

A Pattern Solution to a Meta-Problem

James W. Cooper

Recently, I had to read some files containing document metadata. Metadata, you probably recall, is data *about* the document rather than data that you find directly in a document. Typically metadata contains information like the document's title, date, major topics, source, copyright information and so forth. If you have a whole lot of documents with such metadata, such as a collection of technical or news articles, it is helpful to catalog the metadata so that you can find the documents that fall into whatever categories the metadata provides. Of course, if you have all this metadata, you have to put it into some sort of organized system like a database or search engine index so you can find it when you need it. Here is where we find a little object lesson in a number of good programming techniques.

Let's consider the following hypothetical set of document metadata.

```
Date:20001102
Time:0700
Service:Snow Jones
Topics:Java;Servers;Design Patterns;Factories;
Location:Mountain View;New York;Redmond;
Companies:Fawcette;Sun;Microsoft;IBM;
```

These data consist of tags describing the type of data and values for each tag type. Some data are numeric and some are text data. Some types have multiple values and others do not. We want to write a parser to take these data and store them in tables in a database or a search engine index. One further twist is that we want to store the time and date information in a numeric table so that we can ask for documents that were created in a certain date and time range. You could imagine a few other numeric metadata types as well such as document length or writing level.

So, when we parse these data, we want to put some of them in numeric tables and some in text tables. At first this seems absurdly easy. We just check for the "Time" or "Date" tags and put the data that follows into a numeric table, and put all other data in text tables. So, you could imagine writing code like:

```
public void tableStuffer(String line) {
    int i = line.indexOf (":");
    if(i > 0) {
        String tag = line.substring (0, i);
        String data = line.substring (i + 1);
        if ((tag.equals ("Date")) || (tag.equals("Time")))
            putNumeric(tag, data);
        else
            putText(tag, data);
    }
}
```

But gack! What a terrible piece of code! This certainly won't scale very well, and it hardly is object oriented. In fact, as a general rule, if you have a bunch of *if* tests like this to decide which routine to call, you should start over again and try to design classes that

avoid this kind of testing. Each class should operate on only one kind of data and the class selection should be done without all these ugly if statements whenever possible.

Writing the Indexers

Suppose we write a class to index the text data first. We'll parse it and store the tag-value pairs in a database. This isn't really at all difficult. Here is the entire class:

```
//Indexes text metadata
//and stores it in a database table
public class Indexer {
    protected Databass db;
    public Indexer(Databass datab) {
        db = datab;
    }
    public void addData(String tag, String dataLine) {
        int i = dataLine.indexOf (";");
        while(i > 0) {
            String tval =dataLine.substring(0,i);
            db.addItem (new DBEntry(tag, tval));
            dataLine = dataLine.substring (i+1);
            i = dataLine.indexOf (";");
        }
        if(dataLine.length () >0 ) {
            db.addItem (new DBEntry(tag, dataLine));
        }
    }
}
```

Note that we are using a class called Databass.

```
//An artifishal database
//used to test storage of indexed data
public class Databass {
    private Vector numData, textData;
    public Databass() {
        numData = new Vector();
        textData = new Vector();
    }
    public void addItem(DBEntry dbe) {
        textData.addElement (dbe);
    }
    public void addNumericItem(DBEntry dbe ) {
        numData.addElement (dbe);
    }
    public Vector getNumericData() {
        return numData;
    }
    public Vector getTextData() {
        return textData;
    }
}
```

Since we haven't yet decided on the database we'll need to deploy this system, we are using this fishy substitute that just keeps the name-value pairs in a vector of DBEntry objects. Our DBEntry class just has get and set methods for the name and value.

Since we are using the Databass class as a proxy for a real database that we might introduce later, we could consider this as an example of the Proxy pattern.

The Numeric Indexer

Now we will also need an indexer for the numeric data, and it is even simpler since we can reuse some of the basic Indexer code:

```
//Indexes numeric data
public class NumIndexer extends Indexer {
    public NumIndexer(Databass db) {
        super(db);
    }
    public void addData(String tag, String dataLine) {
        int value = new Integer(dataLine).intValue ();
        DBEntry dbe = new DBEntry(tag, value);
        db.addNumericItem (dbe);
    }
}
```

We also made our DBEntry class do double duty by giving it two constructors and a place to store either text or numeric data. Only one kind can ever exist however, depending on the constructor you call, and one will go in the numeric table in the database and the other in the text table.

```
//holds one name-value pair
//either text or numeric
public class DBEntry {
    private String fieldName, text;
    private int numValue;
    public DBEntry(String field_Name, String t) {
        fieldName = field_Name;
        text =t;
    }
    public DBEntry(String field_Name, int value) {
        fieldName = field_Name;
        numValue = value;
    }
    public String getFieldName() {
        return fieldName;
    }
    public String getText() {
        return text;
    }
    public int getNumValue() {
        return numValue;
    }
}
```

