Code Generator User Guide

Introduction

Codejen Framework is a source code generation framework that allows application developers to write their own code generators. This document tries to guide the users of the framework (the application developers) how to create a simple code generator. Afterwards, the developers can create their own the code generator, codegen config, and templates for specific application.

This document introduces the concept of code generator, describes the configuration to use the framework, goes through the code generator creation process with readers, and describes the specification of the default implementation.

Fundamental Principle of Code Generation: Template + data model = output[1]

There are two elements you need to generate a piece of source code. They are the templates and data model. A template is a piece of document that describing the similar structure of your output. A template has fields to specify the position so that you can put in your data inside. Such data is known as the data model. Take a look at the following JavaBean template as an example:

```
package ${class.packageName};

class ${class.name} {


<#list aKeys as attrName>
    private ${attributeSet[attrName]} ${attrName};
</#list>

<#list aKeys as attrName>
    public ${attributeSet[attrName]} get${attrName?cap_first}(){
        return this.${attrName};
    }

    public void set${attrName?cap_first}(${attributeSet[attrName]}
${attrName}){
        return this.${attrName} = ${attrName};
    }

</#list>
}
```

Package of the bean

Name of the bean

Fields of the bean

Getters and setters of the bean

The template above describes a JavaBean that has a package name, a class name, property fields, property getters and property setters (**bold** font). However, it depends on your data model (grey font), to generate the source code of the JavaBean.

The idea looks simple. However, writing a code generator is more than apply a model to a template. Unlike dynamic page templates like JSP or PHP, a code generator generates a semi-finished product (if not the full system). For example, to generate a Java EE web application, the code generator may need to generate JSP, struts-config.xml, Struts Forms, Struts Action, BO, DAO, Hibernate hbm files etc. If it is a web service, JSP and Struts components is no longer necessary, the code generator should generate deploy.xml for Apache Axis.

In the other words, different application will have different code generation configuration (codegen config). Handling it with a simple template engine is possible, but it may be very hard to maintain. The author of Codejen Framework have about three year experience developing source code generator. Codejen Framework is the framework for the author to develop his third generation of code generator. The framework is simple and flexible (lightweight) for code generator development. The following sections will give the reader the step-by-step guideline to write a code generator.

[1] **Template + data model = output** is a chapter title of Freemarker Manual

Framework Configuration

For quick start, it is recommended to download the all-in-one package. It has the following third-party library dependencies:

- Apache Jakarta Commons BeanUtils (http://jakarta.apache.org/commons/beanutils/)
- Apache Jakarta Commons Digester (http://jakarta.apache.org/commons/digester/)
- Apache Jakarta Commons Logging (http://jakarta.apache.org/commons/logging/)
- FreeMarker (http://freemarker.sourceforge.net/)
- Mozilla Rhino (http://www.mozilla.org/rhino/)

FreeMarker and Mozilla Rhino dependency can be removed by downloading the standalone package of Codejen core framework library.

To use Codejen framework, just put the dependency libraries into the class path (run-time) or build path (compile time).

Your First Code Generator

A code generator is composed of three parts:

1. Model factory – creates a model from a file (driver mode) or look up a model from an existing application (plug-in mode).
2. Codegen config – specifies what template is going to be used, the output of the source code. If the model is used to generate many files, it configuration will also need to know how to extract the elements from the model and how to name the output files.
3. Templates – specifies the result of the output. The syntax is depends on the template engine used. The common template engines in Java are Apache Jakarta Velocity, Eclipse JET, and FreeMarker.

In this section, we are going to build a code generator that simply gets the model from a properties file. The properties file uses the class name as the key. The value specifies whether it is a class or an interface:

```
A=interface
B=class
C=interface
```

The code generator will generate

- a class or an interface for each entry
- code generation report that summarizes the result

Model Factory

The model factory simply creates an instance of code generator with the codegen config, loads the properties file as the model, and prints the exceptions:

```
// imports are snipped

public class Main {
    public static void main(String[] args) throws Exception{
        CodeGenerator generator = new CodeGenerator("config.xml");
        Properties prop = new Properties();
        InputStream in = new FileInputStream("model.properties");
        prop.load(in);
        generator.setModel(prop);
        generator.run();
        List<Throwable> errors = generator.getErrors();
        for (Throwable error : errors) {
            error.printStackTrace();
        }
    }
}
```

Load the codegen config.

