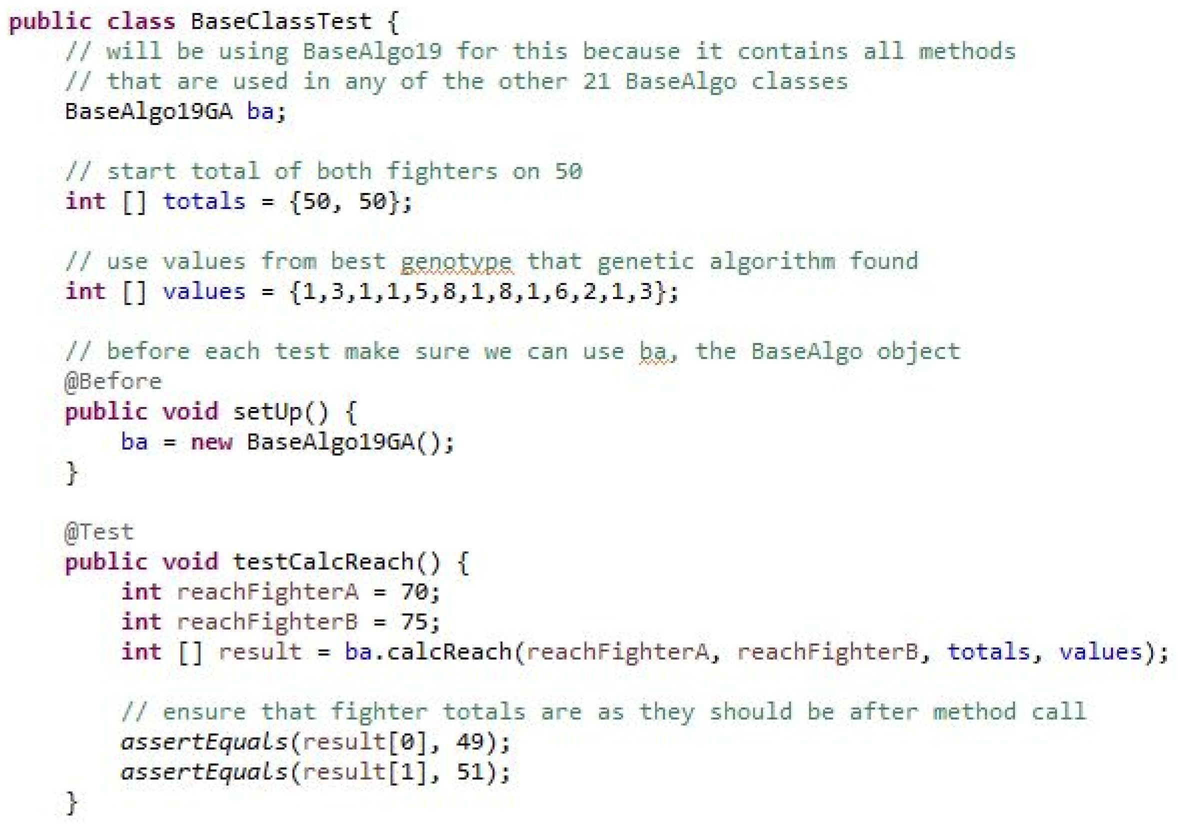
Genetic Algorithm Testing

To test the code which we wrote for the Genetic Algorithms, we decided to use one of the twenty-one algorithms as a basis for the tests. The reason for this was that the algorithm which we chose to use, the nineteenth genetic algorithm, contained every method that any of the other twenty-one algorithms made use of. There was nothing new in any of the other genetic algorithms, they just used a different combination of methods and had different sized genotypes. This meant that after testing everything we could in the nineteenth algorithm, we had essentially tested every genetic algorithm.

We made use of Eclipse and JUnit in order to write test code for this genetic algorithm. Basically, we took one class at a time and wrote test methods which ensured that each method contained in that class functioned as expected and produced correct outputs.

Unit Testing

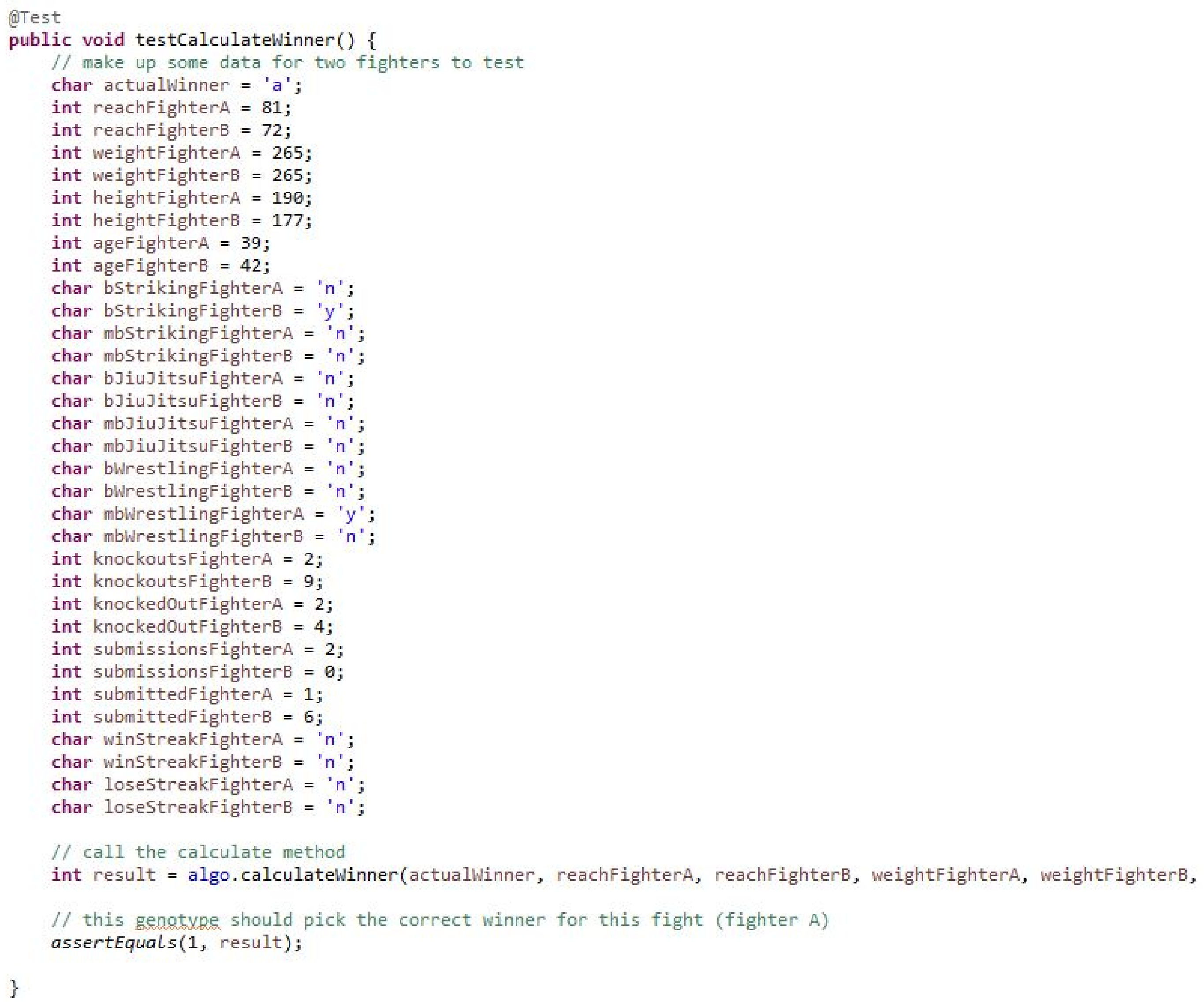
The first class that we unit tested was the BaseClass, which contained all the methods for subtracting and adding amounts from the fighter's total based on data it received from text files. These unit tests were relatively straightforward. We made a genotype and then made variables for each piece of data that could be received. Then, using a BaseClass object, we would assert that the total returned by each method was the correct total. Below is a screenshot of the test we ran on the calcReach method, which passed.



```java
public class BaseClassTest {
    // will be using BaseAlgo19 for this because it contains all methods
    // that are used in any of the other 21 BaseAlgo classes
    BaseAlgo19GA ba;

    // start total of both fighters on 50
    int [] totals = {50, 50};

    // use values from best genotype that genetic algorithm found
    int [] values = {1,3,1,1,5,8,1,8,1,6,2,1,3};

    // before each test make sure we can use ba, the BaseAlgo object
    @Before
    public void setUp() {
        ba = new BaseAlgo19GA();
    }

    @Test
    public void testCalcReach() {
        int reachFighterA = 70;
        int reachFighterB = 75;
        int [] result = ba.calcReach(reachFighterA, reachFighterB, totals, values);

        // ensure that fighter totals are as they should be after method call
        assertEquals(result[0], 49);
        assertEquals(result[1], 51);
    }
```

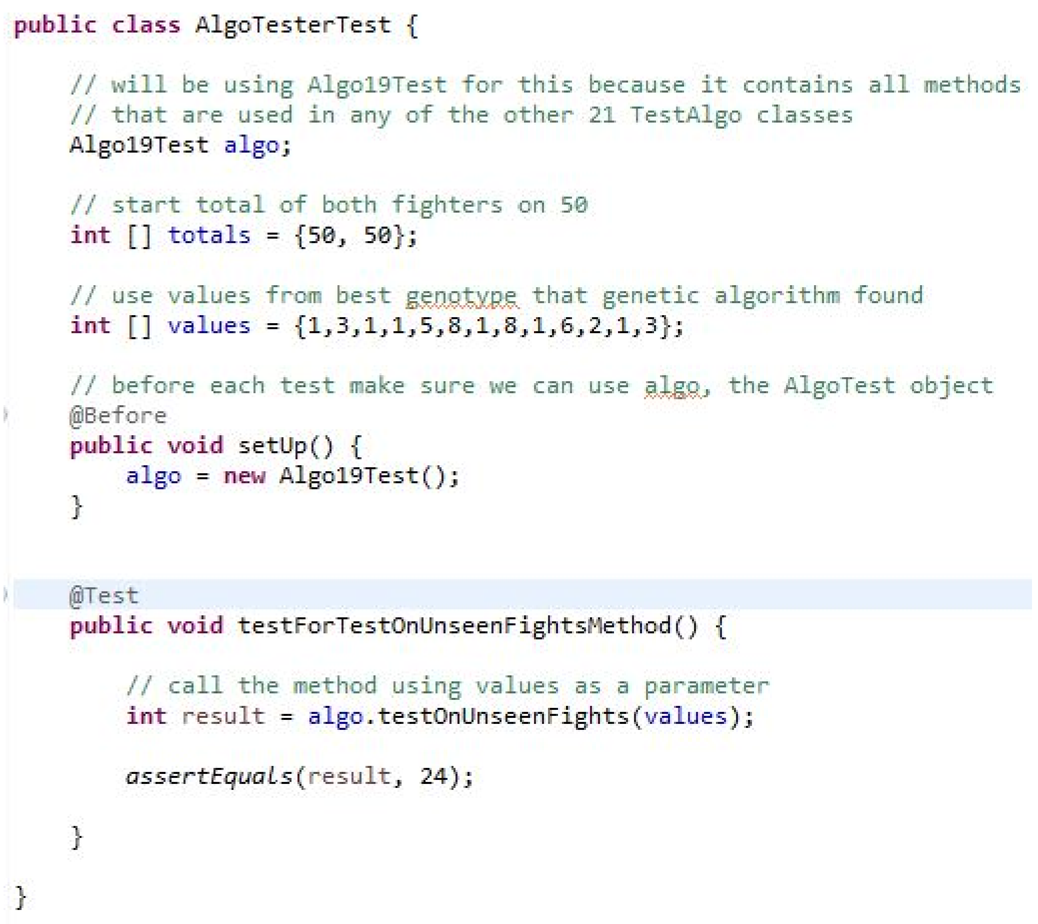
The rest of the tests in this class are much the same as seen above, and they all passed.
The next class we unit tested only contained one method. This method was very important to the

success of the genetic algorithm because it used the genotypes produced by the genetic algorithm to attempt to predict a fight, and returned whether or not it predicted the fight correctly using the values contained in the genotype. So to test this, we used variables to simulate the input coming from a text file. Then coded a genotype into a variable. Using these variables we ran the calculateWinner method and ensured that the method returned a value of 1, indicating that the method predicted the correct winner of the fight. This method can be seen below.



