Generating Java and XML Using StringTemplate

Terence Parr - University of San Francisco

Introduction

Most programs that emit source code or other text output are unstructured blobs of generation logic interspersed with print statements. The primary reason is the lack of suitable tools and formalisms. The proper formalism is that of an **output grammar** because you are not generating random characters--you are generating sentences in an output language. This is analogous to using a grammar to describe the structure of input sentences. Rather than building a parser by hand, most programmers will use a parser generator. Similarly, we need some form of ``unparser generator" to generate text. The most convenient manifestation of the output grammar is a **template engine** such as **StringTemplate**, the engine used in this article.

This small article demonstrates how a Java program may dump out its own field/method interface as Java text using reflection and then generate the interface in XML--all without changing the generation logic. **StringTemplate**'s distinguishing characteristic is that it strictly enforces separation of logic and template which, in the code generation world, means that code generators are guaranteed to be retargetable because you cannot embed any logic in the output templates. For a formal treatment of templates and model-view separation, see **Enforcing Strict Model View Separation in Template Engines**.

Before presenting a template-based solution, let me begin by showing a straightforward, but undignified and unsatisfying solution to this text generation problem. Using Java's reflection API, a Java class can dump itself by traversing and prints the results of methods **Class.getFields()** and **Class.getMethods()**. Here is the core of such a program; it only prints out the fields (output strings are highlighted in orange):

```
import java.lang.reflect.*;

 public class Entangled {

     public int i;
     public String name;
     public int[] data;

     public static void main(String[] args) throws Exception {
            Class c = Entangled.class;
            Field[] fields = c.getFields();
            Method[] methods = c.getDeclaredMethods();
            System.out.println("public class " + c.getName() + " {");

            // dump the fields first
            for (int f = 0; f < fields.length; f++) {
                Class type = fields[f].getType();
                Class arrayType = type.getComponentType();
                String typeS = type.toString();
                if ( arrayType != null ) {
                    typeS = arrayType.toString() + "[]";
                }
                System.out.println("  public " + typeS + " " +  fields[f].getName()
+ ";");
            }

            // dump the methods [ omitted ]
            System.out.println("}");
        }
    }
```

The output looks like:

```
public class Dump {
  public int i;
  public class java.lang.String name;
  public int[] data;
}
```

There are two obvious problems with this common generation approach:

1. the output structure is not at all obvious just from looking at the code even when I have highlighted the output strings in orange.
2. the generator logic and output statements are completely entangled.

Because of entanglement, this program cannot easily be modified to generate different output, such as XML, even though the data model and generation logic would be identical. In the next section, I reimplement the same generator, but using **StringTemplate**. The template-based solution is not only more readable, but it is trivially retargetable as demonstrated in a subsequent section.

Generation of Java Using StringTemplate

Our goal is to generate Java field and method signatures as text strings, which naturally follow the Java syntax. For example, the documentation for this program might describe its output ``by example" such as the following (where again, I have highlighted elements of the output language in orange).

```
class name {
    fields
    methods
}
```

(simplified) fields look like:

```
public type name;
```

and then (simplified) method signatures:

```
public returnType name(arguments);
```

If you associated names with these "exemplars" and called them templates or rules, it would look very much like an output grammar. So, rather than write code that generates output in this format, let's use the templates as **executable documentation**. For example, instead of the above field description, one could write (in **StringTemplate** notation):

```
field() ::= "public <type> <name>;"
```

In reality, the template will be a bit more complicated because types are not just class names--they can be arrays of types also. The **field** template will pass type information to a **type** template that handles the difference in an effort to factor the templates into smaller easier-to-understand pieces.

Here is the complete group of formal templates needed by our generator. I apologize in advance for not exhaustively explaining the syntax and semantics here, but the general idea should be clear (and you can look at the **StringTemplate** documentation for more information).

```
group Java;


class(name,fields,methods) ::= <<
class <name> {
  <fields:field(); separator="\n">
  <methods:method(); separator="\n">
}
```

```
    >>


    field() ::= "public <type(t=it.type)> <it.name>;"


    method() ::= <<

    public <it.returnType> <it.name>(<it.parameterTypes:{<type(t=it)> arg<i>};
separator=", ">);

    >>


    type(t) ::= <<

    <if(t.componentType)><t.componentType>[]

    <else><t.name><endif>

    >>
```

The one key construct at work is that of **template application**, which is similar to python's **map** operation. For example, in the **class** template, **<methods:method()>** means ``apply **method** template to each **Method** Java object in the incoming **methods** attribute." Attribute **it** is the name of an iterated value; templates applied to multi-valued attributes refer to **it** to access each consecutive element of the list. Template reference looks like a macro invocation with named arguments so **type(t=it.type)** sets argument **t** of template **type** to the iterated **Field** object's **type** property (access via **Field.getType()**). The construct works as you'd expect with the caveat that tests the presence or absence of a value in attribute **t**'s **componentType** property.

