

Transforming Java by completely separated phases using ANTLR and StringTemplate

By Oliver Zeigermann¹

This short introductory paper will explain how a combination of the ANTLR² parser generator and the StringTemplate³ template engine can be used to transform Java sources to a new format. Such a transformation can be seen as a translation from one language to another. In such a translation there is an analysis phase that extracts information from the source language – in our Java – and a synthesis phase that uses this information to construct something new. The analysis phase is handled by the parser generator ANTLR together with a standard Java grammar including a tree walker⁴. ANTLR will allow us to create a complete Java parser from that grammar that will reveal the structure of the Java sources we wish to transform⁵. Generally, one would generate the output format by traversing this structure using the tree walker in the synthesis phase.

However, when working in a team, this requires complex compiler design and ANTLR knowledge both for the analysis as well as for the synthesis part. The basic idea here is to decouple both phases even more in order not to require the synthesis people to have any compiler design knowledge. This can be achieved by adding an intermediate step that generates simple Java-Beans as a result of the analysis phase. This is a well known way to represent data that all Java developers should be familiar with – at least much more familiar than with ANTLR abstract syntax trees. Another good thing about Java-Beans is that they are a natural input for the StringTemplate template engine. With StringTemplate and Java-Beans there is no need to traverse or iterate over any data structures as this can very easily be done declaratively.

The rest of this paper will show how this can work, guided by a practical example. To understand it you will need at least basic knowledge of ANTLR⁶ and a general idea of how source-to-source translations work.

The Use case: Generating EJBs from Java Interfaces

Our example deals with transforming Java interfaces to EJB Session Bean delegates which simply call POJO⁷ implementations of these interfaces. The idea behind this is to develop and test services in a straight forward POJO way and finally deploy them as EJBs for better remoting, scalability and fail-safety. Next to the central Session Bean we will need descriptors, and EJBHome/EJBObject interfaces. We will concentrate on the Session Bean, though.

¹ Oliver be reached at oliver@zeigermann.de

² <http://www.antlr.org/>

³ <http://www.stringtemplate.org/>

⁴ That grammar can be found at <http://www.antlr.org/grammar/1093454600181/java15-grammar.zip>

⁵ Why not simply using Java reflection or Javadoc? In fact both ways allow you to easily handle quite a range of problems. But when every aspect of a Java source is important – such as all import statements, which is the case in our example – they simply are not powerful and flexible enough.

⁶ An introduction to ANTLR can be found here

<http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html>

⁷ POJO = Plain Old Java Object, used to make clear no EJB is meant

Another problem is that such a delegate is only useful if the client code that uses this service can be configured to either do a direct call or a remote EJB one. This can be achieved with the *Spring Framework*⁸ which – in theory⁹ – will allow the client code simply not to care which type of call it does. We will not go into details here, but have a look at figure 1 for an illustration.

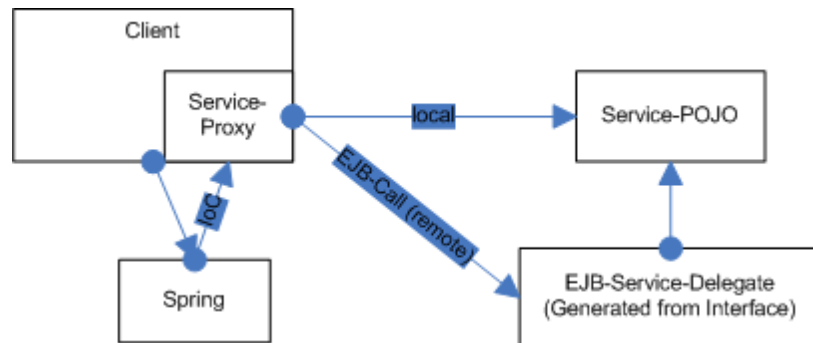


Figure 1: Using Spring to configure calls

The analysis phase: Parsing Java

Generation of the EJB delegate works by parsing the Java service interface with the named standard ANTLR Java grammar and processing the generated AST – abstract syntax tree – that reveals the structure of the interface with the tree grammar that comes along with it. The tree grammar has been augmented by actions that simply create Java-Bean classes containing general information about the interface and its methods. Imagine this is a method declaration of your interface:

```
boolean isMale(Person id) throws ServiceException;
```

The Java parser will generate an appropriate AST structure for this which we can then traverse in a tree rule. The plain rule taken from `java.tree.g` that has not been augmented with any action code would look like this:

```
methodDecl
  : #(METHOD_DEF modifiers
      typeSpec
      methodHead)
  ;
```

It simply traverses a tree with the `METHOD_DEF` token as the root node which contains information about the methods modifiers (e.g. `public`, `final`, `static`), its type and the method head including the methods name, parameters and exceptions. What we have now is a Java-Bean that has properties for all this information:

```
public class Method {
```

⁸ <http://www.springframework.org/>, intro at <http://www.theserverside.com/articles/article.tss?l=SpringFramework>