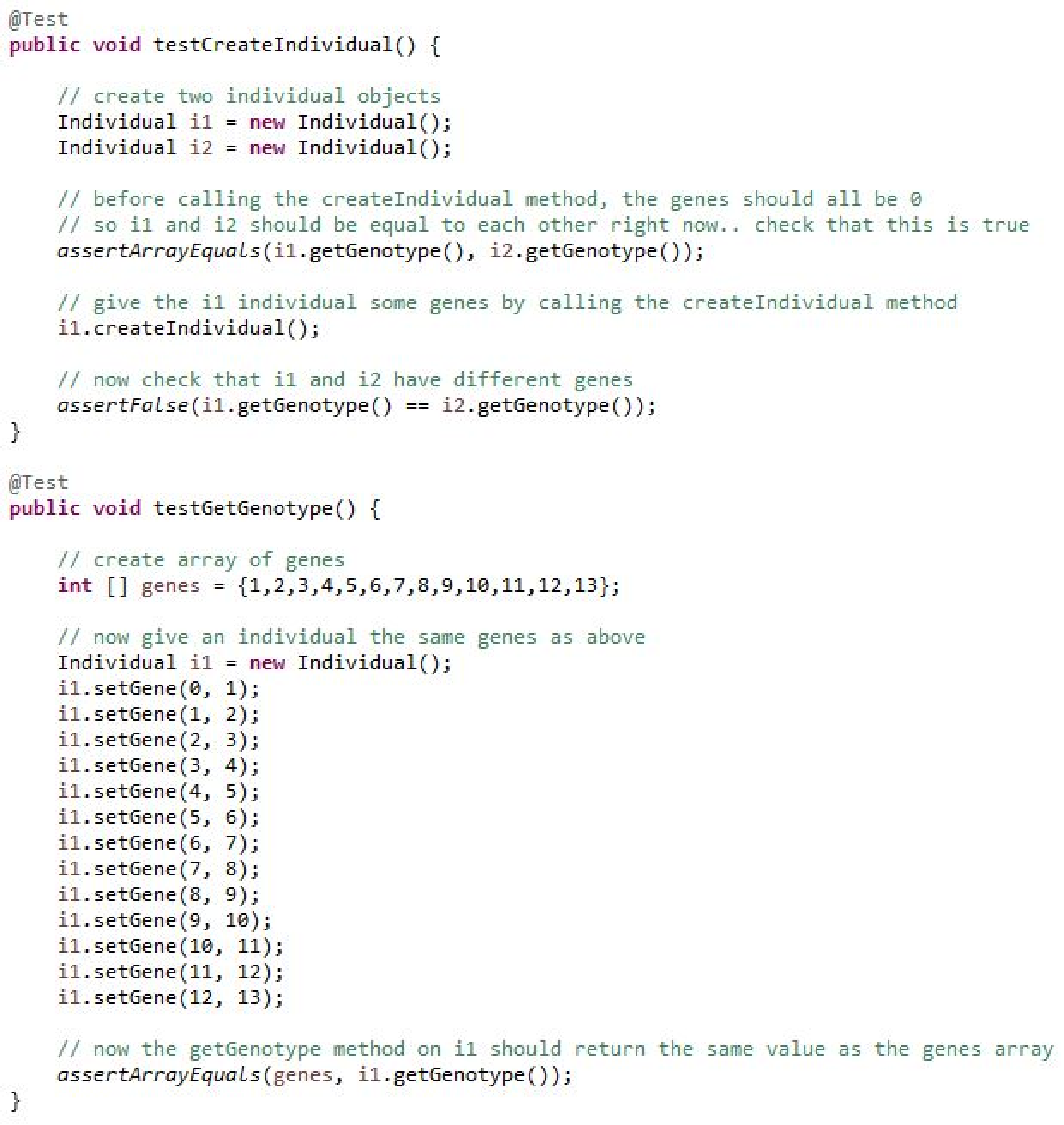
```
@Test
public void testCalculateWinner() {
    // make up some data for two fighters to test
    char actualWinner = 'a';
    int reachFighterA = 81;
    int reachFighterB = 72;
    int weightFighterA = 265;
    int weightFighterB = 265;
    int heightFighterA = 190;
    int heightFighterB = 177;
    int ageFighterA = 39;
    int ageFighterB = 42;
    char bStrikingFighterA = 'n';
    char bStrikingFighterB = 'y';
    char mbStrikingFighterA = 'n';
    char mbStrikingFighterB = 'n';
    char bJiuJitsuFighterA = 'n';
    char bJiuJitsuFighterB = 'n';
    char mbJiuJitsuFighterA = 'n';
    char mbJiuJitsuFighterB = 'n';
    char bWrestlingFighterA = 'n';
    char bWrestlingFighterB = 'n';
    char mbWrestlingFighterA = 'y';
    char mbWrestlingFighterB = 'n';
    int knockoutsFighterA = 2;
    int knockoutsFighterB = 9;
    int knockedOutFighterA = 2;
    int knockedOutFighterB = 4;
    int submissionsFighterA = 2;
    int submissionsFighterB = 0;
    int submittedFighterA = 1;
    int submittedFighterB = 6;
    char winStreakFighterA = 'n';
    char winStreakFighterB = 'n';
    char loseStreakFighterA = 'n';
    char loseStreakFighterB = 'n';

    // call the calculate method
    int result = algo.calculateWinner(actualWinner, reachFighterA, reachFighterB, weightFighterA, weightFighterB,

    // this genotype should pick the correct winner for this fight (fighter A)
    assertEquals(1, result);

}
```

We then unit tested the class which we had been using to test fitness of genotypes against unseen fights, to give us an idea of how well a genotype performed when generalised to fights that it had not learned from. This test only involved writing one method which ran the testOnUnseenFights method using a genotype that we already knew should return a value of 24. As you can see in the following screenshot, we just checked to see it returned the correct value, which it did.



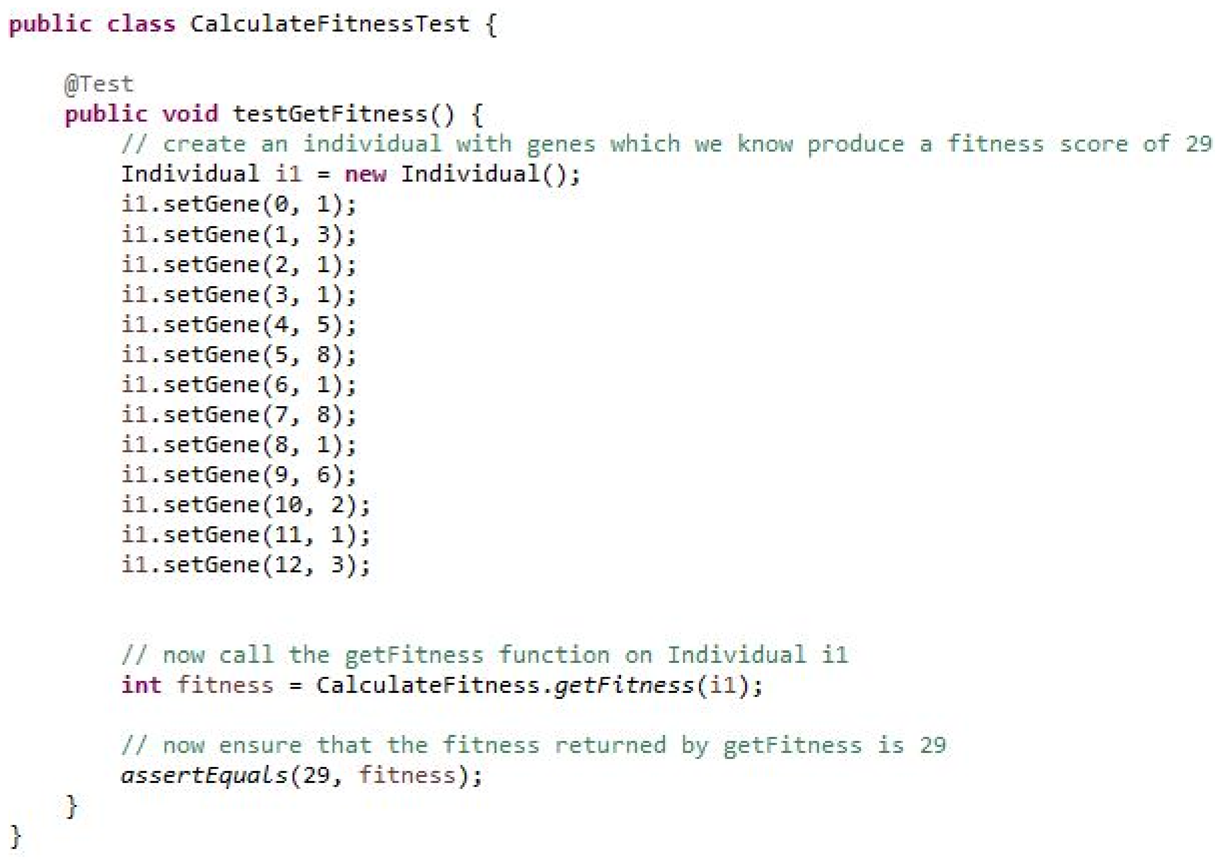
```java
public class AlgoTesterTest {

    // will be using Algo19Test for this because it contains all methods
    // that are used in any of the other 21 TestAlgo classes
    Algo19Test algo;

    // start total of both fighters on 50
    int [] totals = {50, 50};

    // use values from best genotype that genetic algorithm found
    int [] values = {1,3,1,1,5,8,1,8,1,6,2,1,3};

    // before each test make sure we can use algo, the AlgoTest object
    @Before
    public void setUp() {
        algo = new Algo19Test();
    }


    @Test
    public void testForTestOnUnseenFightsMethod() {

        // call the method using values as a parameter
        int result = algo.testOnUnseenFights(values);

        assertEquals(result, 24);

    }

}
```

