Assignment 12
Agile Testing

Shaeq Khan skhan27@gmu.edu

TDD Novice: Work through a simple TDD example one step at a time. Don't skip anything. Koskela, Chapter 2, has an excellent example.

Writing the test

We first create a `Template` object by passing the template text as a constructor argument. We then set a value for the variable "`name`" and finally invoke a method named `evaluate`, asserting the resulting output matches our expectation.

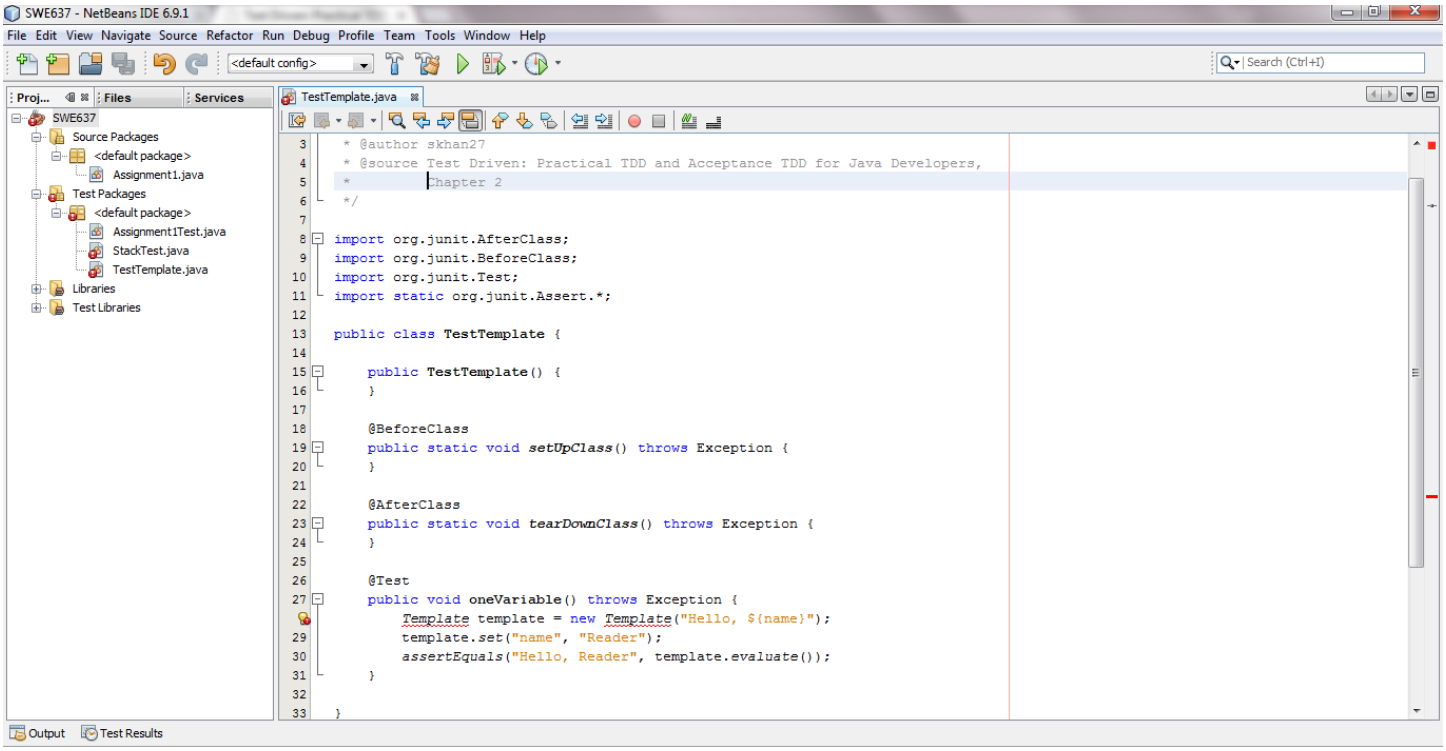

Figure 1 – writing the test

Making the compiler happy

Now, the compiler is eager to remind us that, regardless of our intentions, the class `Template` does not exist so we go ahead and create the class in the package.

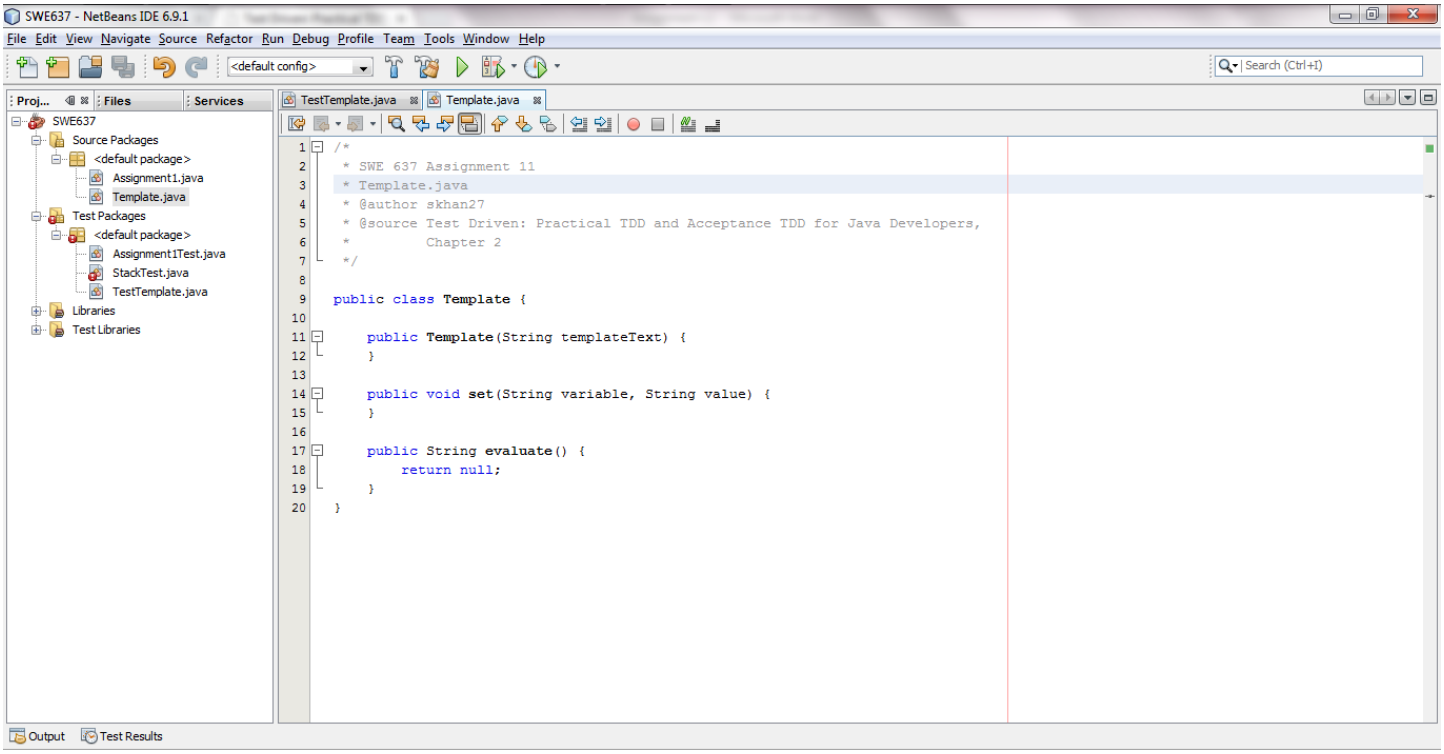

Figure 2 – making the compiler happy

Running the test

When we run our freshly written test, it fails—not surprisingly, because none of the methods we added are doing anything.

