SourceGen Project

Daniel Hoberecht
Michael Lapp
Kenneth Melby III

June 21, 2007

Abstract

Comverse develops and deploys world class billing and ordering applications for telecommunications companies worldwide. The Kenan FX line of products meets a majority of each client's needs for customer services calls to service activation and billing. Custom software for unique client functionality is often written by consultants in the professional services division to ensure that the software has all the functionality that a client may require. The process of starting a new custom project is quite tedious. A consultant often starts with a basic template that gets modified before beginning the customization. These modifications include renaming files and directories, modifying contents of existing files, and copying directories and files. During the modification, if a single error is made it could be very hard to find where the error is. Every custom application for a client begins with this process; there must be a better way.

Our task is to develop an engine that reads in an XML file detailing changes to the template project, executes those changes, and returns the source for a new custom project. The engine would optimally be generic and flexible so that it can be extended for use in multiple settings.

Table of Contents

Table of Figures

1 Introduction

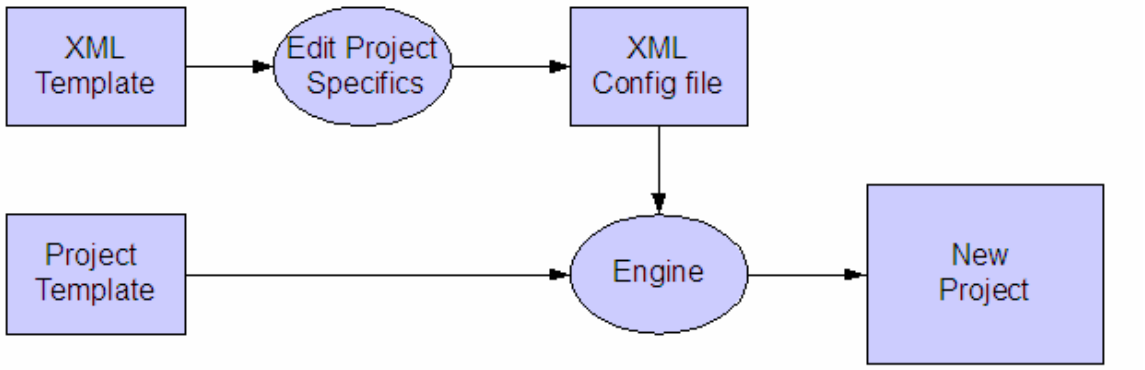
1.1 Purpose

Comverse develops and deploys software and systems for communication service providers worldwide. Kenan FX is a core software solution for everything from customer service calls to service activation and billing. This meets a majority of clients' needs, but not all. Custom client functionality is often written by consultants in the professional services division to add unique functionality to an existing product.

Before starting a new custom project, a basic template project must be modified. The template project contains the basic source code, configuration files, and directories for a plugin to existing software. The modifications include renaming files and directories, changing the contents of specific files, and moving and copying files and directories. This is a tedious process with little room for error. All of these changes must be made before the custom application can even be started.

The purpose of this project is to create an automated process to replace this existing manual, step-by-step method. A template XML file describing basic changes and default parameters will be manually created for each project. A graphical user interface (GUI) will receive modifications from the user and update the XML template file (this will be the XML configuration file). The engine will read in the XML config file and use separate modules to copy, replace, or rename files and directories. A new directory structure ready to be customized will be the result. The overall flow of the project is shown in figure 1.

Figure 1: Simplified Project Flow



2 Requirements

2.1 Functional Requirements

1. Engine must be able to execute the changes stored in the XML config file.
2. Engine must be able to copy files and directories.
3. Engine must be able to rename files and directories.
4. Engine must be able to rename elements in a file.
5. Engine must be able to delete files and directories.
6. Engine must load the XML config file.

2.2 Non-functional Requirements

1. Engine must be modular and flexible.
2. Project specific changes must be stored in an XML file.
3. Engine must use Xerces/Document Object Model (DOM) library to do XML parsing.
4. The XML config file will have a consistent formatting.
5. All code must be written in Java version 1.5.
6. The project must be built using Apache's Ant.