Properties file as the model.

Prints the errors, if exist.

Code Generation Configuration

The codegen config specifies the templates to be used as well as the output path. Like Apache's Ant builder, the XML used does not have a rigid DTD or schema. Generally, the structure of the XML likes the following table[2].

Element	attr[], elements	Description
config	(property | include | template)*	Root element of codegen config.
property	attr[key, value]	Property of the config. It can be used in ${key} syntax.
include	attr[file]	Another config file to be included in the current codegen config.
template	attr[class, ISA], modelExtractor, fileNameGenerator, postProcessor*	A template specification.
modelExtractor	attr[class, ISA]	Extract the element from the data model.
fileNameGenerator	attr[class, ISA]	Generate the file name of each element.
Postprocessor	attr[class, ISA]	Process the generated output.

ISA in the table means **I**mplementation **S**pecific **A**ttributes. ISA allows codegen developers to specify the parameters creating their templates, post processors, model extractors and file name generators. For the detail, please refer to More Sophisticated Code Generator.

Code Generation Report

Generating the report is a one-to-one model. A model is going to generate one and only one output. The following codegen config generates the code generation report:

```
<config>
    <property key="rootDir" value="F:/java/workspace/codejen/src/sample" />
    <property key="templateDir" value="${rootDir}" />
    <property key="jsDir" value="${rootDir}" />
    <property key="outputDir" value="${rootDir}/output" />
    <include file="${rootDir}/class-config.xml" />
    <template class="org.sf.codejen.freemarker.FreeMarkerTemplate"
              templateDir="${templateDir}"
              templateFile="report.ftl"
              outputDir="${outputDir}"
              outputFile="report.txt"
              modelName="map"
              canOverwrite="true">
    </template>
</config>
```

${rootDir} will be expanded to its value.

Includes the config for generating classes and interfaces.

ISA are the properties of FreeMarkerTemplate

[2] The second column of the table **attr[], elements** denotes the attribute within the square brackets, "**[]**", and the nesting element of the current element. "|" within brackets, "**()**", means it can be either one element. "*" means the specified element is optional and can be repeated.

As described in the table above, ${key} of the property element will be expanded to its value. When the value of **outputDir** is **${rootDir}/output**, it means **outputDir** property is **F:/java/workspace/codejen/src/sample/output** in this configuration.

For the ISA of the template, the JavaDoc of org.sf.codejen.freemarker.FreeMarkerTemplate specifies the property of the implementation. All properties can be used as the attributes in the element.

Class and Interface

The following codegen config (class-config.xml) generates the classes and the interfaces:

```
<config>
    <template class="org.sf.codejen.freemarker.FreeMarkerTemplate"
              templateDir="${templateDir}"
              templateFile="javaclass.ftl"
              outputDir="${outputDir}"
              modelName="entry"
              canOverwrite="true">
        <modelExtractor class="org.sf.codejen.js.JsModelExtractor"

    script="${jsDir}/modelExtractor.js;javascript:extractModel(map)"
                        modelName="map" />
        <fileNameGenerator class="org.sf.codejen.js.JsFileNameGenerator"
                           script="javascript:entry.key + '.java'"
                           modelName="entry" />
    </template>
</config>
```

- model name is **map**
- Load modelExtractor.js
- Run extractModel(**map**)

Generating classes and interfaces is a one-to-many model. One model is going to produce many files. In this case, an entry in properties file is going to be a class or an interface. We need a model extractor and a file name generator to set up the model and the name of the output file.