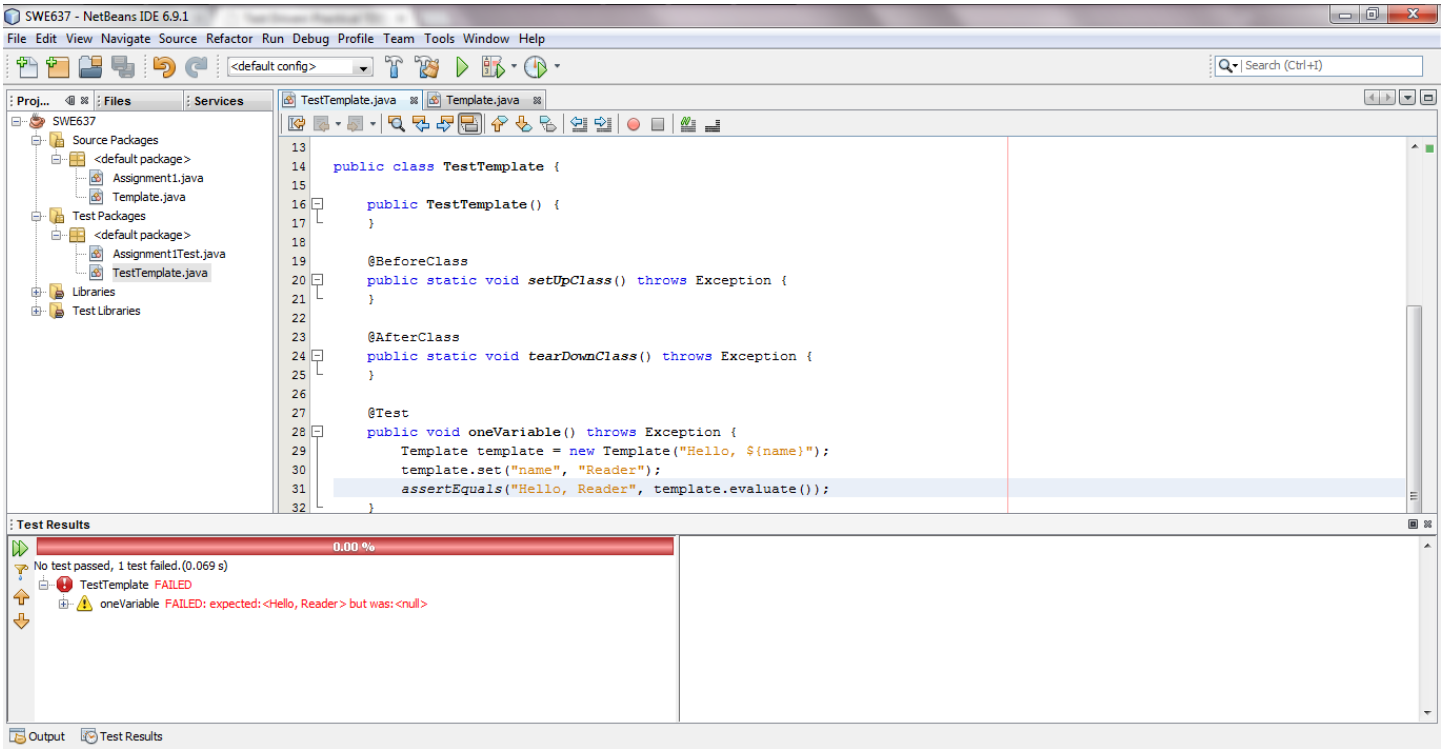

Figure 3 – the first failing test

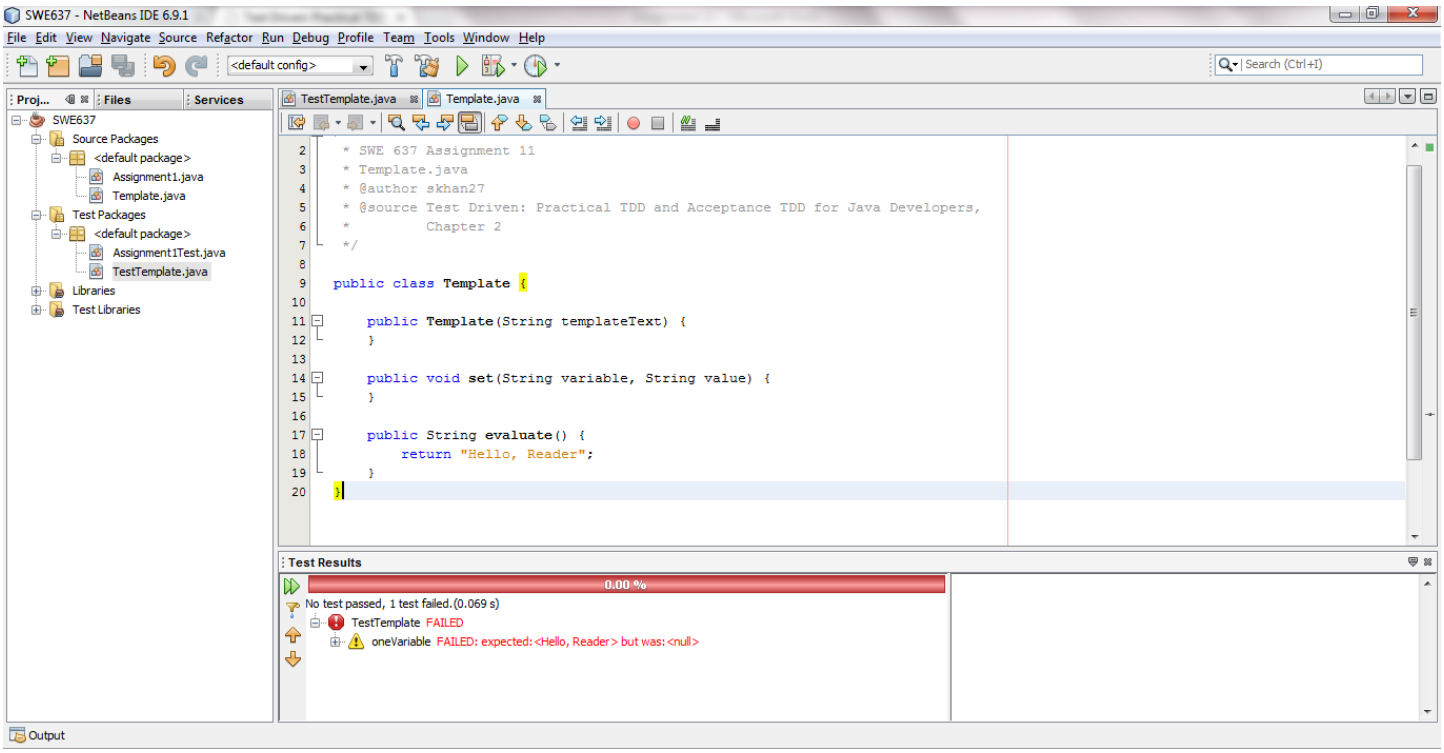

Figure 4 - Passing the test with a hard coded `return statement`.

Forcing out the hard coded return statement with another test.

