Tutorial for Data Modeling

OSMO Tester

MBT tool

v3.3

Teemu Kanstrén

Table of Contents

[Introduction 3](#_Toc372289263)

[Hello world with names 3](#_Toc372289264)

[Generating numerical data 8](#_Toc372289265)

[Conclusions 10](#_Toc372289266)

[References 10](#_Toc372289267)

# Introduction

This tutorial describes the data modeling concepts of OSMO Tester using simple examples. It skips the background, the theoretical descriptions and just shows some examples. Check the OSMO manual for more lengthy descriptions. The reader should be familiar with the information presented in the OSMO Tester basic tutorial.

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.

# Hello world with names

Previously in the basic tutorial we created a model that prints “HELLO” and “WORLD” in that order. Now we extend that to give some names for greetings with “HELLO” and “WORLD”. As a reminder, Listing 1 shows the model program that was developed.

public class HelloModel {

private int helloCount = 0;

private int worldCount = 0;

@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 there() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO");

helloCount++;

}

@Guard("world")

public boolean earth() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD");

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.addTestEndCondition(new Length(5));

tester.addSuiteEndCondition(new Length(2));

tester.generate(52);

}

}

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

WORLD

HELLO

WORLD

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

generated 2 tests.

Figure 1. Example output.

So, let’s extend this with names for “HELLO” and “WORLD” as promised. A model program implementing a “naïve” approach for this is shown in Listing 3.

public class HelloModel {

private int helloCount = 0;

private int worldCount = 0;

@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 there() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

**String name = "teemu";**

**if (Math.random() > 0.5) {**

**name = "bob";**

**}**

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

helloCount++;

}

@Guard("world")

public boolean earth() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

**String world = "mars";**

**if (Math.random() > 0.5) {**

**world = "venus";**

**}**

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

worldCount++;

}

}

Listing 3. Modified model program with data.

Here we have two options of input data for both the “HELLO” and “WORLD” items. In the first phase the “name” variable gets either the value “teemu” or “bob” with a 50% chance. The “world” variable gets either the value “mars” or “venus” with a 50% chance. As the models are Java programs they can make use of any of the Java programming language features and libraries as shown here.

Running this model provides the output shown in Figure 2.

TEST START

HELLO teemu

WORLD venus

HELLO teemu

WORLD venus

HELLO bob

TEST END

TEST START

HELLO teemu

WORLD venus

HELLO bob

WORLD venus

HELLO teemu

TEST END

generated 2 tests.

Figure 2. Example output.

But this is not very pretty, maintainable or anything. If we want to add many more options it becomes quite a mess. OSMO Tester provides a ValueSet object that can be used to make this better. Using this is illustrated in Listing 4.

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");**

@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 there() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

**System.out.println("HELLO "+names.balanced());**

helloCount++;

}

@Guard("world")

public boolean earth() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

**System.out.println("WORLD "+worlds.random());**

worldCount++;

}

}

Listing 4. Using ValueSet.

This example shows two different ways to request a value from the ValueSet. The “random()” just picks one of the available options on random. The “balanced()” always gives values so that the total number of choices taken for each value is as close to each other as possible. For more options, see the Javadocs.

Running this model now gives the output shown in Figure 3.

TEST START

HELLO bob

WORLD venus

HELLO teemu

WORLD mars

HELLO teemu

TEST END

TEST START

HELLO bob

WORLD venus

HELLO bob

WORLD venus

HELLO teemu

TEST END

generated 2 tests.

Figure 3. Example output.

Or perhaps we just want to have randomly generated strings of ASCII text instead of explicit definition? This is shown in Listing 5.

public class HelloModel3 {

private int helloCount = 0;

private int worldCount = 0;

private Text text = new Text(3, 7).asciiLettersAndNumbersOnly();

@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 there() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO "+text.random());

helloCount++;

}

@Guard("world")

public boolean earth () {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD "+text.random());

worldCount++;

}

}

Listing 5. Using Text.

This creates one data variable called “text” and configures it to produce text with length between 3 and 7 characters and containing only valid ASCII letters and numbers. The output is shown in Figure 4.

TEST START

HELLO VZNV

WORLD jcfbVrX

HELLO J4vmw

WORLD klz

HELLO YpYs

TEST END

TEST START

HELLO E1ThDB1

WORLD HabF

HELLO D7N

WORLD OzjV

HELLO XzuEi

TEST END

generated 2 tests.

Figure 4. Example output.

Beyond just getting different values from a ValueSet, it is also possible to reserve and free items in the set. This is especially useful when we need more than one value from a set to perform an action or a related set of actions (inside a test step or spread over several), while wanting to maintain that each item we get is unique. Take the following as an example:

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

String name1 = names.random();

String name2 = names.random();

In this example, there is no guarantee that name1 and name2 are different. There is a good chance that both are “teemu” or both are “bob”.

However, if we write the following:

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

String name1 = names.reserve();

String name2 = names.reserve();

names.free(name1);

names.free(name2);

Now the first reserve() provides a random value from the set and at the same time reserves it. This means future reserve() calls cannot provide this value any more. Both names are not unique and every time either name1 or name2 will be “bob” and the other one will be “teemu”. The free call later frees the value for future use, making it available for reserve() again. It is also possible to choose which one to reserve with reserve(String name) version of the call.

# Generating numerical data

So far we generated strings. But numbers are also needed. So how do we generate those? Listing 6 shows some examples.

public class HelloModel4 {

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);**

@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 there() {

return helloCount == worldCount;

}

@TestStep("hello")

public void sayHello() {

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

helloCount++;

}

@Guard("world")

public boolean earth() {

return helloCount > worldCount;

}

@TestStep("world")

public void sayWorld() {

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

worldCount++;

}

}

Listing 6. Creating numbers.

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. Random choice is used.

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 5.

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 5. Example output.

Besides these data modeling objects, it is also possible to create ValueRangeSets that categorize the input data into partitions and each partition as well as the overall set can be configured with chosen data generation strategies (algorithms). The TestUtils class from OSMO also provides means to directly generate various types of numerical test data if desired (using the configured OSMO Tester seed). However, it is recommended to create a Randomizer object separately for your model objects, as this allows the generator to better control how randomization is applied across different objects (to produce deterministic results in different configurations). See the OSMO Tester manual and Javadocs (or the source code ☺) for more details on the different model objects and OSMO library routines.

# Conclusions

This tutorial showed the basic aspects of using OSMO Tester to model data in model programs. Beyond these any aspects of the Java programming language and libraries can also be used. The elements shown in this tutorial are the just to support the user in making it easier to create and create data in the models.

# References

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