The next class to be unit tested was the Individual class which basically contained the genes of an individual and methods to retrieve them and manipulate them. The first method we tested was the method to create an individual, we did this by checking an Individual object's genes before and after the call to the createIndividual method – before the call its genes should be empty, but after, the genes should contain values. The rest of the methods in this class were getters and setters, and we tested these by coding the inputs and such using variables and then ensuring that the correct values were retrieved by the methods. Some of these methods can be seen below.



```java
@Test
public void testCreateIndividual() {

    // create two individual objects
    Individual i1 = new Individual();
    Individual i2 = new Individual();

    // before calling the createIndividual method, the genes should all be 0
    // so i1 and i2 should be equal to each other right now.. check that this is true
    assertArrayEquals(i1.getGenotype(), i2.getGenotype());

    // give the i1 individual some genes by calling the createIndividual method
    i1.createIndividual();

    // now check that i1 and i2 have different genes
    assertFalse(i1.getGenotype() == i2.getGenotype());
}

@Test
public void testGetGenotype() {

    // create array of genes
    int [] genes = {1,2,3,4,5,6,7,8,9,10,11,12,13};

    // now give an individual the same genes as above
    Individual i1 = new Individual();
    i1.setGene(0, 1);
    i1.setGene(1, 2);
    i1.setGene(2, 3);
    i1.setGene(3, 4);
    i1.setGene(4, 5);
    i1.setGene(5, 6);
    i1.setGene(6, 7);
    i1.setGene(7, 8);
    i1.setGene(8, 9);
    i1.setGene(9, 10);
    i1.setGene(10, 11);
    i1.setGene(11, 12);
    i1.setGene(12, 13);

    // now the getGenotype method on i1 should return the same value as the genes array
    assertArrayEquals(genes, i1.getGenotype());
}
```

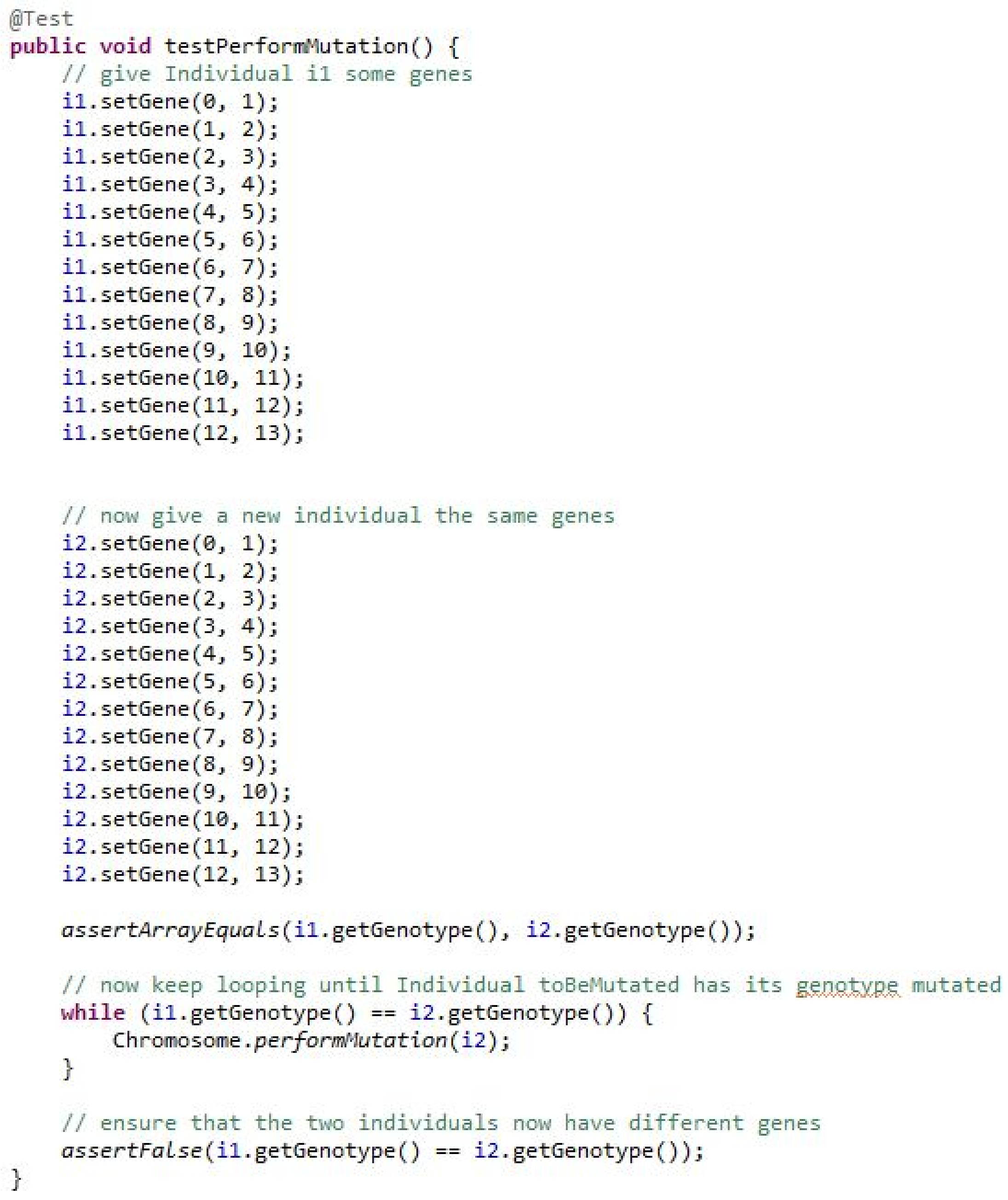
The final class that could be simply unit tested was the CalculateFitness class, which only contained one method that returned the fitness of an individual based on that individual's genotype. Again, we used genes that we knew should return a result of 29.



```
public class CalculateFitnessTest {

    @Test
    public void testGetFitness() {
        // create an individual with genes which we know produce a fitness score of 29
        Individual i1 = new Individual();
        i1.setGene(0, 1);
        i1.setGene(1, 3);
        i1.setGene(2, 1);
        i1.setGene(3, 1);
        i1.setGene(4, 5);
        i1.setGene(5, 8);
        i1.setGene(6, 1);
        i1.setGene(7, 8);
        i1.setGene(8, 1);
        i1.setGene(9, 6);
        i1.setGene(10, 2);
        i1.setGene(11, 1);
        i1.setGene(12, 3);


        // now call the getFitness function on Individual i1
        int fitness = CalculateFitness.getFitness(i1);

        // now ensure that the fitness returned by getFitness is 29
        assertEquals(29, fitness);
    }
}
```

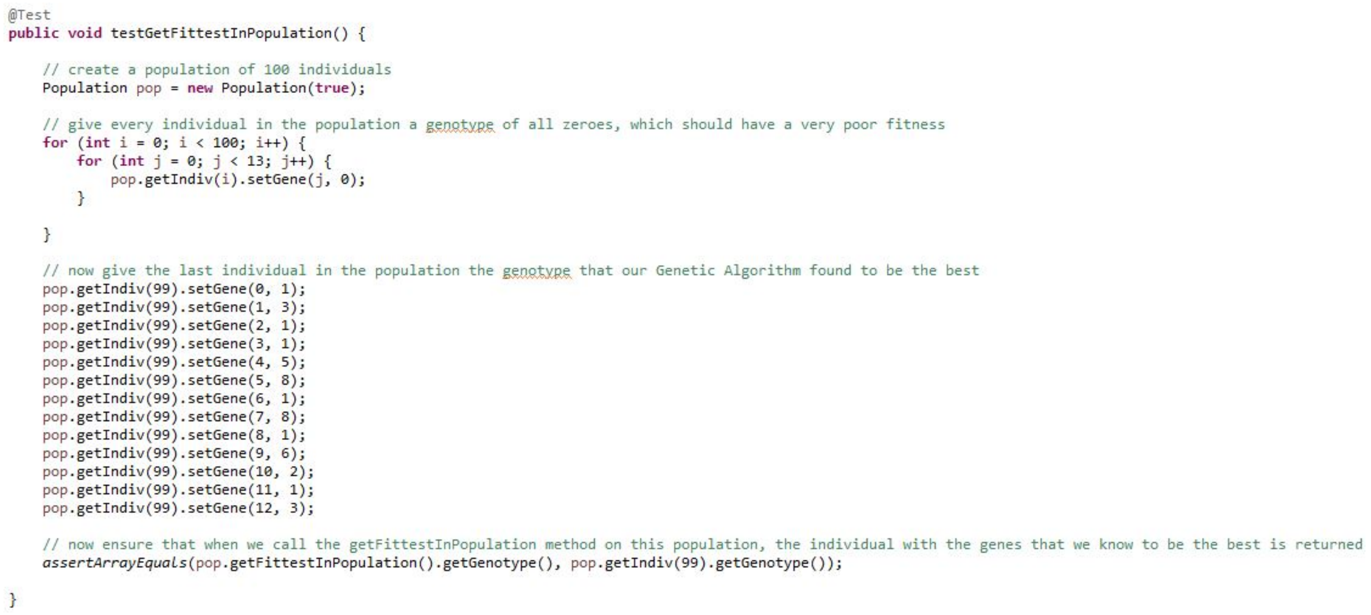
Integration Testing

For the following classes, the lines between unit test and integration test became slightly blurred, because these classes made use of other classes in their methods, so we are going to describe them under the heading of integration tests.
The Chromosome class contained methods to manipulate the genes of Individuals in a Population, using evolutionary methods such as crossover, mutation and natural selection. We tested the performCrossover method by calling it on an empty Individual object and then checking that the object was not empty after the method call. The naturalSelection method was tested by creating a Population object and then calling the naturalSelection method with the Population object passed as a parameter and placing the result into a previously empty Individual object. To see if it worked, we ensured that the Individual object was no longer empty. The performEvolution method was tested in almost the exact same manner as the naturalSelection method, except it returned a Population object instead of an Individual object. The most awkward method to test was the performMutation method, which has a probability of 0.005 of changing a gene of an Individual. To test that this worked, we created two Individuals with identical genotypes. We then tried to mutate the second individual’s genotype by calling the performMutation method. We looped until it changed and then ensured that it had worked by comparing the two genotypes and asserting that they were not equal.



```
@Test
public void testPerformMutation() {
    // give Individual i1 some genes
    i1.setGene(0, 1);
    i1.setGene(1, 2);
    i1.setGene(2, 3);
    i1.setGene(3, 4);
    i1.setGene(4, 5);
    i1.setGene(5, 6);
    i1.setGene(6, 7);
    i1.setGene(7, 8);
    i1.setGene(8, 9);
    i1.setGene(9, 10);
    i1.setGene(10, 11);
    i1.setGene(11, 12);
    i1.setGene(12, 13);


    // now give a new individual the same genes
    i2.setGene(0, 1);
    i2.setGene(1, 2);
    i2.setGene(2, 3);
    i2.setGene(3, 4);
    i2.setGene(4, 5);
    i2.setGene(5, 6);
    i2.setGene(6, 7);
    i2.setGene(7, 8);
    i2.setGene(8, 9);
    i2.setGene(9, 10);
    i2.setGene(10, 11);
    i2.setGene(11, 12);
    i2.setGene(12, 13);

    assertArrayEquals(i1.getGenotype(), i2.getGenotype());

    // now keep looping until Individual toBeMutated has its genotype mutated
    while (i1.getGenotype() == i2.getGenotype()) {
        Chromosome.performMutation(i2);
    }

    // ensure that the two individuals now have different genes
    assertFalse(i1.getGenotype() == i2.getGenotype());
}
```

