

# Framed Aspects: Supporting Variability and Configurability for AOP

Neil Loughran and Awais Rashid

Computing Department, Lancaster University, Lancaster LA1 4YR, UK  
{loughran | awais}@comp.lancs.ac.uk

**Abstract.** Aspect oriented programming (AOP) seeks to decompose concerns which crosscut system structure into more manageable modules. However, current AOP techniques alone lack the configuration mechanisms and generalisation capabilities that are required to realise variability (through clear reuse specifications). Conversely, frame technology provides extensive mechanisms for providing reuse and configuration yet cannot effectively modularise crosscutting concerns. This paper proposes *'framed aspects'* a technique and methodology which combines the respective strengths of AOP, frame technology and Feature-Oriented Domain Analysis (FODA). We argue that framed aspects can alleviate many of the problems the technologies have when used in isolation and also provide a framework for implementing fine-grained variability. The approach is demonstrated with the design and implementation of a generic caching component.

## 1. Introduction

The use of AOP in the production of software is now gaining major backing in the software industry [1]. AOP allows for modularisation of concerns that normally cause crosscutting in object oriented (OO) systems. However, there are no mechanisms available in current AOP languages to support and realise fine-grained configurability and variability, thus, the potential for aspects to be reused in different contexts is limited. This paper demonstrates *framed aspects*, an AOP independent meta language which adds the power of parameterisation, construction time constraint checking and conditional compilation to AOP languages. Parameterisation support for AOP allows an aspect to be customised for a particular scenario, and therefore increases the reusability of an aspect module. Conditional compilation allows for optional and alternative variant features of an aspect module to be included or excluded, thus resulting in optimal usage of code. Finally, constraints define rules for limiting the acceptable values and combinations of features in which an aspect module can be created.

AOP approaches such as AspectJ [2] do not allow the specification for a concern to be written as a separate entity from the aspect itself, thus the developer must have an intricate understanding of the aspect code and thus cannot treat the aspect in a black box manner. The work presented in this paper demonstrates how framed aspects address this problem and improve reusability and evolvability for AOP languages. We focus on one particular AOP technique (Aspect J) but the concepts and meta language are generic and therefore applicable to other AOP approaches. The next section discusses the concept of frame technology and how it can be beneficial to AOP. Section three introduces the framed aspect approach. Section four demonstrates the framed aspect approach, used in conjunction with AspectJ, to create a reusable generic caching component. Section five discusses related work and finally section six concludes the paper.

## 2. Frames and AOP

### 2.1 Frames

Frame technology [3] was conceived in the late 1970s, and the technology has since evolved into XML forms [4]. Frame technology is language independent and allows useful code to be generalised, adapted and thus configured to different requirements and specifications by using code templates organised into a hierarchy of modules known as 'frames'. A developer can then write a specification and by using a frame processor create customised systems, components and libraries. An individual frame (or group of frames) is the separation of a concern, class, method or related attributes. Variability is achieved by allowing variation points, code repetitions, options, slots for new functionality etc., to be marked *invasively*, using meta tags embedded within the program code. Typical commands in frames are <set> (sets a variable), <break> (create a slot for new functionality to be added at a later date) <select> (selects an option), <adapt> (refines a frame with new code) and <while> (creates a loop around code which repeats).

### 2.2 AOP

AOP allows concerns which would traditionally crosscut system structure in OO systems to be encapsulated into single entities known as *aspects*. AOP languages, such as AspectJ, allow existing modules to be refined statically, in a *non-invasive* manner, using *introductions* (add new methods, fields, interfaces and superclasses) or through injection of additional behaviour in the control flow at runtime via *advice*. Additionally, *joinpoints* (points of interest to which we add new behaviour) can be defined using *pointcuts*. Fig. 1 demonstrates a Document caching aspect which uses a defined pointcut and advice.

```
aspect SimpleCacheAspect
{
    private int MAX_CACHE_SIZE = 100;
    private int PERC_TO_DEL = 50;
    private Hashtable cache = new Hashtable();

    pointcut pc1(Editor g, String url) : args (g,url) &&
        call (public void Network.requestInfo(Editor, String));

    void around(Editor g, String url): pc1(g,url)
    {
        PageContent cachedPage=(PageContent) cache.get(url);
        if(cachedPage==null)
        {
            proceed(g,url);
            PageContent page=new PageContent(g.getDocument());
            addToCache(url,page);
        }
        else
            g.setDocument(cachedPage.getContent());
    }

    class PageContent {
        private Document data;
        private int accesses=0;
        public PageContent(Document d) { data = d; }
        public Document getData() { accesses++; return data; }
        public int getAccesses() { return accesses; }
    }
}
```

Creation of **pc1 pointcut** on the method **requestInfo** in **Network** class.

