertEqual** method is very powerful. There is also an assertFalse method, which is not too useful as a "!" can make an **Assert.True** into a test for falseness. However it still would make the tests more readable if there were an **Assert.False** method.

Overall a comparison between **SpartanUnit** and **PyUnit** is not very useful as **PyUnit** is more of a quick and dirty unit test framework and **SpartanUnit** is a framework made to be easily tweaked to run in multiple situations and can be easily ported to different languages. An implementation of **SpartanUnit** and **PyUnit** testing an array search program can be found in the Appendix A.1.

*3.4 Suggestions for improvement of **SpartanUnit***

One major drawback experienced during testing was that TestSuites do not continue after a test fails. It would be preferred that all tests are run regardless of the result.

4: OpenHopla Code Kata #2 (TriangleGuru) Dominators

4.1 Testing Preparation

To prepare to test OpenHopla's triangle testing I constructed flow graph for the function **ClassifyTriangle** in the **TriangleGuru** class. The functions returning Boolean values can be considered basic blocks for the sake of simplicity. The resulting flow graph is shown below in Figure 4.1a with the simplified version in Figure 4.1b.

In Figure 4.1a we can see the code in **TriangleGuru** represented in blocks. These are called basic blocks. A basic block of code is a subset of the functions code that executes in a single path. This means that all of the code in each block is executed. The edges represent where the path changes (In this case **if-**statements).

In Figure 4.1b we can see the simplified version of 4.1a. This simply replaces each basic block with a corresponding number. This makes the flow graph easier to read.

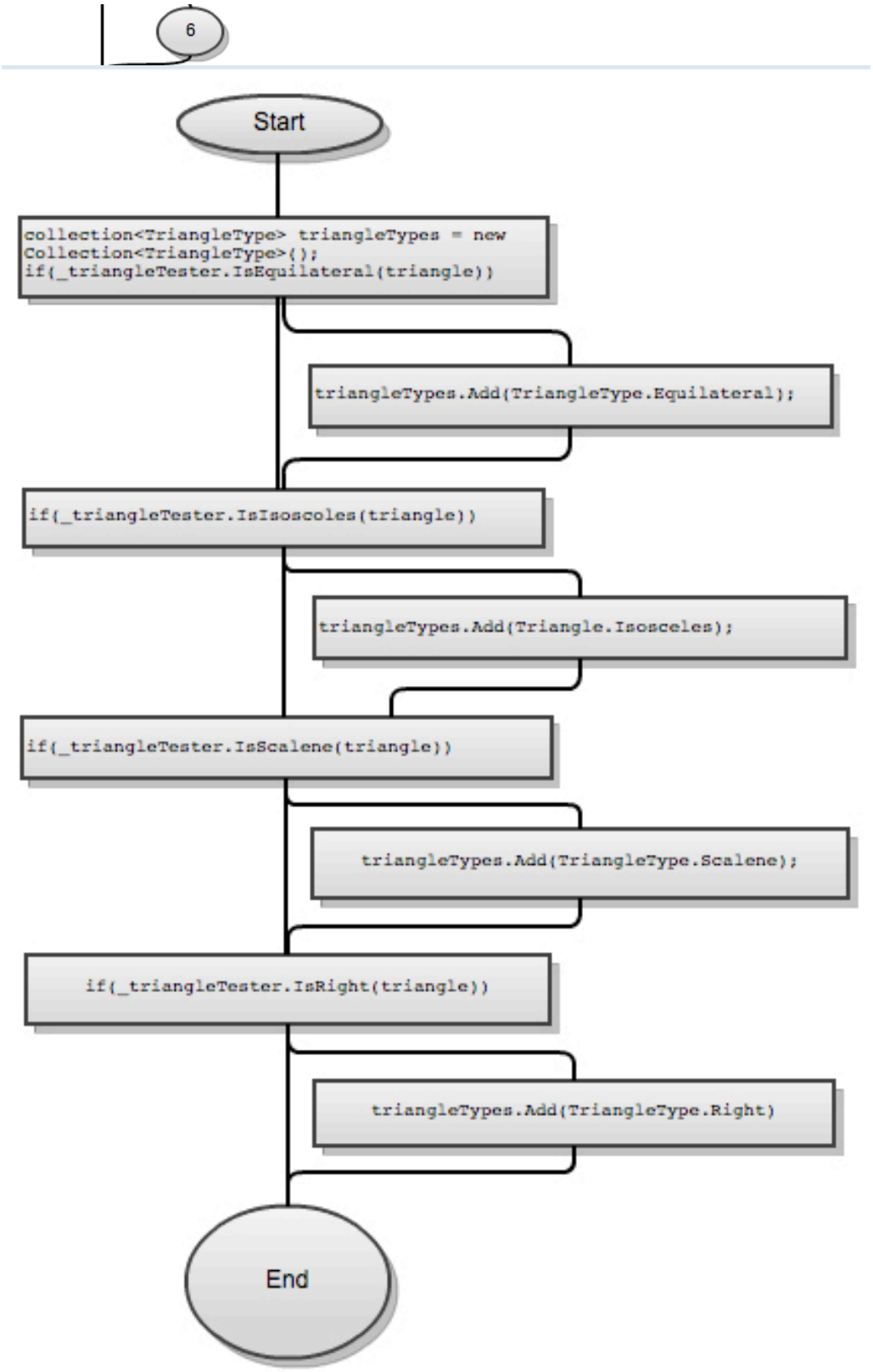
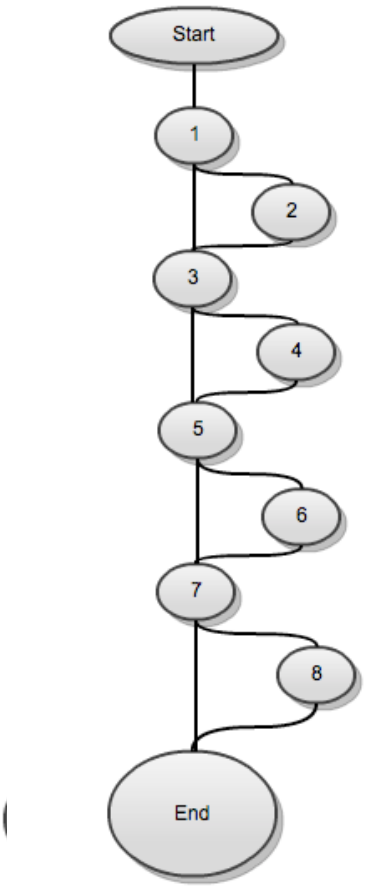
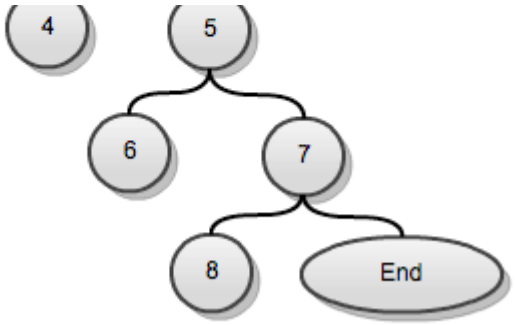