2.3 Use Cases

Functional requirements for each use case are in parenthesis.

1) **Engine loads XML config file detailing changes (6)**

Pre-conditions:
- XML config file exists.
- XML config file updated (Use case 6).

Primary Flow:
1. XML config file is passed into engine and parsed.
2. Parsed tree is preprocessed (Use case 2).

Post-conditions:
- Engine is ready to execute changes detailed in XML config file.

Alternate Flow(s):
1A. If XML file is not found, then error message is logged.

2) **Parsed XML tree is preprocessed (6)**

Pre-conditions:
- XML file has been parsed successfully (Part of use case 1).

Primary Flow:
1. Variable placeholders within the variable block are replaced with values.
2. Variable placeholders within the start tree are replaced with values.
3. Start tree is pulled out of the full raw tree.

Post-conditions:
- Engine has preprocessed start tree.

Alternate Flow(s):
None

3) **Engine executes changes from XML file (1)**

Pre-conditions:
- XML config file has been loaded (Use case 1).

Primary Flow:

1. Engine traverses start tree.

The following are subject to what actions were defined in the XML.

2. Engine copies files and directories (Use case 4).
3. Engine renames files and directories (Use case 5).
4. Engine renames elements in specific files (Use case 6).
5. Engine deletes files and/or directories (Use case 7).

Post-conditions:

- Desired changes from XML config file were executed successfully.

Alternate Flow(s):

None

4) **Engine copies files and/or directories (2)**

Pre-conditions:

- XML config file has been loaded (Use case 1)

Primary Flow:

1. Copy rules are loaded from the DOM tree.
2. Files and/or directories are copied to a new specified location as defined in the XML config file.

Post-conditions:

- Desired files and/or directories were copied successfully.

Alternate Flow(s):

None

5) **Engine renames files and/or directories (3)**

Pre-conditions:

- XML config file has been loaded (Use case 1)

Primary Flow:

1. Rename rules are loaded from the DOM tree.
2. File and/or directories are renamed as defined in the XML config file.

Post-conditions:

- Desired files and/or directories were renamed successfully.

Alternate Flow(s):

None

6) **Engine renames an element in a file (4)**

Pre-conditions:

- XML config file has been loaded (Use case 1)

Primary Flow:

1. Searches the file directories for the desired file.
2. Performs a find and replace inside the file from the specified pattern to a new pattern.

Post-conditions:

- Pattern inside specified file has been changed.

Alternate Flow(s):

2A. Logs if an element to be renamed in file is not found.

7) **Engine deletes a file or directory (5)**

Pre-conditions:

- XML config file has been loaded (Use case 1)

Primary Flow:

1. Searches the file directories for the desired file.
2. Deletes the specified file or directory

Post-conditions:

- Specified file or directory was deleted.

Alternate Flow(s):

2A. Logs an error if a directory had contents and an attempt was made to delete it. The directory is not deleted.

3 Design

3.1 Approach

Originally the design consisted of plugins for copy, rename, delete, and substitution, which would all plug into the Engine class. We developed a specific XML format to go along with this design, which had a block for each action. If the user wanted to do multiple copies there would be multiple copy tags. This seemed tedious and not very straight forward for the user creating an XML file. Upon further examination a more object oriented (OO) design came about. Having file and directory classes that have methods to be copied, renamed, deleted, and for files substituted within appears to be the easier approach to conceptually understand. The XML file is easier and more intuitive to format under the OO approach. The OO approach also lends itself to Java; being an object oriented language.

3.2 Class/Method Specifics

Class methods are explained using a flow, pseudocode, or description of what the method does.

A few definitions of terms used in this section:

"raw tree" - The tree resulting from XML parsing.

"processed tree" - Tree resulting from preprocessing. Has variable values plugged in for placeholders. Consists of only the "start branch."

"start branch" - Everything underneath the start node. Consists of only rules that need to be executed on the directory structure.

The requirement(s) that each method helps satisfy are listed next to that method in parenthesis.

