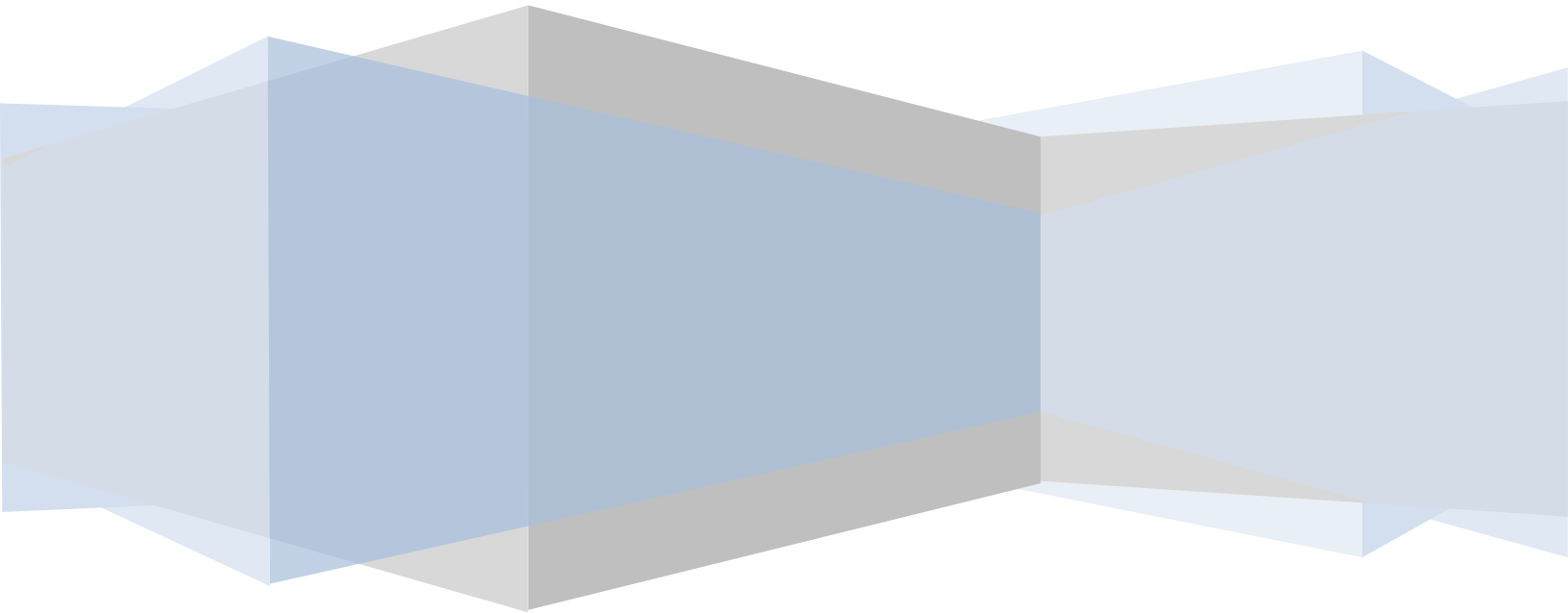
Abed Tony BenBrahim
ba2305@columbia.edu

# JTemplate: Another Approach to Document Template Markup Languages

## COMS W4115 Project Proposal



# Introduction

In domains such as web page development and model driven software development (MDSD), the automated generation of documents is most often accomplished by the use of processing instructions written in a markup language, embedded directly into a document template, to conditionally or iteratively generate sections of the document or programmatically insert content. Mainstream web development technologies such as Java Server Pages, PHP, Ruby on Rails and Microsoft's ASP(X) all employ this method of embedding code inside a document template to generate dynamic content. In the same fashion, popular MDSD frameworks such as openArchitectureWare (openArchitectureWare 2009) make extensive use of templates to transform application models to source code.

The use of templates with embedded processing instructions greatly accelerate the initial development of document generation applications. However this methodology imposes severe penalties in terms of future maintainability. As the ratio of markup instructions to template content increases (Figure 1), it becomes increasingly difficult to locate and correct defects. Often, the template becomes a "write once" document, with each modification becoming prohibitively more expensive and error-prone to undertake.