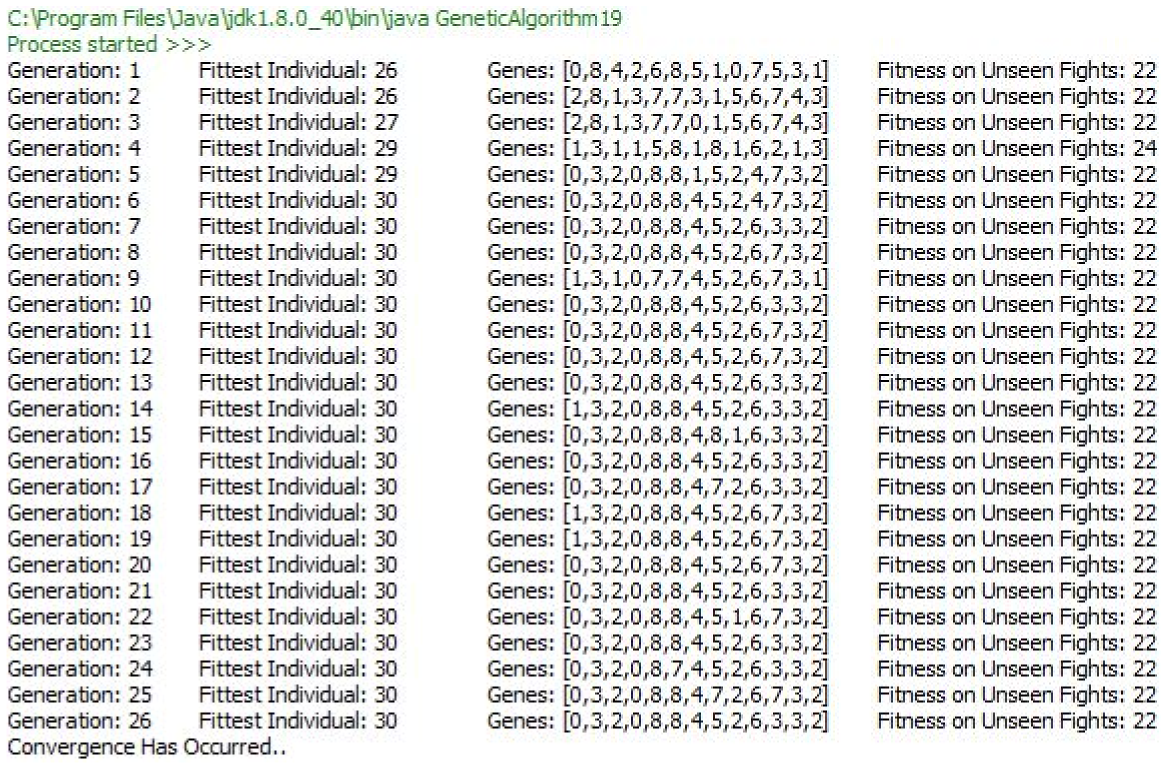
The final class to be tested was the Population class, which is responsible for building and manipulating populations of Individuals. We tested getting and setting of Individual objects in this class in the same manner as before. We also tested the constructors because they took a Boolean as a parameter, which signalled whether or not all the Individuals in the Population should be given random genes or not, because this is needed for the first generation of a genetic algorithm. To test this we called the constructor with false as a parameter and then ensured that attempting to access an individual in the Population threw an exception. Then when we called it with true as a parameter we could access an individual in the Population with no exception being thrown. The most awkward method to test in this class was the getFittestInPopulation method. To test that this worked as expected, we created a population of individuals, all of whom had genes with a value of zero (because this gives a horrible fitness). We then changed one Individual in the population to have genes which we know to have a very high fitness. We then called the method on the population and ensured that it returned the individual with the best genotype as expected. This test can be seen below.



```
@Test
public void testGetFittestInPopulation() {

    // create a population of 100 individuals
    Population pop = new Population(true);

    // give every individual in the population a genotype of all zeroes, which should have a very poor fitness
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 13; j++) {
            pop.getIndiv(i).setGene(j, 0);
        }

    }

    // now give the last individual in the population the genotype that our Genetic Algorithm found to be the best
    pop.getIndiv(99).setGene(0, 1);
    pop.getIndiv(99).setGene(1, 3);
    pop.getIndiv(99).setGene(2, 1);
    pop.getIndiv(99).setGene(3, 1);
    pop.getIndiv(99).setGene(4, 5);
    pop.getIndiv(99).setGene(5, 8);
    pop.getIndiv(99).setGene(6, 1);
    pop.getIndiv(99).setGene(7, 8);
    pop.getIndiv(99).setGene(8, 1);
    pop.getIndiv(99).setGene(9, 6);
    pop.getIndiv(99).setGene(10, 2);
    pop.getIndiv(99).setGene(11, 1);
    pop.getIndiv(99).setGene(12, 3);

    // now ensure that when we call the getFittestInPopulation method on this population, the individual with the genes that we know to be the best is returned
    assertArrayEquals(pop.getFittestInPopulation().getGenotype(), pop.getIndiv(99).getGenotype());

}
```

System Testing

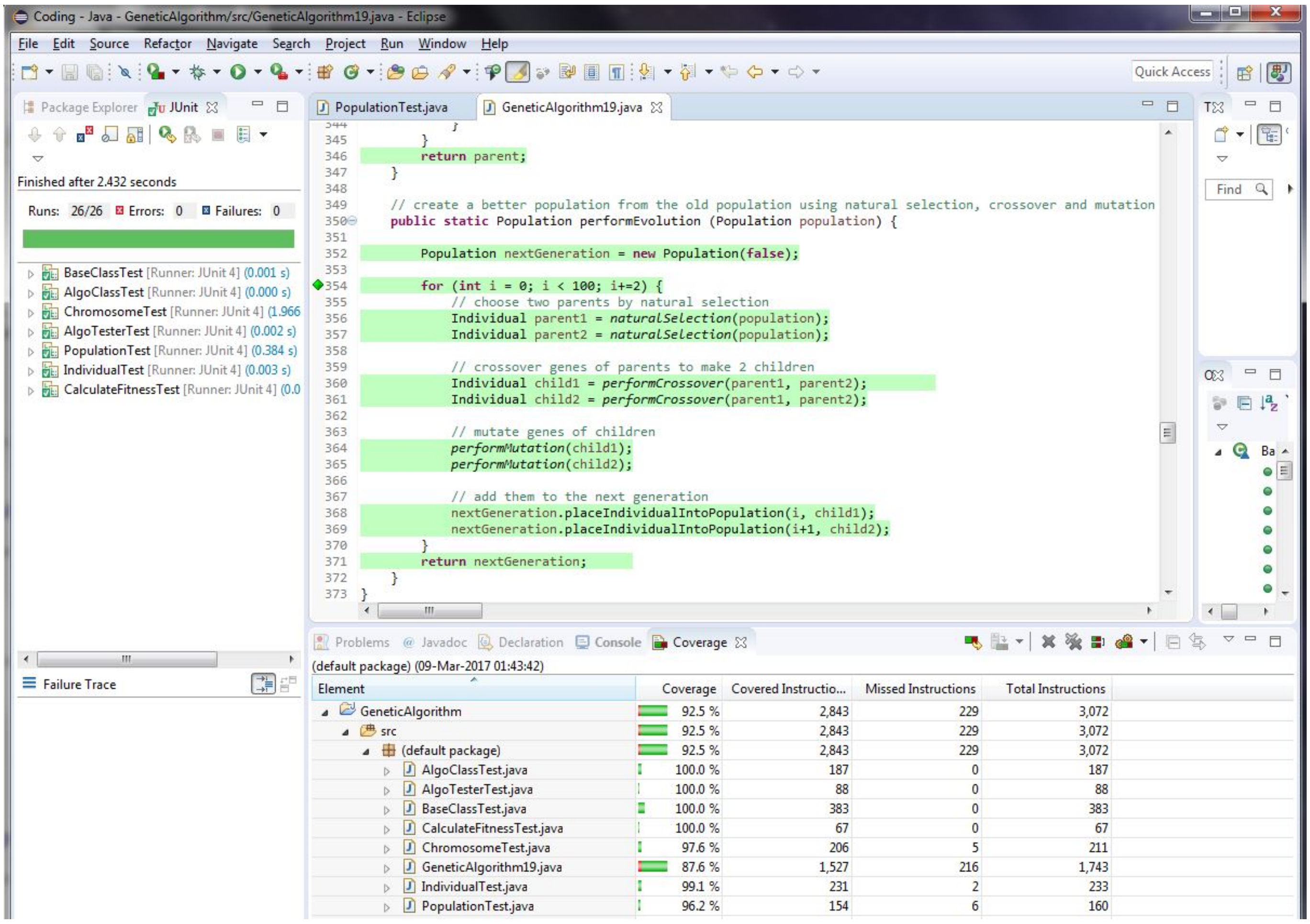
The system test for this genetic algorithm basically consisted of us running it over and over again and seeing that it did indeed evolve and improve in its ability to predict the fight. We wrote a main method which basically created a random population of individuals and then, using mutation, crossover and natural selection, this population was replaced with a new and improved population, which formed the next generation. This cycle continued until the genetic algorithm stopped evolving and the best solution had been found. A screenshot of this process in a terminal can be seen below.



```
C:\Program Files\Java\jdk1.8.0_40\bin\java GeneticAlgorithm19
Process started >>>
Generation: 1     Fittest Individual: 26     Genes: [0,8,4,2,6,8,5,1,0,7,5,3,1]     Fitness on Unseen Fights: 22
Generation: 2     Fittest Individual: 26     Genes: [2,8,1,3,7,7,3,1,5,6,7,4,3]     Fitness on Unseen Fights: 22
Generation: 3     Fittest Individual: 27     Genes: [2,8,1,3,7,7,0,1,5,6,7,4,3]     Fitness on Unseen Fights: 22
Generation: 4     Fittest Individual: 29     Genes: [1,3,1,1,5,8,1,8,1,6,2,1,3]     Fitness on Unseen Fights: 24
Generation: 5     Fittest Individual: 29     Genes: [0,3,2,0,8,8,1,5,2,4,7,3,2]     Fitness on Unseen Fights: 22
Generation: 6     Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,4,7,3,2]     Fitness on Unseen Fights: 22
Generation: 7     Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 8     Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 9     Fittest Individual: 30     Genes: [1,3,1,0,7,7,4,5,2,6,7,3,1]     Fitness on Unseen Fights: 22
Generation: 10    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 11    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 12    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 13    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 14    Fittest Individual: 30     Genes: [1,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 15    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,8,1,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 16    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 17    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,7,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 18    Fittest Individual: 30     Genes: [1,3,2,0,8,8,4,5,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 19    Fittest Individual: 30     Genes: [1,3,2,0,8,8,4,5,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 20    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 21    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 22    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,1,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 23    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 24    Fittest Individual: 30     Genes: [0,3,2,0,8,7,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Generation: 25    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,7,2,6,7,3,2]     Fitness on Unseen Fights: 22
Generation: 26    Fittest Individual: 30     Genes: [0,3,2,0,8,8,4,5,2,6,3,3,2]     Fitness on Unseen Fights: 22
Convergence Has Occurred..
```

Results

Using a tool for Eclipse known as EclEmma, we were able to measure the coverage of the genetic algorithm after performing all of the tests that we had written. We achieved a total coverage score of 92.5%. We would have been closer to 100% if we had been able to write some JUnit tests for the main method of the genetic algorithm, however, this is very awkward, and rather unnecessary. The main method consisted purely of methods that we have fully tested as well as some print statements, and the fact that it evolves when run and becomes better at predicting fights as it runs, is proof enough that the main method is doing everything it should be. A screenshot showing our code coverage results can be seen below.