Engine:

Reads in the XML file and executes all changes detailed within it. Has a Parser.

private readData(File xmlFile): **(6)**

1. Engine reads in XML.
2. Engine passes XML on to Parser.
3. Parser parses XML to "raw tree."
4. Parser passes "raw tree" to Preprocessor.
5. Preprocessor inserts variable values into placeholder locations.
6. Preprocessor returns a "processed tree."

public run(File xmlFile): **(1, 6)**

1. readData(File)
2. start("processed tree")

private start(Node start): **(6)**

Find number of start node's children.
For each child
    create a PSDirectory from the <dir> tag
    traverse(PSDirectory, Node)
endfor

private traverse(PSDirectory dir, Node rules): **(2, 3, 4, 5, 6)**

1. Copies the directory if it has the tag.
2. Traverses the children of the file system.
3. Renames the directory if it has a tag.
4. Deletes the directory if it has a tag.

private traverse(PSFile file, Node rules): **(2, 3, 4, 5, 6)**

1. Copies the file if it has a tag (to a different name also)
2. Goes through file and does all substitutions
3. Renames the file if it has a tag
4. Deletes the file if it has a tag

private Node findNode(Node input, String seek): **(2, 3, 4, 5, 6)**

Takes in a Node and returns the node whose name matches the String passed in from the first set of children.

private String findNewPath(String source, String from, String to): **(3)**

Takes in a source file or directory (pathname) and then uses the from and to string with regular expressions to change the source pathname into the proper "to" path.

Parser:
Parses XML file to a raw tree Has a Preprocessor.

public parseXML(File xmlFile): **(6)**

Parses XML file into a "raw tree."

public preprocess(Node rawTree): **(6)**

Preprocessor.preprocess("raw tree")

public getStart(): **(6)**

Returns the local "start tree" variable.

Preprocessor:
Substitutes variables in start and within other variables and "cleans" the tree.

public preprocess(Node rawTree): **(6)**

1. Saves "raw tree" as a local variable.
2. variableSubstitution()
3. cleanTree()

private variableSubstitution(): **(6)**

1. Finds the NodeList with all <var> nodes and the Node that points to the top of the start tree.
2. loadHash(NodeList variables);
3. substitutionInVariables();
4. substitutionInStart(Node startTree);

private void loadHash(NodeList variables): **(6)**

Loads all the variables in the NodeList into a HashTable. The name of each variable is the key. This provides a fast O(1) retrieval of variable values when placeholders are replaced with variable values.

private void substitutionInStart(Node node): **(6)**

Finds variable placeholders and calls updateVariable() to change the placeholder to an actual variable value in the start tree. The following attributes support variable placeholders:

1. **from** attribute in a dir or file tag.
2. **to** attribute in a copy or rename tag within actions.
3. **seek** or **replace** in a sub tag within subs.
4. **boolean** attribute in a delete tag within actions.

private void updateVariable(Node node, String attributeName): **(6)**

Replaces a variable placeholder in the specified attribute name for the given node with the variable value.

private void substitutionInVariables(): **(6)**

Replaces variable placeholders in the variables themselves with the appropriate variable value. Infinite loops are possible and it is up to the user to avoid situations where they would come up (a variable cannot depend upon itself).

private Node findNode(NodeList list, String seek): **(6)**

Returns a node from the given NodeList that has the name that was passed in.

private cleanTree(): **(6)**

Sets the local "tree" to equal the "start tree."

public getStart(): **(6)**
Returns the local "start tree" variable.

PSFile:
File that can use implemented actions.

public copy(String to): **(2)**
Implements copy for a file.

public remove(): **(5)**
Implements remove for a file.

public substitution(String find, String replace): **(4)**
Opens the file, finds a desired string, and changes that string to a new updated string.

PSDirectory:
Directory that can use implemented actions.

public copy(String to): **(2)**
Implements copy for a directory.

public remove(): **(5)**
Implements delete for a directory. The directory must be empty to be able to be deleted so the contents of the directory must be deleted first.

