Document Generator 3.0 Component Specification

1. Design

The approach used for generating code from the template file is using XSL transformations. The first version of Document Generator used plain XSL files as user editable template and provided an API for generating XML data for them and applying XSL transformations. Unfortunately the XSL files are rather complicated (and messy) with all those XSL tags. A user not familiar with XSL will have serious problems in doing some significant modifications and even an experienced user will run into a lot of syntax errors without some specialized editing tool. That's why I've used as template files, simple text files that use % marked tags for fields and applied a preprocessing step to it to obtain a regular XSL file. The conversion is really trivial. A header needs to be prepended to the text and a footer is appended. Each field tag gets replaced by a XSL value-of tag.

The template uses loop constructs for repeating parts of the template. These loops can be nested and can create very complex templates. Despite this complexity, everything is easy for us because we just have to replace each start of loop tag with a for-each XSL tag and each end of loop tag with the ending for-each tag.

These simple steps give us the power of XSL transformations with reasonably looking text template files. We don't need to take care of parsing XML data, merging template with data, because everything is done by the industry standard APIs, Xerces and Xalan.

As the design is concerned, flexibility was one of the main goals. The process of template transformation is all abstracted using a strategy pattern through the Template and TemplateData interfaces. That means that XSL transformations and XML input can be completely swapped out and replaced with Velocity templates for example. The default implementations of are XsltTemplate and XmlTemplateData but they can be swapped out by simply replacing the static attribute values from DocumentGenerator.

Another flexible thing is the source that is used to store templates. The TemplateSource interface provides an interface which abstracts any storage source that can retrieve the text of a template based on a name. There is an implementation for the file system that stores templates as files. The power of the approach comes from the configurability of the template sources. If you look in the sample configuration file you'll see how the application can use simultaneously one file system source, two future database sources (one Oracle, one MySQL), a future implementation of a CVS source, and a LDAP source. Each source has an identifier and the API allows us to specify where the template should be retrieved from.

The user interface is implemented using DocumentGenerator as façade. Under normal usage, only the method from this class should be used. This class is a manager for the template sources and also coordinates the template transformation process.

There are also a few classes that support the API input requirement. They are based on composite pattern and they represent a structure of fields, loops representing the mapping of the corresponding tags from a template (a template consists of fields and loops, a loop may contain fields and other loops). This API allows changing field values, adding loop iterations and configuring the data for each in an intuitive manner.

Version 3.0 Major Changes:

a) The DocumentGenerator class is changed to acts as a pure API class, configuration and command line feature are moved to other classes, TemplateSources are programmatically configurable

b) DocumentGeneratorCommand class is added, it's used by command line

c) Dependency on Config Manager is removed, Configuration API is used instead

d) DocumentGeneratorFactory is used to create fully configured DocumentGenerator instance by taking an instance of ConfigurationObject

e) More command line arguments are provided

f) The configuration file, template source file, xml data file can be in jar archieve files.

1.1 Design Patterns

Strategy for the template transformation technique

Strategy for the template source

Factory for DocumentGeneratorFactory

1.2 Industry Standards

XML data files and XSL transformations are used for code generation.

1.3 Required Algorithms

1.3.1 The template format

The template format is explained in the following sample. It contains fields marked with %. A field has a name, a description (comments, sample data) and a default value. The template can contain loops marked with %loop:name% … %endloop%. Loops can be nested as in the example below. The template may contain comments (marked with #). The \ character is used as escape char similar to Java.

```
# This is a sample template.
#
# - A comment line starts with #.
# - If # is needed at the start of a line, then \# should be used.
# - The character # can be used freely after the start of the line without applying the previous rule.
# - The template fields have the syntax %fieldname=defaultValue{description}% where =defaultvalue is optional
# - If %, {, =, \ are needed as actual text then \%, \{, \=, \\ should be used instead (similar to \ in Java).
# - Looping constructs can be done using the loop special field.
# - Everything between %loop:field_name% and %endloop% will be repeated as many times as
#   the field_name appears in the input XML.
# - Loops can be nested.

Hello %CODER_HANDLE{The handle of the coder}%,

This email details the %PROJECT_TYPE{Design/Development}% Review Board assignments for the
week of %CURRENT_DATE:6/11 - 6/17%

Application Dates: Screening %SCREENING_DATE{m/dd format ie 6/11}%, Review %REVIEW_DATE%,
                   Aggregation %AGGREGATION_DATE%, Final Review %FINAL_REVIEW_DATE%

%loop:PROJECT{loop for each project}%

Component: %PROJECT_NAME%

%loop:REVIEWER%
%REVIEWER_ROLE%: %REVIEWER_HANDLE% $%REVIEWER_PAYMENT%
```

%endloop%

Product Manager: %PM_HANDLE%

%endloop%

As always, please let us know if you will be unavailable to participate in reviews in the coming weeks.

