Tutorial for Model Modularization

OSMO Tester

MBT tool

v2.2

Teemu Kanstrén

Table of Contents

[Introduction 3](#_Toc315266565)

[Modularizing State 3](#_Toc315266566)

[Modularizing the Test Steps 8](#_Toc315266567)

[Conclusions 10](#_Toc315266568)

[References 10](#_Toc315266569)

# Introduction

This tutorial describes modularization of the model programs in OSMO Tester using simple examples. It builds on the previous basic and data modeling tutorials. The reader should be familiar with these previous 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.

# Modularizing State

Previously in the basic tutorial we created a model that prints “HELLO” and “WORLD” in that order, with some data values associated to each printout. In the previous examples, the complete model program was created all in one (class) file. In this tutorial, we split this into several files and show how the different elements can be merged from different files. Some call this process “flattening” but the terminology is not so important. 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 from the basic tutorial.

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

public class Main {

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.

If you have looked at the scripter tutorial, you have already seen how to create a separate class for scripting and use that as a means to plugin in various scripter implementations. That is one aspect of modularizing the model and is already covered by the scripter tutorial. Now, let’s modularize the state as an object of its own. In this example the state is practically the set of class variables. More generally, it can be thought of as the part of the model program that holds over the different test steps.

So, let’s move out the state as promised. This modification is shown in Listing 3.

public class HelloModularModel {

private SeparateState state = new SeparateState();

@BeforeTest

public void startTest() {

state.reset();

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

}

@AfterTest

public void endTest() {

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

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return state.canHello();

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO "+state.nextName()+"

("+state.nextSize()+")");

state.didHello();

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return state.canWorld();

}

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD "+state.nextWorld()+"

("+state.nextRange()+")");

state.didWorld();

}

}

Listing 3. Model program with separate state.

Of course, we also need the class to hold the state. This is shown in Listing 4.

public class SeparateState {

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);

public SeparateState() {

names.setStrategy(DataGenerationStrategy.BALANCING);

}

public void reset() {

helloCount = 0;

worldCount = 0;

}

public boolean canHello() {

return helloCount == worldCount;

}

public String nextName() {

return names.next();

}

public int nextSize() {

return sizes.next();

}

public void didHello() {

helloCount++;

}

public boolean canWorld() {

return helloCount > worldCount;

}

public void didWorld() {

worldCount++;

}

public String nextWorld() {

return worlds.next();

}

public double nextRange() {

return ranges.next();

}

}

Listing 4. The state class (“model object”).

As we see here, the model is just standard Java code, and what we have actually done here is to extract one concept from the bigger model class to its own class. Of course, this is not always necessary but it is a very good way to separate concerns, increase cohesion, and all those fancy buzzwords. In short, it just makes the code more understandable, maintainable, and so on. But as said, if you don’t like it you don’t need to do it. Some people are just cranky about having any quality in their code.

Running this model provides the output shown in Figure 2.

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 2. Example output.

This should be the same as before, since we just refactored the code to make more sense.

# Modularizing the Test Steps

So we modularized the model state to its own class. Cute. Now let’s see how we can also modularize the rest of it, meaning the test steps. What we do is we put all the “hello” related stuff in one (class) module and all the “world” related in another.

Let’s start with the “hello” part shown in Listing 5.

public class HelloModule {

private final SeparateState state;

public HelloModule(SeparateState state) {

this.state = state;

}

@BeforeTest

public void startTest() {

state.reset();

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

}

@AfterTest

public void endTest() {

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

}

@Guard("hello")

public boolean thisNameReallyIsIrrelevant() {

return state.canHello();

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO "+state.nextName()+"

("+state.nextSize()+")");

state.didHello();

}

}

Listing 5. The “hello” module.

And add to that the “world” in Listing 6.

public class WorldModule {

private final SeparateState state;

public WorldModule(SeparateState state) {

this.state = state;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

return state.canWorld();

}

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD " + state.nextWorld() + " (" +

state.nextRange() + ")");

state.didWorld();

}

}

Listing 6. The “world” module.

So we now have the two things separated. And how do we combine them to create a single model? We initiate OSMO Tester as shown in Listing 7.

public class ModularMain {

public static void main(String[] args) {

**SeparateState state = new SeparateState();**

**OSMOTester tester = new OSMOTester();**

**tester.addModelObject(new HelloModule(state));**

**tester.addModelObject(new WorldModule(state));**

tester.setSeed(345);

tester.addTestEndCondition(new Length(5));

tester.addSuiteEndCondition(new Length(3));

tester.generate();

}

}

Listing 7. The merger.

Notice that here it makes sense to separate state beyond just understandability, maintainability, and code quality. Both modules with test steps can now share the state when we create it separately. And OSMO Tester treats this exactly the same as the initial model we had in Listing 1. OSMO Tester just takes the modules and takes all the annotated parts and data variables from each and merges them all internally into a single model.

We could still take the before and after test parts from the “hello” model and put them separately in another “setup” module if we wanted. But for showing how modularization works as an example this is not necessary as it would be the same process we did here.

And when we run it, we get once again the same output as shown in Figure 3.

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 3. Example output.

That’s it. There is no big magic, OSMO is intended to be simple to use and so we believe it is.

# Conclusions

This tutorial showed how we can modularize different parts of the test model. This is mainly useful to manage larger models and to make models in general easier to understand, maintain and to have higher code quality with less bugs hidden in complexity.

One part that is missing is the creation of a modular scripter, which is shown in the scripting tutorial.

Another part that is not discussed is the use of @Pre and @Post annotations and associating a single guard statement to several test steps (multi-association). The pre- and post-annotations simply define modules that are executed before or after test steps. The association to test steps is done in a way similar to guard statements. The multi-association can be done in a very simple way and is best understood by looking into the OSMO Tester manual and examples.

# References

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