Coding - Java - GeneticAlgorithm/src/GeneticAlgorithm19.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
JUnit
PopulationTest.java
GeneticAlgorithm19.java
Problems
Javadoc
Declaration
Console
Coverage
Failure Trace

Finished after 2.432 seconds

Runs: 26/26 Errors: 0 Failures: 0

```
BaseClassTest [Runner: JUnit 4] (0.001 s)
AlgoClassTest [Runner: JUnit 4] (0.000 s)
ChromosomeTest [Runner: JUnit 4] (1.966
AlgoTesterTest [Runner: JUnit 4] (0.002 s)
PopulationTest [Runner: JUnit 4] (0.384 s)
IndividualTest [Runner: JUnit 4] (0.003 s)
CalculateFitnessTest [Runner: JUnit 4] (0.0
```

```
            }
        }
        return parent;
    }

    // create a better population from the old population using natural selection, crossover and mutation
    public static Population performEvolution (Population population) {

        Population nextGeneration = new Population(false);

        for (int i = 0; i < 100; i+=2) {
            // choose two parents by natural selection
            Individual parent1 = naturalSelection(population);
            Individual parent2 = naturalSelection(population);

            // crossover genes of parents to make 2 children
            Individual child1 = performCrossover(parent1, parent2);
            Individual child2 = performCrossover(parent1, parent2);

            // mutate genes of children
            performMutation(child1);
            performMutation(child2);

            // add them to the next generation
            nextGeneration.placeIndividualIntoPopulation(i, child1);
            nextGeneration.placeIndividualIntoPopulation(i+1, child2);
        }
        return nextGeneration;
    }
}
```

(default package) (09-Mar-2017 01:43:42)

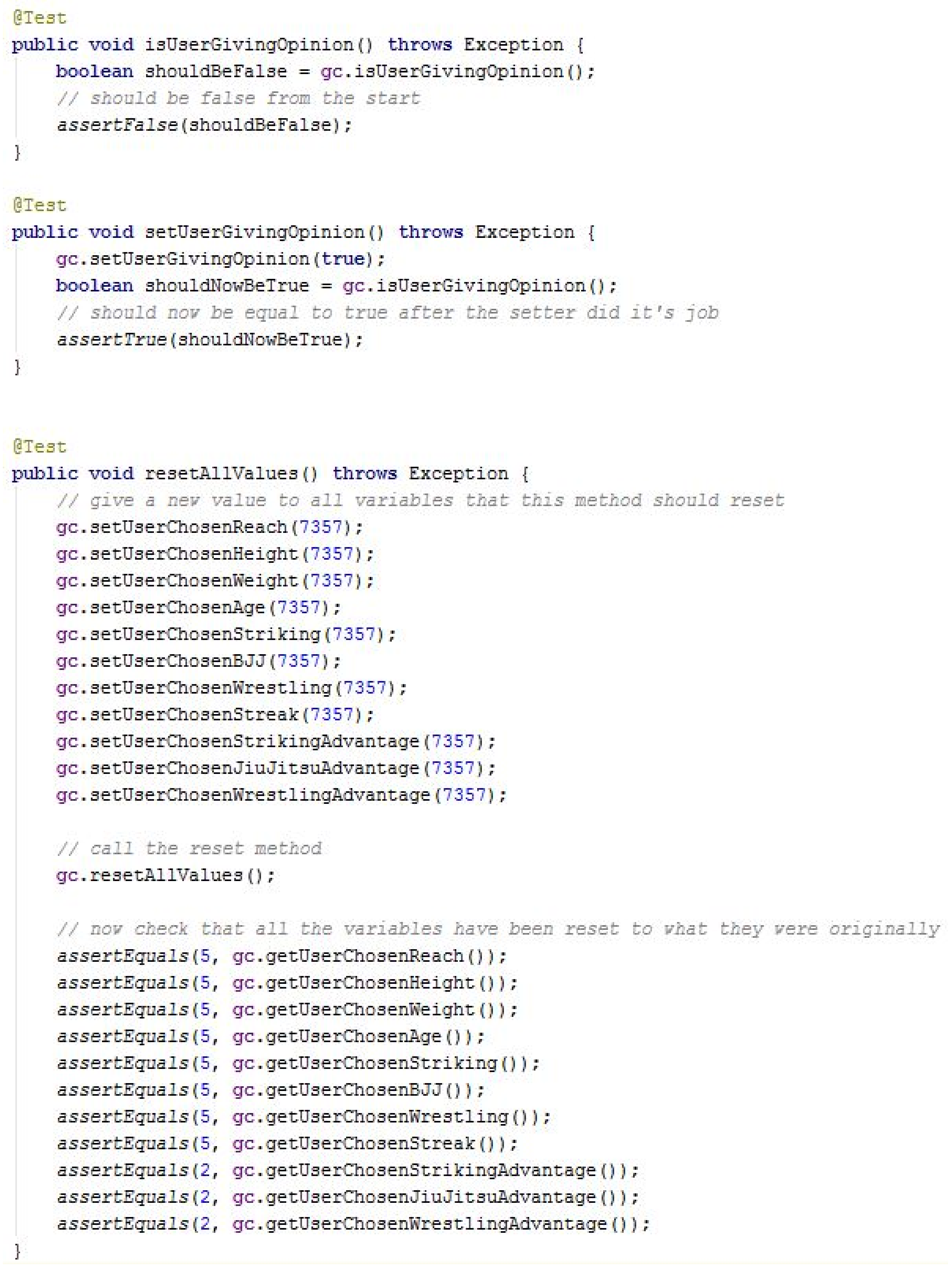
Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
GeneticAlgorithm	92.5 %	2,843	229	3,072
src	92.5 %	2,843	229	3,072
(default package)	92.5 %	2,843	229	3,072
AlgoClassTest.java	100.0 %	187	0	187
AlgoTesterTest.java	100.0 %	88	0	88
BaseClassTest.java	100.0 %	383	0	383
CalculateFitnessTest.java	100.0 %	67	0	67
ChromosomeTest.java	97.6 %	206	5	211
GeneticAlgorithm19.java	87.6 %	1,527	216	1,743
IndividualTest.java	99.1 %	231	2	233
PopulationTest.java	96.2 %	154	6	160

Android Application Testing

To test the code we wrote for our Android Application, we used unit tests and instrumentation tests. Unit tests, in terms of Android applications, are tests that can be run locally, without needing a device or emulator. Whereas, instrumentation tests are tests which require a device or emulator in order to be performed. These tests mainly focus on the User Interface and actual functionality of the Application itself. Instrumentation tests can be used to describe both integration tests as well as unit tests when it comes to Android Applications, which makes some of these tests quite difficult to categorise. So for the purpose of this write-up, the tests described under the heading of unit tests, solely consist of unit tests, however, the tests described under the heading of instrumentation tests can consist of both unit tests and integration tests.
Unit tests were carried out using JUnit and instrumentation tests were carried out using a combination of Espresso and JUnit.

