Metadata driven multi-artifact code generation using Frame Oriented Programming

Frank Sauer
frank.sauer@trcinc.com
The Technical Resource Connection
A wholly owned subsidiary of Perot Systems Corporation

Abstract
This paper describes Frame Oriented Programming (FOP) and shows that this programming technique and its implementation in the *Frame Processing Language (FPL)* is an effective form of template driven code generation and ideal for Software Product Lines and Model Driven Architectures. It demonstrates how multiple artifacts in different target languages can be generated in a single pass from arbitrary XML based meta-data, such as XMI [6] or W3C XML schemas and how frames modularize templates using Object oriented as well as Aspect Oriented techniques.

Introduction to FOP
Frame Oriented Programming is based on Frame Technology which was invented by Paul G. Bassett in 1978 but first published in [1] and culminated in his milestone 1996 book titled "Framing Software Reuse: lessons from the real world[2]". In one sentence, frames are a universal mechanism to package information into components and participate in an automated assembly process in which frames adapt – and are adapted by – other frames. Frames are not particular about the language used to describe the packaged information; this could be any conceivable programming language or even natural languages such as English. This feature of frames makes them an ideal concept for template driven artifact generation. The ability to apply frame technology to arbitrary text allows the modularization techniques of Object Oriented Programming (OOP) as well as Aspect Oriented Programming (AOP) [4] to be applied to any textual artifact, which was previously impossible. [7] Explains how FOP unifies the ideas of both OOP and AOP.

Frame hierarchies
A frame hierarchy defines how frames are combined to make other frames. Frame hierarchies can be thought of as a parts-explosion diagram. A higher-level frame is an assembly of its lower level frames and the arrows in a frame hierarchy diagram can therefore be labeled as '*part-of*'. Figure 1 shows an example frame hierarchy. In this frame hierarchy the top-level frame (or specification frame) `a` has two subassemblies, `b` and `c`. The `b` and `c` frames themselves can be considered specification frames for their respective sub-hierarchies. Frame `e` is part of both `b` and `c`; however, `b` and `c` receive separate copies of frame `e` in the assembly process. The reverse of the part-of relationship between frames is far more interesting and forms the basis for the rest of the ideas set forth by FOP. When frames `b` and `c` are part of frame `a`, frame `a` is said to '*adapt*' frames `b` and `c`.

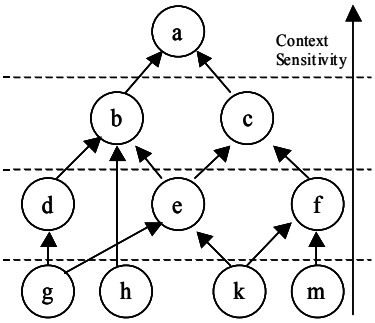

Figure 1 Example frame hierarchy

The adapt relationship on frames
The properties of the adapt relationship can be expressed as follows (from [1], let ≥ denote the adapt relation, and X and Y and Z be frames):

$\forall X:$	$X \geq X$	(reflexive)
$\forall X,Y,Z:$	$X \geq Y \wedge Y \geq Z \Rightarrow X \geq Z$	(transitive)
$\forall X,Y:$	$X \geq Y \wedge Y \geq X \Leftrightarrow X = Y$	(non-symmetric)

These three properties imply that ≥ imposes a partial ordering on the frames. This means that frames indeed form a lattice as shown in Figure 1 and that the terms *frame* and *frame hierarchy* can be used interchangeably. Another way to think of frames adapting each other is to rename the adapt relationship to "*reuses*". In figure 1, frame `a` reuses frames `b` and `c`. In general, if a frame X reuses a frame Y, then frame Y is as reusable as X, or more so. This is easy to see because whenever X is reused, so is Y, but Y can be used in another frame, say Z, without using X. This is the reason why frame hierarchies are drawn the way they are, with the most context-specific frame (the specification frame) at the top, and the most context-free - and therefore most reusable - frames near the bottom of the diagrams.
In the remainder of this paper I will use the term *ancestor frame* to mean the following:

> Frame X is an ancestor of frame Y if X directly or indirectly adapts frame Y. This will place frame X closer to the specification frame than frame Y.