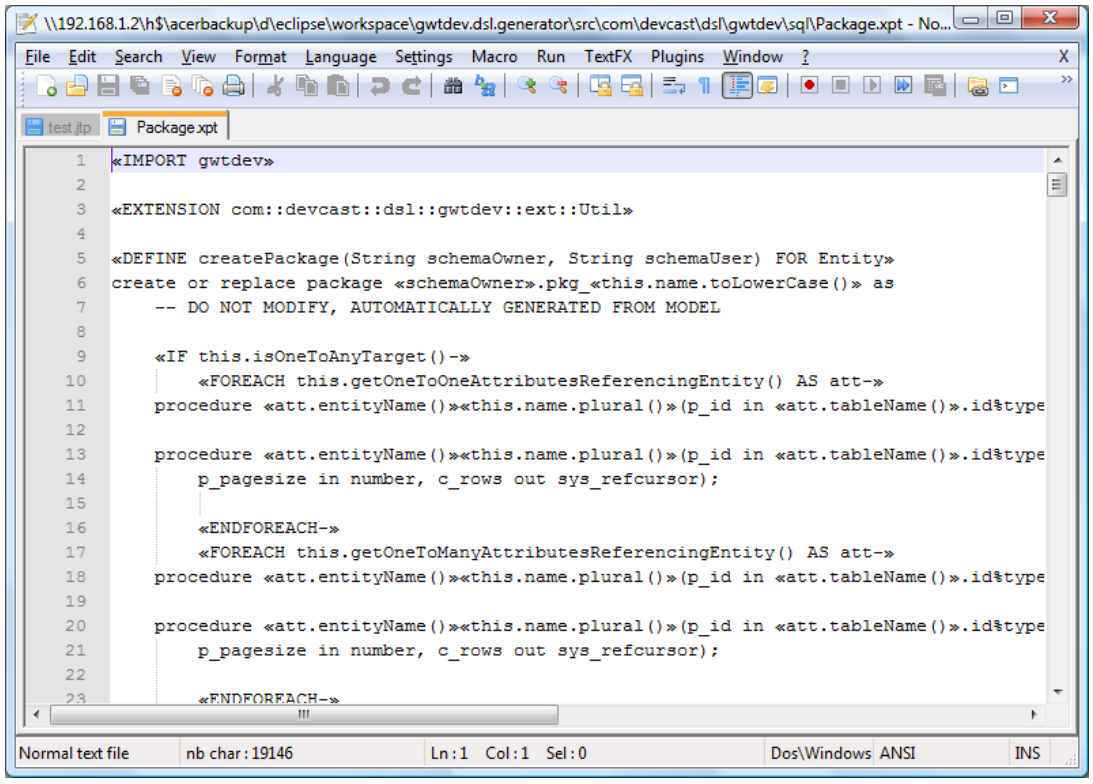



**Figure 1- A view of an openArchitectureWare template to generate an Oracle PL/SQL package**

Several approaches have emerged to achieve the holy grail of separating markup instructions from content. The Apache Wicket web framework (Apache Wicket 2009) takes advantage of the tree structure of HTML to insert content into specified nodes, an approach that works well for

XML like and other hierarchical structures but presents severe challenges for less structured content such as a source code. Terrence Parr's StringTemplate library (Parr 2004) makes use of minimal template markup and an output grammar to generate dynamic content. While this is a promising approach, the use of output grammars and complex recursive constructs has hindered the widespread adoption of this technique. The JTemplate language proposed in this document explores another method of providing clear separation of markup instructions from content.

## The JTemplate Language

### *Defining Templates*

In the JTemplate language, templates are first-class constructs. The following code snippet shows a template for a Java bean, with each line of the template annotated with an optional label (an integer or an identifier) and a start of line marker.

```
template javaBean{
1       #package mypackage;
        #
2       #import java.util.Date;
        #
3       #public class MyClass{
        #
4       #       private Type field;
4       #
5       #       public MyClass(){
        #       }
        #
8       #       public MyClass(fieldList){
9       #               this.field=field;
8       #       }
        #
6       #       public Type getField(){
6       #               return this.field;
6       #       }
6       #
7       #       public void setField(Type field){
7       #               this.field=field;
7       #       }
7       #
        #}
}
```

The labels serve to delineate individual lines or blocks of code that can be manipulated by processing instructions. Likewise, processing instructions are first class constructs in the JTemplate language, as shown in the following code snippet:

```
instructions for javaBean(appOptions, entity){
1 once: mypackage = endWith('.',appOptions.basePackage) + 'model';
```

```
2 when (hasDateField(entity));
3 once: MyClass = toFirstUpper(entity.name);
4 foreach (field in entity.fields): Type=javaType(field), field=field.name;
5 once: MyClass = toFirstUpper(entity.name);
6 foreach (field in entity.fields):
       Type=javaType(field), field=field.name, Field=toUpper(field.name);
7 foreach (field in entity.fields):
      Type=javaType(field), field=field.name, Field=toUpper(field.name);
8 once : MyClass = toFirstUpper(entity.name), fieldList=getFieldList(entity);
9 foreach (field in entity.fields): field=field.name;
}
```

A processing instruction consists of a line or block label matching a label in the corresponding template, an expression specifying how the instruction should be processed (once, conditionally or iteratively) and an optional list of text replacements. Additionally, processing instructions accept arguments in the same way as a function declaration.