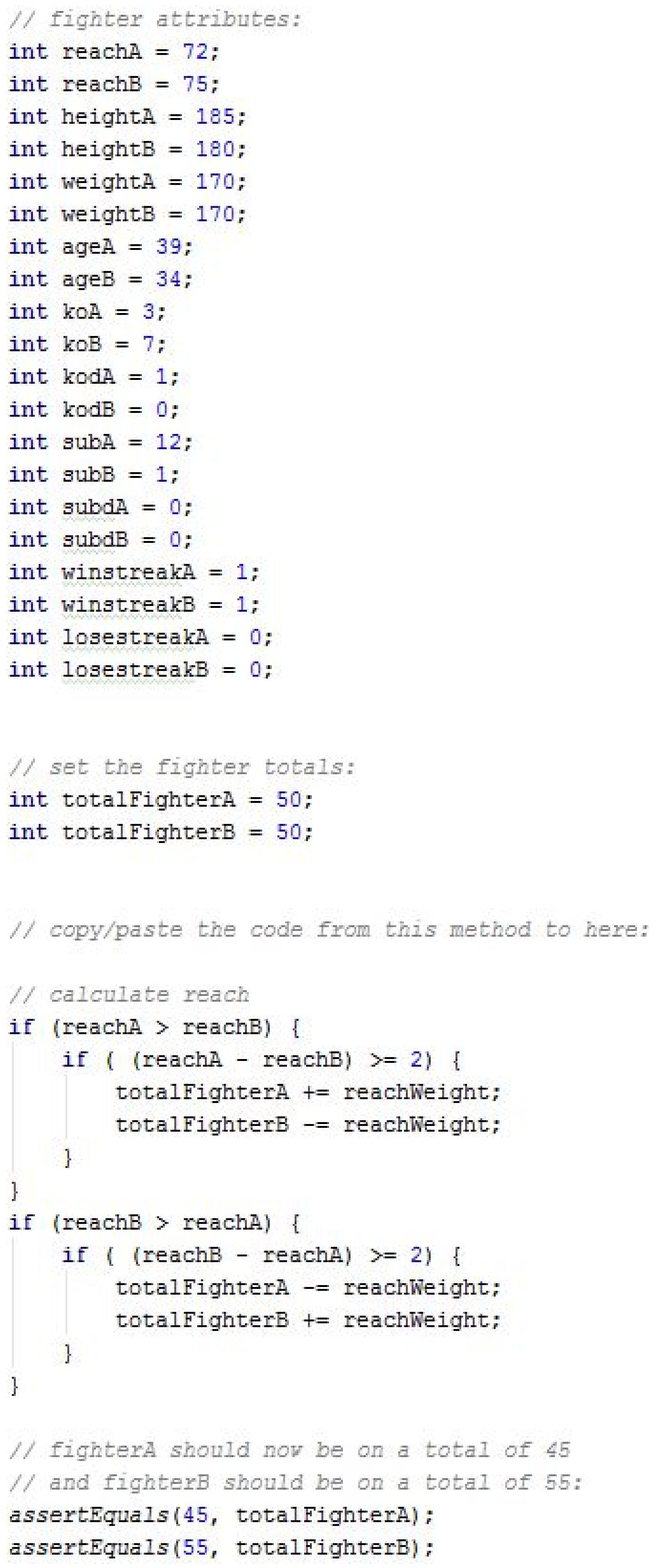
Unit Testing

It was only possible to properly unit test three of the classes in the Application. The first of which was the GlobalClass, which we used to store variables that needed to be passed between different screens on the application using getters and setters. The tests for these are rather self-explanatory, basically just test the getter by retrieving a value and asserting that it is the intended value, and test the setter by changing the value using the setter and then asserting that the value has changed. The only other method in this class was resetAllValues, which we used to reset values that may have been manipulated during a previous run of the application. This essentially tidied up everything before a new run of the application began. We tested this by manually changing all these values, then calling the method, and ensuring that the values had indeed reset. Some of these tests can be seen below.



```java
@Test
public void isUserGivingOpinion() throws Exception {
    boolean shouldBeFalse = gc.isUserGivingOpinion();
    // should be false from the start
    assertFalse(shouldBeFalse);
}

@Test
public void setUserGivingOpinion() throws Exception {
    gc.setUserGivingOpinion(true);
    boolean shouldNowBeTrue = gc.isUserGivingOpinion();
    // should now be equal to true after the setter did it's job
    assertTrue(shouldNowBeTrue);
}


@Test
public void resetAllValues() throws Exception {
    // give a new value to all variables that this method should reset
    gc.setUserChosenReach(7357);
    gc.setUserChosenHeight(7357);
    gc.setUserChosenWeight(7357);
    gc.setUserChosenAge(7357);
    gc.setUserChosenStriking(7357);
    gc.setUserChosenBJJ(7357);
    gc.setUserChosenWrestling(7357);
    gc.setUserChosenStreak(7357);
    gc.setUserChosenStrikingAdvantage(7357);
    gc.setUserChosenJiuJitsuAdvantage(7357);
    gc.setUserChosenWrestlingAdvantage(7357);

    // call the reset method
    gc.resetAllValues();

    // now check that all the variables have been reset to what they were originally
    assertEquals(5, gc.getUserChosenReach());
    assertEquals(5, gc.getUserChosenHeight());
    assertEquals(5, gc.getUserChosenWeight());
    assertEquals(5, gc.getUserChosenAge());
    assertEquals(5, gc.getUserChosenStriking());
    assertEquals(5, gc.getUserChosenBJJ());
    assertEquals(5, gc.getUserChosenWrestling());
    assertEquals(5, gc.getUserChosenStreak());
    assertEquals(2, gc.getUserChosenStrikingAdvantage());
    assertEquals(2, gc.getUserChosenJiuJitsuAdvantage());
    assertEquals(2, gc.getUserChosenWrestlingAdvantage());
}
```

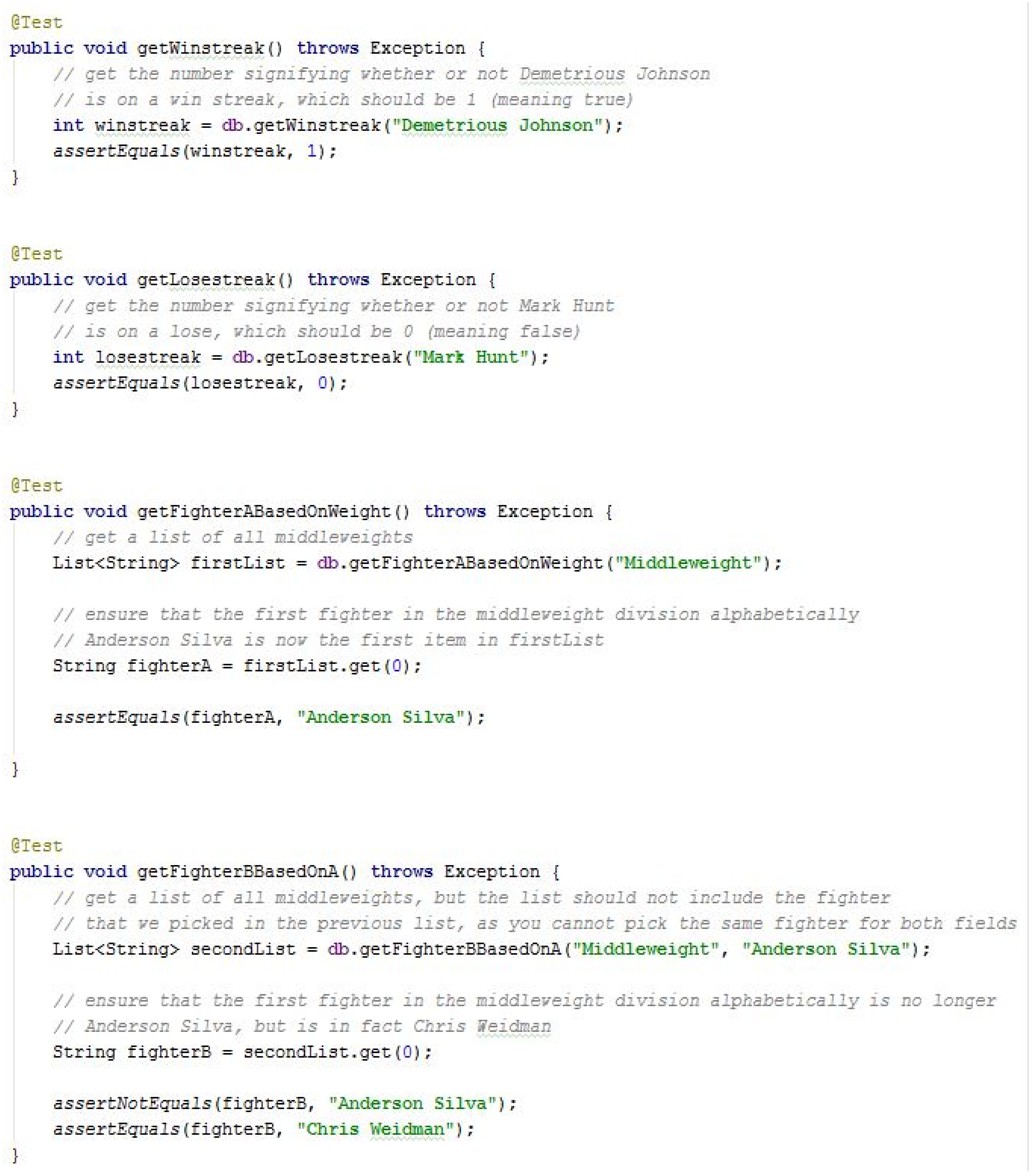
The next two classes that we did unit tests on were the ResultScreen classes that contained the methods for actually predicting a winner of the fight based on the information provided by the user. These methods were similar to some of those used in the genetic algorithm so the tests were relatively similar too. We tested the calculate winner method for when the user chose to give opinion, when the user chose to use data only and for when the user chose to build their own algorithm. The tests for these methods were similar in nature. We manually entered data to simulate the user going through the application in order to receive a prediction. We then ran the method to see if it gave the result that it should have given. In all three cases it did. A screenshot from one of the tests is below.



```java
// fighter attributes:
int reachA = 72;
int reachB = 75;
int heightA = 185;
int heightB = 180;
int weightA = 170;
int weightB = 170;
int ageA = 39;
int ageB = 34;
int koA = 3;
int koB = 7;
int kodA = 1;
int kodB = 0;
int subA = 12;
int subB = 1;
int subdA = 0;
int subdB = 0;
int winstreakA = 1;
int winstreakB = 1;
int losestreakA = 0;
int losestreakB = 0;


// set the fighter totals:
int totalFighterA = 50;
int totalFighterB = 50;


// copy/paste the code from this method to here:

// calculate reach
if (reachA > reachB) {
    if ( (reachA - reachB) >= 2) {
        totalFighterA += reachWeight;
        totalFighterB -= reachWeight;
    }
}
if (reachB > reachA) {
    if ( (reachB - reachA) >= 2) {
        totalFighterA -= reachWeight;
        totalFighterB += reachWeight;
    }
}

// fighterA should now be on a total of 45
// and fighterB should be on a total of 55:
assertEquals(45, totalFighterA);
assertEquals(55, totalFighterB);
```

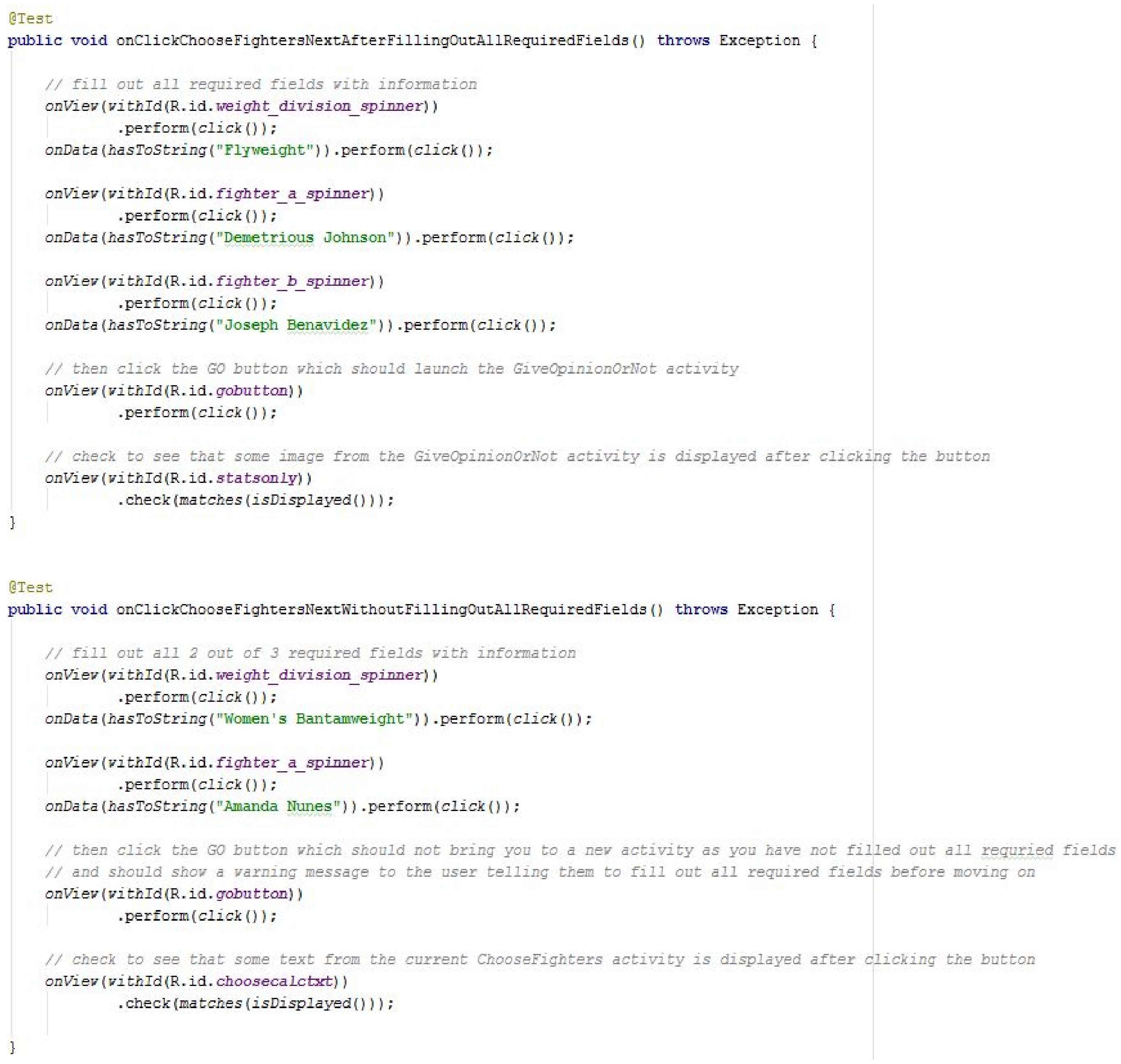
Instrumentation Testing

The first class which required instrumentation testing was the DatabaseHelper class, because in order to carry out tests on the database, the database first needed to be installed on a device or an emulator. Before each test, we created a database on the testing device, and after each test we would close the database for safety. Since our database was already created, all the methods in this class are basically getters. So to test them, we would query the database at a known location and ensure that the value it retrieved was the correct value. We also had two methods to build the lists for the drop-down menu of fighter names. The first of these lists was built based on the weight division selected by the user, and the second list consisted of the same names excluding the name that the user already selected. So we tested that these methods functioned appropriately by choosing a fighter from the list and then ensuring that that fighter did not appear in the second list. Some of these methods can be seen below.



