COMP23420: Design for Testability

Introduction

Background

I want to create my own dice game, which I have decided to call "Five Dice"[1]. I have started by implementing a number of classes to represent:

- A single die (Die)
- A set of dice (Dice)
- A roll of the set of dice (Turn)
- A top-level framework to create turns and examine the outcome (FiveDice)

Broadly, what I eventually hope to achieve is that the player will roll five dice and will be awarded points based on the outcome of the roll. Outcomes that I have identified so far are:

- Five of a kind – all dice showing the same number
- Four of a kind – four of five dice showing the same number
- Large straight – 1, 2, 3, 4, 5 or 2, 3, 4, 5, 6
- Over twenty – the total of all dice adds up to *more than* 20

However, I have got stuck in my implementation because I can't test my code that decides what sort of outcome I have.

I need to refactor my code so that it is testable.

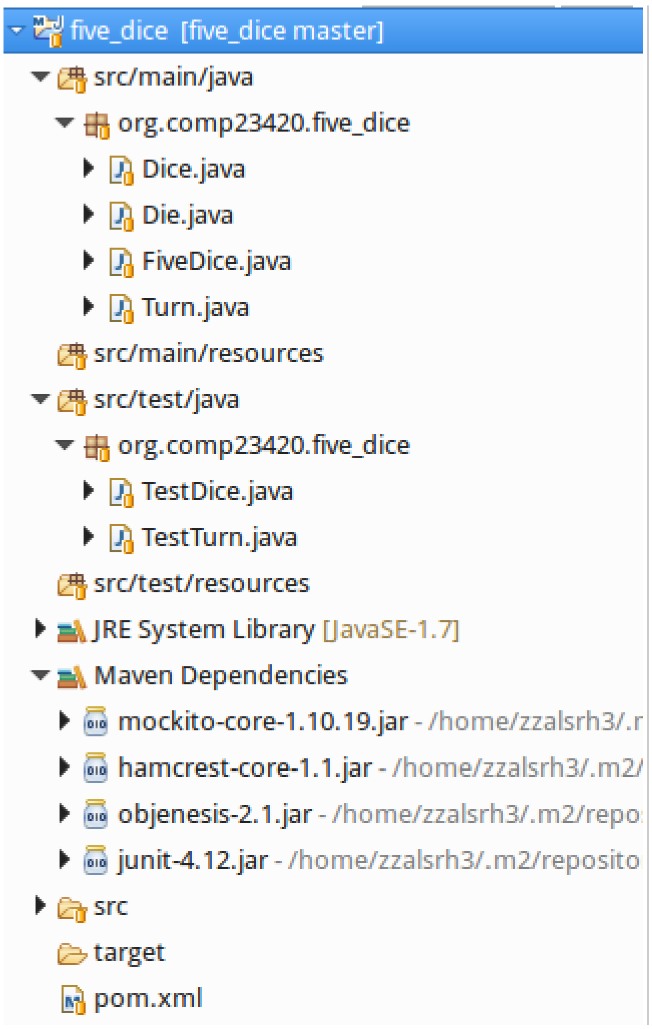
[1] Any resemblance to Yahtzee is entirely coincidental…

Getting the code

My code is kept in GitHub so you can load it into Eclipse in the usual manner – see the workshop exercise from Week 2 if you need a reminder of how to do this. The address of the code repository is:

https://github.com/hainesr/comp23420-five-dice.git

The code is very simple so it doesn't use a build system such as Ant. It does, however, use a tool called maven[2] for dependency management. You can largely ignore maven, but it will need a few moments to download the dependencies we need to help us test the code when you first import the project. When maven has done this you will see a folder in your Eclipse project called "Maven Dependencies". If you expand this folder you will see a few Java libraries, such as JUnit and Mockito in there – see Figure 1. You will also see that source code and test code is kept separate, as you might expect.



```
five_dice [five_dice master]
  src/main/java
    org.comp23420.five_dice
      Dice.java
      Die.java
      FiveDice.java
      Turn.java
  src/main/resources
  src/test/java
    org.comp23420.five_dice
      TestDice.java
      TestTurn.java
  src/test/resources
  JRE System Library [JavaSE-1.7]
  Maven Dependencies
    mockito-core-1.10.19.jar - /home/zzalsrh3/.r
    hamcrest-core-1.1.jar - /home/zzalsrh3/.m2/
    objenesis-2.1.jar - /home/zzalsrh3/.m2/repo
    junit-4.12.jar - /home/zzalsrh3/.m2/reposito
  src
  target
  pom.xml
```

