Tutorial for Online Testing with JUnit

OSMO Tester

MBT tool

v3.0

Teemu Kanstrén

Table of Contents

[Introduction 3](#_Toc351228816)

[Basic of Running Online Tests 3](#_Toc351228817)

[Integrating test runs with JUnit 10](#_Toc351228818)

[Conclusions 12](#_Toc351228819)

[References 12](#_Toc351228820)

# Introduction

This tutorial describes running online tests with OSMO Tester using simple examples. The reader should be familiar with the information presented in the OSMO Tester basic and data tutorials. Previously we have discussed using OSMO Tester to create test scripts to be executed separately, including a tutorial on writing scripters. This is generally called offline testing in model-based testing. The other option is called online testing, where the model is directly executed and can guide the test generation while the test is executing, based on the results of each step. This tutorial shows an example of how to do this with OSMO Tester using JUnit as an online test execution framework.

The reader is expected to have basic knowledge of Java programming and ability to use their own favourite IDE such as Eclipse, IntelliJ, or Netbeans. The code shown in this tutorial is available in the OSMO Tester examples package.

# Basic of Running Online Tests

Previously in the other tutorials we have used a simple model that prints out “HELLO” and “WORLD” in different test steps. In this tutorial we use the same model and show some examples of using typical elements for online testing. In most cases, we would have some real application code to test against. In this case we just show the basics to keep to the point. Listing 1 shows the model program we developed in previous tutorials.

public class HelloModel {

private int helloCount = 0;

private int worldCount = 0;

private ValueSet<String> names = new ValueSet<>("teemu", "bob");

private ValueSet<String> worlds = new ValueSet<>("mars", "venus");

private ValueSet<Integer> sizes = new ValueSet<>(1,2,6);

private ValueRange<Double> ranges = new ValueRange<>(0.1d, 5.2d);

@BeforeSuite

public void init() {

names.setStrategy(DataGenerationStrategy.BALANCING);

}

@BeforeTest

public void startTest() {

helloCount = 0;

worldCount = 0;

System.out.println("TEST START");

}

@AfterTest

public void endTest() {

System.out.println("TEST END");

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO "+names.next()+" ("+sizes.next()+")");

helloCount++;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD "+worlds.next()+" ("+ranges.next()+")");

worldCount++;

}

}

Listing 1. The model program.

Similarly, Listing 2 shows the configuration we set up to run the model program.

public class Main {

public static void main(String[] args) {

OSMOConfiguration.setSeed(52);

OSMOTester tester = new OSMOTester(new HelloModel());

tester.addTestEndCondition(new Length(5));

tester.addSuiteEndCondition(new Length(2));

tester.generate();

}

}

Listing 2. Running the model program.

And as a final reminder, the output from running this model program is shown in Figure 1.

TEST START

HELLO bob (6)

WORLD venus (3.818798374856044)

HELLO teemu (2)

WORLD mars (3.3202641696335067)

HELLO teemu (2)

TEST END

TEST START

HELLO bob (6)

WORLD venus (0.3211659051330242)

HELLO bob (6)

WORLD venus (1.0997927720325893)

HELLO teemu (1)

TEST END

generated 2 tests.

Figure 1. Example output.

This is an example of offline scripting mode, where test scripts are written from the model separately to be executed later by a separate test execution framework. Now we change to an offline test mode where tests are executed directly from the model program as shown in Listing 3.

public class HelloModel {

private int helloCount = 0;

private int worldCount = 0;

private ValueSet<String> names = new ValueSet<>("teemu", "bob");

private ValueSet<String> worlds = new ValueSet<>("mars", "venus");

private ValueSet<Integer> sizes = new ValueSet<>(1,2,6);

private ValueRange<Double> ranges = new ValueRange<>(0.1d, 5.2d);

**private HelloProgram sut = new HelloProgram();**

@BeforeSuite

public void init() {

names.setStrategy(DataGenerationStrategy.BALANCING);

}

@BeforeTest

public void startTest() {

helloCount = 0;

worldCount = 0;

System.out.println("TEST START");

}

@AfterTest

public void endTest() {

System.out.println("TEST END");

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

**String name = names.next();**

**String response = sut.hello(name, sizes.next());**

**assertEquals("hi dude, " + name, response);**

helloCount++;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

**double range = ranges.next();**

**String response = sut.world(worlds.next(), range);**

**assertEquals(range+"? thats pretty mighty, dude", response);**

worldCount++;

}

}

Listing 3. Modified model program with separate scripter.