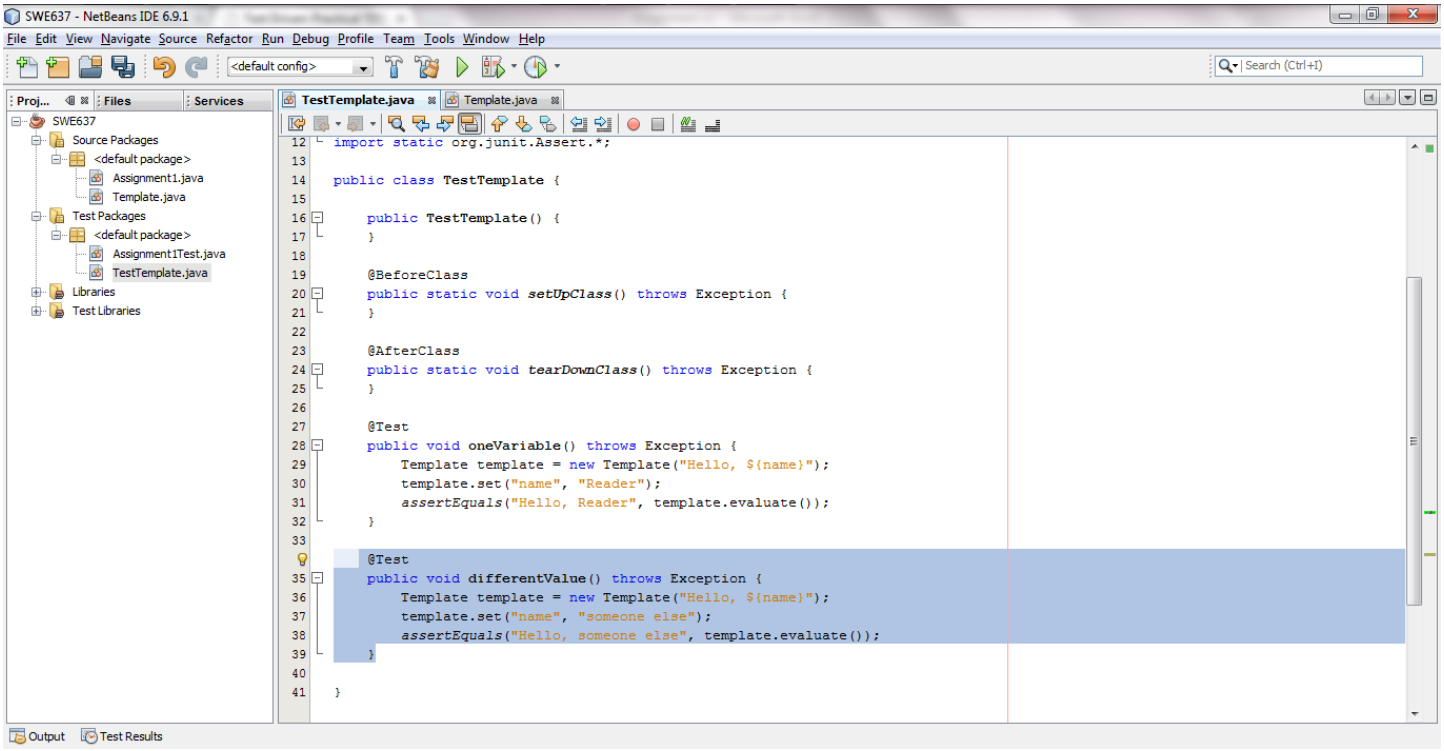

Figure 5 – triangulate with a different value

Making the second test pass by storing and returning the `set value`

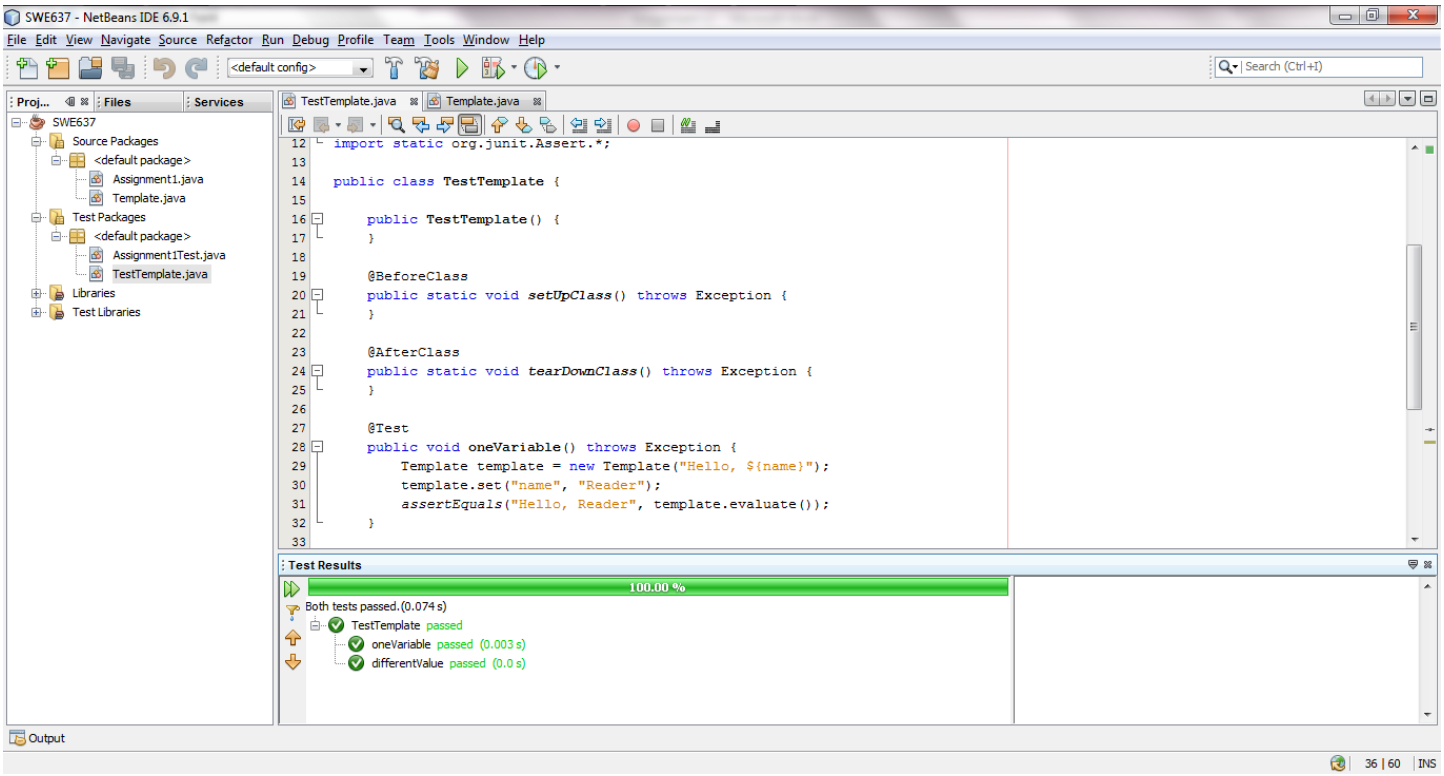

Figure 6 – both test pass

Applying triangulation for the static `template` text

Obviously our hard-coded return statement doesn't cut it anymore

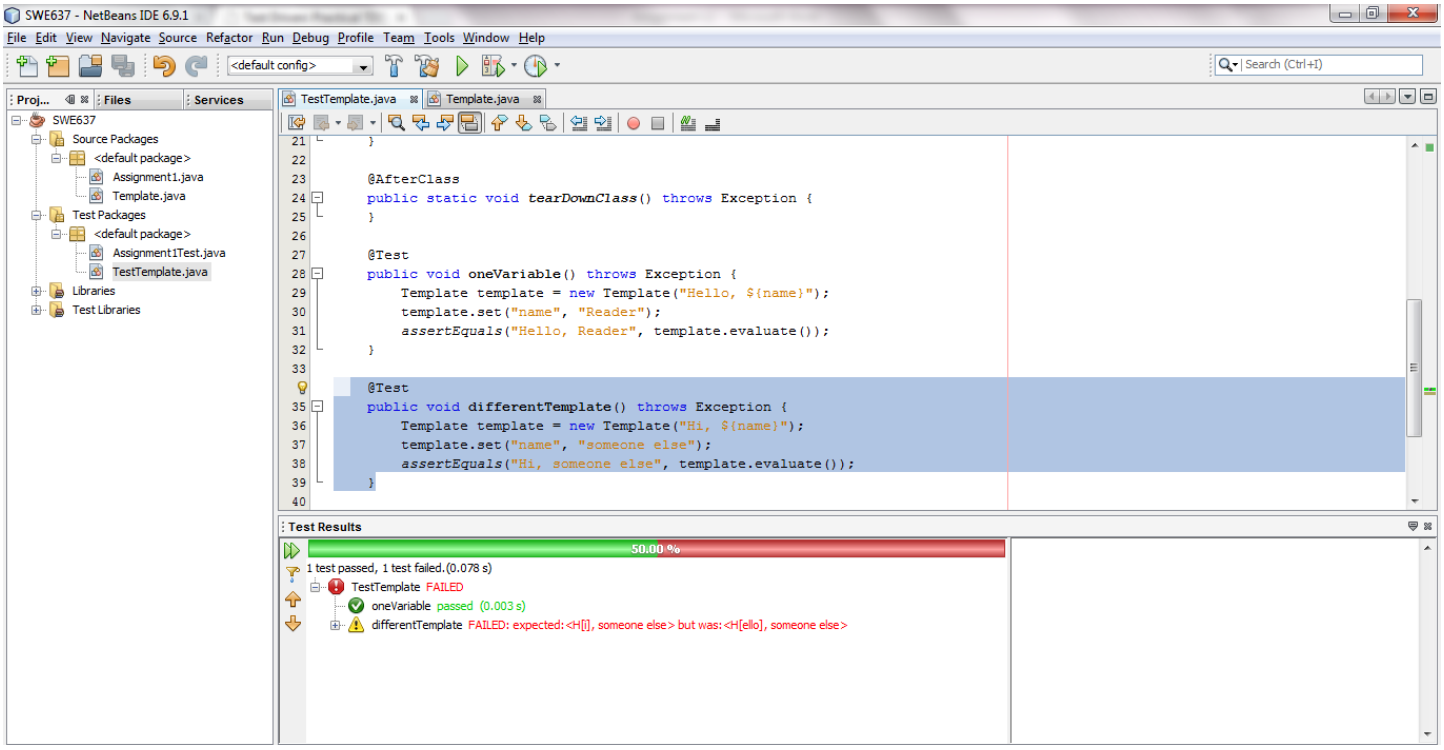

Figure 7 – one test fails.