⁹ Which is not quite possible, unfortunately, as there still are restrictions caused by the cost of each remote call and most important the fact that EJB calls pass objects by value and local calls by reference.

```

    List<String> modifiers = new ArrayList<String>();
    Type type;
    String name;
    List<Parameter> parameters = new ArrayList<Parameter>();
    List<String> exceptions = new ArrayList<String>();

    /* Accessor code has been omitted */
} 10

```

Now we augment the rule to create and populate such a *Method* bean:

```

methodDecl
{
    Method method = new Method();
    Type type;
    List<String> m;
}
: #(METHOD_DEF m=modifiers { method.setModifiers(m); }
    type=typeSpec { method.setType(type); }
    methodHead[method])
  { getInterface().getMethods().add(method); }
;

```

While the method's type and its modifiers are set directly, the *methodHead* rule adds its name, parameters and exceptions in a very similar fashion. We omit the rule for brevity as it does not add any new concepts.

The other beans needed are an Interface Bean, a Parameter Bean and a Type Bean. For a transformation that needs information about the full method definition, not only its declaration you would need beans for expressions, statements, variable declarations, control-structures, etc. as well.

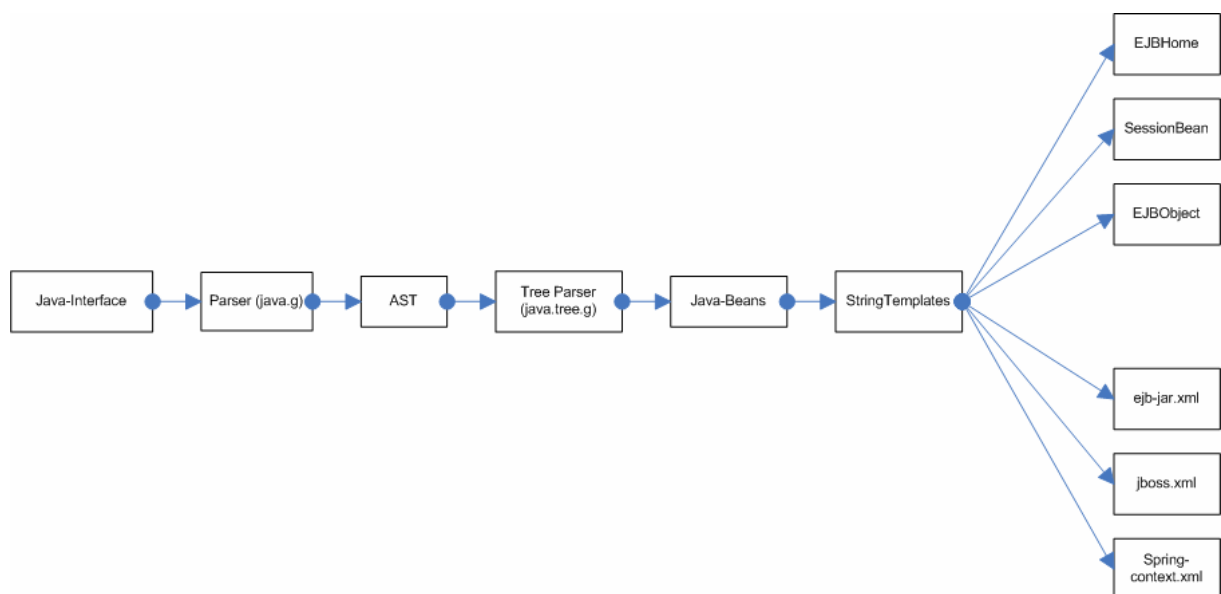


Figure 2: Source Generation Data Flow

¹⁰ We are using Java 5 syntax here even though the original code works with 1.2

Anyway, we restrict ourselves to declarations and use these beans as input data for the StringTemplate template engine to generate what ever we like. An alternative and naïve approach would be to simply generate the EJB while walking the Java structure. You could do this by concatenating Strings and returning them from one rule to another and finally dump it or directly print it out as you go along. As already mentioned the benefit of using Java-Beans and StringTemplate is to enable people without any idea of how Java parsing or ANTLR might work to adjust the generated sources. They will have to know the simple StringTemplate template language, yes, and they will have to understand how to use the Java-Bean classes, but they will not have to mess around with ANTLR¹¹ tree parsers.

An overview of the whole transformation process is displayed in *figure 2*.