Thank you,

TopCoder Software

1.3.2 The XML format (used for input data to the templates)

Below is an XML file that can be used with the template from the previous section. There is a root element always named DATA. Each field in the template has a corresponding leaf element with the value that should replace that field in the output. Each loop in the template has a corresponding composite element. That element contains other leaf and composite elements, as the loop from the template may contain fields and loops. Each nested loop corresponds to a level of depth in the XML data.

```
<DATA>
     <CODER_HANDLE>someone</CODER_HANDLE>
     <PROJECT_TYPE>Design</PROJECT_TYPE>
     <CURRENT_DATE>9/4</CURRENT_DATE>
     <SCREENING_DATE>9/5 - 9/11</SCREENING_DATE>
     <REVIEW_DATE>9/12 - 9/16</REVIEW_DATE>
     <AGGREGATION_DATE>9/17 - 9/19</AGGREGATION_DATE>
     <FINAL_REVIEW_DATE>9/20 - 9/25</FINAL_REVIEW_DATE>
     <PROJECT>
             <PROJECT_NAME>HTTP Utility 1.0</PROJECT_NAME>
             <REVIEWER>
                     <REVIEWER_ROLE>Primary/Failure</REVIEWER_ROLE>
                     <REVIEWER_HANDLE>handle1</REVIEWER_HANDLE>
                     <REVIEWER_PAYMENT>100</REVIEWER_PAYMENT>
             </REVIEWER>
             <REVIEWER>
                     <REVIEWER_ROLE>Stress</REVIEWER_ROLE>
                     <REVIEWER_HANDLE>handle2</REVIEWER_HANDLE>
                     <REVIEWER_PAYMENT>70</REVIEWER_PAYMENT>
             </REVIEWER>
             <REVIEWER>
                     <REVIEWER_ROLE>Accuracy</REVIEWER_ROLE>
                     <REVIEWER_HANDLE>handle3</REVIEWER_HANDLE>
                     <REVIEWER_PAYMENT>70</REVIEWER_PAYMENT>
             </REVIEWER>
             <PM_HANDLE>somepm</PM_HANDLE>
     </PROJECT>
     <PROJECT>
             <PROJECT_NAME>Document Generator 2.0</PROJECT_NAME>
             <REVIEWER>
                     <REVIEWER_ROLE>Primary/Failure</REVIEWER_ROLE>
                     <REVIEWER_HANDLE>handle4</REVIEWER_HANDLE>
                     <REVIEWER_PAYMENT>100</REVIEWER_PAYMENT>
             </REVIEWER>
             <REVIEWER>
                     <REVIEWER_ROLE>Stress</REVIEWER_ROLE>
                     <REVIEWER_HANDLE>handle1</REVIEWER_HANDLE>
                     <REVIEWER_PAYMENT>70</REVIEWER_PAYMENT>
             </REVIEWER>
             <REVIEWER>
                     <REVIEWER_ROLE>Accuracy</REVIEWER_ROLE>
                     <REVIEWER_HANDLE>handle5</REVIEWER_HANDLE>
```

```
                    <REVIEWER_PAYMENT>70</REVIEWER_PAYMENT>
                </REVIEWER>
                <PM_HANDLE>somepm</PM_HANDLE>
        </PROJECT>
</DATA>
```

1.3.3 Parsing the template file

The template file has the structure shown in section 1.3.1. In is a simple text file. During it's processing the following things should be taken into consideration: comments should be eliminated (full line, including CR/LF), the escape char \ should be handled.

Otherwise the parsing is a simple string processing problem. It is just a matter of locating the field tags and the loop tags and making simple text replacements.

To produce an XSL transformation from a template:

- output the XSL header (see the constants in the XsltTemplate class)
- process the text blocks (tags, text between tags) as they come, in a loop
    - if the current block is text then output it as it is
    - if the current block is a field tag, output XSL_VALUE1, the name of the field, then XSL_VALUE2
    - if the current block is a loop tag, output XSL_LOOP1, the name of the loop, then XSL_LOOP2
    - if the current block is a endloop tag, output XSL_ENDLOOP
- output the XSL footer

The validations that need to be performed are the format of the tags and some semantic checking (whether tags nest properly). The tags should respect the format shown in 1.3.1. Otherwise an exception should be thrown.

The values of the constants are given below and they are present in the code stubs too:

```
    /**
     * The header of the XSL tranformation file.
     */
    private static final String XSL_HEADER =
        "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?>\n" +
        "<xsl:stylesheet xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\"\n"
+
        "                xmlns:fo=\"http://www.w3.org/1999/XSL/Format\"\n" +
        "                version=\"1.0\" >\n" +
        "<xsl:output method=\"text\" indent=\"yes\"/>\n" +
        "<xsl:template match=\"/\">\n" +
        "<xsl:for-each select=\"data\">\n" +
        "<xsl:text>";

    /**
     * The footer of the XSL tranformation file.
     */
    private static final String XSL_FOOTER =
        "</xsl:text>\n" +
        "</xsl:for-each>\n" +
        "</xsl:template>\n" +
        "</xsl:stylesheet>\n";

    /**
     * The text to insert for the special loop field, before the loop name.
     */
    private static final String XSL_LOOP1 =
        "</xsl:text>" +
        "<xsl:for-each select=\"";
```

```
/**
 * The text to insert for the special loop field, after the loop name.
 */
private static final String XSL_LOOP2 =
    "\">" +
    "<xsl:text>";

/**
 * The text to insert for the end of loop special field.
 */
private static final String XSL_LOOPEND =
    "</xsl:text>" +
    "</xsl:for-each>" +
    "<xsl:text>";

/**
 * The text to insert for a data field, before the field name.
 */
private static final String XSL_VALUE1 =
    "</xsl:text>" +
    "<xsl:value-of select=\"";

/**
 * The text to insert for a data field, after the field name.
 */
private static final String XSL_VALUE2 =
    "\"/>" +
    "<xsl:text>";
```

1.3.4 *Generating the input API data structure from the template*

This is a combination of the sections 1.3.3 and 1.3.7. The stream of text blocks (as described in 1.3.3) should be fed to two recursive functions (as described in 1.3.7). This processing does not care about text blocks, only about tags. There is one method called for field tags and one for loop tags. Each method has a parameter for the parent node of the current tag. When the field tag method is called, it creates a new Field node using the data from the tag and adds this node to the parent. When the loop method is called, it creates a new Loop node, then it keeps getting tags until the endloop tag is encountered. For each tag is calls recursively the appropriate method (according to the type of the tag) with the new Loop node passed as parent. In the end, after endloop is encountered, the new Loop node is added to the parent node. This algorithm is nothing more than a trivial recursive building of a tree.

