Tutorial for Writing a Scripter

OSMO Tester

MBT tool

v2.1

Teemu Kanstrén

Table of Contents

[Introduction 3](#_Toc315292886)

[Scripting the Hello World to Console 3](#_Toc315292887)

[Scripting Hello World with Velocity 6](#_Toc315292888)

[Conclusions 9](#_Toc315292889)

[References 9](#_Toc315292890)

# Introduction

This tutorial describes writing a scripter for OSMO Tester using simple examples. The reader should be familiar with the information presented in the OSMO Tester basic and data tutorials. This tutorial continues with the model presented in those tutorials.

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.

# Scripting the Hello World to Console

Previously in the basic tutorial we created a model that prints “HELLO” and “WORLD” in that order. In the data tutorial we extended this to add string text names and numerical data values to these items. Now we extend these to show how to write scripters for specific testers. As a reminder, Listing 1 shows the model program that was developed.

public class HelloModel {

private int helloCount = 0;

private int worldCount = 0;

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

private ValueSet<String> worlds =

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

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

private ValueRange<Double> ranges = new ValueRange<Double>(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 ScripterMain {

public static void main(String[] args) {

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

tester.setSeed(345);

tester.addTestEndCondition(new Length(5));

tester.addSuiteEndCondition(new Length(3));

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 (1)

WORLD mars (3.1562892313483015)

HELLO teemu (2)

WORLD mars (1.4289575493440612)

HELLO bob (2)

TEST END

TEST START

HELLO teemu (1)

WORLD venus (3.279034197651822)

HELLO teemu (2)

WORLD mars (2.814722267683214)

HELLO bob (1)

TEST END

TEST START

HELLO teemu (1)

WORLD mars (1.96781339845851)

HELLO bob (6)

WORLD venus (2.7852251942158026)

HELLO bob (1)

TEST END

generated 3 tests.

Figure 1. Example output.

Now we move to a separate scripter as shown in Listing 3.

public class HelloModelWithScripter {

private int helloCount = 0;

private int worldCount = 0;

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

private ValueSet<String> worlds =

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

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

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

**private HelloScripter scripter = new HelloScripter();**

@BeforeSuite

public void init() {

names.setStrategy(DataGenerationStrategy.BALANCING);

}

@BeforeTest

public void startTest() {

helloCount = 0;

worldCount = 0;

**scripter.startTest();**

}

@AfterTest

public void endTest() {

**scripter.endTest();**

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

**scripter.hello(names.next(), sizes.next());**

helloCount++;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

**scripter.world(worlds.next(), ranges.next());**

worldCount++;

}

}

Listing 3. Modified model program with separate scripter.

Of course, we also need the scripter. This is shown in Listing 4.

public class HelloScripter {

public void hello(String name, double size) {

System.out.println("HELLO "+name+" ("+size+")");

}

public void world(String name, double range) {

System.out.println("WORLD "+name+" ("+range+")");

}

public void startTest() {

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

}

public void endTest() {

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

}

}

Listing 4. Separate scripter.

So what’s the point? Now we can modify this to script any format we like in separate classes.

# Scripting Hello World with Velocity

Just writing stuff to the console is not so interesting. So let’s look at writing scripts in a custom format. Now we use Apache Velocity as a template engine. It is also used with OSMO Tester and the included Robot Framework scripter so you can get the jar file from there or from the Velocity website.

Actually we do not need to make any significant changes to the model program. We just modify it to make the scripter pluggable. Listing 5 shows this change.

public class HelloModelWithScripter {

private int helloCount = 0;

private int worldCount = 0;

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

private ValueSet<String> worlds =

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

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

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

**private final HelloScripter scripter;**

**public HelloModelWithScripter(HelloScripter scripter) {**

**this.scripter = scripter;**

**}**

@BeforeSuite

public void init() {

names.setStrategy(DataGenerationStrategy.BALANCING);

}

@BeforeTest

public void startTest() {

helloCount = 0;

worldCount = 0;

scripter.startTest();

}

@AfterTest

public void endTest() {

scripter.endTest();

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

scripter.hello(names.next(), sizes.next());

helloCount++;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

scripter.world(worlds.next(), ranges.next());

worldCount++;

}

}

Listing 5. Model program with pluggable scripter.

That’s pretty much it for this change. But of course wish to see the HelloScripter interface. Find this in Listing 6.

public interface HelloScripter {

public void hello(String name, double size);

public void world(String name, double range);

public void startTest();

public void endTest();

public void write();

}

Listing 6. Pluggable scripter interface.

Note that we had to add the “write” method to allow a more complex scripter to accumulate data and write it to disk at the end. Now the console scripter looks like Listing 6.

public class HelloConsoleScripter implements HelloScripter {

public void hello(String name, double size) {

System.out.println("HELLO "+name+" ("+size+")");

}

public void world(String name, double range) {

System.out.println("WORLD "+name+" ("+range+")");

}

public void startTest() {

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

}

public void endTest() {

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

}

@Override

public void write() {

}

}

Listing 6. Pluggable console interface.

