

Frame Oriented Programming: A Unification of Object Oriented Programming and Aspect Oriented Programming

Frank Sauer

The Technical Resource Connection, Tampa, FL

A wholly owned subsidiary of Perot Systems

frank.sauer@trcinc.com

ABSTRACT

This paper describes Frame Oriented Programming (FOP) and shows that this programming technique unifies the Object Oriented concept of classes and the Aspect Oriented Programming (AOP) concept of aspects into a single coherent concept called a *Frame*. The unification is embodied in the *adapt* relationship between frames that allows frames to reuse and modify other frames in ways that unify the generalization and aggregation relations of Object Oriented Programming (OOP) and at the same time allows for the injection of crosscutting concerns into at “breakpoints” of arbitrary granularity.

Keywords: OOP, AOP, Frame Technology, Code Generation, Reuse, Mode Driven Architecture, Meta Programming, Generic Components, Software Product Lines, Software Development Methods

1. INTRODUCTION

Frame Technology 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 is a very important benefit of frames since modern day system development involves artifacts written in various languages, not limited to programming languages. This ability to apply frame technology to all these artifacts allows the modularization techniques of OOP and AOP to be applied to these artifacts as well, which was previously impossible. In this paper, the XML based *Frame Processing Language (FPL)* implemented by the author is used to illustrate how frames modularize common code (frames do OOP) as well as crosscutting concerns (frames do AOP) and do so on arbitrary artifacts, not just code.

2. 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*.

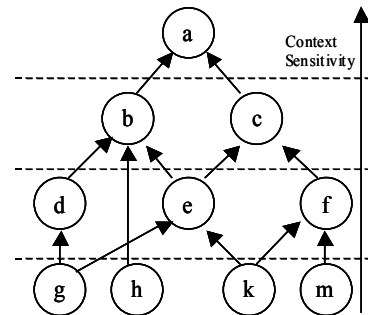


Figure 1 Example frame hierarchy

2.1 The adapt relationship on frames

The properties of the adapt relationship can be expressed as follows (from [1], let \geq 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 \geq imposes a partial ordering on the frames. This means that frames indeed form a lattice as shown in Figure 1. 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*.