AbstractPSFile: Parent of PSFile and PSDirectory that inherits from java.io.File.

public abstract copy(): **(2)**
Both PSFile and PSDirectory will need a copy method, but the implementation will be different between them.

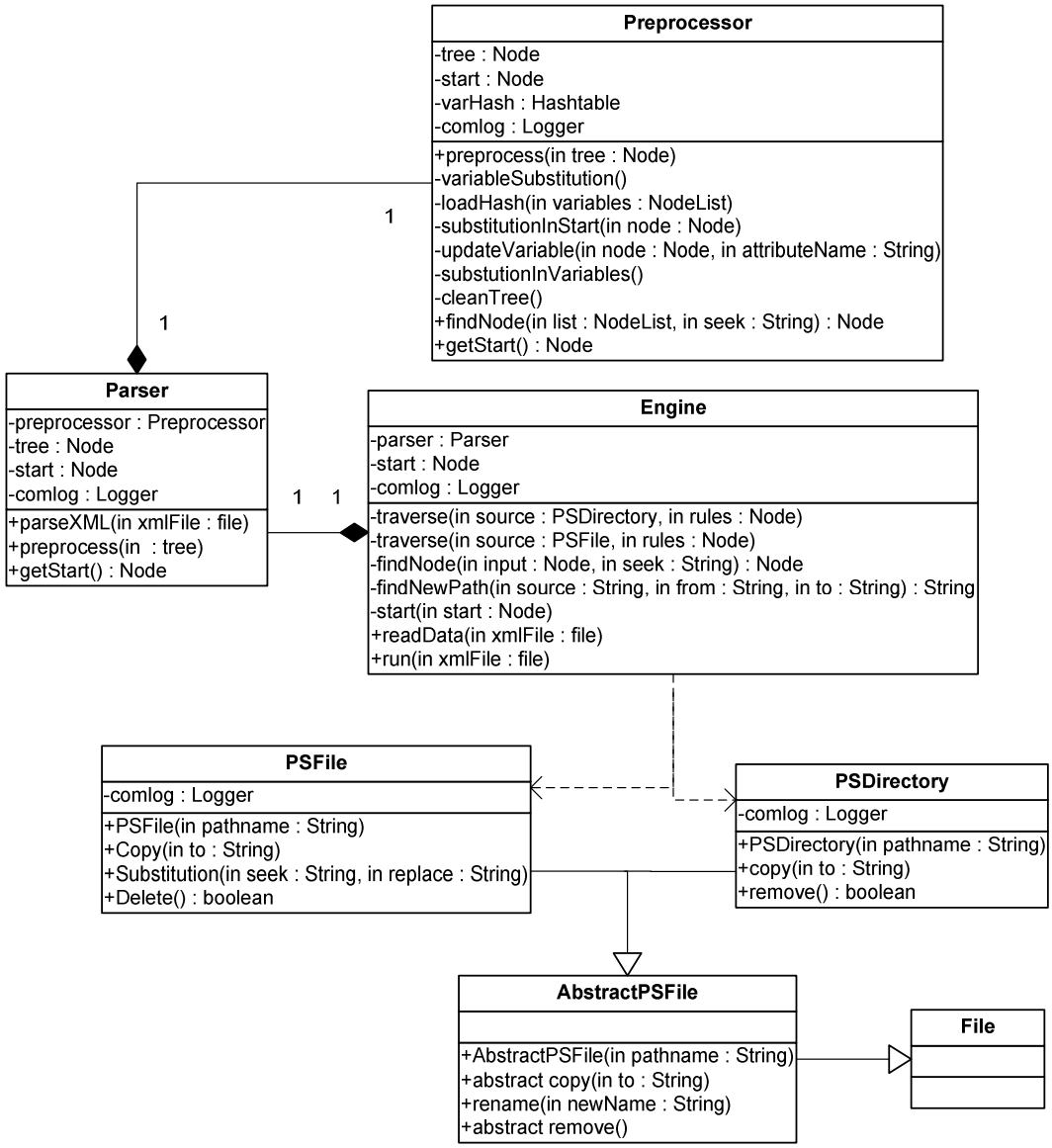
public abstract remove(): **(5)**
Similar to copy() method above.

public rename(String newName): **(3)**
Since both files and directories are treated as Files in Java, renaming them is done the same way.