If we look at the changes we made, there is the class that provides the interface to the program we are testing. In this case it is the “HelloProgram” with the variable name “sut” (for system under test). The test steps are now changed to invoke functionality on the test target and not generate scripts for it. We also save some of the input that is used to assess the correctness of the input received from the test target. Similarly, the model state could be used to guide the output evaluation (test oracle). The input is actually already guided by the state as it defines the allowed order of test steps, but could also form directly part of the input (as it actually does with the names, worlds, sizes, and ranges variables).

Let’s also look at the little program we use as an example of something to test. This is shown in Listing 4.

public class HelloProgram {

private int counter = 0;

public String hello(String name, int size) {

counter++;

return "hi dude, "+name;

}

public String world(String world, double range) {

counter++;

return range+"? swweet, dude";

}

}

Listing 4. The program being tested.

This is really a really simple example to illustrate the concept. So it just returns a response to the caller, modified according to the given input. While this is a simple example of direct invocation (similar to unit testing), the calls could also be to more complex classes or to networked systems using protocols such as REST or SOAP.

What happens if we run it? We get the output shown in Figure 2.

TEST START

TEST END

TEST START

TEST END

generated 2 tests.

Figure 2. Example output.

This is because we do not print anything from the model anymore during test steps being executed, just invoke the test target. So let’s add some minor prints to get an idea of what is happening. This is shown in Listing 5.

public class HelloModel2 {

private int helloCount = 0;

private int worldCount = 0;

private ValueSet<String> names = new ValueSet<>("teemu", "bob");

private ValueSet<String> worlds = new ValueSet<>("mars", "venus");

private ValueSet<Integer> sizes = new ValueSet<>(1,2,6);

private ValueRange<Double> ranges = new ValueRange<>(0.1d, 5.2d);

private HelloProgram sut = new HelloProgram();

@BeforeSuite

public void init() {

names.setStrategy(DataGenerationStrategy.BALANCING);

}

@BeforeTest

public void startTest() {

helloCount = 0;

worldCount = 0;

System.out.println("TEST START");

}

@AfterTest

public void endTest() {

System.out.println("TEST END");

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

**System.out.println("HELLO");**

String name = names.next();

String response = sut.hello(name, sizes.next());

assertEquals("hi dude, " + name, response);

helloCount++;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

**System.out.println("WORLD");**

double range = ranges.next();

String response = sut.world(worlds.next(), range);

assertEquals(range+"? swweeet, dude", response);

worldCount++;

}

}

Listing 5. Model program with more detailed prints.

Running this we get the output shown in Figure 3.

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

generated 2 tests.

Figure 3. Example output.

So now we see a bit more about what is happening. And what if we get errors? Let’s modify the program to see. The Listing 6 shows a modified version that gives invalid input with a given condition.

public class HelloProgram2 {

private int counter = 0;

public String hello(String name, int size) {

counter++;

**if (counter % 15 == 10) {**

**return "whoopsidaisy";**

**}**

return "hi dude, "+name;

}

public String world(String world, double range) {

counter++;

return range + "? swweeet, dude";

}

}

Listing 6. The program being tested with an error.

Now running the model against this program gives the output shown in Figure 4.

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

Exception in thread "main" java.lang.RuntimeException: junit.framework.ComparisonFailure: expected:<[hi dude, teemu]> but was:<[whoopsidaisy]>

at osmo.tester.generator.MainGenerator.unwrap(MainGenerator.java:378)

at osmo.tester.generator.MainGenerator.nextTest(MainGenerator.java:100)

at osmo.tester.generator.MainGenerator.generate(MainGenerator.java:69)

at osmo.tester.OSMOTester.generate(OSMOTester.java:71)

at osmo.tester.examples.tutorial.online.Main3.main(Main3.java:14)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:601)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)

**Caused by: junit.framework.ComparisonFailure: expected:<[hi dude, teemu]> but was:<[whoopsidaisy]>**

at junit.framework.Assert.assertEquals(Assert.java:85)

at junit.framework.Assert.assertEquals(Assert.java:91)

at osmo.tester.examples.tutorial.online.HelloModel3.sayHello(HelloModel3.java:51)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:601)

at osmo.tester.model.InvocationTarget.invoke(InvocationTarget.java:32)

at osmo.tester.generator.MainGenerator.execute(MainGenerator.java:162)

at osmo.tester.generator.MainGenerator.nextStep(MainGenerator.java:138)

at osmo.tester.generator.MainGenerator.nextTest(MainGenerator.java:87)

... 8 more

Figure 4. Example output.