1.3.5 *XSL transformation compilation*

The following code can be used for XSL transformation compilation:

```
String xslt = ….;
TransformerFactory transFactory = TransformerFactory.newInstance();

Transformer xsltTransformer = transFactory.newTransformer (new StreamSource(new
ByteArrayInputStream(xslt.getBytes())));
```

1.3.6 *Applying the XSL transformation*

The following code can be used to apply an XSL transformation:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();

xsltTransformer.transform(new StreamSource(new ByteArrayInputStream(xml.getBytes())), new
StreamResult(out));

return new String(out.toByteArray());
```

1.3.7 *XML generation from the input API*

Although Xerces can be used for this purpose, the generation is just a matter of a few prints so it can be implemented without it. The input API follows the composite pattern (Node, Field, Loop). In order to generate XML from them two recursive functions need to be written that visit a Field and a Loop node. The visit of the Loop node invokes recursively the visit of it's owned subnodes. The visit of a Field simply outputs the start of tag, the value and the end of tag. The visit of a Loop iterates all loop items and for each it outputs a start of tag (named as the loop name), then visits recursively each owned node, then outputs the start tag.

1.3.8 *Configuration file loading logic*

From version 3.0, configuration file are moved to Configuration Object. And we use DocumentGeneratorFactory to create DocumentGenerator, and it needs an Configuration Object.

Fist the "sources" property in the configuration object is looked up to determine the template source ids.

Then for each source id, the property "<id>_class" is lookup up.

This property indicates the template source class implementation to be used by the current id. An instance of that class is created dinamically using the id and configuration object, and the template sources map is updated.

Each class will look up its specific properties based on the id passed in its constructor (see the sample config file for details).

Finally the "default_source" property indicates the default template source (optional).

1.3.9 *Command line interface*

The command line parameters are:

Generate document:

[-s[ource] <source_id>] -t[emplate] <template_name> -x[ml] <xml_data_file> [-o[ut] output_txt_file] [-c[onfig] <configuration_file>] [-n[amespace] <namespace>]

Add template:

[-s[ource] <source_id>] -t[emplate] <template_name> -a[dd] <template_file> [-c[onfig] <configuration_file>] [-n[amespace] <namespace>]

Remove template:

[-s[ource] <source_id>] -t[emplate] <template_name> -r[emove] [-c[onfig] <configuration_file>] [-n[amespace] <namespace>]

-s[ource] <source_id> The template source identifier to look the templates into (the sources are configured in the config file). If it is missing then the default template file will be used (see the sample config file).

-t[emplate] <template_name> The name of template to be used. The name is specific to the template source. For file sources it is the name of the file. For future implementations it might mean other things (For database sources it is an actual name, file name for CVS sources, name for LDAP).

-x[ml] <xml_data_file> The XML file with field data for the template.

-o[ut] <output_file> The output text file. If it is missing then the output will be written to the standard output.

-a[dd] <template_file> The template file to add to the template source. It will overwrite existing templates with the given name.

-r[emove] Removes the template from the template source.

Note: As it is indicated above by the [] paranthesis, either shortcuts (-s, -t, -x, -o) or full names (-source, -template, -xml, -out) can be used.

-c[onfig] #configuration_file# The configuration file to use, DocumentGeneratorCommand.DEFAULT_CONFIGURATION_FILE will be used if it's not provided

-n[amespace] #namespace# The namespace to load ConfigurationObject, DocumentGeneratorCommand.DEFAULT_NAMESPACE will be used if it's not provided

The methods getTemplate and applyTemplate will be used to generate the output.

All exceptions that occur while using the API will be caught and meaningful messages will be displayed.

For processing the parameters the Command Line Utility should be used:

- create a CommandLineUtility instance

- add to it Switch instances for each parameter

- call the parse method and get the data that populates now the Switch instances (check the component's documentation for more details if needed):

1.3.10 Supporting the '}' character

There is a small bug in the XsltTemplate class, specifically in line 436 that damages support for escaping the '}' character. The `(chars[i] == ')')` portion of the code, needs to be changed to `(chars[i] == '}')` note the difference in the characters ')' and '}'. This will allow escaped closing brackets (such as '\}") to be used in the template.

1.3.11 Conditional Template Headers

The <xsl:if> element is going to be used to help with the new functionality introduced in 2.1. The following constants have been introduced:

```
/**
 * The first portion of the XSL:if opening tag.
 */
private static final String XSL_IF1 = "<xsl:if test=\"";


/**
 * The second portion of the XSL:if opening tag.  This part is rendered
 * after the conditional has been parsed successfully.
 */
private static final String XSL_IF2 = "/"><xsl:text>";