The synthesis phase: Generating the Session Bean

The analysis phase is followed by the synthesis phase that uses these beans to fill in the templates for files to be generated. Imagine this is your source interface

```
package test.service;

import test.bo.Person;
import test.core.ServiceException;

public interface PersonService {
    boolean isMale(Person id) throws ServiceException;
    Person populate(Person id) throws ServiceException;
}
```

The corresponding Session Bean delegate might look like this

```
package test.service;

import test.bo.Person;
import test.core.ServiceException;

import javax.ejb.*;

public class PersonServiceSessionBean implements SessionBean,
    PersonService {

    private PersonService service;

    public void setService(PersonService service) {
        this.service = service;
    }

    public boolean isMale(Person id) throws ServiceException {
        return service.isMale(id);
    }
}
```

¹¹ Other reasons to use StringTemplate and separate the logic from the presentation include source-to-source translations with multiple targets. In that scenario you can change the target or add new ones by supplying different sets of templates. A complete example of a source-to-source translation having multiple targets be found here: http://www.codegeneration.net/tiki-read_article.php?articleId=77

```

    public Person populate(Person id) throws ServiceException {
        return service.populate(id);
    }

    /* SessionBean implementations have been omitted */
}

```

StringTemplate

You will notice that this really is straight ahead and just needs a mechanism to fill in the information gathered from the interface in the analysis phase. As already mentioned, StringTemplate is used for this task. A simple template to output the package of the Interface could look like this:

```
package $package$;
```

Gaps marked by `$...$` are filled with values set from Java glue code:

```
template.setAttribute("package", getInterface().getPackageName());
```

As StringTemplate knows how to deal with beans an alternative way to put it would have been to let the template know about the interface bean and get the `packageName` property inside the template:

```
template.setAttribute("interface", getInterface());
package $interface.packageName$;
```

This works as all non primitive data types are considered beans and properties of them are accessed in the well know `<beanName>.<property>` notation.

Templates can be instantiated directly from a string

```
StringTemplate template = new StringTemplate("package $package$;");
```

from a file – e.g. called `template.st` - that literally contains the template

```
StringTemplateGroup group = new StringTemplateGroup("ejb");
StringTemplate template = group.getInstanceOf("template");
```

or by a special format that allows more than one template definition per file. This becomes very handy and helps to reduce the number of template files when you have quite a lot of rather small templates. More about this can be found in the StringTemplate documentation at <http://www.stringtemplate.org/doc/doc.html>

Finally templates are dumped by simply converting them to a string:

```
System.out.println(template.toString());
```

Now we want to add all imports of an interface to the output. Imports are stored as a list of strings. As promised when there is more than one value, there is no need for explicit iteration as StringTemplate handles lists of values specially. This template would concatenate the values of every element in the list:

```
$imports$
```

However, this is not quite what we want as all imports would be in the same line. Adding a separator declaration including the needed semicolon fixes this:

```
$imports; separator=";\n"$
```

Finally, we need the import keyword in front of all imports. This works by applying a specified sub template to every element of the list and concatenating the results:

```
$imports:{n| import $n$}; separator=";\n"$
```

The anonymous template `{n| import n}` gets passed each value as a parameter called *n*. This concept can informally be compared to closures in - say Groovy - or anonymous classes in Java. Just as well as anonymous templates you can use named ones which are defined somewhere else. This statement would use the template named *import* to format each element instead.

```
$imports:import(); separator=";\n"$
```

All this can amount to complex templates like the one that turns a method declaration like

```
boolean isMale(Person id) throws ServiceException;
```

into the delegating method

```
public boolean isMale(Person id) throws ServiceException {
    return service.isMale(id);
}
```

that is part of the Session Bean we have seen above:

```
public $method.type$ $method.name$($method.parameters:parameterDecl();
separator=", "$) throws $method.exceptions; separator=", "$ {
    return service.$method.name$($method.parameters:{n| $n.name$};
separator=", "$);
}
```

Static text has been marked bold and the gaps have been set cursive to make the template more readable. Note that it uses both named as well as anonymous sub templates, but no new concepts apart from the ones described above have been introduced. Here we have access to single valued properties of a bean like in

```
$method.type$
```

and

```
$method.name$
```

as well as an anonymous sub template

```
$method.parameters:{n| $n.name$}; separator=", "$
```

that simply gets the names of the Parameter beans that are in a list. Finally, we have the application of a named sub templates to list valued properties including the specification of a separator:

```
$method.parameters:parameterDecl(); separator=", "$.
```

The sub template *parameterDecl* looks like this:

```
$if(parameter.final)$final $endif$$parameter.type$ $parameter.name$
```

It uses the only logic construct available in StringTemplate to check if the parameter is final. If so the type/name pair is preceded by the *final* modifier. It should be obvious that this *if* statement is very useful to enforce the separation of logic and presentation. Without it, we would have been forced to do this check within the Java glue code moving presentation into it.

Summary

It has been shown that it is easy to create a fully customized Java parser that reveals all information about Java sources with the ANTLR parser generator once you have learned its use and syntax. To use this information to generate any kind of textual output the StringTemplate template engine can be very useful. It allows you to fully separate logic from presentation while remaining easy to learn and intuitive to use. By the introduction of an additional intermediate Java-Bean layer it is possible to fully decouple the parsing from the target generation code. This makes complex and rarely found parsing and compiler design knowledge obsolete for the developer who takes care of the generation part.

Full source of the example including all necessary libraries and an ant build script can be found at <http://www.zeigermann.de/genEJB.zip>.