Choosing the Indexer We Need

Now, how do we go about choosing the right indexer, without going through a bunch of if statements? One way is to use a Hashtable with keys for each expected type. In the IndexFactory class below we create an instance of each kind of indexer (here we only have two) and add them into a hash table with their keys. We also add a couple of text tags we know will always be in the metadata files.

```

public class IndexFactory {
    private Hashtable Indexers;
    private Indexer txt;
    private NumIndexer num;

    public IndexFactory(Databass dbass) {
        //keep them in as hash table
        Indexers = new Hashtable();
        txt = new Indexer(dbass);
        num = new NumIndexer(dbass);
        //store them with appropriate keys
        Indexers.put ("Date", num);
        Indexers.put ("Time", num);
        Indexers.put("Companies", txt);
        Indexers.put("Service", txt);
    }
}

```

Then to get the right indexer out of the factory, we just use the tag as the key to the hash table.

```

//get an indexer based in the tag
public Indexer getIndexer(String tag) {
    Indexer ind = (Indexer)Indexers.get (tag);
    //default is to return a text indexer
    if(ind ==null) {
        ind = txt;
    }
    return ind;
}

```

In this implementation, we require that we specifically identify cases where the numeric indexer is needed, and assume that all other cases can be indexed using a text indexer. We can easily add new indexers as we need them by just expanding the entries in the hash table.

The code that does the indexing by reading from the file just amounts to reading each line, finding the tag, asking for the indexer, and calling its addData method:

```

public void readData(String filename) {
    String tag = "";
    String line = "";
    InputFile fl = new InputFile (filename);
    String s = fl.readLine ();
    while(s != null ) {
        int i = s.indexOf (":");
        if(i > 0) {
            tag = s.substring (0,i).trim();
            line = s.substring (i+1).trim();
            Indexer ind = ixFactory.getIndexer (tag);
            ind.addData(tag, line);
            s = fl.readLine ();
        }
    }
}

```

This IndexFactory class is, of course, an example of the Factory Pattern.

Displaying the Result

We need to be able to inspect our work and see that we have done it correctly. So, I put together a simple graphical program to display the text and numeric data, as shown in Figure 1 and 2.

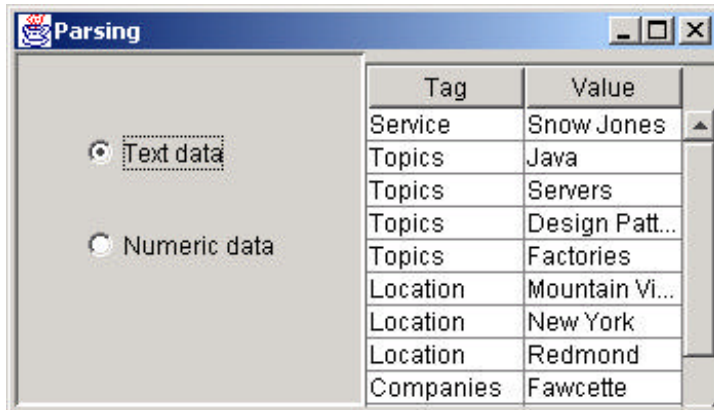


Figure 1 – The text values indexed

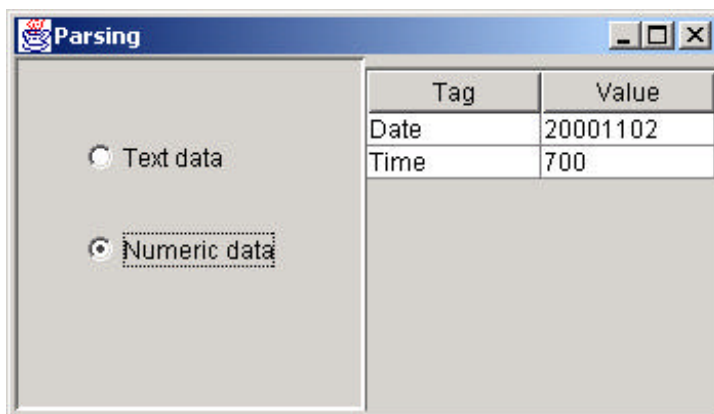


Figure 2 –The numeric values indexed

But we now have some new, interesting problems to solve. When we click on the two radio buttons, we again have the possibility of a couple of if statements, along the lines of

```
public void actionPerformed(ActionEvent ev) {
    Object obj = ev.getSource();
    if(obj == TextRadio)
        //show text data
    if (obj == NumRadio)
        //show numeric data
}
```