3.3 UML Diagram

The engine class is at the heart of our design. It has a Parser to parse and preprocess XML files and will use PSFile and PSDirectory when traversing the "start tree." Parser has a Preprocessor so that all preprocessing is done through the Parser. Both PSFile and PSDirectory inherit from AbstractPSFile, which inherits from java.io.File. Figure 2 further illustrates the relationships between all classes.

Figure 2: UML Diagram



3.4 XML Design

3.4.1 Formatting Specifics

The XML config file must adhere to a strict format in order for the system to work properly. The following are the specifics of each tag needed to create a working XML config file. A listing of valid locations for variable placeholders is above in section 3.2 under the Preprocessor class. File actions should be listed most specific at the top.

`<project-mapping>`
Contains all specifics for the project. XML must contain one.

`<project-name>`
Contains the project name. Optional.

`<project-description>`
Contains a description of the project. Optional.

`<variables>`
Contains all variables that can refer to other variables or contain information for use in the start tree. No limit on number of variables. XML must contain one.

`<var name="" value="" />`
Details a single variable. The name and value of the variable are the attributes. To use a variable put ${your_variable_name} where you want the value for that particular variable to be placed.

`<start>`
Contains all directories and files to be modified and the rules that detail what modifications are to be made.

`<dir from="">`
Can contain other dirs or files. Must have an action block. The XML must contain at least one dir. Attribute from is the location where the dir currently is that the user wants to manipulate.

`<file from="">`

`<actions>`
Every dir and file must have an actions block detailing what is to be done to the dir or file.

`<copy to=""/>`

Copies the file or directory to where the to attribute specifies. If the name of the file or directory is different from the from attribute it will rename and copy at the same time.

`<rename to=""/>`
Renames the file or directory to what the to attribute specifies.

`<subs>`
Contains all the substitutions for find/replace within a file. Only applies to files.

`<sub seek="" replace=""/>`
Attributes seek and replace specify what to look for in a file and what to replace it with.

`<delete boolean="false"/>`
Attribute boolean specifies whether the file or directory should be deleted.

3.4.2 XML Start Tree

Figure 3 is a basic layout of what the start node looks like in the XML config file. Each directory and file has an action block detailing all the changes needed to be made. Directories and files do not necessarily correspond directly to a specific directory or file in the operating system. Using regular expressions and wildcard characters groups or specific types of files or directories can be changed.