### *The JTemplate Language*

The remainder of the JTemplate language consists of a subset of ECMAScript 5 (ECMA International April 2009) operating in strict mode. JTemplate is an interpreted, dynamically typed language. First-class types consist of integers, strings, doubles, Booleans, arrays, maps and functions. All variables must be declared and initialized before being referenced. Control structures include if/else, while, foreach and switch. The following incomplete code snippet shows how the JavaBean template described above might be generated:

```
var model={
  entities: [
    {name: 'customer', fields: [
       {name: 'lastName', type: 'char',  maxLength: 50},
       {name: 'firstName', type: 'char', maxLength: 50}
    ],
    references: [
      {entity: 'address', cardinality: 'one-to-many'}
    ]
  },
    {name: 'address', fields: [
      {name: 'address1', type:'char', maxLength: 100},
      {name: 'address2', type:'char', maxLength: 100}
     ]
    }
  ]
};

var appOptions={basePackage: 'edu.columbia.w4115'};

var hasDateField=function(entity){
  foreach (field in entity.fields){
    if (field.type=='date'){
      return true;
    }
  }
```

```
  return false;
};

var main=function(){
  var path='gen/'+replace(appOptions.basePackage,',','/')+'/model/';
  mkDirs(path);
  foreach(entity in model.entities){
    var text=javaBean(appOptions, entity);
    writeFile(path+toFirstUpper(entity.name)+'.java', text, true);
  }
};
```

### *Built in Libraries*

The design of the built in library functions reflect JTemplate's primary use as a string manipulation and file generation language.

#### String manipulation Functions

| Name | Description |
|---|---|
| length(s) | Returns the length of string s |
| charAt(s,i) | Returns the character at index i in string s |
| indexOf(s,ss) | Returns the 0 based index of substring ss in string s |
| replace(s,ss,rs) | Replaces all occurrences of ss with rs in string s |
| split(s, sep) | Returns an array of substrings from s separated by sep |
| toLowercase(s) | Returns s lowercased |
| toUppercase(s) | Returns s uppercased |
| toFirstUpper(s) | Returns s with the first letter uppercased |
| toFirstLower(s) | Returns s with the first letter lowercased |
| endWith(s,c) | Ends s with string c if it does not already end with c |

#### File manipulation Functions

| Name | Description |
|---|---|
| writeFile(name,s) | Writes string s to file at path given by name |
| mkDirs(s) | Creates the directories indicated by string s |
| fileOpen(s,mode) | Returns a handle for file s with the open mode "r","rw","w" or "a" |
| fileClose(f) | Closes file handle f |
| readLine(f) | Reads a line from file handle f into a string |
| writeString(f, s) | Writes string s to file handle f |
| fileExists(s) | Returns true if file s exists |

#### Language Functions

| Name | Description |
|---|---|
| isDefined(v) | Returns true if v is defined |
| undefine(v) | Undefines variable v |
| typeOf(v) | Returns the type of v as a string |

# Project Plan

### *Schedule*

| Week Ending | Milestone |
|---|---|
| May 30, 2009 | Setup environment, Preliminary lexer, parser and AST, Project Proposal |
| June 6, 2009 | Finalize parser and AST, symbol table implementation, Interpreter for basic constructs (no templates) |
| June 13, 2009 | Testing framework, Language Reference Manual |
| June 20, 2009 | Language Reference Manual |
| June 27, 2009 | Built in Library Implementation |
| July 4, 2009 | Interpreter for template instructions |
| July 11, 2009 | Semantic analysis (detect error in template labels, unused variable warnings, etc…) |
| July 18, 2009 | *Open* |
| July 25, 2009 | *Open* |
| August 1, 2009 | Final Report |
| August 8, 2009 | Final Report |

### *Development Environment*

Project web site: http://code.google.com/p/ojtemplate/

Source code repository: Anon SVN http://ojtemplate.googlecode.com/svn/trunk/jtemplate/

IDE: Eclipse/OcaIDE

Build System: ocamlbuild

# Works Cited

*Apache Wicket.* http://wicket.apache.org/.

ECMA International. *ECMAScript Language Specification (Final Draft Standard ECMA-262).* Geneva: ECMA International, April 2009.

*openArchitectureWare.* http://www.openarchitectureware.org/.

Parr, Terrence. "Enforcing strict model-view separation in template engines." *Proceedings of the 13th international conference on World Wide Web.* New York: ACM, 2004.