Tuesday, June 26, 2007

A Workbook for Practicing Test-Driven Design

Brian Marick
marick@exampler.com

I've come to think that the best way of learning Test-Driven Design (TDD) is to add on to some code that was developed that way. This package provides such code together with a list of features to add. I hope that teaches not only TDD but also:

1. How to use tests to understand code.

2. How TDD lets you cope with "requirements churn."

3. How a test-driven design can leave you, at the end, with code that *looks* as if it had been cleverly designed with full knowledge of requirements right from the beginning - even though many of the requirements came after the first version was finished.

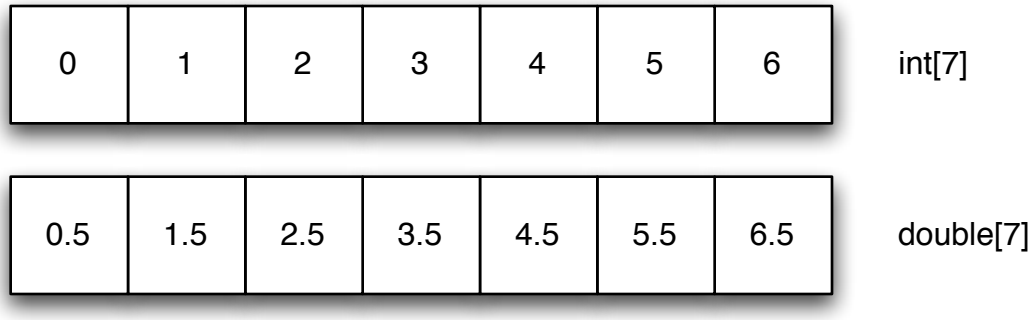
Important: I assume you've already read up on TDD. I've read these books: *Pragmatic Unit Testing* (Hunt and Thomas), *Test Driven Development: By Example* (Beck)*, and Test-Driven Development: A Practical Guide* (Astels). There are many other good books. Here are the top two Google hits for "test-driven design": "Introduction to test driven design" (Ambler) <http://www.agiledata.org/essays/tdd.html> and "Test-driven development" (Wikipedia) <http://en.wikipedia.org/wiki/Test_driven_development>.

You also should know Java and some programming environment. Although I provide an ant script that runs the tests, I *strongly* encourage you to take the time to figure out how to run the tests from within your programming environment. TDD thrives on quick feedback.

Acknowledgements: This code is inspired by the Java interface to the Hierarchical Data Format, an open-source library for storage of large amounts of scientific data, currently supported by The HDF Group <http://www.hdfgroup.org/>. Thanks to Peter Cao and Quincey Koziol for their explanations. This implementation uses no HDF code, and any stupidities in it are my fault, not theirs.

What the package does

This package contains code to convert between the Land of Java and an on-disk data format. As of now, each **member** of a **dataset** is a Java array of numbers. Each array has the same length, called the **dimension**. A 2-member dataset looks like this:



Datasets are defined member-by-member:

```
 int[] ints = {0, 1, 2, 3, 4, 5, 6, 7 };
    double[] doubles = {0.0, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5 };
...
dataset.defineMemberFor(ints);
dataset.defineMemberFor(doubles);
```

The above definition means that the dataset should be prepared to store 4-byte integers and 8-byte floating point numbers to disk. The Java bit layout is replicated on disk.

Java defaults can be overridden. Perhaps all the integers should be stored in three bytes. That can be declared like this:

```
dataset.defineMember(Like.integer(3));
```

Clients of this code can also control byte order:

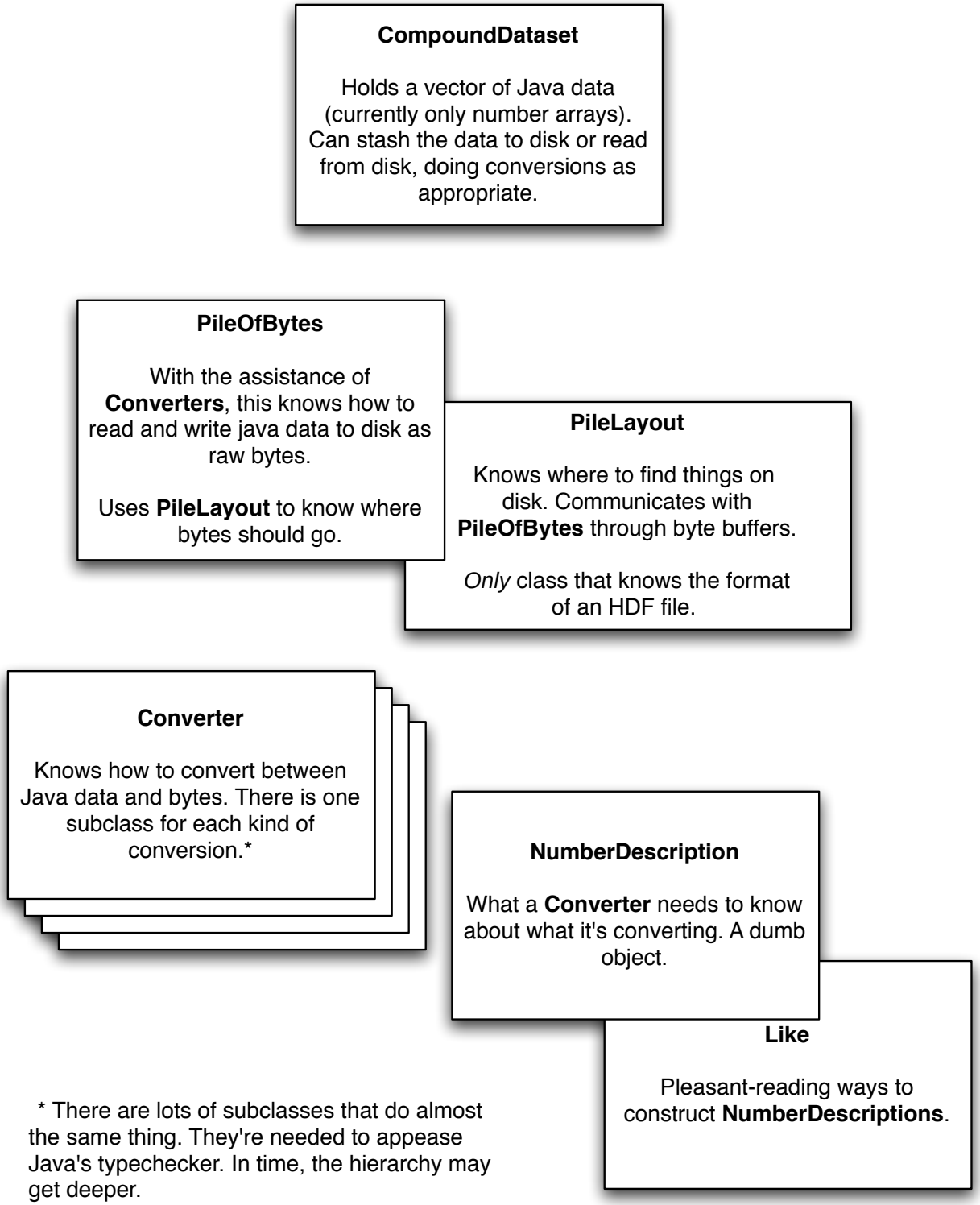
```
dataset.defineMember(Like.integer(3, ByteOrder.LITTLE_ENDIAN));
```

`BIG_ENDIAN` is the default (as it is in Java). For I/O speed, you can also ask the code to use the native ordering, whichever it is.

That's pretty much it. The main classes are sketched on the next page. You should be able to best understand the code in this order:

`CompoundDatasetTests` (then the code in `CompoundDataset`)
`PileOfBytesTest` (then code in `PileOfBytes`)
`PileLayoutTest` (then `PileLayout`)
The various converters and their tests.

Important classes

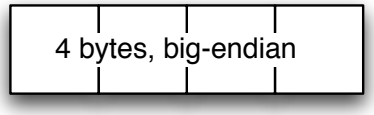


* There are lots of subclasses that do almost the same thing. They're needed to appease Java's typechecker. In time, the hierarchy may get deeper.

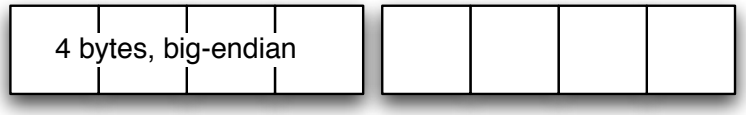
Disk layout

Number of dimensions (each member must have this many elements)

Number of distinct members

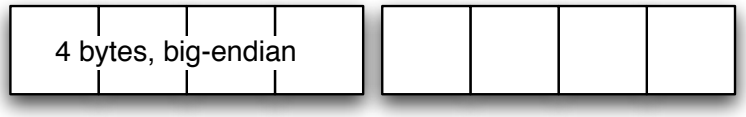


For each member, the type of that member's elements (e.g., INTEGER)



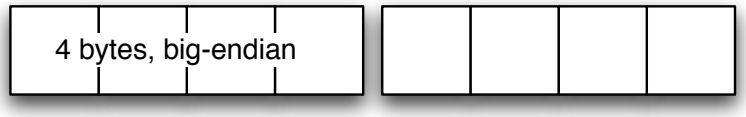
…

For each member, the size (in bytes) of its elements (e.g., 4, 8)



