DDT Reference Documentation

SENACOR
TECHNOLOGIES

January 20, 2008

Contents

1 What is Data Driven Testing? 3
1.1 The Input Problem . . . 3
1.2 Automating Input . . . 3
1.3 Help the Users Help You . . . 4
1.4 Easy-to-use, collaborative data-driven testing . . . 5

2 DDT's Architecture 5
2.1 A matrix of values . . . 5
2.2 ObjectMatrix and StringMatrix . . . 6
2.3 Annotations for DelegatingObjectMap . . . 6
2.4 Embedded Annotations . . . 7
2.5 ObjectMap and BeanFiller . . . 8
2.6 BeanFiller row name syntax . . . 9
2.7 BeanAccessStrategy . . . 9

1 What is Data Driven Testing?

We all love seeing the green bar light up after a test run. It shows us that our code works as we think it should, and we can confidently move on to the next step in our work. Of course, nobody in their right mind will claim that every part of a program is equally well tested for all possible scenarios. 100% test coverage is hardly possible, neither with line-by-line coverage nor with all possible input values. 100% coverage is also unnecessary, if we choose our tests carefully.

1.1 The Input Problem

It's easy to write good tests if you are testing a small unit of functionality - perhaps one method with one or two simple parameters, returning a simple value and possibly performing some kind of side effect. But the more complex your application becomes, the more complex your methods will inevitably become.[1]

Imagine, for example, you have a method that takes only a single parameter: The root of a tree of objects, which represents a central part of your problem domain. Your method is supposed to perform a simple operation with the data in this tree, and so you write a simple test that calls the method and checks the result for correctness. You get a green bar and move on.

But have you tested what happens if the data looks a little different? A value here or there is (legitimately) `null`? Something in a list is not quite in the order which you expect?[2] You come up with a slightly varied test setup and write another quick test to verify your implementation.

In many cases, this is enough. But it does not scale. Rich, complex applications often require equally rich and complex input. It is tedious at best, tiring and error-prone at worst to manually write test cases for each interesting edge case or test setup you might need to verify. Worse, you end up writing almost the same code for each test, because you are still testing that one method, which you always call in the same way. The only things that change between the test cases are the input data and the expected outcome that you are looking for. People who are easily bored (many programmers are, I know I am) tend to become careless with repetitive, tedious tasks. The result: You don't test as much as you should or want to. This is where data-driven testing comes in.

1.2 Automating Input

If the only thing that changes between test cases is setting up the input values and determining the expected outcome, the obvious solution is to just write the actual test code once,

[1] Of course, we all try to keep this complexity down, but most applications contain some inherently complex business logic.

[2] This can easily happen with user-supplied data, for example, or if you read legacy data written by other applications.

and read the input and output data from some kind of data source. This can be something as simple as opening a text file and reading a few lines with `BufferedReader#readLine()`. So you whip up a quick file format, parse it into your test, and off you go to meet the green bar.

Almost anybody's first attempt at this will look somewhat like this: You open your data source, you read line by line of data, slowly assembling the graph of objects you want to run your test on:

```
...
myObject.setFoo(file.read("foo"));
myObject.setBar(file.read("bar"));
myObject.getFoo().setBaz(file.read("baz"));
...
// and so on for 50 more lines...
```

That's a whole lot of code, typing and fragile `String` literals to set up an object tree. Wouldn't it be great if you could just say `fillBean(myObject)`?

This is where DDT comes in.

1.3 Help the Users Help You

Your product is delivered, everybody is happy, until a few weeks later, you get a critical bug report - something doesn't quite work as the customer expected. You have a look at the bug data and scratch your head: This input was never specified anywhere, no wonder it didn't work! But of course the customer insists that this is exactly what he meant when he wrote the specifications.

How can you avoid this? In an ideal world, you'd get good example data from your customer along with the specification. In the real world, you'll be lucky if you even get moderately coherent specs. You have to come up with your own test data, among other things. You almost inevitably will miss something that the user expects to "just work".

Why not make the customer give you usable test data? "Right", you say, "as if those barely literate MBAs could come up with exhaustive data sets, let alone in the format my test code needs!"

You're right about that, of course. But they will still come and nag you with every single bug, so it will quickly pay off to try and help them. Work together with them, helping them not only with specifying what the application should do. Help them with giving you good test data. Give them a tool they are familiar with and friendly descriptions of what you need instead of The Matrix.[3]

[3] Of course your data files are perfectly logical and easy to understand. But you know that whenever you point the customers at something as strange as a programmer's text editor, their eyes glaze over and they only see weird green falling letters, no matter what is actually on the screen.

What tool do the users really know? Any suit worth his necktie knows how to type stuff into Excel sheets. And to be honest, it is nice to quickly whip up some tables of data. Let's keep them happy inside their spreadsheets, and use this to our advantage.