Figure 3: XML Start Tree Structure

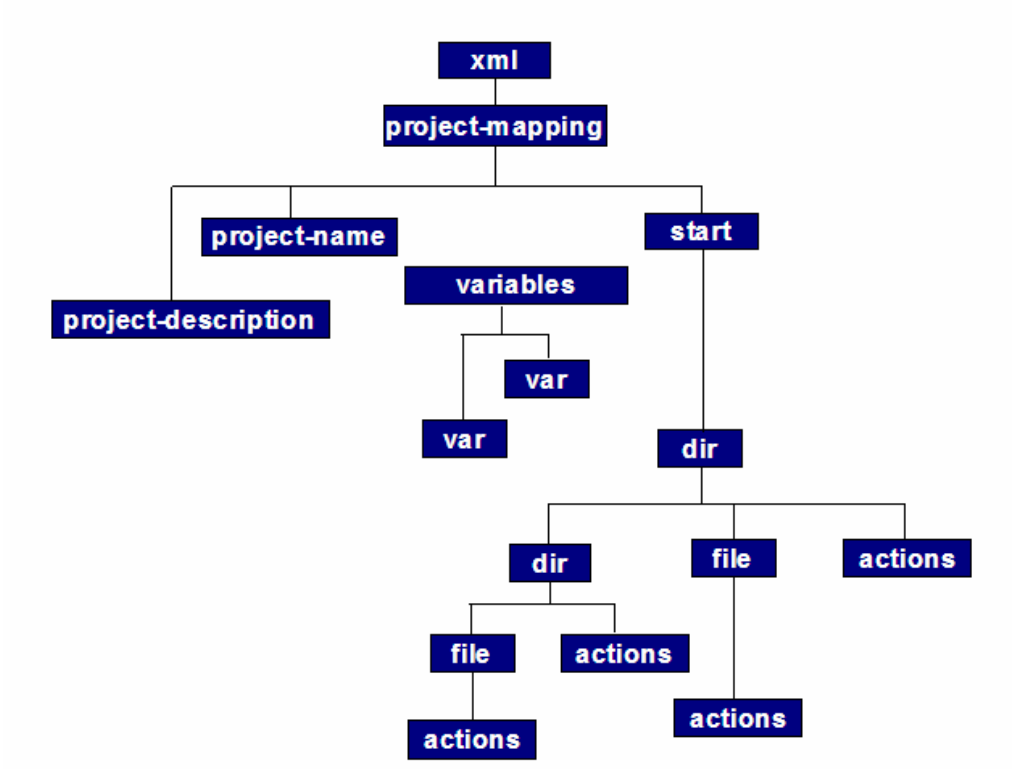
```
<start>
      <dir from="">
            <dir from="" >
                  <file from="">
                        <actions>
                              <copy to=""/>
                              <rename to=""/>
                              <subs>
                                    <sub seek="" replace=""/>
                              </subs>
                              <delete boolean="false"/>
                        </actions>
                  </file>
                  <actions>
                        <copy to=""/>
                        <rename to=""/>
                        <delete boolean="false"/>
                  </actions>
            </dir>
            <file from="">
                  <actions>
```

```
                        <copy to=""/>
                        <rename to=""/>
                        <subs>
                              <sub seek="" replace=""/>
                        </subs>
                        <delete boolean="false"/>
                  </actions>
            </file>
            <actions>
                  <copy to=""/>
                  <rename to=""/>
                  <delete boolean="false"/>
            </actions>
      </dir>
</start>
```

3.4.3 Parsed XML Tree

After the XML file is parsed it will be stored in a tree. This tree structure can be seen in figure 4. The processed tree that will be executed by the engine is the entire start branch on the right. Sample XML code for this start branch is detailed above in section 4.4.2. Each tag in the XML file corresponds to a node in the tree.

Figure 4: Parsed XML DOM Tree



4 Implementation Details

There are many pieces of the design that were updated during implementation. Inside the XML file, variable dependency support was added so that variables can depend on other variables. The decision was also made to execute actions (copy, rename, etc.) on the original directory or file that the actions block resides in. Before that decision all actions were executed on whatever file or directory had been detailed in the copy tag. We used paired programming extensively to catch errors in programming and improve code as we worked.

4.1 Technical Requirements

The entire project was written in Java 1.5. The client requested Java as the language and their organization is currently using 1.5 so this project should be compatible within their codebase.

The Apache Xerces library was used to do Document Object Model (DOM) XML parsing. Our client suggested using this library, which supports both the Simple API for XML (SAX) and the Document Object Model (DOM) for XML parsing. Comverse has a review board that approves the use of third party packages or libraries. The Xerces library has already been approved and is already being used in other projects. Upon further research we found that DOM is the better technique for our implementation when compared to SAX. DOM builds an entire tree of the XML file, which our project traverses to execute changes [1].

The project can be built with Apache Ant, which was also requested by our client.

Debug statements, error messages, and exceptions are all written to a log file using Log4j. Log4j is an easy to use system for logging within Java and is another piece if open source software from Apache. All classes within the project have a Logger to write to the log file.

4.2 Issues Resolved

While attempting to do renames and deletes of files, the engine would only change random files. After some investigating we found that during the file copy we would open a filestream but never actually close the file stream, so when the file was trying to either be renamed or deleted it would just ignore the file as write-protected. To fix this we simply closed the file stream.