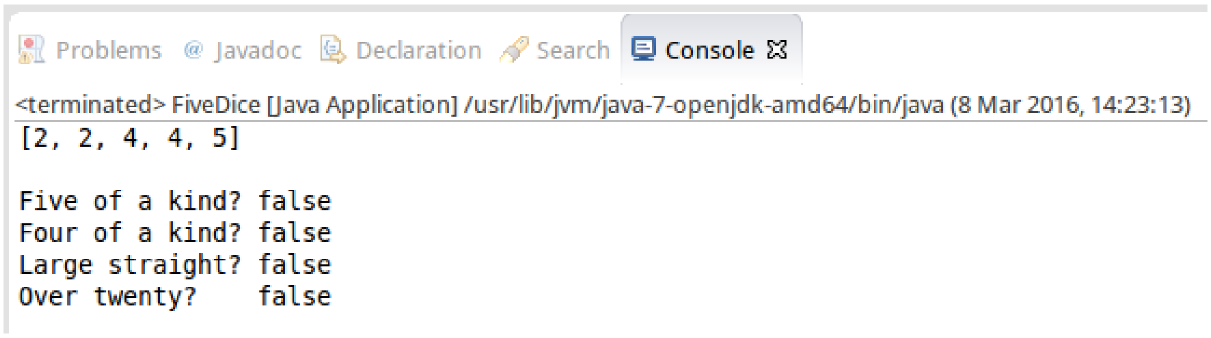
Figure 1:The Five Dice project view.

Running the code

Open the `FiveDice.java` source file by double clicking on it. As you can see it just creates a Turn object and the prints out some information about it. Run it by right-clicking on the code, or the

[2] https://maven.apache.org/

filename in the project view, and selecting "Run As -> Java Application". You should see some output in the Eclipse console – see Figure 2.



```
<terminated> FiveDice [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (8 Mar 2016, 14:23:13)
[2, 2, 4, 4, 5]

Five of a kind? false
Four of a kind? false
Large straight? false
Over twenty?    false
```

Figure 2: Five Dice console output.

Running the tests

I haven't set up a complete test suite for this project yet but you can run individual tests by right-clicking on a test file, `TestDice.java` for example, and selecting "Run As -> JUnit Test". Open the `TestTurn.java` file and have a look at the tests that are defined in there.

Exercise 1

Take a few minutes to investigate the questions below and we will discuss them with the group.

Try running this test class a few times and look at the results. What do you notice?

Look at the source code of the class we're testing here – `Turn.java` – and others that it relies on. Can you see why it's hard to test?

Why is it impossible to test this class as it stands?