6
Start
collection<TriangleType> triangleTypes = new
Collection<TriangleType>();
if(_triangleTester.IsEquilateral(triangle))
triangleTypes.Add(TriangleType.Equilateral);
if(_triangleTester.IsIsoscoles(triangle))
triangleTypes.Add(Triangle.Isosceles);
if(_triangleTester.IsScalene(triangle))
triangleTypes.Add(TriangleType.Scalene);
if(_triangleTester.IsRight(triangle))
triangleTypes.Add(TriangleType.Right)
End

Figure 4.1a





Figure

The fl or tree
(Figure 4.2) tested. As
shown below er code
blocks will be

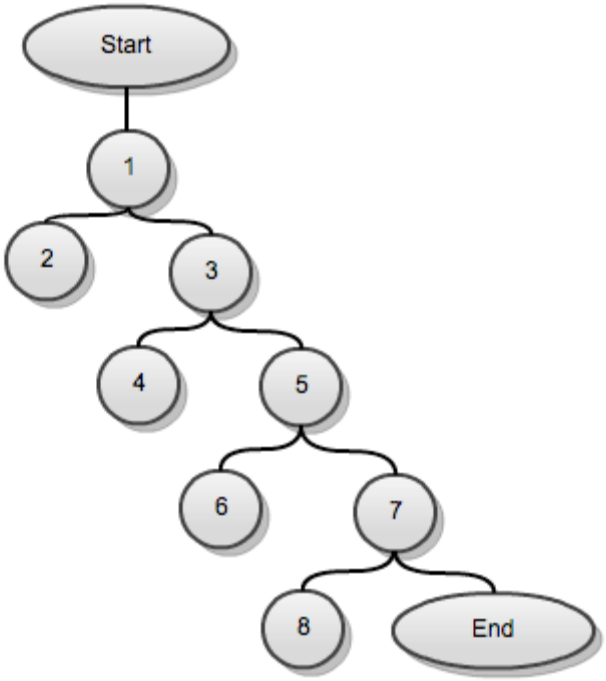


Figure 4.2

Since code blocks 2, 4, 6 and 8 are considered most important, we first need to examine the code in these particular blocks. These four blocks of code are executed if the conditional statement before it is evaluated to true.

```
2: triangleTypes.Add(TriangleType.Equilateral);
4: triangleTypes.Add(TriangleType.Isosceles);
6: triangleTypes.Add(TriangleType.Scalene);
8: triangleTypes.Add(TriangleType.Right);
```

It is obvious why these 4 lines are the most important. These are the 4 possible types of triangles that **TriangleGuru** can classify. We can have triangles containing the following types of triangle in a collection.

Collection 1	**Collection 2**	**Collection 3**	**Collection 4**
Equilateral	**Isosceles**	**Isosceles**	**Scalene**
Isosceles*	**Right**		

4.2 Test Design

All of the above Collections in Section 5.2 are the equivalence classes in an enumeration. So we need to test all possible collections, and this will also cover all of our important finds from the dominator tree. These tests can be found in the Appendix A.2.

4.3 Test Results

All tests of a single triangle of collections 2, 3 and 4 passed as expected. However the test that tested a triangle of being both equilateral and isosceles failed. This would have to do with the accuracy of floating point values and the reason would be unknown to a black box tester but a white box tester would quickly find that **TriangleGuru** is directly comparing the number of *unique* values in the triangle sides. Since the sides are not exactly equal, but as close as possible, they do not pass the test.

4.4 Suggestions for improvement

OpenHopla needs to add an epsilon difference as there is in the method *IsRight*. This would be about equal to the difference between the square root of three and the actual value that could be easily calculated on the fly.

5: OpenHopla Code Kata #2 (TriangleGuru) Other Tests

Multiple types of tests were executed on the triangle tester. These types are enumerated below:

Functions	I have test every single functions of the triangle to make sure it works properly. Do find problems in function public bool IsRight(Triangle triangle) in triangleTester. (same as in object based test)
Correctness	1) All types of input such as int, double, str, char, positive and negative value. 2) Object based test of invalid triangle: less or more than 3 sides, side of length 0, inputting same coordinate, not a closed graph. (Problems found in function public bool IsRight(Triangle triangle) in triangleTester) 3) boundary value test for all types of triangle. 4) every possible type of triangle
Accuracy	Using epsilon to test how low we can represent triangle's accuracy. As expected using an epsilon value lower than that hardcoded in the IsRight function caused IsRight to produce unexpected results.

Stress	Tried to add as many test cases as possible (up to 20) to see whether program fails
Regression	Tried to test program with same test cases after I add the classification of obtuse
User	Let my friends to randomly pick some coordinates to see whether program works properly.

These test cases can be found in the Appendix A.4

5.5 Suggestions For Improvement

5.5.1 Types of the triangles

Based on the reason to help the project team to understand more about the triangle, we may classify the triangle into more types. We can add obtuse triangle and acute triangle besides current equilateral, Isosceles, scalene, and right. In the following, Definitions of the obtuse and acute triangles follow.

Obtuse triangle: a triangle that has one angle over 90 degrees.
Acute triangle: a triangle in which all angles \less than 90 degrees.

The code for verifying obtuse and acute triangle is easy and the method for verifying a triangle as a right triangle was used with changes to the "=" operator to either ">" or "<". For the right triangle, there is a special triangle that may be useful to test for. These are the 3-4-5, the 30-60-90, and the right isosceles triangles.

For a 3-4-5 triangle, the ratio of length of the sides is 3:4:5. We can get the length of one side by knowing the other two without complicated calculations. Hence, we may have the 11-12-13 triangle by extending this area. For the 30-60-90 triangle, the ratio of the length of the three sides is 1: $\sqrt{3}$:2. One can get the degree of the angles by knowing the lengths or get lengths by knowing angles.

For right isosceles triangle, we have the ratio as 1:$\sqrt{2}$:$\sqrt{2}$. One may get the degree of the angles by knowing the lengths or get lengths by knowing angles.

5.5.2 Conditions to classify the triangles

As in the code, all triangles have to pass every **if-**statement to the end to get the types of the triangle. However, such as in the case of a right triangle, it must not be an equilateral triangle. Hence it is not necessary to pass the **if-**statement. Based on this reason, one can enumerate the following possibilities for triangle classification including the obtuse and acute triangles.