Frame commands

In addition to the content being assembled, frames contain assembly instructions called *frame commands*. This section will examine the most relevant frame commands. The syntax used in this section and the rest of the paper is from the author's frame processing language called FPL. This is an XML based implementation of frame technology and uses XML tags for the frame commands and XML content as the information to be assembled, much like XSLT does in XML transformations. The FPL syntax was also inspired by XVCL (XML based Variant Configuration Language [3]). Here is a basic example of a frame in FPL:

```
<frame name="f1" language="text">
 this is normal content
 <break name="optional">
 this is the default content
 </break>
</frame>
```

Each FPL frame is enclosed in a root <**frame**> tag and contains content and other FPL commands. This example has two lines of content with the second line embedded inside an FPL <**break**> command. The <**break**> command and its companion <**adapt**> are the core FPL commands that enable frames to adapt other frames.

> A <break> is a named point of variation allowing ancestor frames to modify the frame containing the break.

An ancestor frame adapting frame `f1` does so with the <**adapt**> command which is the *only* mechanism by which frames are combined. Figure 2 shows the XML syntax of the <**adapt**> command.

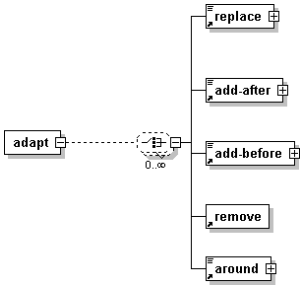

Figure 2 the <adapt> command structure

The syntax in Figure 2 shows that a frame adapting another frame has an opportunity to replace, remove or add content (and more FPL commands) to descendant frames. These actions occur at the breakpoints labeled by the <**break**> commands in the descendant frames. The allowed content of the <**replace**>, <**add-before**>, <**add-after**> and <**around**> commands is any sequence of valid FPL commands, not just content. These FPL commands are executed *in the context of the* <**break**> *being adapted*, as if they were written in the frame containing the <**break**>. This is an important feature of FPL and will be explored in much more detail in following sections. If no ancestor frame removes or replaces the break, the original default content of the break will be evaluated. The following example is a frame that replaces the optional content of frame `f1` with alternative content.

```
<frame name="f2" language="text">
 <adapt frame="f1">
  <replace break="optional">
  this is alternative content
  </replace>
 </adapt>
</frame>
```

Here is yet another frame `f3` that adapts `f2` and inserts some extra content after the break:

```
<frame name="f3" language="text">
 <adapt frame="f2">
  <add-after break="optional">
  second line of alternative content
  </add-after>
 </adapt>
</frame>
```

We now have a three level frame hierarchy; `f3` adapts `f2`, which in turn adapts `f1`. The result of running `f3` through the frame processor is a text file with this content (ignoring indentation):

```
this is normal content
this is alternative content
second line of alternative content
```

Note that `f3` directly modifies a <**break**> defined in `f1`, an *indirect* descendant. Frame `f2` - being adapted by `f3` - also adapts the same <**break**> in `f1`. In general, any frame along the path from the specification frame to a <**break**> can adapt that <**break**>. Table 1 enumerates the precedence rules FPL follows to determine which adapt operations to perform. In this table, a value of "*ancestor*" means that only the ancestor operation will be performed and a value of "*both*" indicates that both operations execute. The columns are ancestor operations and the rows are descendant operations on the same break.

