Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Using Metadata

Martin Fowler

I occasionally come across people who describe their programming tasks as tedious, which is often the sign of a design problem. One common source of tedium is pulling data from an external source. You almost always do the same thing with the data, but because the data differs each time, it's difficult to reduce such tedious programming. This is when you should consider using metadata.

To illustrate the approach, consider a simple design problem: build a module that will read data out of a simple file format into memory. One example of this file is a tab-delimited format with the first line containing the names of the fields (see Table 1).

```
class ExplicitReader...
       public String FileName;
       TextReader reader;
       static char[] SEPARATOR = {'\t'};

       public ExplicitReader (String fileName) {
               FileName = fileName;
       }
       public IList ReadBatsmen() {
               IList result = new ArrayList();
               reader = File.OpenText (FileName);
               reader.ReadLine(); //skip header
               String line;
               while ((line = reader.ReadLine()) != null) {
                       String[] items = line.Split(SEPARATOR);
                       Batsman bat = new Batsman();
                       bat.Name = items[0];
                       bat.Matches = Int32.Parse(items[1]);
                       bat.Innings = Int32.Parse(items[2]);
                       bat.Runs = Int32.Parse(items[3]);
                       result.Add(bat);
               }
               return result;

       }
}
public class Batsman...
       public String Name;
       public int Matches;
       public int Innings;
       public int Runs;
```

Figure 1. A simple, explicit solution for reading data from a tab-delimited file.

Explicit and implicit reads

Figure 1 offers perhaps the most straightforward approach to this problem—reading each column of data into a record structure. As a program, it's pretty simple, because it's easy to read and to write. Trouble rears, however, if you have a lot of files to read. You have to write this program for each file, which is a tedious job, and tedium usually has a bad smell—indicating worse troubles. In this case, the trouble would be duplication—always something worth avoiding.

Table 1

Actual listing 1

Name	Matches	Innings	Runs
DG Bradman	52	80	6,996
RG Pollock	23	41	2,256
GA Headley	22	40	2,190
H Sutcliffe	54	84	4,555
AC Gilchrist	31	44	2,160
E Paynter	20	31	1,540
KF Barrington	82	131	6,806
ED Weekes	48	81	4,455
WR Hammond	85	140	7,249

```
public class ImplicitReader...
        public String FileName;
        TextReader reader;
        static char[] SEPARATOR = {'\t'};

        public ImplicitReader (String fileName) {
                FileName = fileName;
        }
        public IList Read() {
                IList result = new ArrayList();
                reader = File.OpenText (FileName);
                IList headers = parseHeaders();
                String line;
                while ((line = reader.ReadLine()) != null) {
                        result.Add(parseLine(headers, line));
                }
                return result;
        }
        IList parseHeaders() {
                IList result = new ArrayList();
                String[] items = reader.ReadLine().Split(SEPARATOR);
                foreach (String s in items) result.Add(s);
                return result;
        }
        IDictionary parseLine (IList headers, String line) {
                String[] items = line.Split(SEPARATOR);
                IDictionary result = new Hashtable();
                for (int i = 0; i < headers.Count; i++)
                        result[headers[i]] = items[i];
                return result;
                }
```

Figure 2. An implicit design solution for reading in data from multiple files.

```
abstract class AbstractReader {
        public AbstractReader (String fileName);
                FileName = fileName;
        }
        public String FileName;
        protected TextReader reader;
        protected static char[] SEPARATOR = {'\t'};

        public IList Read() {
                IList result = new ArrayList();
                reader = File.OpenText (FileName);
                skipHeader();
                String line;
                while ((line = reader.ReadLine()) != null) {
                        String[] items = line.Split(SEPARATOR);
                        result.Add(doRead(items));
                }
                return result;
        }
        private void skipHeader() {
                reader.ReadLine();
        }
        protected abstract Object doRead (String[] items);
        }
class ExplicitReader2 : AbstractReader ...
        public ExplicitReader2 (String fileName) : base (fileName){}
        override protected Object doRead(String[] items) {
                Batsman result = new Batsman();
                result.Name = items[0];
                result.Matches = Int32.Parse(items[1]);
                result.Innings = Int32.Parse(items[2]);
                result.Runs = Int32.Parse(items[3]);
                return result;
        }
```

Figure 3. An explicit design that uses substitution on the variable part of the program.