/**
 * The closing tag for xsl:if.  This part is rendered after the
 * conditional template text is written.
 */
private static final String XSL_IFEND = "</xsl:text></xsl:if>";
```

1.3.12 Supporting Conditional Template Text

Conditional template text is added to the template by using an %if% template tag. The example format for this tag is:

%if:[fieldname] [comparator] '[value]' {[description]}%

insert conditional template text here.

%endif%

fieldname is the template field value that should be tested against the specified value.

comparator is the operation that is used to compare the field value, against the stated value. The supported comparator Strings are “<”, “>”, “=”, “<=”, “>=” and “!=”. Note that “<” and “>” may possibly not exist in pure character form (depending on developer’s implementation) at this time - they may be > or < Strings instead (For example, “>=” might be represented as “>=”). See the next section for more details.

value is the value to test against the field value. Note that for “>”, “<”, “<=”, and “>=” operations, only numeric values are allowed. A TemplateFormatException should be thrown if a different value exists.

description is merely there to provide a helpful String for other users who read the template.

Note that no default value is supported, since this is a conditional field where such a default value will rarely be used, and where adding it would require a more complex for of syntax.

The steps to render the conditional text are:

- If the current block is an opening %if% block then first strip the prefix “%if:” and postfix “%” strings to get the block opening String.
- Parse out the field name by separating the String part where a “<”, “>”, “=”, “<=”, “>=” or “!=” occurs. If no comparator String occurs, then throw TemplateFormatException.
- If “{“ and “}” occur at the end of the block opening String, it means a description exists. Parse out the description too.
- Note that value needs to be enclosed in the ‘ character. If it is not enclosed, throw TemplateFormatException. Also note that the value needs to be a number for “<”, “>” “<=” and “>=” comparators.
- Write out the XSL_IF1 header.
- Write the [fieldname] [comparator] ‘[value]’ conditional String.
- Write out the XSL_IF2 header.
- Delegate parsing back to the normal parsing behavior until a matching %endif% has been found. When this %endif% is reached, then write out the XSL_ENDIF header. If no matching %endif% is found at the end of the document, throw TemplateFormatException.
- Create a Condition object with fieldname, description, and conditionalString and add to internal set of NodeLists.

Example:

The following template text:

```
%if:COMPONENT_PAYMENT=‘2000’ {Display special text for high-paying ‘rush’ components}%
    Note that this component needs to be urgently done, which is why there are increased payments.
%endif%
```

Should result in the following XSL Template:

```
<xsl:if test=”COMPONENT_PAYMENT=’2000’”/><xsl:text>
    Note that this component needs to be urgently done, which is why there are increased payments.
</xsl:text></xsl:if>
```

And the following Condition created for it would have the following true statements:

condition.getName().equals(“COMPONENT_PAYMENT”);

condition.getDescription().equals(“Display special text for high-paying ‘rush’ components”);

condition.getConditionalStatement().equals(“COMPONENT_PAYMENT=’2000’”);

condition.getDefaultValue() == null;

1.3.13 Supporting the ‘<’ and ‘>’ Characters

The ‘<’ and ‘>’ characters are not supported properly, because they are an integral part of XML, and so the generated template mistakes them for XML delimiters, and this results in a syntax-error. The problem is easily fixed by replacing ‘>’ with ‘>’ and ‘<’ with ‘<’. The developer needs to take care as to *when* to perform the replacement. The Condition’s conditional statement should still display “<” and “>” instead of “<” and “>”. A suggested approach would be to convert it together in handleEscapeChars() private method of XsltTemplate, and to take care when parsing a Condition.

Example:

The following template text:

%if:COMPONENT_PAYMENT>=‘2000’ {Display special text for high-paying ‘rush’ components}%

Note that this component needs to be urgently done, which is why there are increased payments.

%endif%

Should result in the following XSL Template:

<xsl:if test=”COMPONENT_PAYMENT>=’2000’”/><xsl:text>

Note that this component needs to be urgently done, which is why there are increased payments.

</xsl:text></xsl:if>

1.4 Component Class Overview

DocumentGenerator:

This is the main class of the design and the only API class that the users will interact with in most cases.

It is also a facade for the rest of the classes, hiding the details of the document generation process.

It exposes methods for getting a template from the template source (file system, in the future database, CVS and LDAP maybe), for parsing a given template (as text) and for applying a template (to data coming from XML or data configured using the API classes for field data entering).

Version 3.0:
This class is changed to a pure API class, it's not a command line client and all configuration related stuff is moved to DocumentGeneratorFactory. The template sources are programmatically configurable, so this class is no longer immutable and therefore no longer thread-safe.

Template:

Abstracts a template processing class. This class is responsible for parsing the template defined by this component to some precompiled internal format so it can be subsequently applied to multiple input data.

Note that the format of the text template can be changed easily in the future. Only XmlTemplateData and XsltTemplate depend on this format. Obviously, the entire fieldConfig API depends on the logical structure of a template, but not on the actual syntax.

The implementations must have a public default constructor (with no parameters).

Currently an implementation based on XML files and XSLT transformations is provided but other methods can be used as well (such as Velocity templates).

TemplateData:

This interface abstract the input data used by templates to the rest of the design.

Only the template implementation has intimate knowledge of the implementation of TemplateData. Typically there should be pairs of Template and TemplateData implementation (such as XmlTemplateData and XsltTemplate).

The interface exposes method for setting the template data, from raw text (currently XML but that depends only on the current implementation of the TemplateData, so it can be changed easily) and from programmatically configured template data, and for getting the template data.

The implementations must have a public default constructor (with no parameters).

TemplateSource:

This interface abstracts a template source, some kind of repository where templates can be retrieved from, based on a given name. The name has different interpretations specific to each implementation (the value of a field in the database source, a file name in the file source, a file name in a future CVS source implementation).

Version 3.0:
Implementations are NOT required to provide a constructor accepting two strings since ConfigManager is not used, but implementations should provide a constructor accepting a single ConfigurationObject instance.

XsltTemplate:

This class is an implementation of the Template interface that uses XSL transformations as method for applying template transformations to input data, in order to generate text content.

The idea behind this approach is to convert the template format defined by this component to XSLT by simple text replacements. The input data is given in XML format (directly or indirectly thought the API for programmatic data input). This way the text generation is reduced to simple XSL transformations.

This class has been changed to support conditional template text, and also to have some bugs fixed.

XmlTemplateData:

Implementation of the TemplateData interface that uses XML as format for template input data.

FileTemplateSource:

Implementation of a template source that retrieves template from the file system.
In this implementation the name of the template is the filename of the template.

Version 3.0:
The constructor +FileTemplateSource(namespace:String,sourceId:String) is changed to take a ConfigurationObject since ConfigManager is removed.

Node:

This interface is part of the API for entering data for template fields.

It designates a node which can be a simple field or a composite loop node.
The field node represents data about a template field.
The loop node represents data about a loop in the template and can contain other loop and field nodes, following the composite pattern.

Field:

The field class is part of the API for programmatically configuring field values.
It corresponds to a field in the template and exposes methods for getting the name, a description (containing comments, sample data) and getting/setting the value.).

An Accessor has been added to determine whether this is read-only.

Loop:

The loop class is part of the API for programmatically configuring field values.
It corresponds to a loop in the template.

The class has a method to retrieve the list of children nodes (the fields and other loops inside the loop) in a read-only structure as a "sample". The idea is to provide useful information for an interactive GUI (the only reason why anyone would use this API - otherwise it's much easier to write XML data, and of course for mass mail generation).

A loop may be repeated 0 to many times but the user needs to see some information about the loop before deciding to create data for loop items and that's what the "sample" is for.

An instance itself can be read-only if it is placed inside such a "sample" parent loop.
Read-only means deep read-only, that is every element, up to any depth is read-only.

The loop data can be configured using standard methods for adding, inserting, removing, clearing and getter loop items.
A "loop item" means the data that is used for one repetition of the loop.
For example repeating the loop 3 times would require 3 "loop items" containing the data for each repetition.

An Accessor has been added to determine whether this is read-only.

Condition:

This is an implementation of Node that has similar information to that of a Field class. It also includes information on the conditional expression used to determine if some template text is displayed.

NodeListUtility:

This is a Utility class that is used to conveniently populate a NodeList from different data sources, such as Java objects with accessors, and Maps.

NodeList:

An immutable list of nodes used for better handling of the nodes (fields and loops) within a loop.).

TemplateFields:

This class represents the root of the structure used to configure programmatically the template data. It is a subclass of NodeList.

DocumentGeneratorCommand:

The DocumentGeneratorCommand class is the command line class used to execute document generation. This class loads configurations via Configuration Persistence component, and creates an instance of DocumentGenerator via DocumentGeneratorFactory. It executes document generation via the created DocumentGenerator instance. Command Line Utility is used to process command line arguments and related stuff.

This class is new in version 3.0.

DocumentGeneratorFactory:
The DocumentGeneratorFactory class defines a static method to create DocumentGenerator instance based on the passed in ConfigurationObject.

This class is new in version 3.0.

1.5 Component Exception Definitions

IllegalArgumentException:
Usually thrown for null arguments and empty strings. The javadocs contain full coverage of the situation is which this exception should be thrown for each method.

IllegalStateException:
The Loop and Field classes throw this exception when modification attempts are made for read-only instances.

ArrayIndexOutOfBoundsException:
Thrown in the Loop class, in the insertLoopItem method, when the index is out of bounds .

IOException:
Thrown in case of I/O errors in the DocumentGenerator methods, when error occurs while reading from InputStreams (for those methods that have InputStreams as params) or while writing to the OutputStreams (for those methods that have OutputStreams as params).

TemplateSourceException:
Exception thrown in the DocumentGenerator.getTemplate() methods to indicate an error related to getting a template from a template source:

- there is no template source with a given id

- there is no template with the given name in the template source

- other template source implementation specific exception (wrapped in this exception - such as SQLException, IOException)

TemplateFormatException:
Exception thrown to signal an invalid template in DocumentGenerator.getTemplate, parseTemplate, applyTemplate, getFields and in the Template implementations:

- can be signaled by the template processing class (Template instance)

- other implementation specific exception indicating a template format problem

(wrapped in this exception - such as an XSLT exception).

TemplateDataFormatException:
Exception thrown to signal invalid template data in DocumentGenerator.applyTemplate method and Template implementations.

- this wraps some implementation specific error indicating bad template data (such as an XML exception or an XSLT exception).

DocumentGeneratorConfigurationException:
Thrown by DocumentGeneratorFactory to indicate that any error occurs during configure DocumentGenerator.

This exception is defined in version 3.0 to replace the old InvalidConfigException.

1.6 Thread Safety

Most classes do not keep state or are immutable. The input API is not thread safe from obvious reasons: it gets input from code programmatically so it is up to that code to make consistent and logical updates

(it's like updating an array from two threads and expecting predictable results).

The most important class DocumentGenerator is no longer thread-safe in version 3.0 since the TemplateSources are programmatically configurable, and therefore the whole component is not thread-safe.

2. Environment Requirements

2.1 Environment

- JDK 1.4.2 or higher is required for compilation.
- JRE 1.4.2 or higher is required for running the test cases and for usage.

2.2 TopCoder Software Components

- Command Line Utility 1.0 – for command line switches processing
- Configuration API 1.0 – for configuring document generators.
- Configuration Persistence 1.0 – for loading configurations from file
- Base Exception 2.0 – all exceptions extend from BaseNonCriticalException.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.file

com.topcoder.util.file.templatesource

com.topcoder.util.file.xslttemplate

com.topcoder.util.file.fieldconfig

3.2 Configuration Parameters

A sample config file follows. Each property is explained in the comments. The file is available also in the conf directory.

Configuration file content:

```
<?xml version="1.0"?>
<CMConfig>
<Property name="com.topcoder.util.file">
<Value>com/topcoder/util/file/default.xml</Value>
</Property>
</CMConfig>
```

The content of default.xml

```
<CMConfig>
    <Config name="com.topcoder.util.file">
        <Property name="sources">
            <Value>file</Value>
            <Value>my_file</Value>
```