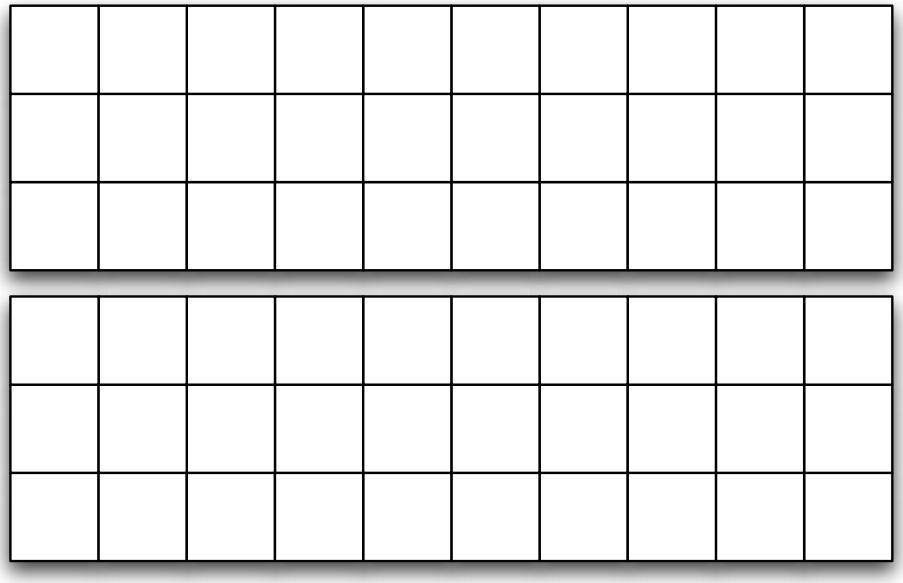
…

For each member, the endianness of its elements (big or little)



…

For each member, a block of bytes large enough to hold all its elements.



Features to add

1. Currently, you can define a member by giving it a number description:

   ```
   dataset.defineMember(Like.integer(4, ByteOrder.BIG_ENDIAN));
   ```

   Or you can define it "by example":

   ```
   dataset.defineMemberFor(new int[50]);
   ```

   Extend, test-first, the latter way so that that sizes and byte orders can be given:

   ```
   dataset.defineMemberFor(new int[50], 3);

   dataset.defineMemberFor(new int[50], 3, ByteOrder.LITTLE_ENDIAN);
   ```

2. Add the ability to read and write unsigned 4-byte integers, something like:

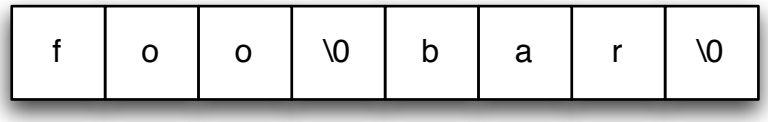
   ```
   dataset.defineMember(Like.unsignedInteger(4));
   ```

   It seems to me that the resulting Java array should be of longs.

3. Add the ability to read and write arrays of Strings. The disk format is a block of *n* C-strings (null-terminated 7-bit ASCII, aka US-ASCII, aka `ISO646-US`). So this:

   ```
   String[] { "foo", "bar" }
   ```

   would turn into this block of bytes:

   Because the array of strings must obey the dimensions of the entire dataset, the code reading strings from disk will know how many null-terminated strings to read.

4. Oops. We changed our mind and want a new disk format for storing strings. This one is as above, but it adds an 8-byte, bigendian prefix containing the number of bytes devoted to the member (not including the prefix). The prefix makes it easier to skip over blocks of code.

   Because we have an influential legacy user, continue to read the old format. However, the code that writes the dataset should only write the new format.

5. We'll change our mind *one last time*. To simplify things further, the `PileLayout` should add a fourth per-member field (not just type, element size, and byte order). It should be an 8-byte bigendian, and it marks the absolute byte index of the beginning

of the data for the member.

Although this really, truly is the *last time* we'll change our mind, maybe it would be a good idea to start the file off with a version number. (Hint: you can assume no existing dataset has enough members to need a left-most 1 bit in the number-of-dimensions field.)

You must continue to support the two older ways to represent string data.

6. Time has passed and there's no more need to support the old way of representing Strings. Remove that code. (Strive to make your code look as if the latest string format had been planned from the beginning.)

7. A new feature: one of the dataset's members can be a complete dataset itself. There is no limit to how many nested datasets there can be.

8. The code as it stands will allow a client to store 8-byte longs into 4-byte unsigned integers. Change the code so that it throws an exception in such a case. Add such sanity checking for all the types (such as string characters that won't fit into 7-bit ASCII). Try to keep the error checking from turning the code into a confusing mass of `if` statements.