Faking details a little longer

First, we'll need to start storing the variable `value` and the template `text` somewhere. We'll also need to make `evaluate` replace the placeholder with the value.

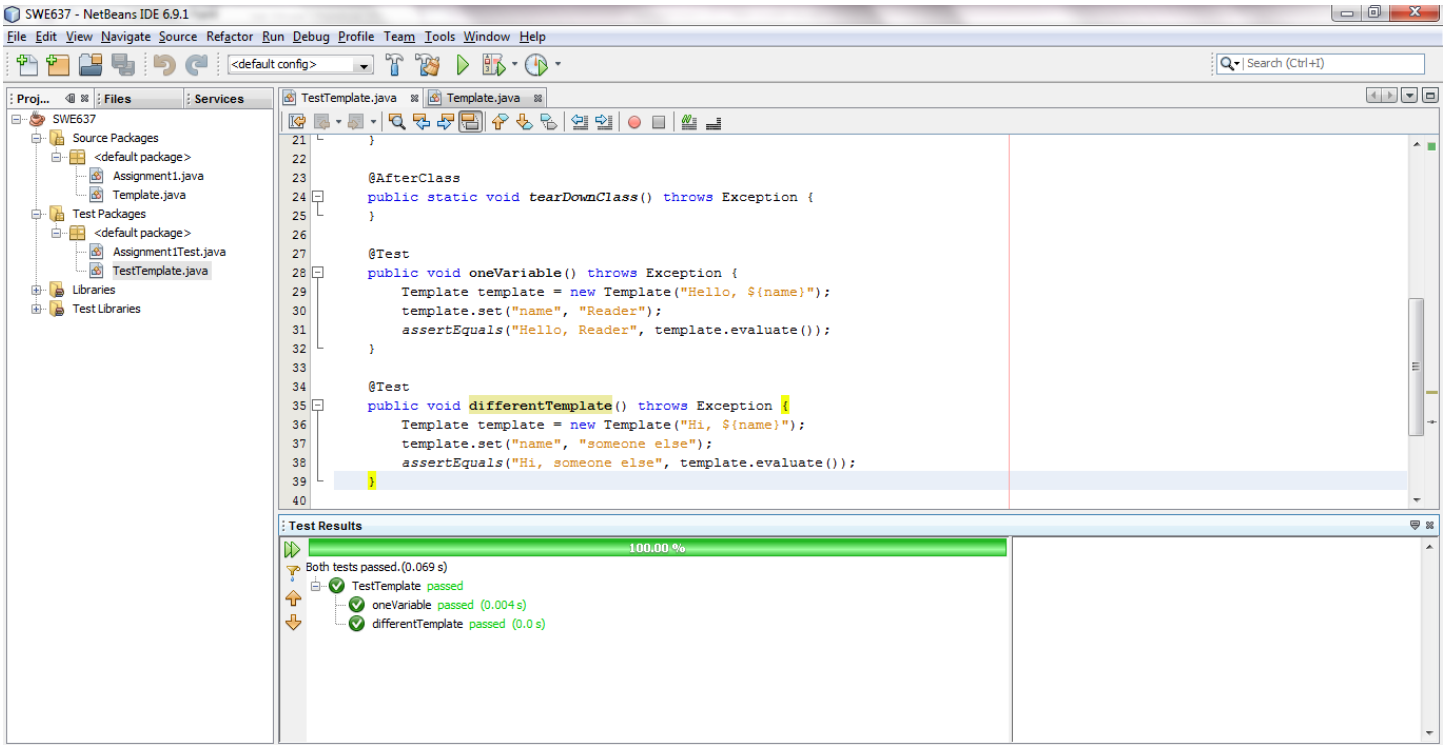

Figure 8 - Our first attempt at handling the variable for real

Squeezing out the fake stuff

The `multipleVariables` test fails right now, telling us that `evaluate` returns the template text as is instead of “1, 2, 3” (which is hardly a surprise because our regular expression is looking for just “${name}”).

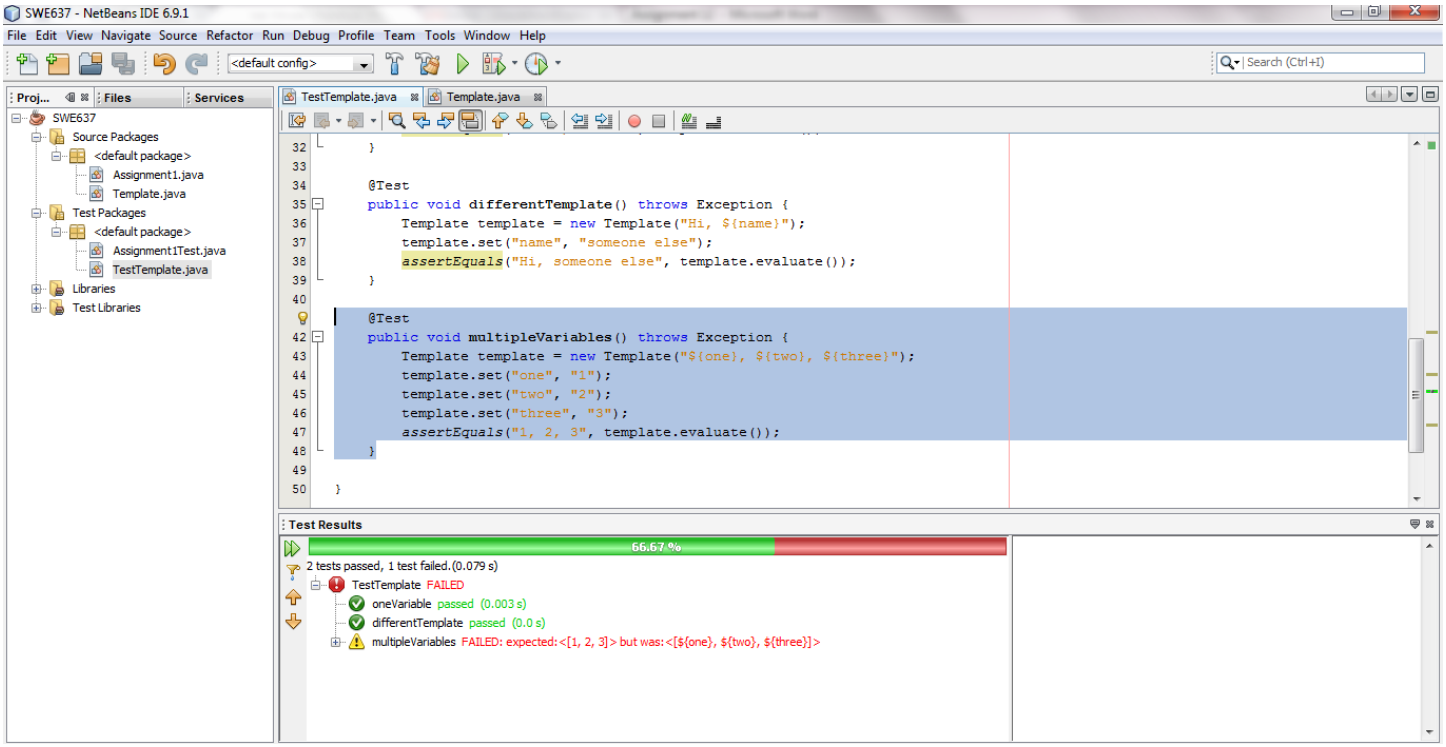

Figure 9 - test for multiple variables on a `template`

Applying the search-and-replace approach to pass our current failing test.