This is also where DDT comes in.

1.4 Easy-to-use, collaborative data-driven testing

DDT combines the solutions to these two problems. It defines a clean, easy to understand test format (the `ObjectMatrix`) and allows you to choose from a number of filetypes you can read, including the Excel spreadsheet files your users love so much. It fills complex object trees with a simple method call. It makes it easy and clean to read the ugly binary blobs that Microsoft calls a file format.[4] And it even runs your tests for you, ensuring that for each test case your code gets exactly the data it needs.

To just get up and running quickly, go have a look at the Quick Start tutorial. The reference documentation you are reading right now will take you through all important concepts and explain how to use all parts of DDT and how they work.

2 DDT's Architecture

This chapter will introduce you to the way DDT is built.

2.1 A matrix of values

The core abstraction DDT uses is a matrix of data cells, each accessible by column and row names. If this sounds suspiciously like a spreadsheet: That's where it comes from.

	narf	zorg
foo	A	B
bar	C	D
baz	E	F

Unlike the spreadsheet, though, DDT never uses integer indices (except internally in the Excel file reader). So in the above example cell A is addressed by `(narf, foo)`, and cell D by `(zorg, bar)`.

Each cell can contain one value of an arbitrary type. Additionally each cell can have an arbitrary number of annotations, which are simple key-value string pairs.

[4] We use Andy Khan's excellent JExcel library for the actual reading. DDT provides a comfortable abstraction for test data on top of that, and on top of other file formats like CSV or Properties.

2.2 ObjectMatrix and StringMatrix

All this is defined by the `ObjectMatrix` interface. This interface offers a number of accessor methods to get values of the common types - numbers, dates, strings, booleans - and the accompanying annotations, which are represented as `java.util.Properties`.

How these values and annotations are read and parsed is implementation-dependent. You could easily provide your own implementation of `ObjectMatrix` to read some kind of esoteric file format, read from a database or any other kind of data source.

DDT provides one implementation of this interface called `DelegatingObjectMatrix`. This class defines all of `ObjectMatrix`' operations on top of a simpler object of type `StringMatrix`.

`StringMatrix` is essentially the same as an `ObjectMatrix`, but it only returns string values. `DelegatingObjectMatrix` then uses Transformers (see later section) to coerce these strings into the desired typed values. This approach has the advantage that only a very simple `StringMatrix` has to be implemented in order to read from various sources such as Excel files, CSV files or Properties files. All the interesting stuff is already done in `DelegatingObjectMatrix`.

If the string returned by the underlying `StringMatrix` is null or blank (i.e. empty or whitespace only) all of `DelegatingObjectMatrix`' accessor methods will return null as the value for this cell. For historical reasons, only `getString()` will return an empty string in this case. If you need the string to be null, or want to make the nulling of another type more explicit, use the null *annotation* (see below).

The default implementation of `StringMatrix` is called (surprise!) `DefaultStringMatrix` and provides yet more common operations. The only thing left to do to read alternate data sources is to implement the interface `StringMatrixReader`, which basically defines a method to read a string at position (x, y).

Most data sources like Excel and CSV files only provide integer-based indices to the columns and rows, but we want name-based indices. `DefaultStringMatrix` reads these names from one column and one row, respectively. By default, it uses the topmost row for column names and the leftmost column for row names, but you can set this via constructor parameters.

2.3 Annotations for DelegatingObjectMap

`DelegatingObjectMap` understands the following annotations:

- `null`
- `ref=otherColumnName`: This is deprecated and only included for historical reasons. It acts as a reference to the value of a cell in the same row, in the given column. This will be replaced by a more general reference construct in a future version.

- `default-value=someValue`: Use this as a row- or column-level annotation to specify a default value for all cells. If no value is found in a cell itself, this default value will be used instead. Use the null annotation to explicitly set a value to null.

2.4 Embedded Annotations

`DelegatingObjectMatrix` does not define where the annotations come from. With a database or other file formats these may come from any kind of external metadata, for example. Usually when using `DefaultStringMatrix`, though, you will have an `EmbeddedAnnotationMatrixDecorator` sitting between the `DefaultStringMatrix` and the `DelegatingObjectMatrix`.

This allows you to specify annotations inline, along with the cell values. The string in a cell can be either:

```
value
```

Which is just the value you want to see. Or it can one of the following:

```
value~annotation-key
value~annotation-key=
value~annotation-key=annotation-value
~~annotation-key=annotation-value
```

The tilde (˜) separates the "normal" cell value from the annotation, and the equals (=) separates the annotation key from its (optional) value. If there is no equals sign, the value will be an empty string. If there is the equals sign but no value, the annotation is *dynamic*, see below for more details. The final form without cell value but starting with a double tilde (˜˜) is a global annotation. Put this form into the naming row or column and this annotation will be shared by all cells in the matrix.