The script property of **JsModelExtractor** and **JsFileNameGenerator** shares the same behavior. It loads (and compiles) the scripts separated by OS path separator (";" for Windows, ":" for Linux). The text after **javascript:** will be treated as interpretable script which will be executed directly.

The script in **modelExtractor.js**:

```
function extractModel(map) {
    var iter = map.entrySet().iterator();
    var result = new java.util.ArrayList();
    while (iter.hasNext()) {
        result.add(iter.next());
    }
    return result;
}
```

Templates

In Codejen Framework, FreeMarker is used as the default template implementation[3]. The template of the report will like this:

```
<#function article noun>
    <#if noun?lower_case?starts_with('a') ||
         noun?lower_case?starts_with('e') ||
         noun?lower_case?starts_with('i') ||
```

[3] Codejen Framework can be configured to use other template engines, even within the same codegen config.

```
          noun?lower_case?starts_with('o') ||
          noun?lower_case?starts_with('u')>
          <#return 'an'>
     <#else>
          <#return 'a'>
     </#if>
</#function>
Source Code Generation Report
==============================
<#assign keySet = map?keys>
<#list keySet as key>
${key} is ${article(map[key])} ${map[key]}.
</#list>
==============================
${keySet?size} files are generated.
```

The template for generating classes and interface is much simpler:

```
public ${entry.value} ${entry.key} {
}
```

Output

Four files will be generated under the output directory.

report.txt

```
Source Code Generation Report
==============================
A is an interface.
C is an interface.
B is a class.
==============================
3 files are generated.
```

A.java

```
public interface A {
}
```

B.java

```
public class B {
}
```

C.java

```
public interface C {
}
```

More Sophisticated Code Generator

In most cases, you only need to specify the **modelExtractor** and the **fileNameGenerator**. However, **postProcessor** may required for situation like source code formatting (a.k.a. beautification) or further syntax checking. The **postProcessor** element can be nested within the **template** element like this:

```
        <postProcessor class="org.sf.codejen.js.JsTemplateProcessor"
                             script="formatting.js;javascript:format(tpl)"
                             modelName="tpl" />
```

Default Implementation Specifications

To make the framework more ready to use, the default template specification, the model extractor, the file name generator, and the template processor are already implemented with FreeMarker and Mozilla Rhino (a JavaScript engine).

Template Specification

Using the all-in-one framework library, **org.sf.codejen.freemarker.FreeMarkerTemplate** can be used with the following implementation specific attributes (ISA):

ISA	Required	Default	Description
templateFile	Yes	N/A	The path of template file. Relative to **templateDir**.
outputFile	Yes when no **modelExtractor** is nested within the **template** element.	N/A	Filename of the generated file. Relative to **outputDir**. If **modelExtractor** does not exist and **outputFile** is empty, no file will be generated.
templateDir	No	working directory	The directory to load the template.
outputDir	No	working directory	The output directory of the source codes generated.
modelName	No	model	The model name used within the template. Accessed by ${model}.
canOverwrite	No	true	Indicate whether the generated output should overwrite the existing file.

Model Extractor, File Name Generator, and Template Processor

Model extractor, file name generator, and template processor (used in postProcessor) in Mozilla Rhino implementation are **org.sf.codejen.js.JsModelExtractor**, **org.sf.codejen.js.JsFileNameGenerator**, and **org.sf.codejen.js.JsTemplateProcessor** respectively. They share the same ISA:

ISA	Required	Default	Description

script	Yes	N/A	The path of the script files to be used separated by OS dependent path separator. i.e. "**;**" for Windows, "**:**" for Linux/Unix.
			If it is only a one line script, **javascript:** can be used to execute the script directly. For example:
			`javascript: entry.key + '.java'`
			Furthermore, script files and script can be mixed together like:
			`${jsDir}/modelExtractor.js;`

			javascript:extractModel(map)

modelName	No	model	Name of the context variable. For example, if **modelName** is **entry**, the **script** attribute will be
			`javascript: entry.key + '.java'`
			If **modelName** is **model**, the **script** attribute will be
			`javascript: model.key + '.java'`