D \ A	remove	replace	around	add-before	add-after
remove	ancestor	ancestor	both	both	both
replace	ancestor	ancestor	both	both	both
around	ancestor	ancestor	*ancestor*	both	both
add-before	ancestor	ancestor	both	both	both
add-after	ancestor	ancestor	both	both	both

Table 1 Precedence rules for adapt operations

In general, anything added before, after or around a break becomes part of the break and ancestor

<**remove**> or <**replace**> operations take precedence over *all* operations performed on the same break at lower levels in the frame hierarchy because higher level frames have more knowledge of the integration context and are thus better suited to decide how to modify the sub-assembly represented by the lower level frames. The <**add-before**> and <**add-after**> operations have a cumulative effect, and are executed top-down and bottom-up respectively. There is no theoretical reason why only one <**around**> should be allowed. This is a current implementation deficiency of FPL and will be fixed in a future version. At that time, the <**around**> operation will behave like <**add-before**> and <**add-after**> and the results will be cumulative.

The <**break**> command allows for a seamless evolution of components over time because the insertion of an extra <**break**> in an existing frame does not alter its structure or behavior, unless an ancestor frame is explicitly adapting it. This means that frames developed in one context can be easily evolved and adapted to other unanticipated contexts and this will not have *any* effect on existing ancestor frames already reusing the frame. Since these existing ancestors are unaware of the new <**break**>, its original content will be used. In other words, instead of changing the frame itself, it becomes (more) changeable, but the change occurs where it is needed, namely in the new context (the new ancestor frame). This is a critical feature of FPL that makes it ideal for application in a Software Product Lines [5]. Figure 3 depicts the evolution of a component over time. During the development of App2, a modification of C2 is needed. C2 is modified by surrounding some existing content in C2 in a <**break**> command and App2 uses this new <**break**> to modify C2 in the context of App2. When App1 needs to be rebuilt, we can safely use C2' for the reasons outlined above.

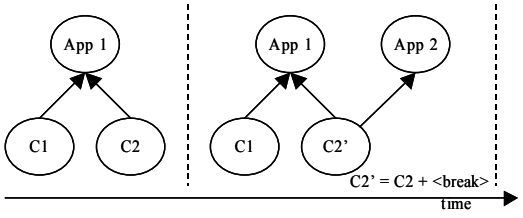

Figure 3 Evolution of Components

Frame Variables

In addition to the <**break**>/<**adapt**> mechanism outlined in the previous section, FPL supports variables as another mechanism to customize frames. FPL variables follow a scope rule similar to the precedence rule for <**adapt**> operations:

> The scope of a variable *v* defined in frame X is the entire sub-hierarchy rooted at X and the value of *v* will override the value of any variable named *v* defined in the sub-hierarchy rooted at X.

In other words, here too, context-sensitive overrides context-free. This allows for default values to be set anywhere in a frame hierarchy and override the defaults in any ancestor frame whenever the context dictates that they should have a different value. Variables are set with the <**set**> command and in addition to a name and a *value* can contain any number of named *facets*, each with its own value. In the following example a variable named class1 is created with two facets:

```
<set var="class1" value="SavingsAccount">
  <facet name="template" value="bean.fpl"/>
  <facet name="superclass" value="Account"/>
</set>
```

FPL commands can access variable and facet values in any of their attributes, as in the following example:

```
<adapt frame="${class1.template}" outfile="${class1}.java">
  <replace break="extends">extends ${class1.superclass}
  </replace>
</adapt>
```

The ${} syntax[1] to access variables allows for variables to be used to construct the name of other variables, e.g. in the previous example ${class${index}} will evaluate to "SavingsAccount" if ${index} has the value 1. This allows for the representation of elaborate data structures with simple variables. The use of variables in the *frame* attribute of the <**adapt**> command allows the structure of the frame hierarchy itself to be variable. In this example, the SavingsAccount is constructed from a generic Javabean frame generating a Java class according to the Javabean specification.