Refactoring the Turn class

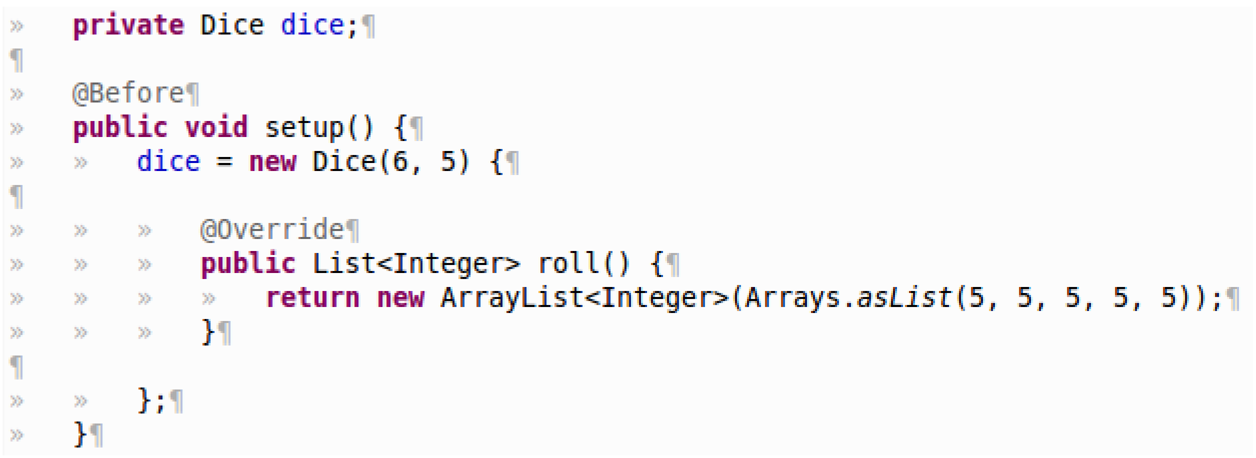
We need to refactor the Turn class so that its constructor takes a Dice object as a parameter, and then use that to perform the roll. Don't forget to also refactor the FiveDice class and the tests for Turn.

Feel free to have a go at this, but it's a fairly mechanical process, so to speed things up you can just checkout or merge in the "turn-refactor" branch at this stage to get the code.

Overriding methods to help testability

So now we have a Turn object that we can pass an arbitrary Dice object to, how does that help us test it more easily?

Well, now that we can pass in an arbitrary Dice object, we can create one with an overridden `roll()` method, like in Figure 3, here:

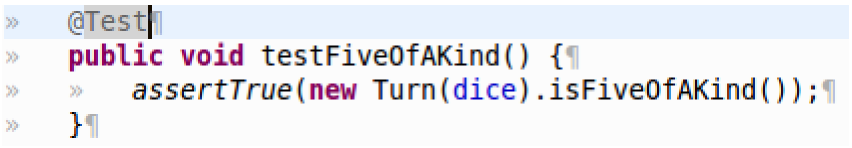


```
	private Dice dice;

	@Before
	public void setup() {
		dice = new Dice(6, 5) {

			@Override
			public List<Integer> roll() {
				return new ArrayList<Integer>(Arrays.asList(5, 5, 5, 5, 5));
			}

		};
	}
```

Figure 3: Overriding roll() to fix its output.

Implement this in your code.

Here we've added a method that is run before each of the tests in the TestTurn class, and we can pass this into the constructor for Turn in each of the test methods – `testFiveOfAKind()`, and so on, like in Figure 4.



```
	@Test
	public void testFiveOfAKind() {
		assertTrue(new Turn(dice).isFiveOfAKind());
	}
```

Figure 4: Testing with an overridden method.

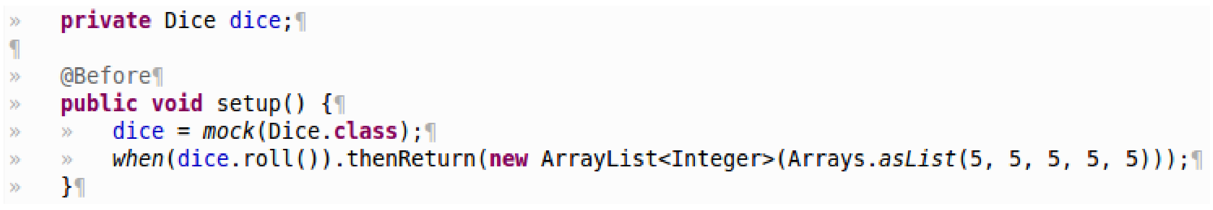
Exercise 2

Try this for all the methods in TestTurn, and see how the test results look now.

What do you notice? Why is this happening?

Mocking classes for deeper testing

So this is an improvement, but all this manual overriding is a fair amount of work, and we're not even getting all the tests to pass yet. We could override the method in each test, or we could use a mock instead.



```
    private Dice dice;

    @Before
    public void setup() {
        dice = mock(Dice.class);
        when(dice.roll()).thenReturn(new ArrayList<Integer>(Arrays.asList(5, 5, 5, 5, 5)));
    }
```

Figure 5: Mocking roll() to fix its output.