Figure 2 offers one approach to avoiding this tedium, a generic way to read in any data from a file. The advantage is that this single program will read in any file, providing it follows the general format. If you have a hundred of these kinds of files to read, writing a single program like this takes a lot less effort than writ-

```
public class ReflectiveReader ...
        public String FileName;
        TextReader reader;
        static char[] SEPARATOR = {'\t'};
        public Type ResultType;

        public ReflectiveReader (String fileName, Type resultType) {
               FileName = fileName;
               ResultType = resultType;
        }
        public IList Read() {
               IList result = new ArrayList();
               reader = File.OpenText (FileName);
               IList headers = parseHeaders();
               String line;
               while ((line = reader.ReadLine()) != null) {
                      result.Add(parseLine(headers, line));
               }
               return result;
        }
        IList parseHeaders() {
               IList result = new ArrayList();
               String[] items = reader.ReadLine().Split(SEPARATOR);
               foreach (String s in items) result.Add(s);
               return result;
        }
        Object parseLine (IList headers, String line) {
               String[] items = line.Split(SEPARATOR);
               Object result = createResultObject();
               for (int i = 0; i < headers.Count; i++) {
                      FieldInfo field = ResultType.GetField((String)headers[i]);
                      if (field == null)
                             throw new Exception ("Unable to find field: " + headers[i]);
                      field.SetValue(result, Convert.ChangeType(items[i],field.FieldType));
               }
               return result;
       }
       Object createResultObject() {
               Type[] constructorParams = {};
               return ResultType.GetConstructor(constructorParams).Invoke(null);
       }
}
```

Figure 4. A reflective programming design.

ing an explicit program (as in Figure 1) for each file.

The problem with this generic style is that it produces a dictionary, which is easy to access (especially when your language supports a simple index mechanism as C# does) but is not explicit. Consequently, you can't just look at a file declaration to discover the possible fields you must deal with, as you can with the Batsmen class in Figure 1. Furthermore, you lose all type information.

So, how can you have your explicit cake while eating only a small amount of code? One approach is to parameterize the assignment statements from Figure 1 by enclosing them in a single substitutable function. Figure 3 does this with the object-oriented style of an abstract superclass. In a more sophisticated programming language, you could just pass the block of assignment statements in as a function argument. By parameterizing the assignment statements, you can reduce duplication. You can also reduce—but not eliminate—the tedium. All those assignments still must be written, both for reading and writing (if you are supporting both). However, by taking advantage of the metadata in both the target class and file structure, you can avoid writing any assignments at all.

The metadata is available in two forms. The field heading at the top of the data file is a simple metadata that supplies the field names (XML tag names give the same information). If the target class's fields match the data file's names (or if we can make them match), we have enough to infer the assignments. If we can query the target class's metadata, we can also determine the types for the target class's fields. This lets us handle the type conversions properly.

Two ways of using the metadata

We can use the metadata in two ways: *reflective programming* and *code generation*. The reflective programming approach leads us to a program that uses reflection at runtime to set field values in the target class (see Figure 4). Many modern platforms provide this kind of runtime reflection. The resulting reader class can read any file that conforms to the format and has a matching target class.

The code generation style aims to generate a class that's similar to the hand-written one in Figure 3. We can

```
public class ReaderGenerator ...
        String DataFileName;
        Type Target;
        String ClassName;
        TextWriter output;
        public void Run() {
                Console.WriteLine(output);
                output = new StringWriter();
                writeClassHeader();
                writeConstructor();
                writeDoRun();
                writeClassFooter();
                Console.WriteLine(output);
                writeOutputFile();
        }
        void writeClassHeader() {
                output.WriteLine("using System;");
                output.WriteLine("namespace metadata");
                output.WriteLine("{");
                output.WriteLine(String.Format("class {0} : AbstractReader ", ClassName));
                output.WriteLine("{");
        }
        void writeClassFooter() {
                output.WriteLine("}");
                output.WriteLine("}");
        }
        void writeConstructor() {
                output.Write(String.Format
                       ("\t public {0} () : base (\"{1}\")", ClassName, DataFileName));
                output.WriteLine("{}");
        }
        static char[] SEPARATOR = {'\t'};
        void writeDoRun() {
                output.WriteLine("\toverride protected Object doRead(String[] items) {");
                output.WriteLine(String.Format ("\t\t{0} result = new {0}();", Target));
                writeFieldAssignments();
                output.WriteLine("\t\treturn result;");
                output.WriteLine("\t}");
        }
        void writeFieldAssignments() {
                TextReader dataReader = File.OpenText (DataFileName);
                String[] headers = dataReader.ReadLine().Split(SEPARATOR);
                dataReader.Close();
                for (int i = 0; i < headers.Length; i++) {
                       FieldInfo field = Target.GetField((String)headers[i]);
                       if (field == null)
                             throw new Exception ("Unknown Field: " + headers[i]);
                       output.WriteLine(String.Format(
                             "\t\t result.{0} = ({1})Convert.ChangeType(",
                             headers[i], field.FieldType));
                       output.WriteLine(String.Format(
                             "\t\t\titems[{0}],typeof({1}) );",
                             i, field.FieldType));
                }
        }
        void writeOutputFile() {
                StreamWriter outFile = new StreamWriter(File.Create(ClassName + ".cs"));
                outFile.Write(output);
                outFile.Close();
        }
}
```

Figure 5. A generator.

use the style presented in Figure 1, because we don't have to worry about duplication in the generated code. Figure 5 shows the kind of class we could use to perform the generation, and Figure 6 shows the resulting class. Although I'm using the same language in this case, there's no reason why the generator must be the same language as the class it's generating—scripting languages often make good languages for generation due to their powerful string handling.