As stated in the previous section, the <**adapt**> operations can contain any valid FPL command, and this includes the <**set**> command. The previous section also defined that any commands contained in an <**adapt**> operation execute in the context of the <**break**> being adapted. In the case of the <**set**> command this implies that the scope of the variable will be the frame containing the <**break**>, *not* the frame in which the variable is being defined. In fact, the <**set**> command is not even executed until the <**break**> is reached and the precedence rules dictate that the <**adapt**> operation should be performed. This means that the value of a variable defined inside an <**adapt**> operation can depend on any variables defined in any frame between the adapting frame and the <**break**> being adapted. The scope of a variable defined in the context of an <**adapt**> operation can be modified to be the *defining* frame rather than the <**break**> in which it is being executed using the *samelevel="true"* attribute on the <**set**> command. This allows for information to be passed *up* the frame

[1] Adopted from the popular xml based build tool Ant

hierarchy, as opposed to the normal flow down the hierarchy. The <**adapt**> command itself also supports the *samelevel* attribute which when set to true will lift up the scope of all variables defined in the adapted frame to that of the adapting parent frame.

In addition to the scope modifier *samelevel*, the evaluation of any ${} variable references within the value or facets of a variable can be deferred to the time the variable itself is being referenced by adding *defer="true"* to the <**set**> command.

Functions

Often, the value of a variable needs to be modified in some way to be useful in the context where it is being used. For example, if a variable contains a java package name and is used to define both the package and the directory structure, the '.'s in the package name need to be replaced with '/'s. To support this kind of functionality FPL defines a number of pre-defined functions as well as an extension mechanism to add custom functions (written in java) to the language. The following example frame shows how a package name can be morphed into a directory name using the replace function. Note the use of double brackets for the start and end of the parameter list. This syntax was chosen in order not to conflict with any of the languages that might be used as content.

```
<adapt frame="${class1.template}"
        outdir="replace[[${package},.,/]]
        outfile="${class1}.java">
  <replace break="extends">extends ${class1.superclass}
  </replace>
</adapt>
```

Other functions include substring, date, uppercase, lowercase, index-of, sum, length, etc. Functions can be used anywhere variables can be used. At this time FPL does not support infix expressions, only functions and variables.

More FPL Commands

Iteration over a sequence of FPL commands is achieved with the <**while**> command. This command can either iterate over a set of list variables or use the existence of a variable as its terminating condition. The following two examples demonstrate each technique. The index attribute defines the name of a variable that will be incremented on each iteration of the <**while**> command.

```
<while defined=" class${c}" index="c" start="1">
<adapt frame="${class${c}.template}"
        outdir="replace[[${package},.,/]]
        outfile="${class${c}}.java">
  <replace break="extends">extends ${class${c}.superclass}
  </replace>
</adapt>
</while>
```

The above example will iterate over the variables named *class1, class2, etc.* for as long as such a variable exists. For each class it will then adapt the appropriate template and generate a separate file named after the class. The next example will do something very similar, but using a list variable to do the iteration.

```
<set var="classes" list="Account,AccountList"/>
<set var="templates" list="bean.fpl,list.fpl"/>
<while listvars="classes,templates" >
  <adapt frame="${templates}"
        outdir="replace[[${package},.,/]]
        outfile="${classes}.java">
  </adapt>
</while>
```

Note that no index variable is needed to access the elements of the lists. Within the body of the <**while**> command, references to the list variables being iterated over will automatically advance to the next element on each iteration. Note also that both lists must have the same number of elements.

The main conditional control structure in FPL is the <**select**> command. Figure 4 shows the syntax of this command and the following example illustrates it.

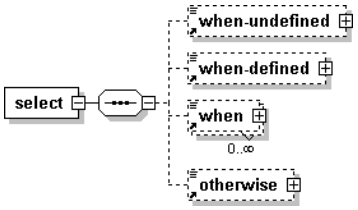

Figure 4 structure of the <select> command

```
<select var="v">
  <when-undefined>
variable v does not exist
  </when-undefined>
  <when-defined>
variable v exists
  </when-defined>
  <when value="1" comparator="greater-than">
The value of v is greater than 1
  </when>
  <when value="foo">
The value of v is "foo"
  </when>
  <otherwise>
Otherwise should not execute in this example
  </otherwise>
</select>
```