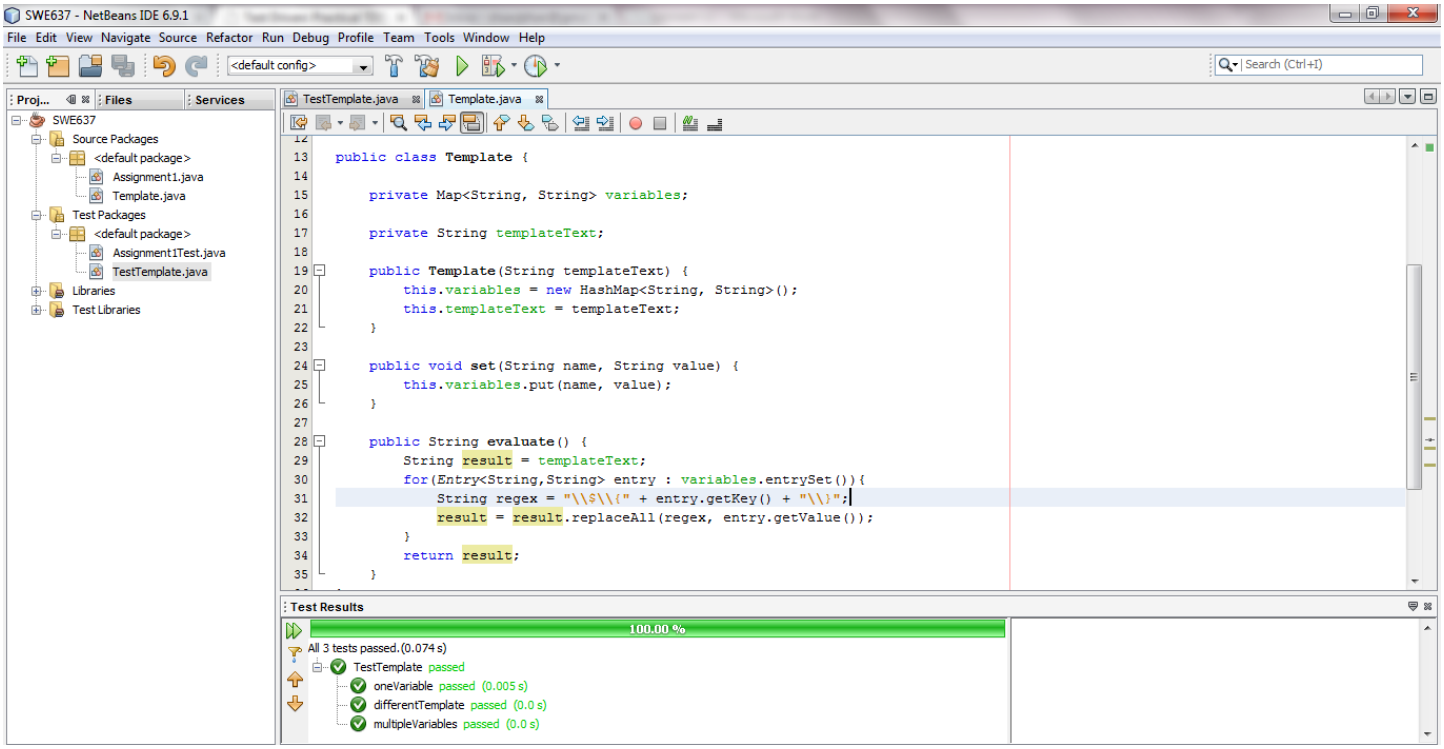

Figure 10 - The search-and-replace-based implementation gets us back to green

Testing for a special case

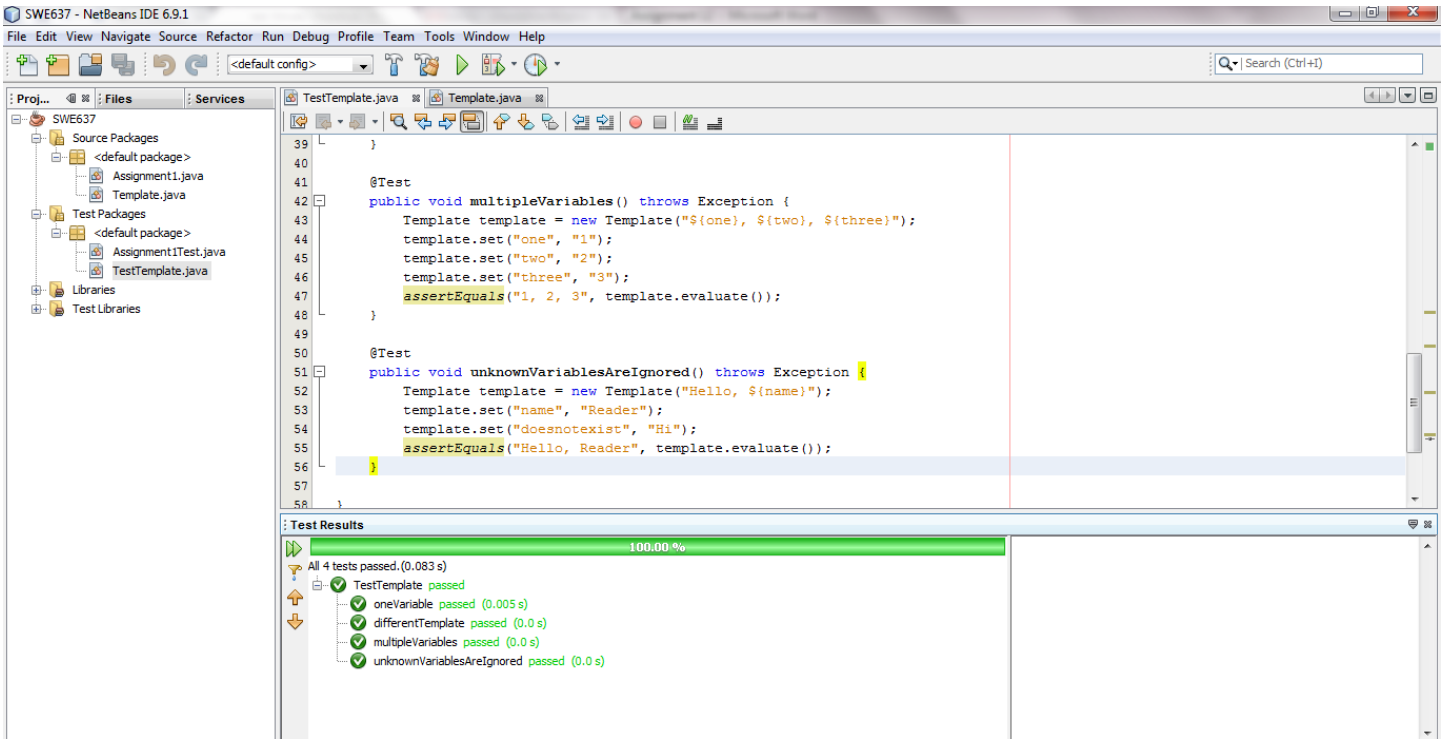

Figure 11 - In this case, the code really does pass the new test with no changes.

Refactoring the test code

As you can see, we were able to mold our tests toward using a single template text and common setup, leaving the test methods themselves delightfully trivial and focusing only on the essential—the specific aspect of functionality they're testing.