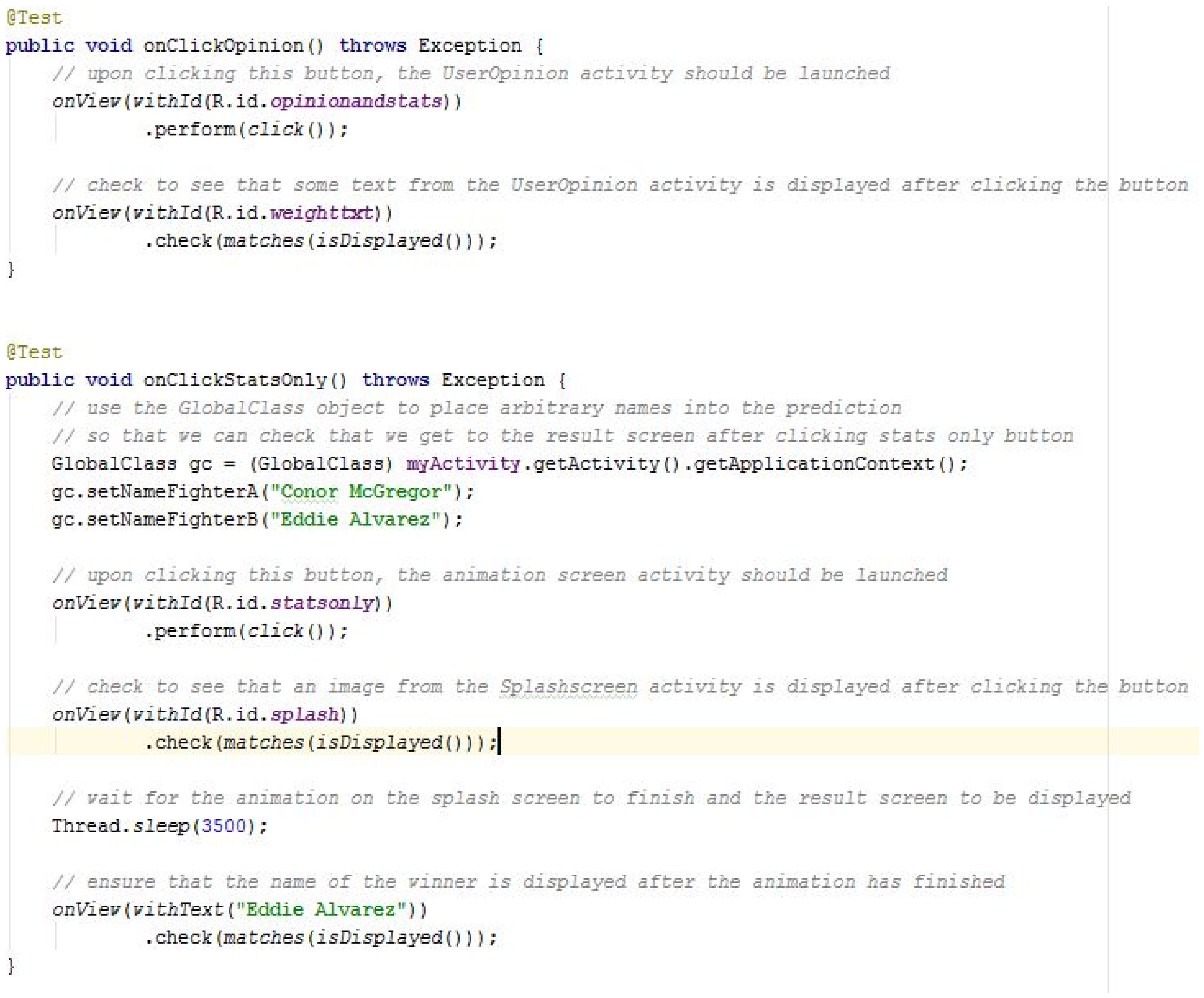
```
@Test
public void getWinstreak() throws Exception {
    // get the number signifying whether or not Demetrious Johnson
    // is on a win streak, which should be 1 (meaning true)
    int winstreak = db.getWinstreak("Demetrious Johnson");
    assertEquals(winstreak, 1);
}


@Test
public void getLosestreak() throws Exception {
    // get the number signifying whether or not Mark Hunt
    // is on a lose, which should be 0 (meaning false)
    int losestreak = db.getLosestreak("Mark Hunt");
    assertEquals(losestreak, 0);
}


@Test
public void getFighterABasedOnWeight() throws Exception {
    // get a list of all middleweights
    List<String> firstList = db.getFighterABasedOnWeight("Middleweight");

    // ensure that the first fighter in the middleweight division alphabetically
    // Anderson Silva is now the first item in firstList
    String fighterA = firstList.get(0);

    assertEquals(fighterA, "Anderson Silva");

}


@Test
public void getFighterBBasedOnA() throws Exception {
    // get a list of all middleweights, but the list should not include the fighter
    // that we picked in the previous list, as you cannot pick the same fighter for both fields
    List<String> secondList = db.getFighterBBasedOnA("Middleweight", "Anderson Silva");

    // ensure that the first fighter in the middleweight division alphabetically is no longer
    // Anderson Silva, but is in fact Chris Weidman
    String fighterB = secondList.get(0);

    assertNotEquals(fighterB, "Anderson Silva");
    assertEquals(fighterB, "Chris Weidman");
}
```

The ChooseFighters class was another class which we performed instrumentation testing on. It consisted of drop-down menus that allow the user to select fighters based on a weight division of their choosing. We tested onCreate methods by ensuring that appropriate information was displayed on the screen when it was launched. We also tested that when the user selects a weight division, a fighter from the appropriate weight division is part of the list that is displayed to the user on the next drop-down menu. We also tested that if the user attempts to move on to the next screen before filling out all required fields, they will not be able to, and will be provided with a message asking them to fill out all required fields before moving on. Some of these methods are shown below.



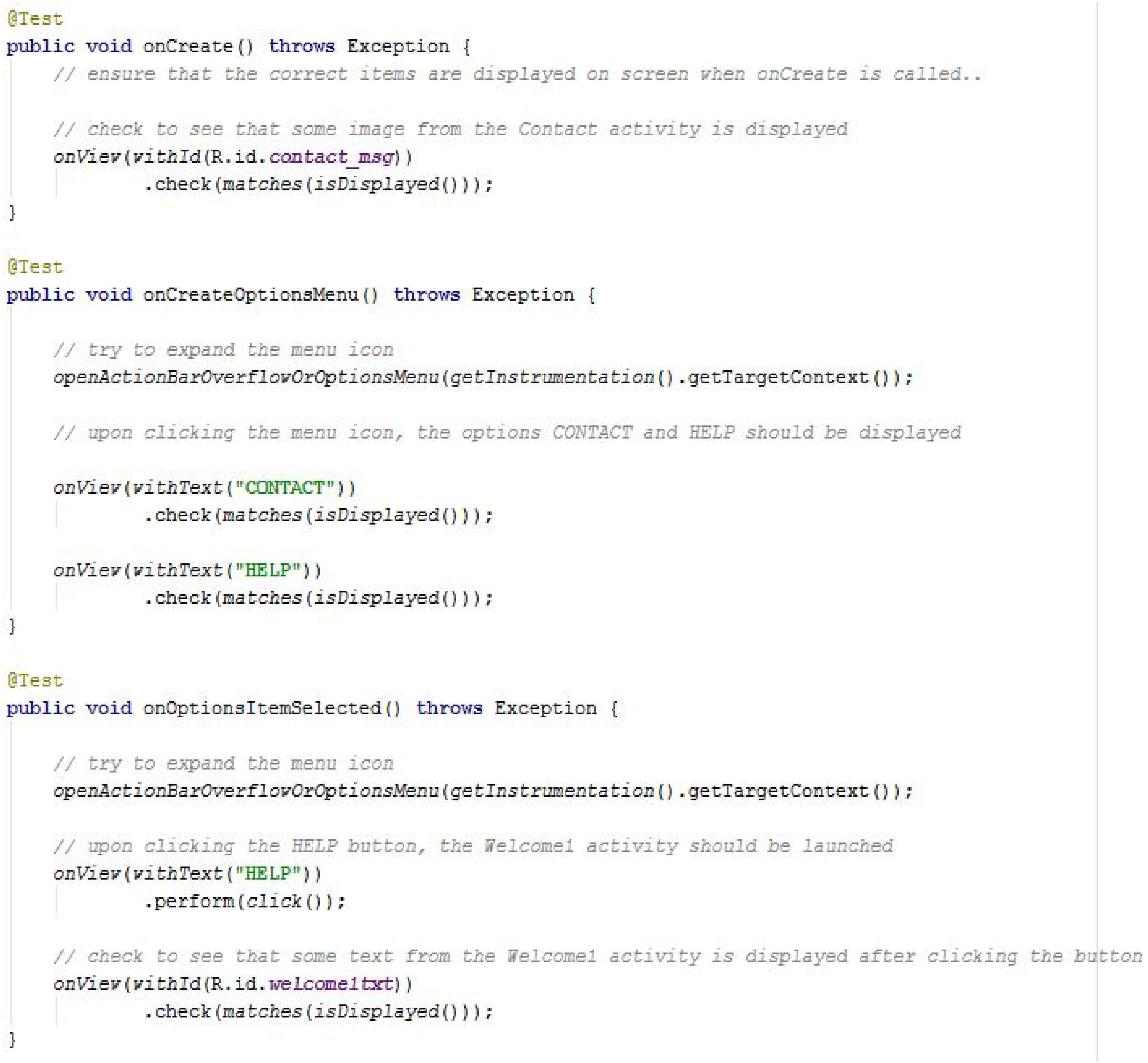
```java
@Test
public void onClickChooseFightersNextAfterFillingOutAllRequiredFields() throws Exception {

    // fill out all required fields with information
    onView(withId(R.id.weight_division_spinner))
            .perform(click());
    onData(hasToString("Flyweight")).perform(click());

    onView(withId(R.id.fighter_a_spinner))
            .perform(click());
    onData(hasToString("Demetrious Johnson")).perform(click());

    onView(withId(R.id.fighter_b_spinner))
            .perform(click());
    onData(hasToString("Joseph Benavidez")).perform(click());

    // then click the GO button which should launch the GiveOpinionOrNot activity
    onView(withId(R.id.gobutton))
            .perform(click());

    // check to see that some image from the GiveOpinionOrNot activity is displayed after clicking the button
    onView(withId(R.id.statsonly))
            .check(matches(isDisplayed()));
}


@Test
public void onClickChooseFightersNextWithoutFillingOutAllRequiredFields() throws Exception {

    // fill out all 2 out of 3 required fields with information
    onView(withId(R.id.weight_division_spinner))
            .perform(click());
    onData(hasToString("Women's Bantamweight")).perform(click());

    onView(withId(R.id.fighter_a_spinner))
            .perform(click());
    onData(hasToString("Amanda Nunes")).perform(click());

    // then click the GO button which should not bring you to a new activity as you have not filled out all requried fields
    // and should show a warning message to the user telling them to fill out all required fields before moving on
    onView(withId(R.id.gobutton))
            .perform(click());

    // check to see that some text from the current ChooseFighters activity is displayed after clicking the button
    onView(withId(R.id.choosecalctxt))
            .check(matches(isDisplayed()));

}
```