Type	Can Be	Must Be	Must Not Be
Equilateral	N/A	Acute, Isosceles *	Right, Obtuse, Scalene
Isosceles	Right, Obtuse, Acute, Equilateral	N/A	Scalene
Right	Isosceles, Scalene	N/A	Obtuse, Acute, Equilateral
Scalene	Obtuse, Acute, Right	N/A	Isosceles, Equilateral
Obtuse	Isosceles, Scalene	N/A	Right, Equilateral, Acute
Acute	Isosceles, Equilateral, Scalene	N/A	Right, Obtuse

An equilateral triangle is a special type of isosceles triangle. An equilateral triangle is also an isosceles triangle and an isosceles triangle may also be equilateral. But in this project, we may only say equilateral triangle is a triangle with three equal sides and isosceles triangle with 2.

One suggestion is to add an **if**-statement under the outer **if** to verify the "can be" part in the table (if a triangle is isosceles, then we tried to verify if it is equilateral , right, obtuse or acute) and add an **else-if** clause. Even though it may look more complicated but is more efficient and save time. It also may help better understand the types of the triangle.

6: OpenHopla Code Kata #4 (DysonNumbers)

6.1 Test Preparation

It was decided to use boundary value analysis (BVA) as a useful technique for generating tests for the *DysonNumbers* class as it is a class that handles ranges of numbers. The following graph shows the possible values obtained using this technique (denoted by X):

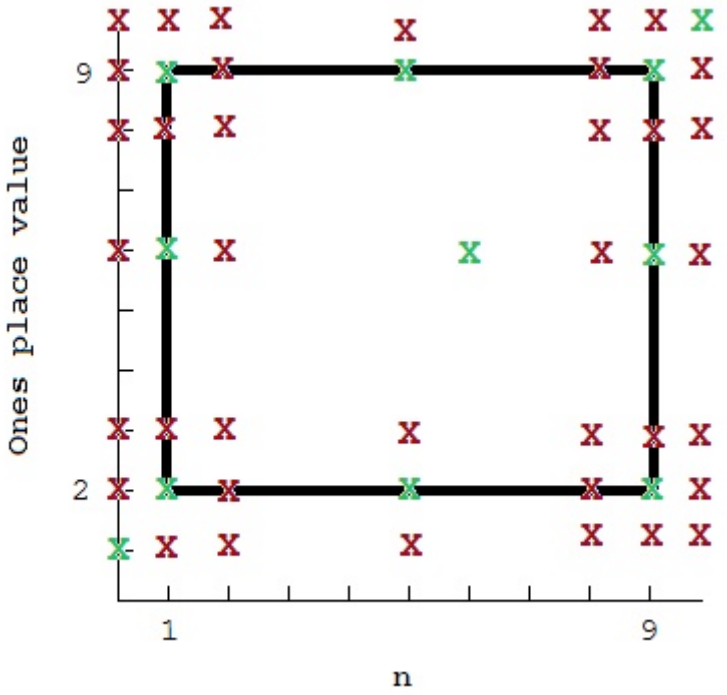


Figure 6.1

The red Xs denote test cases that were not chosen due to time constraints, but would be part of a more complete test suite. The green Xs were chosen because they covered a particular area of interest: on boundary, outside the boundary, or inside the boundary. The internal X, 6:6, was chosen because its answer can be found on Wikipedia's page on Parasitic Numbers.

6.2 Test Design

The best way to test this program would be to select a set of values from the BVA to test, then one could determine if the test yielded the correct result. However, the actual value of any problem computed by this program is not easily found in the literature, and computing them by hand is a time consuming task. Therefore there are three main test designs: ones that are supposed to fail (out of boundary), ones that should succeed but have no provided answer, and ones with answers. The first two were tested by simple control flow: if illegal inputs did not throw an exception, the test failed; if legal inputs did not throw an exception, the test passed unless it was able to be compared to a value.

6.3 Test Results

Unfortunately, most tests failed to yield correct results when n > 2. Reasons for this are brought up in the following section.

6.4 Suggestions for Improvement

It was found early in this Code Kata that for this problem the magnitude of the answers are far beyond the maximum value of the basic integral types. The largest such type in C#/.Net is the **Int64** wrapper type, which has a maximum of ~$9x10^{18}$. Taking into account the 6th Dyson number, ~$1x10^{53}$, this is not adequate.