This is just as bad an idea here in the user interface code as it was in the indexer choosing code. And as we have done in the past, we can do away with this kind of decision, by making each radio button know about what it has to do. We derive `TextRadio` and `NumRadio` from the `JRadioButton` class, but also have each implement the `Command` interface:

```
public interface Command {
    public void Execute();
}
```

```
}
```

Then, whatever the radio buttons need to do when clicked can be put inside those Execute methods. Here is the TextRadioButton class:

```
public class TextRadio extends JRadioButton implements Command {
    private Mediator md;

    public TextRadio(Mediator med) {
        super("Text data");
        md = med;
    }
    public void Execute() {
        //do something to display data
    }
}
```

Putting the Data in the Table

You will note that I left out what exactly the radio button has to do in the TextRadioButton class above, because we have another problem to solve here. The data we are going to display in the JTable on the right side of the display has to get there somehow. With a JTable this happens by setting a different TableModel into the JTable class. So to display the text data, for example, we need to write a class that implements the AbstractTableModel class. Fundamentally, it has to implement the getValueAt method as well as the getColumnCount and getRowCount methods. Here is the class for the Text display:

```
public class TextModel extends AbstractTableModel {
    private Databass db;
    protected Vector data;
    public TextModel(Databass dbase) {
        db = dbase;
        data = db.getTextData ();
    }
    public java.lang.Object getValueAt(int row, int column) {
        if ((row < data.size ()) && (row>=0)) {
            DBEntry dbe = (DBEntry)data.elementAt (row);
            switch (column) {
                case 0:
                    return dbe.getFieldName ();
                case 1:
                    return dbe.getText ();
                default:
                    return "";
            }
        }
        else
            return "";
    }
    public int getColumnCount() {
        return 2;
    }
    public int getRowCount() {
        return data.size ();
    }
}
```

```

public java.lang.String getColumnName(int column) {
    switch(column) {
        case 0:
            return "Tag";
        case 1:
            return "Value";
        default:
            return "";
    }
}
}

```

Now we begin to see the scope of our problem. The Execute method for each of the radio buttons needs to set an instance of the TableModel into the JTable, and this implies that the radio button derived class needs to have instances of the correct table model, the JTable and probably the Databass class. This seems like a lot of clutter, especially since it has to be duplicated in both of the radio button classes.

A Mediator for our Problem

A better solution is for another class to keep track of these objects and pass them to the table as needed. Then the two radio buttons need only call the correct method on that class. This “other” class is a Mediator class, because it mediates between three user interface objects and two TableModel classes. It also turns out to be a good place for the data to be read in and placed in the database in the first place. Our complete Mediator class looks like this:

```

public class Mediator {
    private Databass db;
    private IndexFactory ixFactory;
    private JTable jtb;
    private NumModel numModel;
    private TextModel textModel;

    public Mediator(Databass dbase, JTable jtab) {
        db = dbase;
        jtb = jtab;
        ixFactory = new IndexFactory(db);
        textModel = new TextModel(db);
        numModel = new NumModel(db);
    }

    public void readData(String filename) {
        String tag = "";
        String line = "";
        InputFile fl = new InputFile (filename);
        String s = fl.readLine ();
        while(s != null ) {
            int i = s.indexOf (":");
            if(i > 0) {
                tag = s.substring (0,i).trim();
                line = s.substring (i+1).trim();
                Indexer ind = ixFactory.getIndexer (tag);
                ind.addData(tag, line);
                s = fl.readLine ();
            }
        }
    }
}

```

```

    }
}
public void setTextData() {
    jtb.setModel (textModel);
}
public void setNumericData() {
    jtb.setModel(numModel);
}

```

Note the two methods, `setTextData` and `setNumericData`, These place the correct table model in the `JTable`, and are the methods called by our radio button's `Execute` method.

```

public void Execute() {
    md.setNumericData();
}

```

Now the only class that the radio buttons need to know about is the `Mediator`, and it takes care of all the heavy lifting and keeps classes from unnecessary knowledge about each others internal workings.

Conclusions

We started out with a simple parsing problem what we wanted to solve in the best OO fashion we could think of. We ended up using a `Proxy` pattern (for our database) a `Factory` pattern for our `Indexers`, a `Command` pattern for the `Radio` buttons, a `Mediator` pattern for loading the `JTable`, and in fact also are using an `Observer` pattern, because the `JTable` observes changes in the `TableModel` and displays them. So again, design patterns are all around us, and these really are ones we use pretty much every day to write better, more scalable code.