The generator also uses the language's reflection capabilities to deter-

Figure 6. Example code that Figure 5 generated.

```
using System;
namespace metadata
{
class ExplicitReader3 : AbstractReader
{
public ExplicitReader3 () : base ("batsmen.txt"){}
        override protected Object doRead(String[] items) {
                 metadata.Batsman result = new metadata.Batsman();
                 result.Name = (System.String)Convert.ChangeType(
                        items[0],typeof(System.String) );
                 result.Matches = (System.Int32)Convert.ChangeType(
                        items[1],typeof(System.Int32) );
                 result.Innings = (System.Int32)Convert.ChangeType(
                        items[2],typeof(System.Int32) );
                 result.Runs = (System.Int32)Convert.ChangeType(
                        items[3],typeof(System.Int32) );
                 return result;
        }
}
}
```

mine the field types; however, it does it at compile time rather than at runtime. The generated classes don't use the language's reflection capabilities.

Given these two styles of metadata-based programs, the obvious question is when to use each style. The reflective program offers a single compact class to carry out the mapping. There are, however, some disadvantages. Many people find reflection somewhat hard to use, and it might defeat some of your environment's tooling, such as intelligent reference searches and automated refactorings. In addition, in some environments, reflective calls can be significantly slower than direct method calls.

Generation also has its problems. You need discipline to ensure that developers don't hand-edit the generated files. You must also ensure that generation is done with every significant change—the best way of doing this is to make it part of an automated build process. With many files, generation might lead to a larger code bulk, which might affect footprint and build times. I usually prefer generation to reflective programs, but you have to weigh your decision based on your concerns.

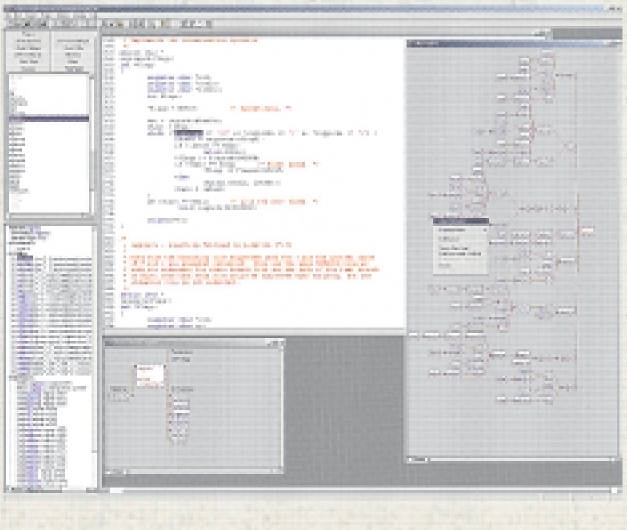
There's also the question of whether to use metadata-based techniques at all. For something like this, I wouldn't bother for a few classes. I'd just use a technique to separate the varying code from the constant code. I can't give a hard number for when it's better to use metadata—it's more a reflection of the degree to which the assignment's monotony is affecting development.

Martin Fowler is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. Contact him at fowler@acm.org.

Reverse Engineering Tools

Our tools help developers understand, document, and maintain impossibly large or complex amounts of source code

They parse **Ada 83, Ada 95, FORTRAN 77, FORTRAN 90, FORTRAN 95, K&R C, ANSI C** and **C++,** and **Java** source code to reverse engineer, automatically document, calculate code metrics, and help your engineers understand, navigate and maintain source code that has grown too large for one person (or even a group) to know



Key Features:

- Fast on big projects
- Quick and easy to use—no complicated or fussy setup, immediately useful
- PERL/C/C++ API for custom reports/documentation
- Automatic creation of graphics and documentation
- Export to common graphics formats and Visio
- Cross reference everything in source
- Variety of hierarchical and graphical views (including With Trees, Call Trees, Include Trees, Extended-By Trees, Ada Structure Graphs, and many others)
- Code colorizing source editor and printing
- Rapid code navigation and editing

Supported Languages:

- Ada 83 and Ada 95
- Java
- ANSI C, K&R C, and C++
- FORTRAN 77, 90, 95
- Can create custom languages on request

Supported Languages:

- Windows 95, 98, NT 4.0, 2000, XP
- Linux (Intel)
- Solaris
- HP-UX
- SGI Irix
- Alpha (DSF)

Big projects aren't a problem. The tools can parse and later manipulate very large amounts of code. 1,000,000 SLOC projects (and larger) are common among our customers.

We also focus on exceptional customer support based on rapid response from a real engineer

Download and try on your code:

http://www.scitools.com/

All downloads are fully functional, just time limited.

Scientific Toolworks, Inc.
info@scitools.com
(802) 763-2995