**Around advice** which executes whenever the method defined in **pc1** is called. If record doesn't exist in cache, **proceed** with original call then store the result in the cache. Otherwise populate editor with cached content.

Data structure for storing **Document** content of editor pane. The access scheme is incremented every time the document is accessed from the cache.

Fig. 1. A simple editor pane caching aspect in AspectJ

However, varying the aspect for different contexts is difficult due to the lack of parameterisation and configuration support. AspectJ does not allow the separation between specification and the aspect code. The aspect is, therefore, a white box component and an understanding of what the aspect does at code level is required.

## 2.3 Comparing Frames and AOP

Frames and AOP share commonalities, such as the ability to refine modules or add code to defined points of interest. However, the mechanisms by which they achieve this is different in both technologies. The explicit invasive approach employed by frames, while being very flexible as customisations can be added anywhere in the code (compared to the restricted join point model in AOP), can lead to poorly modularised, heavily tagged and hard to maintain code. The strengths and weaknesses of both technologies are summarised in table 1. The strengths of one technique are the weaknesses of the other and vice versa. A hybrid of the two approaches can provide the combined benefits thus increasing configurability, modularity, reusability, evolvability and longevity of the aspects.

**Table 1.** Comparing frames and AOP

Capability	Framing with OO	AOP
Configuration Mechanism	Very comprehensive configuration possible	Not supported natively, dependent on IDE
Separation of Concern	Only non crosscutting concerns supported	Addresses problems of crosscutting concerns
Templates	Allows code to be generalised to aid reuse in different contexts	Not supported
Code Generation	Construction time mechanism allows generation of code and refactoring via parameterisation.	Generates code which (in the case of advice) is bound at run time
Language Independence	Supports any textual document and therefore any language	Constrained to implementation language, although language independent AOP forms exist
Use on Legacy Systems	Limited	Supports evolution of legacy systems at source and byte code level
Variation Point Identification	Invasive breakpoints	Non invasive joinpoints
Dynamic Runtime Evolution	Not supported	Possible in JAC and JMangler. Future versions of AspectJ will have support.

## 3. Framed Aspects

### 3.1 Rationale for Framed Aspects

The framed aspect approach uses AOP to modularise crosscutting and tangled concerns and utilises framing to allow those aspects to be parameterised and configured to different requirements. Many commands in frames (such as `<break>`, which has before and after forms similar to AOP advice) are in our opinion better implemented in AOP languages and in a much cleaner non-invasive way, thus we developed the Lancaster Frame Processor (LFP) meta language. LFP is essentially a cut down version of the XVCL [4] frame processor (albeit with some added commands for constraints and a simpler syntax) and utilises only a subset of the commands used in traditional framing tools forcing the programmer to use AOP for the remainder. This balance of AOP and frames reduces the meta code induced by frames (due to their invasive nature) and at the same time provides effective parameterisation and reconfiguration support for aspects. We are of the opinion that aspects can also have concerns within themselves, especially as aspect modules become larger as new variants and features are added. Breaking up the aspect into smaller modules helps to localise these inner concerns in a manner in which inheritance cannot and also provide a framework for development. Moreover, allowing the aspect to be broken into sub-components allows pointcuts, advice, introductions and members to be modelled independently from one another as opposed to being tightly coupled<sup>1</sup>.

<sup>1</sup> Similar discussions have taken place on the AspectJ users list. cf. G. Kiczales and C. Beust 10<sup>th</sup> July 2003

### 3.2 Framed Aspect Composition

A framed aspect composition is made up of three distinct modules:

- **Framed Aspect Code.** This module consists of the normal aspect code and parameterised aspect code.
- **Composition Rules:** This module maps out possible legal aspect feature compositions, combinations, constraints and controls how these are bound together.
- **Specification:** Contains the developer's customisation specifications. The developer will usually take an incomplete template specification and fill in the options and variables s/he wishes to set.