Another class which we did instrumentation testing on was the GiveOpinionOrNot class which allowed the user to choose what kind of prediction they wanted. If they decided to use statistics only, it would launch a SplashScreen animation and then display a result. To test this, we used GlobalClass to manually set the names of two fighters, to simulate the user having selected them in the previous screen. We then checked that once the animation had ended, the correct fighter was displayed as the winner to the user. The user could also choose to give their opinion as part of the prediction, in which case, the UserOpinion activity was launched. To test that this worked, we simulated the user pressing the button to give their opinion (using Espresso methods) and then ensured that some text from the appropriate activity was now being displayed to the user. Some of these methods can be seen in this screenshot.



```
@Test
public void onClickOpinion() throws Exception {
    // upon clicking this button, the UserOpinion activity should be launched
    onView(withId(R.id.opinionandstats))
            .perform(click());

    // check to see that some text from the UserOpinion activity is displayed after clicking the button
    onView(withId(R.id.weighttxt))
            .check(matches(isDisplayed()));
}


@Test
public void onClickStatsOnly() throws Exception {
    // use the GlobalClass object to place arbitrary names into the prediction
    // so that we can check that we get to the result screen after clicking stats only button
    GlobalClass gc = (GlobalClass) myActivity.getActivity().getApplicationContext();
    gc.setNameFighterA("Conor McGregor");
    gc.setNameFighterB("Eddie Alvarez");

    // upon clicking this button, the animation screen activity should be launched
    onView(withId(R.id.statsonly))
            .perform(click());

    // check to see that an image from the Splashscreen activity is displayed after clicking the button
    onView(withId(R.id.splash))
            .check(matches(isDisplayed()));

    // wait for the animation on the splash screen to finish and the result screen to be displayed
    Thread.sleep(3500);

    // ensure that the name of the winner is displayed after the animation has finished
    onView(withText("Eddie Alvarez"))
            .check(matches(isDisplayed()));
}
```

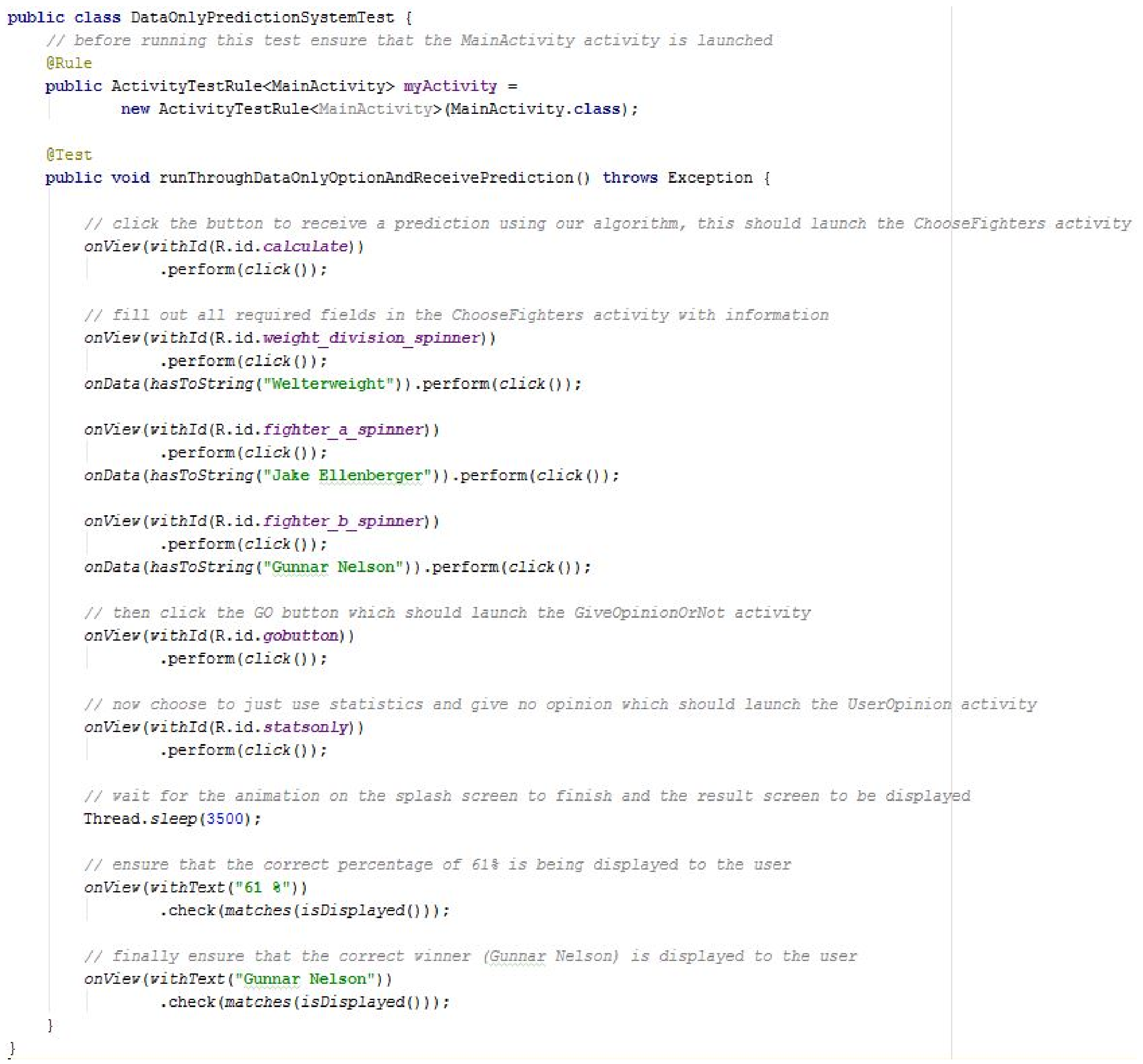
The rest of the classes were mainly tested for functionality. So we used Espresso methods to simulate the user clicking all the buttons that they could on each screen and then ensuring that the correct action was performed upon the button being clicked. We also tested that the onCreate methods of these classes worked appropriately by ensuring that a piece of text or an image from the chosen screen was, in fact, being displayed to the user upon launching it. We also tested the user clicking the back buttons, either the back button on the actual device, or the back button that is on the top left of the screen. These tests were carried out on all classes and some of them can be seen below.



```
@Test
public void onCreate() throws Exception {
    // ensure that the correct items are displayed on screen when onCreate is called..

    // check to see that some image from the Contact activity is displayed
    onView(withId(R.id.contact_msg))
            .check(matches(isDisplayed()));
}

@Test
public void onCreateOptionsMenu() throws Exception {

    // try to expand the menu icon
    openActionBarOverflowOrOptionsMenu(getInstrumentation().getTargetContext());

    // upon clicking the menu icon, the options CONTACT and HELP should be displayed

    onView(withText("CONTACT"))
            .check(matches(isDisplayed()));

    onView(withText("HELP"))
            .check(matches(isDisplayed()));
}

@Test
public void onOptionsItemSelected() throws Exception {

    // try to expand the menu icon
    openActionBarOverflowOrOptionsMenu(getInstrumentation().getTargetContext());

    // upon clicking the HELP button, the Welcome1 activity should be launched
    onView(withText("HELP"))
            .perform(click());

    // check to see that some text from the Welcome1 activity is displayed after clicking the button
    onView(withId(R.id.welcome1txt))
            .check(matches(isDisplayed()));
}
```

System Testing

We wrote 3 separate system tests using Espresso for our Android Application. These system tests simulate a user going from the home screen all the way through to receiving a prediction. Each system test simulated the user choosing a different path through the app, by either creating their own algorithm, using our algorithm with only data, or using our algorithm as well as providing their own opinion. During these system tests we would ensure that the user was being shown the correct information at all times, and that the correct result was provided to the user on the final result screen based on the information that they had entered. A screenshot of one of these tests is provided below. The other two system tests are very similar to this one shown here.



```java
public class DataOnlyPredictionSystemTest {
    // before running this test ensure that the MainActivity activity is launched
    @Rule
    public ActivityTestRule<MainActivity> myActivity =
            new ActivityTestRule<MainActivity>(MainActivity.class);

    @Test
    public void runThroughDataOnlyOptionAndReceivePrediction() throws Exception {

        // click the button to receive a prediction using our algorithm, this should launch the ChooseFighters activity
        onView(withId(R.id.calculate))
                .perform(click());

        // fill out all required fields in the ChooseFighters activity with information
        onView(withId(R.id.weight_division_spinner))
                .perform(click());
        onData(hasToString("Welterweight")).perform(click());

        onView(withId(R.id.fighter_a_spinner))
                .perform(click());
        onData(hasToString("Jake Ellenberger")).perform(click());

        onView(withId(R.id.fighter_b_spinner))
                .perform(click());
        onData(hasToString("Gunnar Nelson")).perform(click());

        // then click the GO button which should launch the GiveOpinionOrNot activity
        onView(withId(R.id.gobutton))
                .perform(click());

        // now choose to just use statistics and give no opinion which should launch the UserOpinion activity
        onView(withId(R.id.statsonly))
                .perform(click());

        // wait for the animation on the splash screen to finish and the result screen to be displayed
        Thread.sleep(3500);

        // ensure that the correct percentage of 61% is being displayed to the user
        onView(withText("61 %"))
                .check(matches(isDisplayed()));

        // finally ensure that the correct winner (Gunnar Nelson) is displayed to the user
        onView(withText("Gunnar Nelson"))
                .check(matches(isDisplayed()));
    }
}
```

Results

It is somewhat harder to get a good measure of coverage when it comes to Android Applications, because of the need to do instrumentation tests to carry out most of the testing. However, we feel that having written 83 tests, consisting of unit tests, integration tests as well as system tests, that we have tested our Android Application to a very high level. We have the application on our phones for over a week now and have been trying to find problems with it, but so far we have had no crashes and have noticed no bugs.
As far as testing the Application as an actual product, we used it last week to predict 10 real fights that took place in an organisation known as the Ultimate Fighting Championship. We then placed a 50 cent bet on each fighter that it predicted would be the winner. At the end of the fights, our Application had predicted 7 out of a possible 10 fights correctly, and one of the fights that it predicted incorrectly is being investigated on the basis that the judges may have made a wrong decision. So the application was very close to predicting 8 out of 10 fights correctly. It also picked 3 of the biggest underdogs on the card to win, and they all did. We placed a total of 5 euro in bets

and ended up making 2.27 euro in profit, which is just over 45% profit. Considering that we used only data, and absolutely no opinion, we were very pleased with these preliminary results, and believe that our application has very good potential as a product.