Another issue was that when we implemented delete it "worked" but created null pointer errors during our testing. We then noticed that delete was actually being done before the engine would traverse its children or rename, and because of this it would not "exist" and so when trying to access the parent the

engine would error out. Fixing this involved simply moving the delete part of traverse to be the last action taken, creating a semi "order of operations."

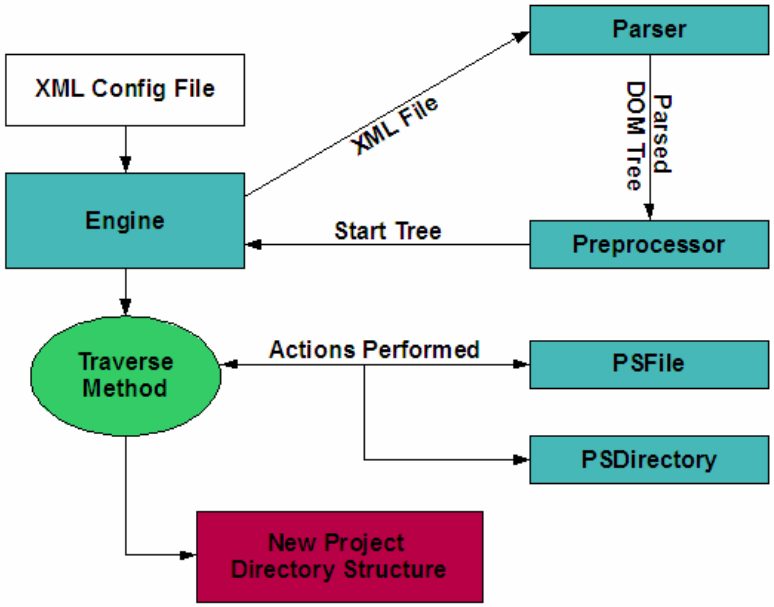
The XML formatting went through several versions including our initial idea of having everything classified based off of the action being performed, i.e. group all copies together. Then for user ease we switched to a formatting that not only made better sense conceptually but also mimicked the file structure. From this idea we made a few adjustments including how to define what actions to be performed, solved by adding an actions tag, and also what to perform it on, which we have finally decided on the source (whether you copy or not the action is preformed on the source not where you copied to).

We cleaned the code for traverse in order to find a bug in the code that was causing a slight issue in our result. Through this process we identified the actual error to be a regular expression issue, in which (.*) is a greedy regular expression, which would match all patterns and not just the ones we wanted. After some searching we found that changing it to (.*?) made the pattern matching work correctly.

4.3 Project Diagram

The entire project flow is detailed below in figure 5. An XML config file is passed to the Engine. The Engine then passes the XML config file to the Parser. Parsing and preprocessing produce a "start tree" ready to be traversed. The traverse method in Engine recursively moves through the "start tree" and performs actions (copy, rename, etc.). When Engine finishes running, a new directory structure is the result.

Figure 5: Project Diagram



5 Scope

Emphasis was placed on developing an engine for the source generator first. This needed to be flexible and useful for a wide variety of projects. A GUI wizard to receive user input and update the XML config file was a secondary task. The "Engine" part of the project was completed successfully. It consists of packages for parsing and preprocessing, files and directories, and the engine itself. Due to many different challenges during implementation we did not have enough time to complete the GUI part of the project.

6 Conclusions

This was a challenging, but very rewarding project. We were able to complete a quality "engine," but did not have enough time to create a GUI. The GUI is a possible future add on. Our idea for it was to have a JTable generated from the variables in an existing XML config file and then update the values of the variables. The absolute path of the XML template file would be provided to the GUI so that it could be modified. After making changes, the GUI would call the Engine class to execute all the changes detailed in the XML config file.

The project should be useful for those in Comverse Professional Services that will not have to manually ready a project template for a custom plugin.

Our client was responsible for testing and was satisfied with the result. Most errors encountered stemmed from incorrect XML usage. The other errors were small and easily fixed. Overall, it was a successful and worthwhile project.

References

[1] "Parsing XML Efficiently," http://www.oracle.com/technology/oramag/oracle/03-sep/o53devxml.html. Accessed May 18, 2007.