The Java code needed to use these templates is very simple. It only has to load the template group from a file (called **Java.stg**, in this case), make an instance of a **class** template, stuff data into that template, and ask the template to print itself out:

```
    StringTemplateGroup templates =

     new StringTemplateGroup(new FileReader("Java.stg"),
AngleBracketTemplateLexer.class);

    StringTemplate classST = templates.getInstanceOf("class");

    classST.setAttribute("name", c.getName());

    classST.setAttribute("fields", fields);

    classST.setAttribute("methods", methods);

    System.out.println(classST);
```

where the fields and methods are obtained as above in the entangled generator. The full program is provided at the end of this article. The output generated by this program is:

```
    class Dump {

      public int i;

      public java.lang.String name;
      public int[] data;

      public void main(class java.lang.String[] arg1);

      public class java.lang.Class class$(java.lang.String arg1);

      public void foo(int arg1, float[] arg2);

      public class java.lang.String bar();

    }
```

Each realm of this implementation, logic and view, is fully encapsulated making it easy to change one without affecting the other. Equally importantly templates are much easier to understand than a bunch of print statements. In the next section, I show how to dump the fields and methods as XML rather than Java code by swapping in a new template file; that is, the output will become XML instead of Java without touching the actual program.

Generating XML

Now imagine that you would like output in XML format instead of the Java syntax field and method signatures. For example, you would like classes to be defined as follows:

```
  <class>

    <name>name</name>

    fields

    methods

  </class>
```

For the **Dump** class, you would like the XML to look like:

```
<class>

    <name>Dump</name>

    <field>

        <type>int</type>

        <name>i</name>

    </field>

    <field>

        <type>java.lang.String</type>

        <name>name</name>

    </field>

    <field>

        <type>int[]</type>

        <name>data</name>

    </field>
    <method>

        <returnType>void</returnType>

        <name>foo</name>

        <arg><type>int</type><name>arg1</name>

        <arg><type>float[]</type><name>arg2</name>

    </method>

    <method>

        <returnType>class java.lang.String</returnType>

        <name>bar</name>

    </method>

    <method>

        <returnType>void</returnType>

        <name>main</name>

        <arg><type>class java.lang.String[]</type><name>arg1</name>

    </method>

</class>
```

The code already pulls data from Java's reflection model and pushes it into a template called **class** so one only needs to change the definition of **class** and the output will be different--all without modifying the code.

Because the angle-brackets used previously for template attribute expressions use the same delimiters as XML tags, **$...$** delimiters make more sense; you'll see below that an argument to the **StringTemplateGroup** constructor determines how **StringTemplate** locates attribute expression. Here are the templates that will dump this XML:

```
group XML;


class(name,fields,methods) ::= <<

<class>

  <name>$name$</name>

  $fields:field(); separator="\n"$

  $methods:method(); separator="\n"$

</class>

>>


field() ::= <<

<field>
```

```
      <type>$type(t=it.type)$</type>

      <name>$it.name$</name>

    </field>

    >>


    method() ::= <<

    <method>

      <returnType>$it.returnType$</returnType>

      <name>$it.name$</name>

      $it.parameterTypes:{<arg><type>$type(t=it)$</type><name>arg$i$</name>};
separator="\n"$

    </method>

    >>


    type(t) ::=
"$if(t.componentType)$$t.componentType$[]$else$$t.name$$endif$"
```

As you can see, the only difference between this group and the Java template group is that the XML templates surround the template attributes with XML tags instead of Java syntax.

Complete Java/XML Generator Program

Here is the complete listing of the **Dump** class that can generate its interface in Java or XML format via a command-line argument. Note the lack of output strings in the code, making it possible to retarget this code generator without modifying the code. By default **Dump** dumps its interface in XML.

```
     import org.antlr.stringtemplate.*;

     import org.antlr.stringtemplate.language.*;

     import java.lang.reflect.*;

     import java.io.*;


     public class Dump {


        // some dummy fields

        public int i;

        public String name;

        public int[] data;
```

```
       public static void main(String[] args) throws Exception {
        //StringTemplateGroup templates = new StringTemplateGroup(new
FileReader("Java.stg"), AngleBracketTemplateLexer.class);

        StringTemplateGroup templates = new StringTemplateGroup(new
FileReader("XML.stg"), DefaultTemplateLexer.class);

          Class c = Dump.class;
          Field[] fields = c.getFields();
          Method[] methods = c.getDeclaredMethods();
          StringTemplate classST = templates.getInstanceOf("class");
          classST.setAttribute("name", c.getName());
          classST.setAttribute("fields", fields);
          classST.setAttribute("methods", methods);
          System.out.println(classST);
       }

       // dummy methods
       public void foo(int x, float[] y) {;}
       public String bar() {return "";}
     }
```

Summary

Translators and other code generator programs should use output grammars for generating text because any output structure conforms to a language. The most convenient output grammar manifestation is a template. While many template engines exist, most do not enforce strict separation of model and view, therefore, allowing fully entangled templates/translation logic. Strict separation supports retargetable code generators because there can be no logic in the template and no output phrases in the code.

StringTemplate distinguishes itself by its strict model-view separation and is freely available under the BSD license at **http://www.stringtemplate.org**.