Only one <**when-undefined**> is allowed inside a <**select**> and it has to be the first child. <**when-defined**> can also exist only once. Any number of <**when**> clauses can be defined and every one for which the comparator evaluates to true with the

variable's value on the left hand side and the <**when**> value on the right hand side will be executed. If none of the previous clauses was executed and there is an <**otherwise**> clause it will be executed.
In many situations the syntax of <select> is too verbose and for this reason FPL defines two commands <**ifdef**> and <**ifndef**> which serve as syntactic sugar for the corresponding version of <**select**> without an <**otherwise**>, e.g. <**ifdef** var="v"> is equivalent to:

```
<select var="v">
  <when-defined>
  </when-defined>
</select>
```

<**ifndef** var="v"> is the same as:

```
<select var="v">
  <when-undefined>
  </when-undefined>
</select>
```

The FPL language modifies the original Frame Technology as described in [1] and [2] and XVCL [3] in some fundamental ways. The differences are these:

- Addition of <**around**>/<**proceed**>,
- Use of regular expressions in the <**adapt**> operations to match <**break**> names,
- Addition of functions,
- Modified variable syntax and new or modified commands,
- Use of XML namespaces to make framing XML content easier.
- In XVCL, the precedence rule for <**add-before**> and <**add-after**> is *ancestor* and the effects are not cumulative.
- Integration with XSLT through the <**transform**> command,
- Commands for dynamic XML construction with the same semantics as <**xsl:element**>, <**xsl:attribute**> and <**xsl:comment**>.

Generating multiple artifacts from UML

FPL was initially developed to turn a diverse family of applications for the insurance industry into a software product line [5] customizable to the insurance product each application deals with. The insurance products are described with XML metadata that is transformed (using XSLT) into FPL variables and facets that drive the customizations and assembly of generic components to create individual members of the application family. A single specification frame drives the customization of multiple artifacts in the areas of business logic (EJBs), presentation logic (JSP and HTML), configuration files (XML and property files) and database schemas.

FPL supports the use of XML metadata and XSLT transformations with a <**transform**> command. The following examples show how to generate Java code and W3C XML schemas from UML models represented in XMI metadata [6]. Similarly, W3C XML schemas themselves are a good source of metadata for generating custom Java/XML bindings. The construction time architecture of a group of XML based applications is depicted in Figure 5. Each application (App1, … 3) defines a different XMI source in a XMI variable. The XMLApp frame uses this variable and a fixed stylesheet to transform the XMI to a FPL data structure using variables and facets. This data structure is used by the lower level frames to customize the two template frames named *generic class* and *generic schema*. This decoupling of the actual metadata from the frames has the advantage that metadata can be easily changed (for example XMI-1.0 to XMI-1.1) simply by plugging in a different XSL stylesheet.

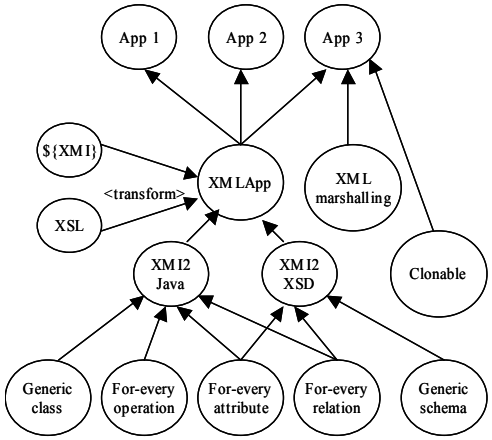

Figure 5 Generating multiple artifacts from UML metadata

The *XMI2Java* frame uses the metadata to generate a Java class for each class found in the metadata, the *XMI2XSD* frame uses it to generate elements in the schema for each class encountered in the metadata. Both these frames use common frames called *for-every-attribute* and *for-every-relation*. The following is the for-every-attribute frame:

```
<frame name="for-every-attribute" language="fpl>
<ifdef var="class-${classname}.attribute-list">
  <set var="attributes"
       list="${class-${classname}.attribute-list}"/>
  <while listvars="attributes" index="attr">
      <set var="attrname" value="${attributes}"/>
      <set var="typename"
           value="${class-${classname}.${attributes}-type}"/>
      <set var="initialvalue" value="null"/>
      <ifdef var="class-${classname}.${attributes}-initial">
        <set var="initialvalue"
         value="${class-${classname}.${attributes}-initial}"/>
      </ifdef>
      <break name="attribute"/>
  </while>
```

```
</ifdef>
</frame>
```