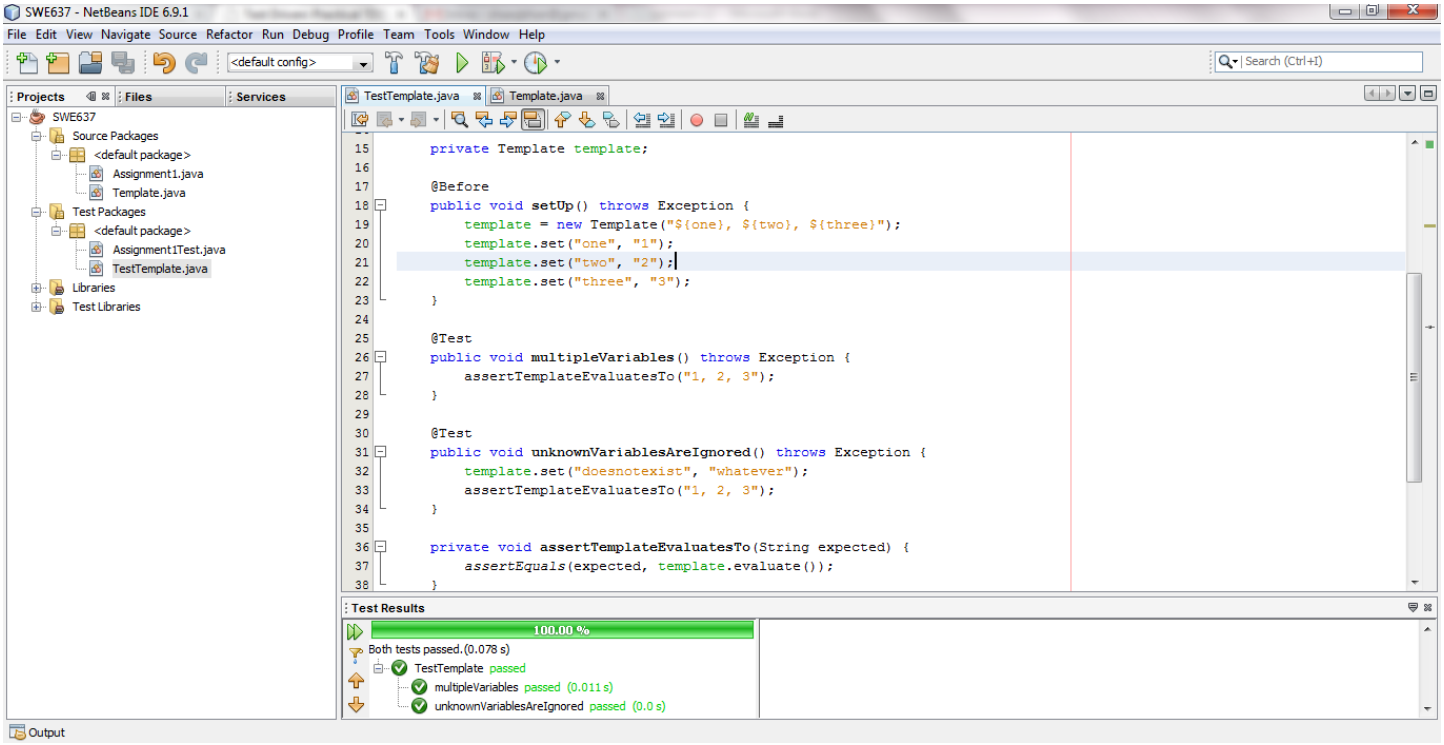

Figure 12 – refactored test class

Adding a bit of error handling

We've got a test that's failing. Well, at first it's not even compiling, but adding an empty MissingValueException class makes that go away.

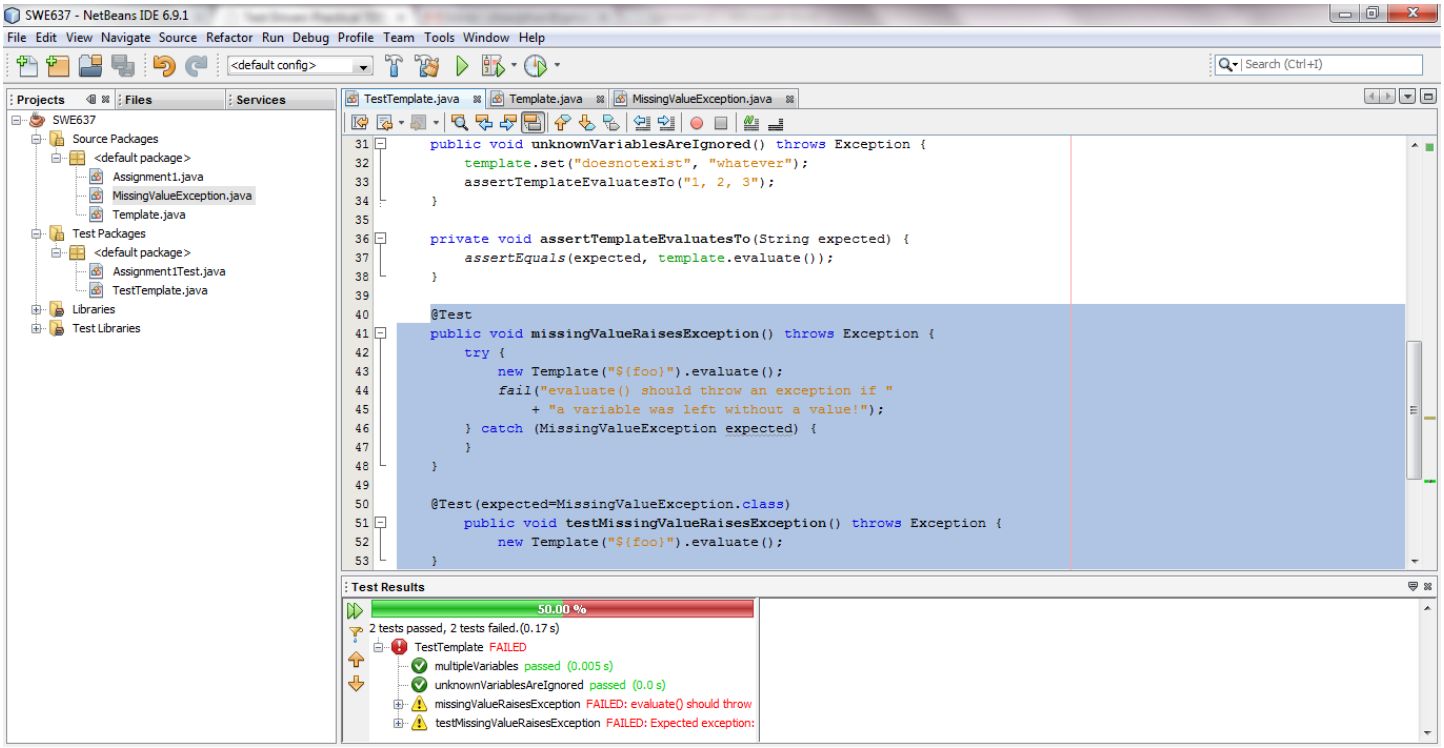

Figure 13 - We have the red bar again, and we're ready to make the test pass. And that means we have to somehow check for missing variables.