2.2 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 [3] and XVCL (XML based Variant Configuration Language [9]). Here is a basic example of a frame in FPL¹:

```
<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 and exhibit the OOP and AOP behaviors this paper claims are being unified by FOP.

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.

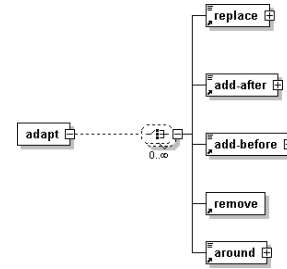


Figure 2 the `<adapt>` command syntax

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

¹ The required FPL namespace declaration <http://www.trcinc.com/2002/FPL/10> has been omitted.

`<break>`. 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.

Table 1 Precedence rules for adapt operations

D \ A	remove	replace	around	add-before	add-after
remove	ancestor	ancestor	both	both	both
replace	ancestor	ancestor	both	both	both
around	ancestor	ancestor	<i>ancestor</i>	both	both
add-before	ancestor	ancestor	both	both	both
add-after	ancestor	ancestor	both	both	both

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>` around existing FPL text 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 frame². Instead of changing the component itself, it becomes (more) changeable, but the change occurs where it is needed, namely in the new context (the new ancestor frame).

FPL has many more commands, mainly in the area of control structures like `<while>`, `<select>`, etc. but these are not needed in the context of this paper or to demonstrate the claim of unification of OOP and AOP.

2.3 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 `${}` syntax³ to access variables allows for variables to be used to construct the names of other variables, e.g. in the previous example `${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 Javabeen frame generating a Java class according to the Javabeen 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 frame in which it is being executed using the `samelevel="true"` attribute on the `<set>` command. This allows for information to be passed *up* the

² Since these ancestors are unaware of the new `<break>`, its original content will be used.

³ Adopted from the popular xml based build tool Ant

frame 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.

2.4 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 is used in order not to clash 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.

3. FOP DOES OOP

In Figure 3, the left diagram shows part of a standard frame hierarchy with two frames adapting a common, more general frame. When this diagram is turned upside down, as done in the right side of Figure 3, it looks suspiciously like two classes inheriting from a common superclass. This is in fact one of the simplest forms of frame adaptation. Since a frame (hierarchy⁴) is a generic component that can be specialized by setting variable values or adding content to `<break>` commands, the adapt relation between frames can be used to model generalization quite easily.

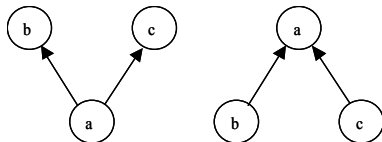


Figure 3 Frame adaptation is Generalization

⁴ To reiterate, each frame X in a frame hierarchy serves as a façade to the sub-hierarchy rooted at X and we can thus use the terms frame and frame-hierarchy interchangeably.

Since a frame can adapt multiple frames and replace or remove pieces of each, frame adaptation allows for a quite natural form of multiple inheritance as well, providing the mechanisms to resolve the conflicts of traditional OO multiple inheritance.

At this time it is important to recall that at the very beginning of this paper the arrows in the frame hierarchy were labeled as “part-of”. In Figure 3 however they are used as “is-a”. The localization of all context sensitive specializations in a single frame does in fact *unify* the “is-a” and “has-a” relationships that are so distinctly different in - and cause so many modeling headaches in - object oriented programming.

Note that another important property of Object Orientation, namely Encapsulation, is completely ignored by Frame Oriented Programming. The reason for this is that encapsulation is a very desirable feature of an object oriented system at *runtime*, but serves no purpose at *construction time* of a system, and Frame Oriented Programming is a metalevel activity, taking place at construction time only.

Bassett shows in [2] that Frame Technology is in fact an advanced form of Object orientation and solves many of the weaknesses of OO. A full treatment of all these benefits is beyond the scope of this paper and not needed to show that frames can be used to apply Object Orientation to artifacts like plain text, HTML or XML, for which this was previously hard or impossible.

3.1 FOP localizes polymorphic variants

Although the focus of this paper is not to describe the benefits of FOP over OOP, one important benefit merits mentioning in the context of this paper. In OOP, polymorphic variations are by definition fragmented into different subclasses. In FOP however, these can be encapsulated in a single frame using the `<select>` command.

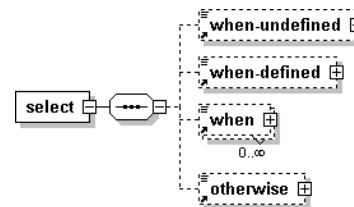


Figure 4 structure of the `<select>` command

`<select>` is an alternative to `<break>` when the inserts (`<replace>`s) are themselves reusable -- rather than several ancestors inserting essentially the same frame text (or one of several common patterns), you write them as `<select>` alternatives and simply set a switch.

Figure 4 shows the structure of the `<select>` command and the following example illustrates its use.

```
<select var="v">
  <when-undefined>
  variant 1; variable v is not defined
</when-undefined>
  <when-defined>
  variant 2; variable v is defined
```

```

</when-defined>
<when value="1" comparator="greater-than">
variant 3: The value of v is greater than 1
</when>
<when value="foo">
variant 4: The value of v is "foo"
</when>
</select>

```

As opposed to OOP where the granularity of polymorphic variation is restricted to methods⁵, the granularity of the variants in FOP is completely arbitrary. Ancestor frames select a variant simply by setting a variable. An interesting use of `<select>` is in the evolution of software over time. By carrying the old version along side the new ones in a single frame, in a `<select>` on a version (or date!) sensitive variable, a frame gains the ability to morph into quite different forms, a form of polymorphism that generates a tremendous amount of redundancy in OOP.

3.2 FOP does Templates

In addition to supporting the standard Object oriented concepts, frames can be used to describe generic types – or templates – as well. Here is an example frame that defines a generic subclass of the standard Java ArrayList with a typed interface. Setting the `PACKAGE`, `CLASSNAME` and `ELEMENTCLASS` (abbreviated to `EC` in example) variables in an adapting ancestor frame customizes the frame:

```

<frame name="ArrayList" language="java">
  <break name="ArrayList-Parameters"/>
  <requires vars="CLASSNAME, EC"/>
  <ifdef var="PACKAGE">
package ${PACKAGE};
  </ifdef>
import java.util.ArrayList;
import java.util.Iterator;
<break name="${CLASSNAME}-IMPORTS"/>
public class ${CLASSNAME} extends ArrayList {
<break name="${CLASSNAME}-BODY">
  public ${EC} get${EC}(int index) {
    return (${EC})super.get(index);
  }
  public boolean add${EC}(${EC} object) {
    return super.add(object);
  }
  <!-- Many more of the same omitted -->
<break name="${CLASSNAME}-METHODS"/>
</break>
</frame>

```

⁵ You either override the entire method, refactor, or maintain redundant modified versions of the method in each subclass. Unfortunately, the smaller the difference is, the larger the redundancy will be.

Note that this frame allows for additional import statements (for example for the element class) as well as the addition of extra methods. The `<break>` commands for imports and methods are named with the value of the `CLASSNAME` variable as part of their name so that they can be targeted specifically in an environment where an ancestor frames adapts this frame multiple times for different element types. Ancestor frames can use regular expressions with wildcards to target multiple `<break>` commands in multiple frames with a single modification. Here is one possible ancestor frame that generates a CustomerList and an AccountList class with Customer and Account elements and gives the AccountList an extra method to return the total balance in all the accounts.

```

<frame name="Lists" language="java">
<adapt frame="ArrayList.fpl" outfile="CustomerList.java">
  <replace break="ArrayList-Parameters">
    <set var="CLASSNAME" value="CustomerList"/>
    <set var="ELEMENTCLASS" value="Customer"/>
  </replace>
</adapt>
<adapt frame="ArrayList.fpl" outfile="AccountList.java">
  <replace break="ArrayList-Parameters">
    <set var="CLASSNAME" value="AccountList"/>
    <set var="ELEMENTCLASS" value="Account"/>
  </replace>
  <replace break="ArrayList-Methods">
    public double getTotalBalance(){
      // omitted implementation
    }
  </replace>
</adapt>
</frame>

```

4. FOP DOES AOP

In the previous section we have seen how FOP can fulfill the OO promise of modularization of commonalities. Aspect Oriented Programming on the other hand is concerned with the modularization of crosscutting concerns [5]. In this section I will use the terminology of AspectJ [6] to compare FOP to AOP and demonstrate that FOP can in fact also modularize crosscutting concerns, and do so on arbitrary textual artifacts.

Doug Orleans writes in [7]: “*Languages that support incremental programming, that is, the construction of new program components by specifying how they differ from existing components, allow for clean separation of concerns.*”

The previous sections on FPL have demonstrated that this is exactly what frame adaptation does; a frame adapts other frames and describes the differences with the adapt operations or variable overrides.

The analogy with AspectJ can be understood by realizing that the AspectJ *joinpoints* are analogous to the named `<break>` commands in FPL, and any other frame that adapts a frame containing a `<break>` can contain multiple advices that are

applied to the `<break>` in the form of `<add-before>`, `<add-after>`, `<remove>`, `<replace>` or `<around>`.

By nesting `<adapt>` commands in any of these operations, entire separately modularized concerns can be injected into multiple artifacts automatically. Recall the “ArrayList” template from the previous section, as well as its ancestor frame named “Lists”. Some time after the development of these frames we discover that it would be nice if all our lists would have a static inner class with a typed sub-interface of the standard *Iterator* interface from Java. This can be accomplished easily by wrapping the “Lists” frame with a new frame with this content:

```
<frame name="ListsWithIterators" language="java">
<adapt frame="Lists">
  <add-before break="*.BODY">
    <adapt frame="TypedIterator.fpl"/>
  </add-before>
</adapt>
</frame>
```

Note the use of the ‘*’ wildcard in the `<add-before>` command. This will insert the result of the adaptation of `TypedIterator.fpl` just before the named breaks matching the wildcard in both `CustomerList` and `AccountList` created by the `Lists` frame. This example inserts a modularized crosscutting concern – the static inner class for a typed iterator – in possibly many other classes. Similarly, with the proper placement and naming of `<break>` commands more fine-grained joinpoints can be advised just as easily. Here is an example adapted from a real world application of FOP.