That is, the execution of the second test case stopped at the third “HELLO” when the test assertion comparing the expected output with the actual output failed. The @AfterTest and @AfterSuite annotated methods are always executed to allow for test teardown code to execute.

Note that by default the generator tries to unwrap some of the exceptions that would be layered on top by the generator executing your code to make the exception reports produced a bit more readable. If you wish to disable this you can do so as shown in Listing 7. This same listing also shows how you can set the test generation to stop completely or continue generating more tests if an exception is thrown by the model. Check the Javadocs for more details.

public class Main4 {

public static void main(String[] args) {

OSMOConfiguration.setSeed(52);

OSMOTester tester = new OSMOTester(new HelloModel3());

tester.addTestEndCondition(new Length(5));

tester.addSuiteEndCondition(new Length(2));

**tester.getConfig().setUnwrapExceptions(false);**

**tester.getConfig().setFailWhenError(true);**

tester.generate();

}

}

Listing 7. Unwrapping runtime exceptions.

# Integrating test runs with JUnit

So above we created some online tests and executed them through OSMO Tester. In practice this is much like running tests on tools like JUnit (from which we imported the assertions already in the examples above). To run these tests over JUnit, we can apply some specific OSMO Tester components. Listing 8 shows how we can do this.

@RunWith(OSMORunner.class)

public class JUnitMain {

@OSMOConfigurationProvider

public static OSMOConfiguration configure() {

OSMOConfiguration config = new OSMOConfiguration();

config.setSeed(52);

config.addModelObject(new HelloModel3());

config.setJUnitLength(2);

config.addTestEndCondition(new Length(5));

return config;

}

}

Listing 8. Using JUnit to run the tests.

Now we can run this as a set of JUnit tests. JUnit requires knowing the number of tests before test execution is started. So to be able to generate tests on the fly, we must define how many we will generate. This is what the *config.setJUnitLength()* is for. Running this produces the output shown in Figure 5.

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

java.lang.RuntimeException: junit.framework.ComparisonFailure:

Expected :hi dude, teemu

Actual :whoopsidaisy

<Click to see difference>

at osmo.tester.generator.MainGenerator.unwrap(MainGenerator.java:378)

at osmo.tester.generator.MainGenerator.nextTest(MainGenerator.java:100)

at osmo.tester.reporting.junit.OSMOJUnitTest.execute(OSMOJUnitTest.java:44)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:44)

at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)

at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:41)

at osmo.tester.reporting.junit.OSMOJUnitTest.invokeExplosively(OSMOJUnitTest.java:35)

at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)

at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:274)

at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:70)

at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:48)

at org.junit.runners.ParentRunner$3.run(ParentRunner.java:242)

at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:58)

at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:240)

at org.junit.runners.ParentRunner.access$000(ParentRunner.java:48)

at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:233)

at org.junit.runners.ParentRunner.run(ParentRunner.java:303)

at org.junit.runner.JUnitCore.run(JUnitCore.java:157)

at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:76)

at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:195)

at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:63)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)

Caused by: junit.framework.ComparisonFailure: expected:<[hi dude, teemu]> but was:<[whoopsidaisy]>

at osmo.tester.examples.tutorial.online.HelloModel3.sayHello(HelloModel3.java:51)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at osmo.tester.model.InvocationTarget.invoke(InvocationTarget.java:32)

at osmo.tester.generator.MainGenerator.execute(MainGenerator.java:162)

at osmo.tester.generator.MainGenerator.nextStep(MainGenerator.java:138)

at osmo.tester.generator.MainGenerator.nextTest(MainGenerator.java:87)

... 28 more

Figure 5. Example output.

Looks pretty much the same as before, so what’s the point? Well you can now make use of any integration with your IDE’s (IntelliJ, Eclipse, Netbeans, …) and build systems (Jenkins, …?) where possible. Figure 6 shows an example of running our test model through the JUnit front using the code shown in Listing 8. In this figure we see how the full JUnit integration works and we can see how the IDE has analyzed the test results, produced hyperlinks for errors, provides results comparisons and so on. In this case, the execution happens by right-clicking the class name to run the “test class” and all the JUnit tests it contains. The tests are then generated on the fly as JUnit asks for them, and reports are updated on the IDE as the tests are run.