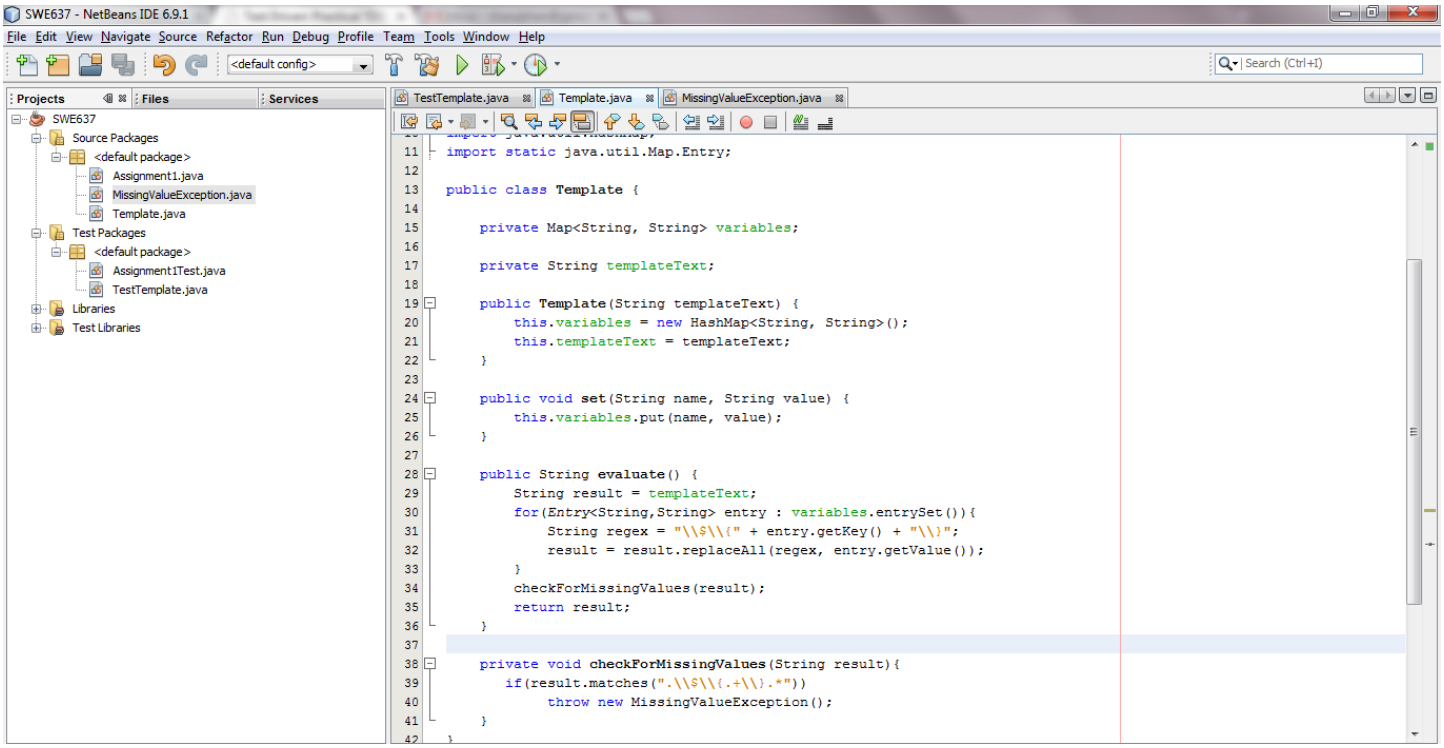

Figure 14 - Refactoring for smaller methods

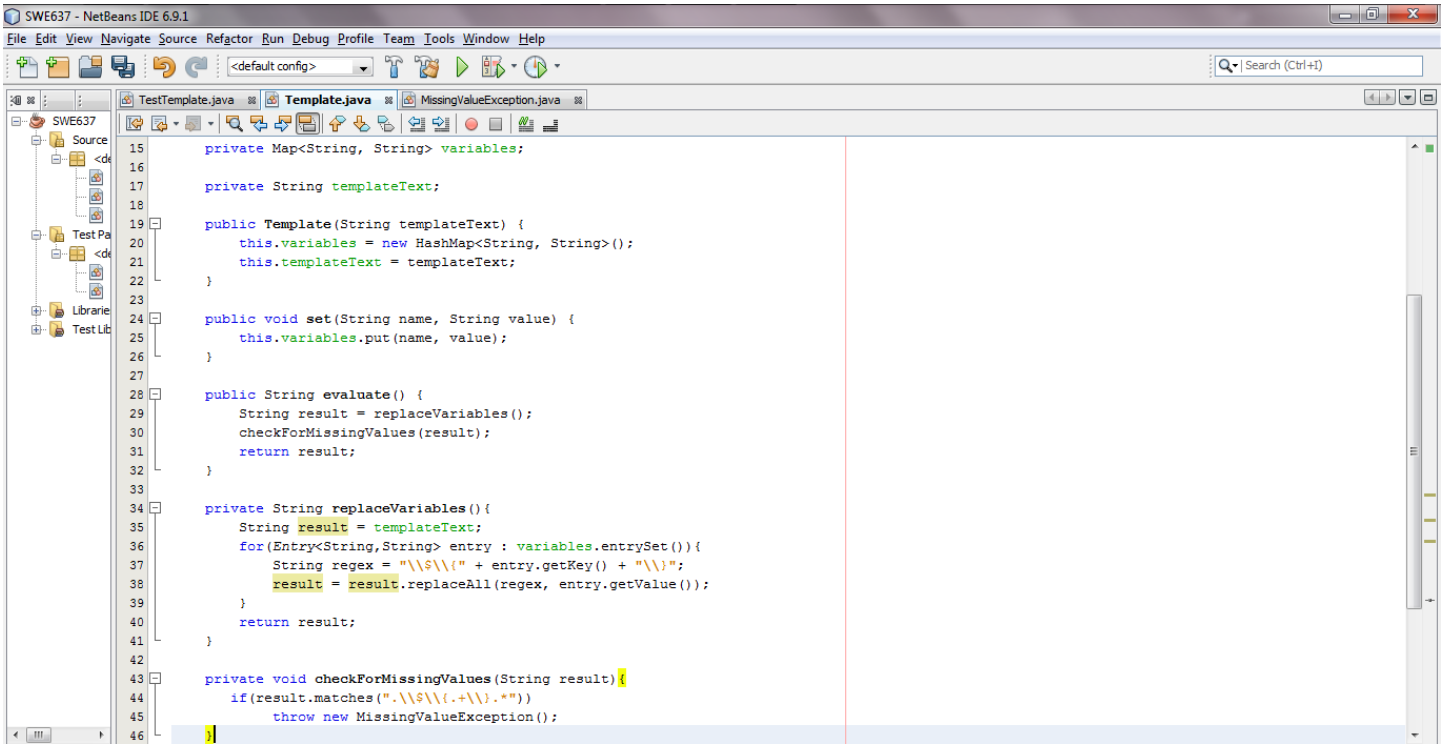

Figure 15 – Keeping methods in balance

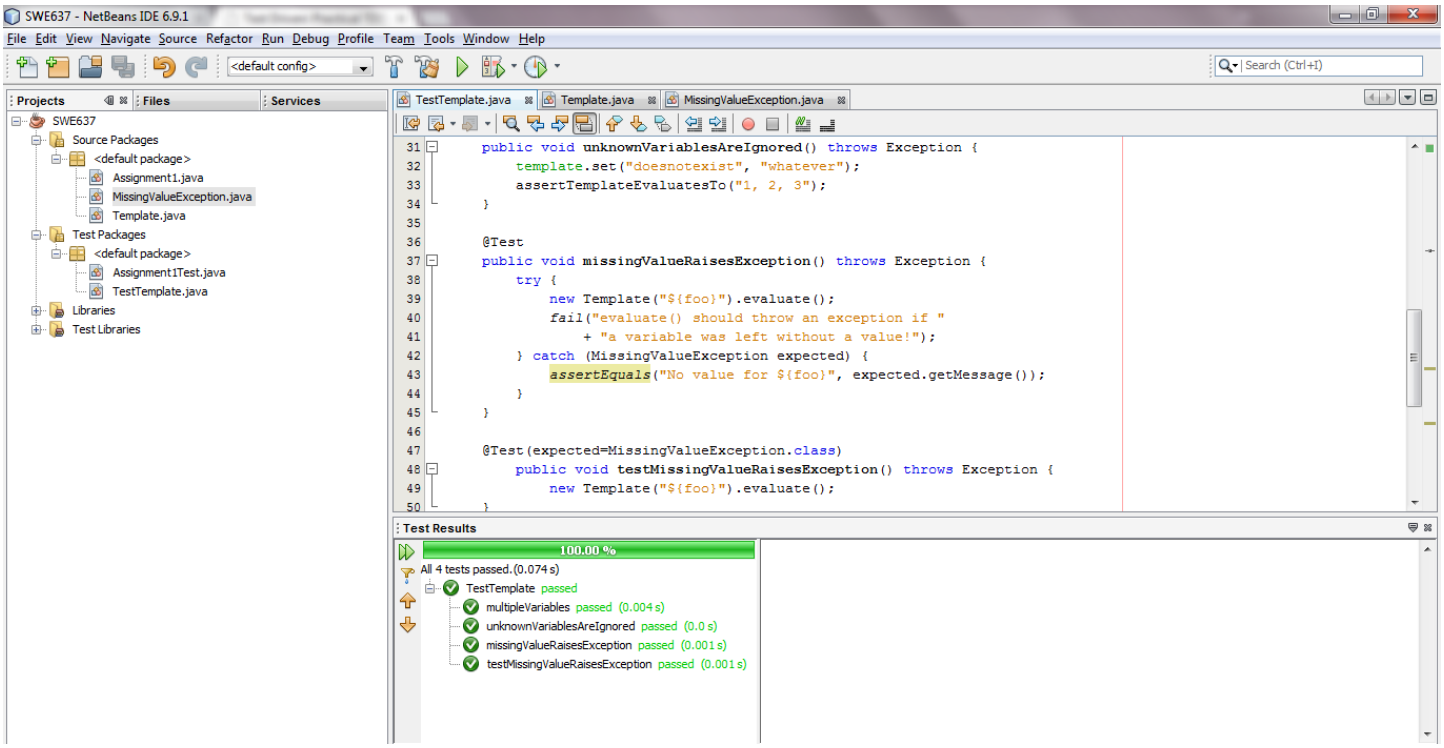

Figure 16 – test run of finalized classes

Looming Design dead end

Regarding the remaining test for variable values that contain "${" and "}", things are starting to look more difficult. For one thing, we can't just do a search-and-replace over and over again until we've covered all variable values set to the template, because some of the variable values rendered first could be re-rendered with something completely different during later rounds of search-and-replace.