Figure 5 is a refactor of the `setup()` method from Figure 3 to use a mock instead of an overridden method.

Hint: you'll need the following extra imports to get the above working in your code:

- `import static org.mockito.Mockito.mock;`
- `import static org.mockito.Mockito.when;`

The `mock()` method creates our special subclass of Dice – this is our mock object.

What is the `when()` method doing? If we run the tests again, what happens now?

Exercise 3

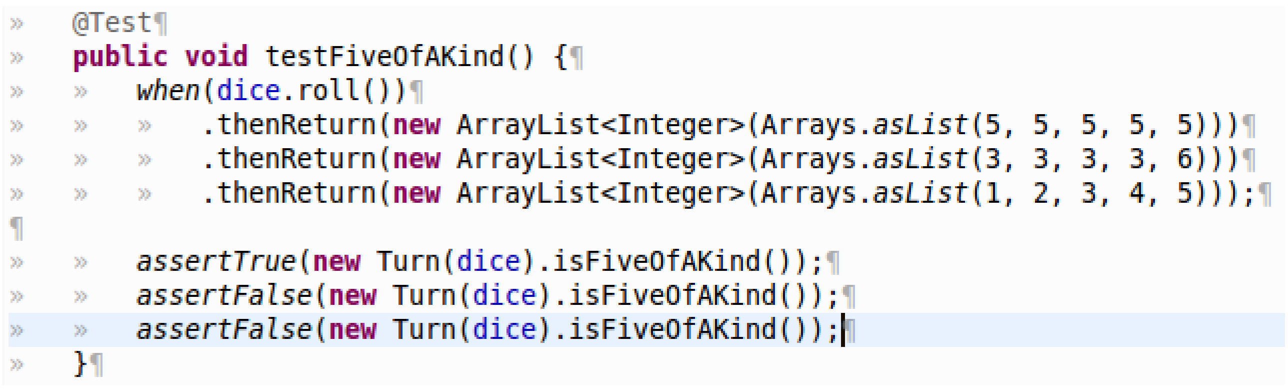
We still have a problem: we've just mocked out the `dice.roll()` method to return all 5s every time, so we have the same behaviour as we had with the overridden method.

Move the `when()` method out of the `setup()` method and copy it into each of the tests in that class, before the `assertTrue()` statements, changing the value returned by `thenReturn()` so that each test passes.

Exercise 4

But this still seems like a lot of work, what is mocking giving me other than a shortcut to overriding a method?

Well, I should really test my methods more than once, and at least check that they don't return true when they shouldn't. Here is where I start to see some real benefit to using mocks: I can simply tell my mocks to return different values each time they are called. I've done this in Figure 6 to check that my method to detect five of a kind rolls, doesn't return true when the roll isn't a five of a kind.



```
	@Test
	public void testFiveOfAKind() {
		when(dice.roll())
			.thenReturn(new ArrayList<Integer>(Arrays.asList(5, 5, 5, 5, 5)))
			.thenReturn(new ArrayList<Integer>(Arrays.asList(3, 3, 3, 3, 6)))
			.thenReturn(new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5)));

		assertTrue(new Turn(dice).isFiveOfAKind());
		assertFalse(new Turn(dice).isFiveOfAKind());
		assertFalse(new Turn(dice).isFiveOfAKind());
	}
```

Figure 6: Multiple returns from a single mock, multiple tests.

I specify three different return values and they are returned, in order to the three calls below.

Repeat this for the other methods in the TestTurn class to ensure that everything is tested properly. Did you find any problems?

Extra exercises for after the workshop (optional)

- Refactor the Die class so that it doesn't use any `static` members
- Allow the Die class to be injected into the Dice class, much like we injected the Dice class into the Turn class above.
- Mock the `roll()` method in the Die class so that you can ensure that it is called exactly five times for each turn.