A cell can not only have its own annotation, but also inherit annotations from the containing row or column:

	C1	C2	
R1˜annot1=someValue	K	L	(row 1)
R1	K	L	(row 2)
R2˜annot2=	X	Y	(row 3)
R2˜annot3=value	M	N	(row 4)
R2˜annot4	C	D	(row 5)
R3˜annot5=value	E	F	(row 6)

The simplest case for this is for row "R3": All cells in this row inherit the annotation given in the naming column. So the cell (C1,R3) has the value "E" and the annotation "annot5"→"value".

The row "R1" is more interesting: As you can see, there are actually three of them. In this case, the rows are collapsed into one row, with all annotations being combined into one Properties object, and only the values of the last column "winning", unless there is one such row without an annotation. Confusing? The example will make it all clear in a moment.

The other interesting thing is the *dynamic* annotation, marked by the trailing equals sign. In this case, the value of the annotation will be taken from the cell in the column that we will be looking at.

Alright, let's look at the example.

The cell at (C1,R1):

- has the value "A", because that is the value in row 2 - the only one without annotations
- has the annotation "annot1"→"someValue", because that annotation belongs to the row "R1" as well, and is a fixed-value annotation. The value "K" in column "C1" is discarded, because the rows are collapsed into row 2
- has the annotation "annot2"→"X", because annot2 is a dynamic annotation which takes its value from the column we're looking at, which is C1.

The cell at (C2,R1) is similar to (C1,R1), except:

- the value is "B"
- the annotation "annot2" now has the value "Y"

The cell at (C1,R2)

- has the annotations "annot3"→"" and "annot4"→""
- has the value "C". If no row is present without annotation markings, the last matching row in the matrix wins. In this example, there are two rows (4 and 5) with the name "R2", so row 5 wins.

2.5 ObjectMap and BeanFiller

`ObjectMatrix` can (and is) used for many different purposes, but most of the time it forms the basis for DDT's data-driven testing system. In this system, the dataset for one test is one column in the matrix. This view is provided by `ObjectMap`, which is basically a one-dimensional `ObjectMatrix`, locked to one row or one column. `ObjectMatrix` defines creator methods for `ObjectMaps`.

The most interesting part of DDT (and the simplest to use, once you understand the matrix and the annotations) is the `BeanFiller`, which sits on top of an `ObjectMap` and lets you fill entire object graphs at will.

The central method is `fillBean(beanName, bean)`, which takes your bean and fills it with data from all rows (or columns) whose name starts with the given `beanName`.

The central method is `fillBean(beanName, bean)`, which takes your bean and fills it with data from all rows (or columns) whose name starts with the given `beanName`. There are a few variations of this method, check out the Javadoc for more info.

2.6 BeanFiller row name syntax

`BeanFiller` uses the row names it finds to build the object graph. The syntax is very similar to the usual expression languages like PropertyUtils, JSF EL and JSP. Given the `beanName` "foo", a tree could look like this:

```
foo.someField
foo.someOtherField
foo.aReference.yetAnotherField
foo.anArray[0].someField
foo.anArray[0].someOtherField
foo.anArray[1].someField
foo.anArray[1].someOtherField
foo.aList[0].someField
foo.aSet[0].someField
foo.aSet[0].someOtherField
foo.aMap[bar].someField
```

As you can see, the tree can be arbitrarily deep and contain collections, maps and arrays. The default implementation can read and write both public and private fields and normal JavaBean-style get/set-properties.

It might be surprising to see a `Set` reference with an index notation. This is a faked index - of course a `Set` doesn't have an index itself, but `BeanFiller` provides one so you can address the same object in more than one row, as shown above.

The tree doesn't have to be completely prepared in the bean you pass in - `BeanFiller` will try and instantiate all objects it needs on its way along the graph. In the above example, if `foo.aReference` is null, `BeanFiller` will try to determine the type of `aReference` via reflection and then instantiate an object of the needed type. There are sane defaults for well-known interfaces like `List`, `Map` and `Set`.

Sometimes you will want to specify a different type, or, for example in the case of collections, `BeanFiller` can't find the type via reflection. In this case you need to provide a type annotation (see above on how to give annotations).

2.7 BeanAccessStrategy

The way `BeanFiller` works with your objects is pluggable. If you work with POJOs (aka Java Beans) then you have to do nothing. The default `JavaBeanAccessStrategy` will try hard to get into your objects, both via standard getters and setters, as well as trying direct field access via reflection, even to protected and private fields. This default should work in almost all cases. But in case you have a really weird object model, have a look at the `BeanAccessStrategy` interface and provide an implementation suitable to your setup.