Running the test tells us that we certainly have a problem. This test is causing an IllegalArgumentException to be thrown from the regular expression code invoked from evaluate, saying something about an "illegal group reference," so our code is definitely not handling the scenario well.

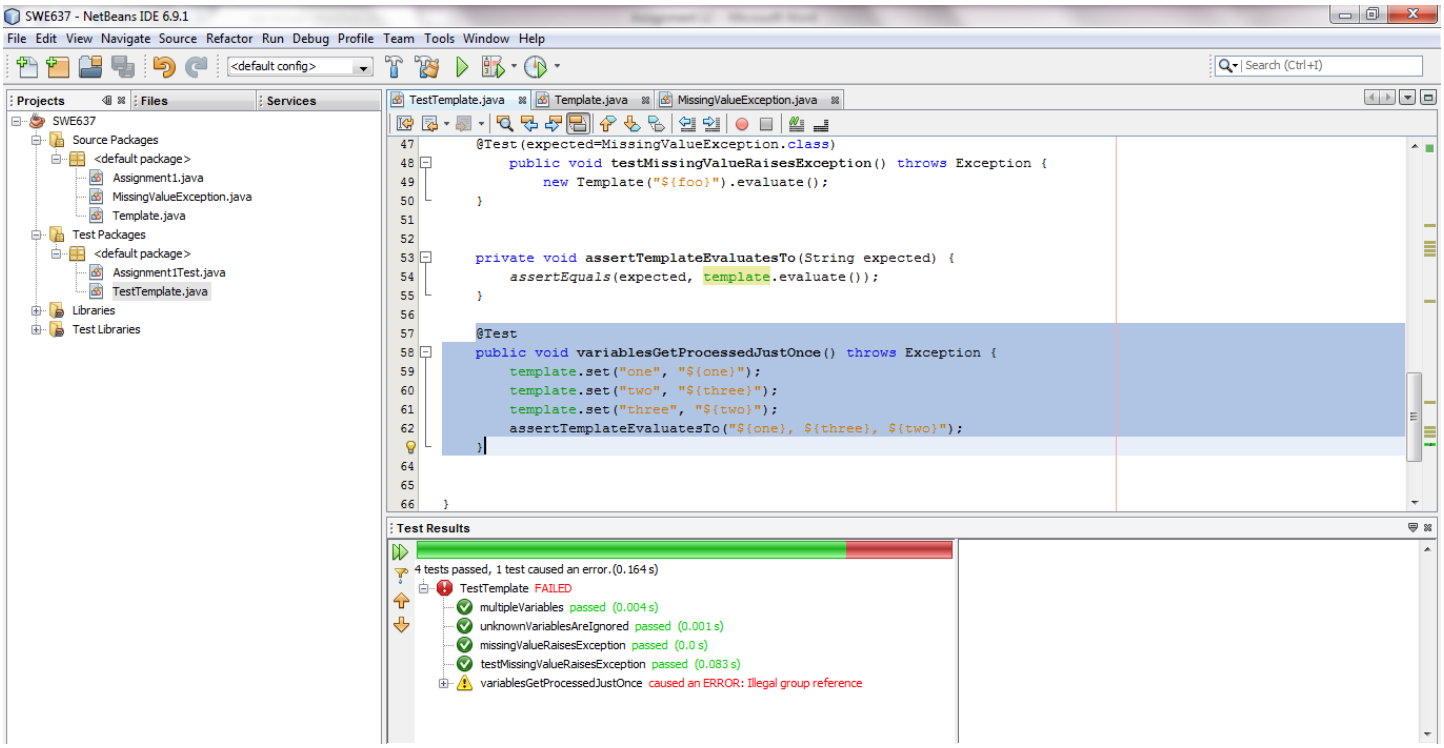

Figure 17 – a double-rendering issue forces us to back out of the test.

Test-driven development is a powerful technique that helps us write better software faster. It does so by focusing on what is absolutely needed right now, then making that tiny piece work, and finally cleaning up any mess we might've made while making it work, effectively keeping the code base healthy. This cycle of first writing a test, then writing the code to make it pass, and finally refactoring the design makes heavy use of programming by intention—writing the test as if the ideal implementation exists already—as a tool for creating usable and testable designs.

```
/*
 * SWE 637 Assignment 11
 * Template.java
 * @author skhan27
 * @source Test Driven: Practical TDD and Acceptance TDD for Java Developers,
 *         Chapter 2
 */

import java.util.Map;
import java.util.HashMap;
import static java.util.Map.Entry;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Template {

    private Map<String, String> variables;

    private String templateText;

    public Template(String templateText) {
        this.variables = new HashMap<String, String>();
        this.templateText = templateText;
    }

    public void set(String name, String value) {
        this.variables.put(name, value);
    }

    public String evaluate() {
        String result = replaceVariables();
        checkForMissingValues(result);
        return result;
    }

    private String replaceVariables(){
        String result = templateText;
        for(Entry<String,String> entry : variables.entrySet()){
            String regex = "\\$\\{" + entry.getKey() + "\\}";
            result = result.replaceAll(regex, entry.getValue());
        }
        return result;
    }

    private void checkForMissingValues(String result) {
        Matcher m = Pattern.compile("\\$\\{.+\\}").matcher(result);
        if (m.find()) {
            throw new MissingValueException("No value for " + m.group());
        }
    }


}
```

```
/*
 * SWE 637 Assignment 11
 * TestTemplate.java
 * @author skhan27
 * @source Test Driven: Practical TDD and Acceptance TDD for Java Developers,
 *         Chapter 2
 */

import org.junit.*;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestTemplate {

    private Template template;

    @Before
    public void setUp() throws Exception {
        template = new Template("${one}, ${two}, ${three}");
        template.set("one", "1");
        template.set("two", "2");
        template.set("three", "3");
    }

    @Test
    public void multipleVariables() throws Exception {
        assertTemplateEvaluatesTo("1, 2, 3");
    }

    @Test
    public void unknownVariablesAreIgnored() throws Exception {
        template.set("doesnotexist", "whatever");
        assertTemplateEvaluatesTo("1, 2, 3");
    }

    @Test
    public void missingValueRaisesException() throws Exception {
        try {
            new Template("${foo}").evaluate();
            fail("evaluate() should throw an exception if "
                + "a variable was left without a value!");
        } catch (MissingValueException expected) {
            assertEquals("No value for ${foo}", expected.getMessage());
        }
    }

    @Test(expected=MissingValueException.class)
        public void testMissingValueRaisesException() throws Exception {
            new Template("${foo}").evaluate();
    }


    private void assertTemplateEvaluatesTo(String expected) {
        assertEquals(expected, template.evaluate());
    }
```

```
@Test
    public void variablesGetProcessedJustOnce() throws Exception {
        template.set("one", "${one}");
        template.set("two", "${three}");
        template.set("three", "${two}");
        assertTemplateEvaluatesTo("${one}, ${three}, ${two}");
    }


}


/*
 * SWE 637 Assignment 11
 * MissingValueException.java
 * @author skhan27
 * @source Test Driven: Practical TDD and Acceptance TDD for Java Developers,
 *         Chapter 2
 */

public class MissingValueException extends RuntimeException {
    public MissingValueException(String message){
        super(message);
    }
}
```