This frame accesses the transformed metadata representing the model and iterates over each attribute defined for a given class. It sets a few simple variables such as *attrname*, *typename* and *initialvalue* for use by the adapting ancestor frames. It does not define any output but instead allows this frame to be reused many times by both the XMI2Java and XMI2XSD frames. These frames <**adapt**> this frame and <**replace**> the <**break name="*attribute*"**/> with whatever they want to generate for each attribute in a class. Similarly, iterating over every relation defined in the UML model is encapsulated in this reusable frame[2]:

```
<frame name="for-every-relation" language="fpl">
<ifdef var="class-${classname}.relation-list">
 <set var="relations"
      list="${class-${classname}.relation-list}"/>
 <while listvars="relations" index="rel">
  <set var="min"
       value="${class-${classname}.${relations}-min}"/>
  <set var="max"
       value="${class-${classname}.${relations}-max}"/>
  <set var="typename"
       value="${class-${classname}.${relations}-type}"/>
  <set var="rolename" value="${relations}"/>
  <break name="relation"/>
   <select var="max">
    <when value="1">
     <break name="singlevalued-relation"/>
    </when>
    <otherwise>
       <break name="multivalued-relation"/>
    </otherwise>
   </select>
 </while>
</ifdef>
</frame>
```

This frame is slightly more complicated than the *for-every-attribute* because it decides from the metadata whether a relation is a single-valued or a multi-valued relation. This gives ancestor frames the opportunity to reuse this logic and to simply <**adapt**> the <**break**> for the applicable type of relation.

The following is the *XMI2Java* frame, which transforms the metadata and iterates over the classes in the model.

```
<frame name="XMI2Java" language="java>
  <break name="XMI2Java-Parameters"/>
  <break name="transform">
   <requires vars="XMI"/>
     <transform xml="${XMI}" xsl="/xmi/xmi2fpl.xsl"/>
  </break>
  <set var="classlist" list="${classes}"/>
  <requires vars="package"/>
  <while listvars="classlist" index="class">
    <set var="classname" value="${classlist}"/>
    <log message="generating class ${classname}"/>
    <adapt frame="generic-class.fpl"
              outdir="replace[[${package},.,/]]"
              outfile="${classname}.java"/>
  </while>
</frame>
```

The following is a fragment of the *generic-class* frame, which generates the attribute declarations using the for-every-attribute and for-every-relation frames:

```
<!-- attribute declarations -->
<adapt frame="/xmi/for-every-attribute.fpl">
  <replace break="attribute">
     <break name="${classname}-${attrname}-declaration">
 private ${typename} ${attrname} = ${initialvalue};
      </break>
  </replace>
</adapt>

<!-- allow ancestors to insert extra attributes -->
<break name="${classname}-attributes"/>

<!-- relationship declarations -->
<adapt frame="/xmi/for-every-relation.fpl">
   <replace break="singlevalued-relation">
     <break name="${classname}-${rolename}-declaration">
 private ${typename} ${rolename} = null;
    </break>
   </replace>
   <replace break="multivalued-relation">
    <break name="${classname}-${rolename}-declaration">
   private List ${rolename} = null;
    </break>
   </replace>
</adapt>
```