Here the new “write” method does nothing but in the Velocity scripter it does more. This scripter is shown in Listing 6.

public class HelloVelocityScripter implements HelloScripter {

private Collection<TestCase> tests = new ArrayList<TestCase>();

private TestCase test = null;

public void hello(String name, double size) {

TestStep step = new TestStep("hello");

step.addArg("name", name);

step.addArg("size", ""+size);

test.addStep(step);

}

public void world(String name, double range) {

TestStep step = new TestStep("world");

step.addArg("name", name);

step.addArg("range", ""+range);

test.addStep(step);

}

public void startTest() {

test = new TestCase();

tests.add(test);

}

public void endTest() {

}

@Override

public void write() {

String script = giefScript();

System.out.println(script);

}

public String giefScript() {

/\*\* For template->script generation. \*/

VelocityEngine velocity = new VelocityEngine();

/\*\* For storing template variables. \*/

VelocityContext vc = new VelocityContext();

vc.put("tests", tests);

velocity.setProperty("resource.loader", "class");

velocity.setProperty("class.resource.loader.class", "org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader");

StringWriter sw = new StringWriter();

velocity.mergeTemplate("osmo/tester/examples/helloworld/scripter/hello-template.vm", "UTF8", vc, sw);

return sw.toString();

}

}

Listing 6. Pluggable Velocity scripter.

Notice that unlike the console scripter, this one stores the test steps in memory until the end, when the whole script is generated and printed out. In our case only to the console but it could also be printed to a file or any other location. This is where the “write” method is used.

We also have to change the startup of the model to change to the new scripter. This is shown in .

public class ScripterMain {

public static void main(String[] args) {

HelloVelocityScripter scripter = new HelloVelocityScripter();

HelloModelWithScripter model = new HelloModelWithScripter(scripter);

OSMOTester tester = new OSMOTester(model);

tester.setSeed(345);

tester.addTestEndCondition(new Length(5));

tester.addSuiteEndCondition(new Length(3));

tester.generate();

scripter.write();

}

}

Listing 6. Modified main method.

This now modularizes the scripter, and we could just change to another one by changing “HelloVelocityScripter” in the above to “HelloConsoleScripter”.

Let’s not forget the template used to format the stuff itself. This is shown in .

<testsuite>

#foreach ($tc in $tests)

<testcase>

#foreach ($ts in $tc.steps)

<step name="$ts.name">

#foreach ($arg in $ts.args)

<arg name="$arg.name" value="$arg.value"/>

#end

</step>

#end

</testcase>

#end

</testsuite>

Listing 6. Velocity template.

What does all that mean? You could just read the Velocity docs to find out. But basically it starts by iterating all available test cases, taking them one at a time, iterating all steps in each test case, and all arguments for a test step. All of this is then formed to produce a complete XML document (complete as in it has all the required parts, describes all generated tests).

Basically in this template we assume there is a variable named “tests” that contains the test objects. Each is stored in a template variable “tc” one at a time. Test steps for each test case (tc) are then iterated as in a template variable named “ts” one at a time. Finally, same applies for arguments in each test step.

Now we run the test generator using the Velocity scripter and what we get is shown in Figure 2.

Now we created a ValueSet “sizes” that contains integer numbers 1, 2, and 6. Each name is then given a “size” in the printout that is picked from this set. The algorithm to pick one is random choice since no explicit configuration is done.

We also created a ValueRange of double precision floating point numbers called “ranges”, ranging from 0.1 to 5.2. Each world is then given a “range” of a random double value between these bounds.

Running this new model now produces the output shown in Figure 2.

<testsuite>

<testcase>

<step name="hello">

<arg name="name" value="bob"/>

<arg name="size" value="1.0"/>

</step>

<step name="world">

<arg name="name" value="mars"/>

<arg name="range" value="3.1562892313483015"/>

</step>

<step name="hello">

<arg name="name" value="teemu"/>

<arg name="size" value="2.0"/>

</step>

<step name="world">

<arg name="name" value="mars"/>

<arg name="range" value="1.4289575493440612"/>

</step>

<step name="hello">

<arg name="name" value="bob"/>

<arg name="size" value="2.0"/>

</step>

</testcase>

<testcase>

<step name="hello">

<arg name="name" value="teemu"/>

<arg name="size" value="1.0"/>

</step>

<step name="world">

<arg name="name" value="venus"/>

<arg name="range" value="3.279034197651822"/>

</step>

<step name="hello">

<arg name="name" value="teemu"/>

<arg name="size" value="2.0"/>

</step>

<step name="world">

<arg name="name" value="mars"/>

<arg name="range" value="2.814722267683214"/>

</step>

<step name="hello">

<arg name="name" value="bob"/>

<arg name="size" value="1.0"/>

</step>

</testcase>

<testcase>

<step name="hello">

<arg name="name" value="teemu"/>

<arg name="size" value="1.0"/>

</step>

<step name="world">

<arg name="name" value="mars"/>

<arg name="range" value="1.96781339845851"/>

</step>

<step name="hello">

<arg name="name" value="bob"/>

<arg name="size" value="6.0"/>

</step>

<step name="world">

<arg name="name" value="venus"/>

<arg name="range" value="2.7852251942158026"/>

</step>

<step name="hello">

<arg name="name" value="bob"/>

<arg name="size" value="1.0"/>

</step>

</testcase>

</testsuite>

Figure 2. Example output.

Whooppee. Of course we also need all the classes and objects to describe the test case, the test step, and arguments. Check the OSMO Tester examples for these files.

# Conclusions

This tutorial showed how to create custom scripters for OSMO Tester. It was further shown how to modularize these in a nice and shiny way. For further modularization of more model aspects when needed, check the modularization tutorial.

# References

OSMOTester home page, discussion forums & source code: <http://code.google.com/p/osmo/>