Considering the size of numbers, the **double** data type initially seems like a good choice, as the maximum integral value representable is ~$1x10^{300}$. However, this too has its drawbacks. First, there is no defined limit for the size of these numbers, so even ~$1x10^{300}$ could be insufficient. Second, using a floating-point data type means that the program would need to be further altered to account for floating point arithmetic. The better choice is the **BigInteger** type, which supports arbitrary precision and acts like an integral type.

The arbitrary precision of the **BigInteger** structure brings into view another potential problem. In the event that the program fails to find the answer, it will continue to calculate until the computer runs out of memory or the tester gets frustrated for waiting too long. Therefore, it is suggested that there be a defined 'overflow' value, perhaps around 400 decimal places. This would allow it to be more accurate than using a **double**, yet prevent the program from calculating indefinitely.

Finally, there was an error in the current implementation of **DysonNumbers.LeftDigit.** The current implementation always computes the **carryDigit** to be 0 or 1. This works fine for n=2, but for anything larger it could become a problem. It was found that setting the **carryDigit** to **nTimesRightDigitPlusCarryDigit/10** allows the program to successfully compute other Dyson numbers, but fails for n=6. Hence a better solution needs to be found.

7: Acknowledgements

We thank John Spurgeon of Intel for the opportunity to test this test framework. We would also like to thank our professor Aditya Mathur for teaching us these insightful ways of testing software.

Appendix A1

Below is the code used to implement an array search test in **SpartanUnit** and after the code used to implement an array search test in Python's **PyUnit.**

SpartanUnit:

```
/* TestController.cs */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SpartanUnit;
using OpenHopla.Geometry;
using System.Collections.ObjectModel;

namespace SpartanUnit.UnitTests.Root
{
    public class TestControllerTest : TestSuite
    {
        public int[] array;
        public TestControllerTest()
        {
            this.TestCases.Add(TestNumberPass);
            this.TestCases.Add(TestNumberFail);
        }
        public override void Setup()
        {
            base.Setup();
            array = new int[10];
            for (int i = 0; i < 10; i++)
            {
                array[i] = i;
            }
        }
        public override void Teardown()
        {
            base.Teardown();
            array = null;
        }
        private void TestNumberPass()
        {
            Assert.True(ArraySearch.find(2, array) == 2);
        }
        private void TestNumberFail()
        {
            Assert.True(ArraySearch.find(2, array) == 3);
        }
    }
}

/* TestSubscriber.cs */
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SpartanUnit;
using SpartanUnit.UnitTests.Root;

namespace SpartanUnit.UnitTests
{
```

```
    public partial class MainTestController : BroadcastingTestController
    {
        private SampleSubscriber _subscriber = new SampleSubscriber();

        public MainTestController()
        {
            _subscriber.SubscribeToBroadcast(this.Broadcast);
            this.TestSuites.Add(new TestControllerTest());
        }

        public string TestResults
        {
            get
            {
                return _subscriber.TestResults;
            }
        }
    }
}
```

PyUnit:

```
/* test.py */
import unittest
import arraysearch

class TestArraySearch(unittest.TestCase):
       def setUp(self):
              self.array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

       def tearDown(self):
              self.array = None

       def testNumberPass():
              self.assertEqual(arraysearch.find(2, array), 2)

       def testNumberFail():
              self.assertNotEqual(arraysearch.find(2, array), 3)

if( __name__ == “__main__” ):
       unittest.main()
```