Note that the actual transformation in *XMI2Java* is enclosed inside a <**break**> command. Ancestor frames may decide to <**remove**> this <**break**> and do the transformation themselves so its results can be used in a context larger than the *XMI2Java* frame hierarchy. This applies to this example because the *XMLApp* frame shares the transformation between the *XMI2Java* and *XMI2XSD* frames, as shown in this frame:

```
<frame name="XMLApp" language="fpl">
 <set var="XMI" value="models/claimInfo.xmi"/>
 <!-- do the transform here so it can be shared between the
      xsd and java generation -->
 <transform xml="${XMI}" xsl="/xmi/xmi2fpl.xsl"/>
 <!-- generate the schema -->
 <adapt frame="XMI2XSD "
         outdir="schemas" outfile="app3.xsd"
         emit-xml-header="true">
  <remove break="transform"/>
```

[2] Code to deal with qualified relations has been omitted for brevity

```
  </adapt>
   <!-- generate the java code -->
 <adapt frame="XMI2Java ">
    <remove break="transform"/>
  </adapt>
</frame>
```

Observe in Figure 5 that *App3* consists not only of an *XMLApp* frame, but also includes the *xmlmarshalling* and *clonable* frames. These two additional frames each describe a separate concern that needs to be injected into the standard generated code of the *XMLApp* frame, only in the context of App3. The *xmlmarshalling* frame contains two methods, toXML() and fromXML(). The *clonable* frame contains a clone() and a deepClone() method that will recursively traverse all relationships defined in the model and generate a deep copy of the receiving object. This is a nice example of separation of crosscutting concerns, a capability emphasized by AOP[4], a capability of FOP that is outlined in more detail in [7].

Including method implementations

In contrast to most code generators, which generate method stubs at best, FPL is well suited to generate and more importantly, regenerate, entire object models, including the method implementations. The following fragment from the *for-every-operation* frame shows how this is accomplished.

```
<break name="${classname}-${opname}-method">
public ${returntype} ${opname} (${arguments}) {
<break name="${classname}-${opname}-implementation"/>
}
</break>
```

The *for-every-operation* frame picks up the method signature from the UML (XMI), and generates the appropriate method stub. However, it embeds a <**break**> which ancestor frames use to infuse the actual method implementation. For example, *App1* might include the following <**adapt**> operations:

```
<add-before break="Period-length-method">
   /**
    * calculate the length of the Period
    */
</add-before>
<replace break="Period-length-implementation">
            return to.getTime() – from.getTime();
</replace>
```

Since these commands execute each time the code is regenerated, the method implementation does not get lost when the code is regenerated, a classic problem with traditional code generators, and the resulting code is free from special comments and forbidden zones, which is usually the solution to this problem offered by non-frame-based code generators.

Conclusions

This paper has shown how Frame Oriented Programming is an effective method for template driven code generation and how it allows for the use of meta-data to drive the customization of generic components in the form of frame hierarchies. The mechanisms of FOP allow for a smooth evolution of components over time, which makes FOP based code generation ideally suited for Software Product Lines.

Acknowledgements

I want to thank Paul Bassett for inventing Frame Technology and for his constructive feedback on this paper. I also want to thank my colleagues; Joey White for giving me a reason to do this work, and Oscar Chappel for the many debates that served to improve FPL, this paper, and my understanding of CLOS.

References

[1] Paul G. Basset**,** "*Frame-Based Software Engineering*", IEEE Software, Vol. 4, No. 4, pp. 9-16, July 1987

[2] Paul G. Bassett, *Framing software reuse: lessons from real world*, Yourdon Press Computing Series, Prentice Hall, 1996

[3] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H.Y. *"XML Implementation of Frame Processor"* Symposium on Software Reusability, SSR'01, Toronto, Canada, May 2001, pp. 164-172

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Lopez, Jean-Marc Loingtier, and John Irwin. *Aspect Oriented Programming*. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 – Object Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220-242. Springer-Verlag, New York, NY, 1997

[5] Paul Clements, Linda M. Northrop. *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, 2001

[6] http://www.omg.org/technology/documents/formal/xmi.htm

[7] Frank Sauer, *"Frame Oriented Programming. A Unification of Object Oriented Programming and Aspect Oriented Programming"*. Unpublished, to be submitted to AOSD 2003.