The specifics for using your IDE or build environment may vary, and while this has been tested at the time of writing this tutorial only with the IntelliJ integration, you should give it a try with your environment (and let us know).

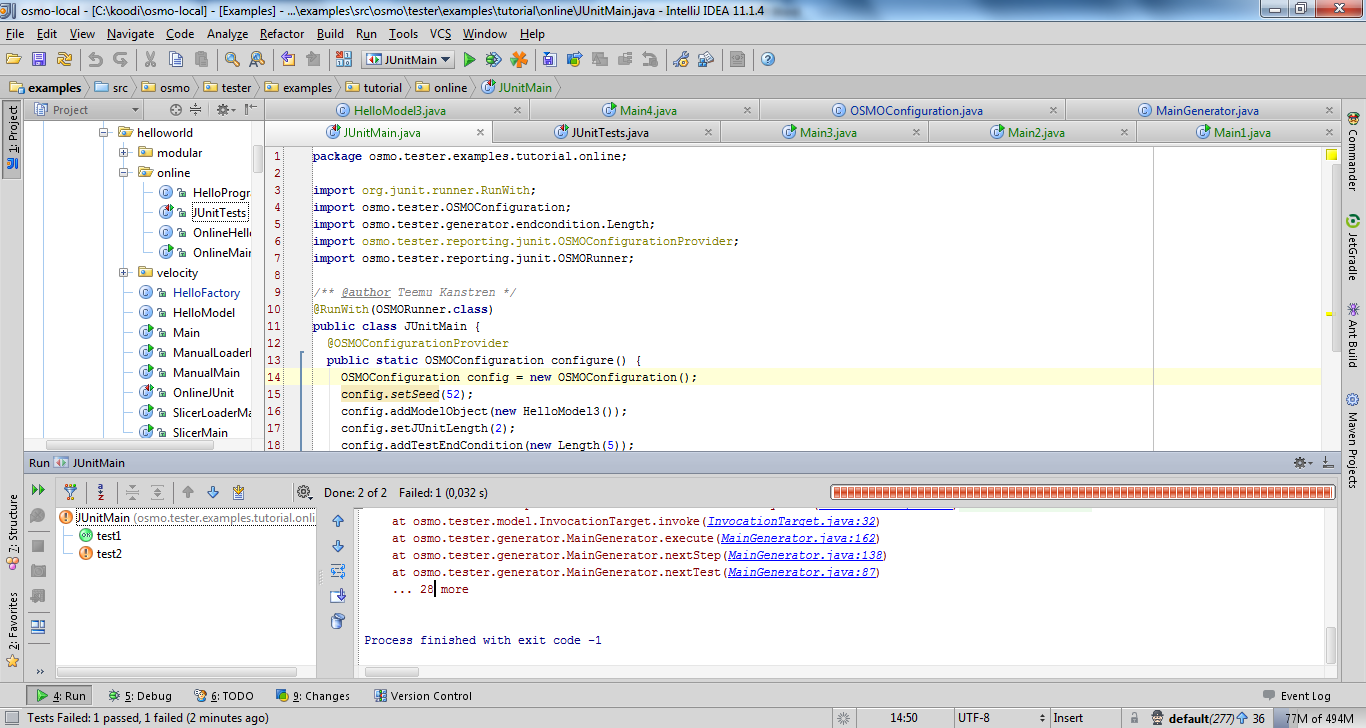


Figure 6. Running the model program through IntelliJ JUnit integration.

# Conclusions

This tutorial showed how to create and run simple online tests with JUnit and OSMO Tester. The examples shown apply to locally executed tests similar to Java unit tests. The same approach can be applied to other platforms as well, such as testing networked systems by just making the test steps send data (e.g., REST or SOAP requests) and evaluate the results received. In practice it just requires that you integrate your test execution environment with your models. JUnit is an easy example used here, but commonly writing your own should not be too hard as it should be not much more than making some API calls.

We also showed how to run your tests through JUnit integration in an IDE. Hopefully you can also manage a similar feature for your build environment.

# References

OSMOTester home page: <http://code.google.com/p/osmo/>