Appendix A2

Below are test cases in **OpenHopIa** for the triangle tester class. As shown the first test case fails as explained in the report.

```
public void CollectionOne() //Fails
{
    List<OrderedPair> vertices = new List<OrderedPair>();
    vertices.Add(new OrderedPair(0, 0));
    vertices.Add(new OrderedPair(6, 0));
    vertices.Add(new OrderedPair(3, 3 * System.Math.Sqrt(3)));
    Triangle t = new Triangle(vertices);
    Assert.True(_triangleTester.IsIsosceles(t));
    Assert.True(_triangleTester.IsEquilateral(t));
    Assert.True(!_triangleTester.IsRight(t));
    Assert.True(!_triangleTester.IsScalene(t));
}

public void CollectionTwo()
{
    List<OrderedPair> vertices = new List<OrderedPair>();
    vertices.Add(new OrderedPair(0, 0));
    vertices.Add(new OrderedPair(5, 0));
    vertices.Add(new OrderedPair(0, 5));
    Triangle t = new Triangle(vertices);
    Assert.True(_triangleTester.IsIsosceles(t));
    Assert.True(!_triangleTester.IsEquilateral(t));
    Assert.True(_triangleTester.IsRight(t));
    Assert.True(!_triangleTester.IsScalene(t));
}

public void CollectionThree()
{
    List<OrderedPair> vertices = new List<OrderedPair>();
    vertices.Add(new OrderedPair(4, 5));
    vertices.Add(new OrderedPair(1, 1));
    vertices.Add(new OrderedPair(-2, 5));
    Triangle t = new Triangle(vertices);
    Assert.True(_triangleTester.IsIsosceles(t));
    Assert.True(!_triangleTester.IsEquilateral(t));
    Assert.True(!_triangleTester.IsRight(t));
    Assert.True(!_triangleTester.IsScalene(t));
}
public void CollectionFour()
{
    List<OrderedPair> vertices = new List<OrderedPair>();
    vertices.Add(new OrderedPair(0, 0));
    vertices.Add(new OrderedPair(0, 3));
    vertices.Add(new OrderedPair(-2, 5));
    Triangle t = new Triangle(vertices);
    Assert.True(!_triangleTester.IsIsosceles(t));
    Assert.True(!_triangleTester.IsEquilateral(t));
    Assert.True(!_triangleTester.IsRight(t));
    Assert.True(_triangleTester.IsScalene(t));
}
```

Appendix A3

Below are the test cases for OpenHopla's **DysonNumber** class.

```
public void failExample()
{
            string message = "OPV=OUTOFRANGE, n=OUTOFRANGE";
            try
            {
                DysonNumbers.SmallestNParasiticNumber(OUTOFRANGE, OUTOFRANGE);
                Assert.Fail(message + ": false success");
            }
            catch (ArgumentException)
            {
                Assert.True(true, message);
            }
            catch (OverflowException)
            {
                Assert.Fail(message + ": overflow");
            }
}

public void succeedExample()
{
            string message = "OPV=INRANGE, n=INRANGE";
            try
            {
                DysonNumbers.SmallestNParasiticNumber(INRANGE, INRANGE);
                Assert.True(true, message + ": likely success");
            }
            catch (ArgumentException)
            {
                Assert.Fail(message + ": argument problem");
            }
            catch (OverflowException)
            {
                Assert.Fail(message + ": overflow");
            }
}

public void SimpleSix()
{
  try
  {
    Assert.True(DysonNumbers.SmallestNParasiticNumber(6, 6.Equals(BigInteger.Parse(
                  "10169491525423728813559932203389830508474576271186440677966")));
  }
  catch (OverflowException)
  {
    Assert.Fail("Overflow while testing onesPlaceValue=6  n=6");
  }
}
```

Appendix A4

Function Test:

Every function of the triangle was tested to make sure it works properly

Correctness Test:

int (0,0), (0,3), (3,4); (5,5), (7,-7),(-7,-6)

int+ double (0,0), (0,3.1), (3.1,4)

string/char: (0,0), (0,ab),(3,4)

one side: (0,0),(3,4)

four sides (0,0),(0,1),(1,0),(1,1)

**** 3 points with 2 have the same coordinate (-2,0) ,(0,2),(-2,0) it did not call exception when going through TriangleTesterShould.Classify345TrianglesAsRight but failed in TriangleTesterShould.Classify345TrianglesAsScelene

Because in public bool IsRight(Triangle triangle), codes get the length of the sides by ignoring the positive and negative sign of the coordinate.

It means, the function Triangle = new Triangle(vertices); does not drop exception when creating triangles with even same coordinate

Boundary Value Test:

for right isosceles triangle (-2,0),(0,2),(2,0), test if any value of 3 coordinates has a very minor changes would cause isRight and isIsosceles Fails.

(-2.1,0),(0,2),(2,0);
(-2,0.1),(0,2),(2,0);
(-2,0),(0.1,2),(2,0);
(-2,0),(0,2.1),(2,0);
(-2,0),(0,2),(2.1,0);
(-2,0),(0,2),(2,0.1)

Every Possible Types of Triangle: 3 test cases in codes

Equilateral: failed, DistinctSegmentLengths(triangle).Count == 1 in test case (-1,0),(0,System.Math.sqrt(3)),(1,0)

Special Right Triangle:

3-4-5: Passed (0,0),(3,0),(3,4)

30-60-90: Passed (0,0),(System.Math.sqrt(3),0),(System.Math.sqrt(3),1)

Right and Isosceles Passed: (-2,0), (0,2), (2,0)

Obtuse and Acute Triangle: may add after changes the codes