```
        </Property>

        <!-- the identifier of the template source to be used as default -->
        <Property name="default_source">
            <Value>my_file</Value>
        </Property>

        <!-- here follow custom properties for each template source -->
        <!-- the <sourceidentifier>_class property is mandatory -->
        <!-- the other <sourceidentifier>_<property_name> properties are specific
for each source -->

        <!-- file template source properties -->
        <Property name="file_class">
            <Value>com.topcoder.util.file.templatesource.FileTemplateSource</Value>
        </Property>

        <!-- my_file template source properties -->
        <Property name="my_file_class">
            <Value>com.topcoder.util.file.MyFileTemplateSource</Value>
        </Property>

        <Property name="my_file_dir">
            <Value>test_files/my_files/</Value>
        </Property>
    </Config>
</CMConfig>
```

3.3 Dependencies Configuration

The dependencies do not need configuration.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

The stress tests are covered in their specific section. The failure tests should be created using the invalid and valid arguments present in the Poseidon documentation.

The accuracy tests should cover the following areas:

- retrieval, addition and removal of templates from the file system
- document generation using reasonably complex templates (a few nested loops)
- the data input through the API
- the configuration file with several template sources
- test the command line interface

4.2 Required steps to use the component

The steps to use the component are:

```
// 1. instantiate the Document Generator
ConfigurationObject someConfig = ...;
DocumentGenerator instance = DocumentGeneratorFactory.getDocumentGenerator(someConfig);

// 2. get a template
Template template = instance.getTemplate(“db1”, “build_templ”);
```

```
// 3. generate the text output
instance.applyTemplate(template,new FileReader(xmlFile), new FileWriter(output_file));

// 4. using input API
TemplateFields root = instance.getFields(template);
Node[] nodes = root.getNodes();
for (int i = 0; i <  nodex.length; i++) {
    if (nodes[i] instanceof Field) {
        ((Field) nodes[i]).setValue("abc");
    } else {
        // do something recursive with (Loop) nodes[i]
        // and process it as root is processed here
    }
}

instance.applyTemplate(root, new FileOutputStream(output_file));
```

4.3 Demo

4.3.1 Create DocumentGenerator and Manage TemplateSources

```
        // Create DocumentGenerator from constructor
        DocumentGenerator docgen = new DocumentGenerator();

        // A workable ConfigurationObject
        ConfigurationObject config =
TestHelper.createConfigurationObject("demo.properties",
            "com.topcoder.util.file");
        // Create DocumentGenerator from factory
        docgen = DocumentGeneratorFactory.getDocumentGenerator(config);

        // Modify TemplateSources
        TemplateSource ts = new FileTemplateSource("file_source", config);
        // Any number of TemplateSource can be added
        docgen.setTemplateSource("file_source", ts);
        docgen.getTemplateSource("file_source");
        // The getter should return ts
        docgen.removeTemplateSource("file_source");
        // ts should be removed
        docgen.clearTemplateSources();
        // All template sources should be removed

        // Modify default TemplateSource
        docgen.setDefaultTemplateSource(ts);
        // Now ts is used as the default one
        docgen.getDefaultTemplateSource();
              // ts should be returned from getter
```

4.3.2 User Scenario of Generating a Document

One of the biggest hassles right now for designers is the fact that the Cobertura build task is present even in the build.xml present in a design distribution. The current enhancement will allow XML to be generated easily, and the conditional statement will allow the build task to be conditionally added. The following template fragment is feasible:

```
<!-- DOCUMENT PACKAGE -->
<property name\="dist_docpackage" value\="$\{builddir\}/doc_package" />
<property name\="docpackage.jar" value\="$\{distfilename\}_docs.jar" />

%if:projectType='development'{Only Display Cobertura task for dev}%
            <!-- codertura task definitation -->
            <property name\="cobertura.dir" value\="$\{ext_libdir\}/cobertura/1.8" />
```

```
                <path id\="cobertura.classpath">
                </path>
                <taskdef classpathref\="cobertura.classpath"
            resource\="tasks.properties" />
                <property name\="cobertura.datafile"
             value\="$\{testlogdir\}/cobertura.ser" />
                <property name\="instrumented.dir"
             value\="$\{builddir\}/instrumented" />
  %endif%

  <!-- TCS Dependencies -->
  %loop:dependencies{Loop around all dependencies}%
  <property name\="%componentName%.version" value\="%componentVersion%"/>
  <property name\="%componentName%.jar.name" value\="%componentLongName%.jar"/>
  <property name\="%componentName%.path"
      value\="%componentName%/$\{%componentName%.version\}"/>
  %endloop%
```

The new NodeList utility can also be used to support easy programmatic modification of the build files. Take the following Project and Component classes:

```
/**
 * <p>
 * This is a data class used to populate data for <code>NodeList</code>.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 2.1
 */
public class Project {
    /**
     * <p>
     * Represents the project type.
     * </p>
     */
    private String projectType;

    /**
     * <p>
     * Represents the project dependencies.
     * </p>
     */
    private final List dependencies = new ArrayList();

    /**
     * <p>
     * Gets the project type.
     * </p>
     *
     * @return the project type.
     */
    public String getProjectType() {
        return projectType;
    }

    /**
     * <p>
     * Sets the project type.
     * </p>
     *
```

```
     * @param projectType the new project type
     */
    public void setProjectType(String projectType) {
        this.projectType = projectType;
    }

    /**
     * <p>
     * Gets the dependencies for the project.
     * </p>
     *
     * @return the dependencies for the project.
     */
    public Component[] getDependencies() {
        return (Component[]) dependencies.toArray(new
Component[dependencies.size()]);
    }