```
<adapt frame="for-every-lineitem.fpl">
  <around break="lineitem-loop">
    // header goes here
    <proceed/>
    // footer goes here
  </around>
  <replace break="lineitem">
    // line items go here
  </replace>
</adapt>
```

The above frame adapts a frame named *for-every-lineitem.fpl* that contains logic to determine what the line items are, loops over every line item, and contains a `<break>` to allow ancestor frames determine what to do with each line item. It also contains a `<select>` that checks to see if there are any line items at all, and if there are none, the frame does nothing. When a frame like that is used within a header and footer environment, the possibility exists that we end up with a header and footer even if there are no line items. To avoid that scenario, the loop embedded within the `<select>` is enclosed in a `<break>` named *lineitem-loop*, so that an ancestor frame can conditionally emit the header and footer only if the `<select>` in a lower level frame decides that they are

needed. The loop itself contains a `<break>` named *lineitem*, which will trigger the ancestor’s `<replace>` for each line item. The line items will be inserted in the location of the `<proceed>` since the `<around>` is adapting the `<break>` containing the loop. The example below shows the *for-every-lineitem.fpl* frame.

```
<frame name="for-every-lineitem.fpl">
<select var="lineitems">
  <when-defined>
    <set var="lines" list="{lineitems}"/>
    <break name="lineitem-loop">
      <while listvars="lines">
        ... other frame commands to obtain line item data
      <break name="lineitem"/>
    </while>
  </break>
</when-defined>
</select>
</frame>
```

The `<select>` command serves as a way to implement cross-cutting concerns when each cross-cut instance may require unique code. For example, suppose the target language exists in several dialects. By setting a variable in the toplevel specification frame, the many dialectic variants located throughout a large body of text in multiple frames can be stored in `<select>`s. At each location the particular variant needed may be unique, but all locations will select the right variant, thus assembling the correct dialect.

There are a few considerable differences between FOP and AOP. For example, the granularity of the breaks in FPL (its joinpoints) is completely arbitrary, whereas AspectJ has a fixed set of joinpoints. A `<break>` in FPL can contain anything from nothing at all to an entire frame. On the other hand, FPL can only do construction time (static) AOP, not runtime AOP. It is the author’s opinion however, that it is at construction time when most is to be gained from a clean separation of concerns since the modularization of cross-cutting concerns is for the programmer’s benefit, not that of the runtime system.

5. FOP IN PRACTICE

FPL was initially developed to turn a diverse family of applications for the insurance industry into a software product line ([4]) customizable to the insurance product each applications 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, presentation logic, configuration files and database schemas.

FPL supports the use of arbitrary metadata and transformations with a `<transform>` command.

Using metadata to drive code generation has many interesting applications; it has been used to generate Java code from W3C XML schemas in order to generate custom Java/XML bindings.

Similarly, it is currently being used to generate Java code and W3C XML schemas from UML models represented in XML metadata [10]. Details on this approach can be found in [8].

The FPL language modifies the original Frame Technology as described in [1], [2] and [3] and XVCL [9] in some fundamental ways. The differences are these:

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

Most of these modifications are not necessary to enable the unification of OOP and AOP; the original Frame Technology from [1] and [2] inherently has this capability. The `<around>` command and the use of regular expressions to match break names do however greatly enhance the AOP capabilities of FPL with respect to the other implementations.

6. CONCLUSIONS

Frame Oriented Programming and specifically the mechanism of frame adaptation using breakpoints and the `<adapt>` operations to insert, replace or remove content is a superset of both Object Oriented Programming and (at least static) Aspect Oriented Programming and allows the modularization concepts of both to be applied to arbitrary textual artifacts.

7. 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.

8. REFERENCES

- [1] Paul G. Bassett, "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] Paul G. Bassett, *The theory and Practice of Adaptive Components*, keynote at the 2nd International Symposium on Generative and Component-Based Software Engineering, GCSE 2000 in Erfurt, Germany, October 9-12, 2000, Springer Lecture Notes in Computer Science, LCNS2177, (Eds. G. Butler and Stan Jarzabek), pp. 1-14
- [4] Paul Clements, Linda M. Northrop. *Software Product Lines: Practices and Patterns*, SEI Series in Software Engineering, Addison-Wesley, 2001
- [5] 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
- [6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. *An Overview of AspectJ*. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327-353, Springer-Verlag, 2001
- [7] Doug Orleans, *Incremental Programming with Extensible Decisions*, in *AOSD 2002, 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, 2002 Conference Proceedings, pages 56-64. ACM Press, editor Gregor Kiczales.
- [8] Frank Sauer "Metadata driven multi-artifact code generation using Frame Oriented Programming" in OOPSLA 2002 Workshop "Generative Techniques in the Context of Model Driven Architecture", November 5, 2002, Seattle, WA.
- [9] 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
- [10] <http://www.omg.org/technology/documents/formal/xmi.htm>