    /**
     * <p>
     * Adds a dependency for the project.
     * </p>
     *
     * @param component a dependency of the project
     */
    public void addDependency(Component component) {
        if (component != null) {
            this.dependencies.add(component);
        }
    }

    /**
     * <p>
     * Sets the dependencies for the project.
     * </p>
     *
     * @param components the dependencies of the project
     */
    public void setDependencies(Component[] components) {
        dependencies.clear();

        if (components != null) {
            for (int i = 0; i < components.length; i++) {
                addDependency(components[i]);
            }
        }
    }
}

/**
 * <p>
 * This is a data class used to populate data for <code>NodeList</code>.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 2.1
 */
public class Component {
    /**
     * <p>
```

```
     * Represents the component name.
     * </p>
     */
    private String name;

    /**
     * <p>
     * Represents the component version.
     * </p>
     */
    private String version;

    /**
     * <p>
     * Represents the component long name.
     * </p>
     */
    private String longName;

    /**
     * <p>
     * Sets the component name.
     * </p>
     *
     * @param name the new component name
     */
    public void setComponentName(String name) {
        this.name = name;
    }

    /**
     * <p>
     * Sets the component long name.
     * </p>
     *
     * @param name the new component long name
     */
    public void setComponentLongName(String longName) {
        this.longName = longName;
    }

    /**
     * <p>
     * Sets the component version.
     * </p>
     *
     * @param name the new component version
     */
    public void setComponentVersion(String version) {
        this.version = version;
    }

    /**
     * <p>
     * Gets the component name.
     * </p>
     *
     * @return the component name
     */
    public String getComponentName() {
```

```
            return name;
        }

        /**
         * <p>
         * Gets the component long name.
         * </p>
         *
         * @return the component long name
         */
        public String getComponentLongName() {
            return longName;
        }

        /**
         * <p>
         * Gets the component version.
         * </p>
         *
         * @return the component version
         */
        public String getComponentVersion() {
            return version;
        }

    }
```

Are initialized in the following manner:

```
        Project project = new Project();
        project.setProjectType("design");

        Component configManager = new Component();
        configManager.setComponentName("configmanager");
        configManager.setComponentLongName("configuration_manager");
        configManager.setComponentVersion("2.1.5");

        Component baseException = new Component();
        baseException.setComponentName("baseexception");
        baseException.setComponentLongName("base_exception");
        baseException.setComponentVersion("2.0");

        project.setDependencies(new Component[] {configManager, baseException });

        // A document can be programmatically retrieved by using:
        // A workable ConfigurationObject
        ConfigurationObject config =
TestHelper.createConfigurationObject("demo.properties",
            "com.topcoder.util.file");
        // Create DocumentGenerator from factory
        DocumentGenerator docGen =
DocumentGeneratorFactory.getDocumentGenerator(config);

        Template buildTemplate = docGen.getTemplate("fileSource",
"test_files/buildTemplate.txt");

        TemplateFields root = docGen.getFields(buildTemplate);
        Node[] nodes = root.getNodes();
        NodeList nodeList = new NodeList(nodes);
```

```
        // This is a lot simpler than looping through all the nodes and looking
        // up the respective project property. Note that it is also possible to
        // do this manually.
        NodeListUtility.populateNodeList(nodeList, project);
        // Applying the template can also be done without using the Document
Generator,
        // using Template#applyTemplate method.
        String designBuildTemplate = docGen.applyTemplate(root);
        System.out.println(designBuildTemplate);

        XmlTemplateData xslTemplateData = new XmlTemplateData();
        xslTemplateData.setTemplateData(root);

        project.setProjectType("development");
        NodeListUtility.populateNodeList(new NodeList(nodes), project);
        String devBuildTemplate = docGen.applyTemplate(root);
        System.out.println(devBuildTemplate);
        displayNodes(nodes);
```

The XML Data for the populated node list would look like this:

```
<DATA>
  <projectType>design</projectType>
  <dependencies>
    <componentName>configmanager</componentName>
    <componentVersion>2.1.5</componentVersion>
    <componentName>configmanager</componentName>
    <componentLongName>configuration_manager</componentLongName>
    <componentName>configmanager</componentName>
    <componentName>configmanager</componentName>
    <componentName>configmanager</componentName>
  </dependencies>
  <dependencies>
    <componentName>baseexception</componentName>
    <componentVersion>2.0</componentVersion>
    <componentName>baseexception</componentName>
    <componentLongName>base_exception</componentLongName>
    <componentName>baseexception</componentName>
    <componentName>baseexception</componentName>
    <componentName>baseexception</componentName>
  </dependencies>
</DATA>
```

The generated document (using the template specified above), would look like (note that cobertura build is not rendered because the project type is “design”):

```
    <!-- DOCUMENT PACKAGE -->
    <property name="dist_docpackage" value="${builddir}/doc_package" />
    <property name="docpackage.jar" value="${distfilename}_docs.jar" />

    <!-- TCS Dependencies -->

    <property name="configmanager.version" value="2.1.5"/>
```

```
    <property name="configmanager.jar.name"
value="configuration_manager.jar"/>
    <property name="configmanager.path"
          value="configmanager/${configmanager.version}"/>

    <property name="baseexception.version" value="2.0"/>
    <property name="baseexception.jar.name" value="base_exception.jar"/>
    <property name="baseexception.path"
          value="baseexception/${baseexception.version}"/>
```

Programmatic inspection of the nodes is also possible:

```
        for (int x = 0; x < nodes.length; x++) {
            if (nodes[x] instanceof Condition) {
                Condition condition = (Condition) nodes[x];
                System.out.println(condition.getName());
                System.out.println(condition.getValue());
                System.out.println(condition.getDescription());
                System.out.println(condition.getConditionalStatement());
            }

            if (nodes[x] instanceof Field) {
                Field field = (Field) nodes[x];
                System.out.println(field.getName());
                System.out.println(field.getValue());
                System.out.println(field.getDescription());
            }

            if (nodes[x] instanceof Loop) {
                Loop loop = (Loop) nodes[x];
                System.out.println(loop.getLoopElement());
                System.out.println(loop.getDescription());
            }
    }
```

4.3.3 *Generate Document from Command Line*

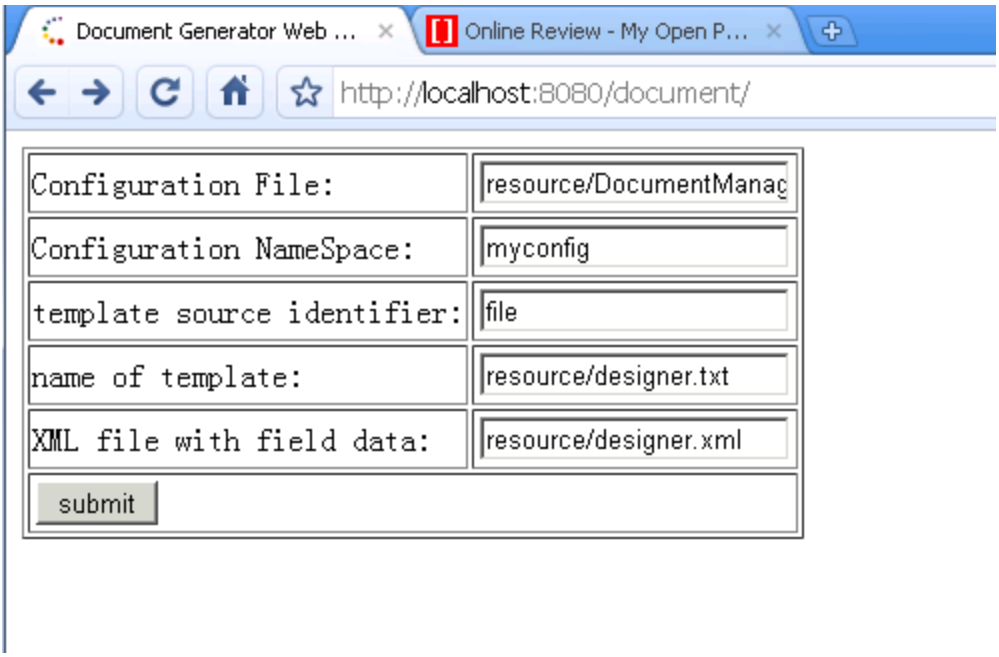
Assume that default configuration file is provided, we can generate document from command line as below:

```
~$ java DocumentGeneratorCommand –s “file_source” –t “template” –x “data_file” –c
“DocumentManager.xml” –n “myconfig”
```

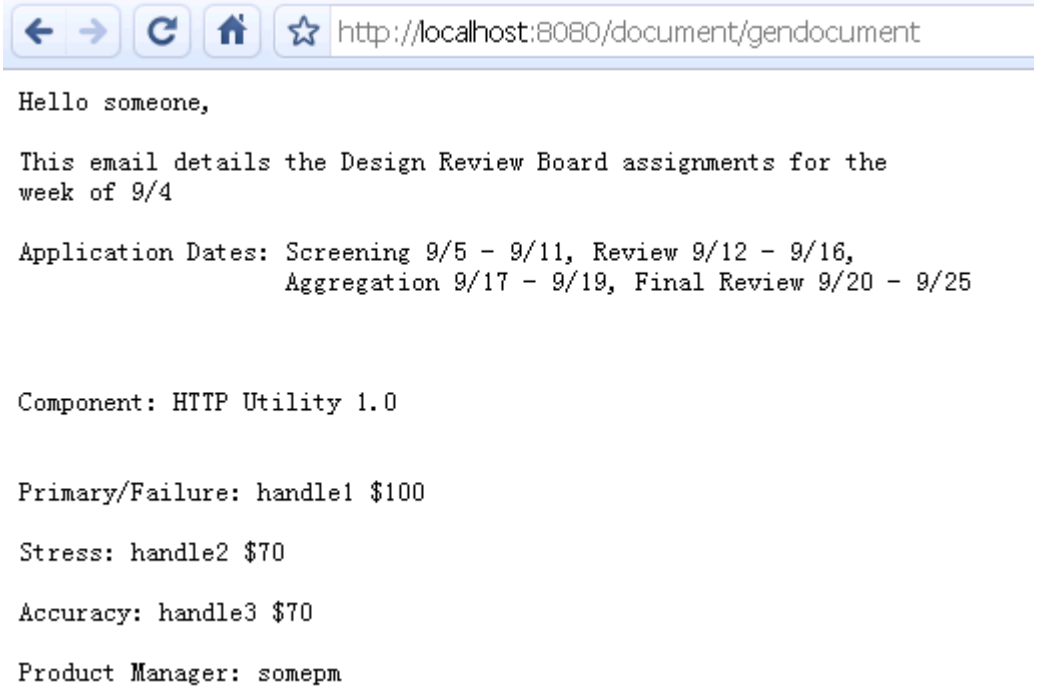
Refer to DocumentGeneratorCommand class for all arguments.

4.3.4 *Web Demo*

Component provides a simple web demo to show how it is working in JBoss 5. It can read configuration file, template file and xml data file from jar achieve file.
1. Open /document in a browser



2. Fill all the fields, click the submit button, you will see the generated document.



5. Future Enhancements

The XML files are not so convenient to use and the input API invites an interactive GUI for this component that would considerably improve the usability of the component.

The design already outlines the possibility to use databases, CVS repositories and LDAP servers as source for templates.