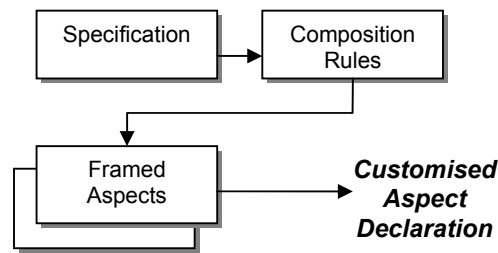


Fig. 2. Framed aspect composition

Fig. 2 demonstrates how the specification, compositions rules and framed aspects (parameterised aspect code) are bound together in order to generate the source code.

### 3.3 Delineating Frames

Creating the framed aspects requires careful consideration of the variants and scope for which the aspect is intended. The first step after discovering these variants is to create a feature diagram using FODA [5], which describes the dependencies, options and alternative characteristics of the feature aspect. The feature approach provides a natural design method for use with framed aspects and aspect frames can be deduced by simply delineating the boundaries between the different options and alternatives in the model. Figs. 3 (a) – (e) demonstrate how the boundaries are delineated for features X, Y and (in the case of alternative features) Z. This gives the programmer a starting point for developing the frames. However, as development progresses, there might arise a need for new frames to capture code that is duplicated across multiple modules.

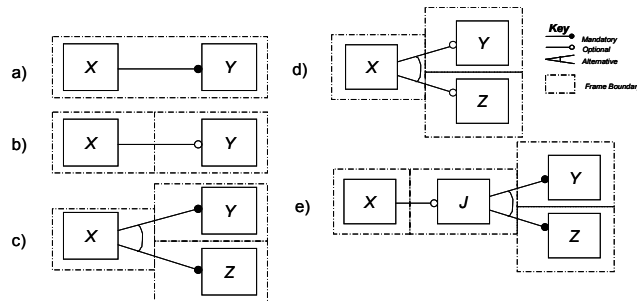


Fig. 3. Delineating frame boundaries of a) mandatory, b) optional and c) alternative features, and frame refactoring showing d) original and e) transformation.

Fig. 3(d) and fig. 3(e) demonstrates how alternative variants (Y and Z) could contain duplicate code (for instance an algorithm), and thus would benefit from an extra layer (frame J) which contains the common code or by simply moving the duplicated code to the parent frame. Moreover, frames can break down large aspect modules into smaller and, therefore, more manageable modules and hide away less important information from the main concern. A frame can enhance reusability by allowing a component (for example, an algorithm) to be framed separately from the main codebase and thus reused in other contexts. Frame commands are utilised for finer grained variability, parameterisation and constraints, while AOP is used for integrating the concern in a non-invasive manner. AOP is also used where a coarser grained functionality is required or when a particular concern crosscuts multiple modules. The process is described in much greater detail in section 4 where we implement a generic caching component.

### 3.4 Framing Aspects

#### 3.4.1 Utilising Parameterisation

Parameterisation in frames can be used with any textual representation such as a type or object, a method, joinpoint or pointcut designator. We consider this form of parameterisation to be very powerful and while languages such as Java 1.5 promise to bring generics as standard, the frame approach can be much more flexible as *any* programming construct can be parameterised. Parameterisation can also be applied to aspects to allow scope of aspect behaviour to be customised, to change the method or methods a pointcut is bound to, or introduce new members into a specified class.

```
private int MAX_CACHE_SIZE = <@MAX_CACHE_SIZE>;
private int PERC_TO_DEL = <@PERC_TO_DEL>;
pointcut pcl(<@EDITOR_NAME> g,String url):args(g,url) &&
call (public void <@NETWORK_CLASS>.<@REQUEST_MTHD>
(<@EDITOR_NAME>, String));
void around(<@EDITOR_NAME> g, String url): pcl(g,url)
{ // impl }
class PageContent
{
private <@DOC_TYPE> data;
private int accesses=0;
public PageContent(<@DOC_TYPE> d) { data =d; }
public <@DOC_TYPE> getData() { accesses++; return data }
public int getAccesses() { return accesses; }
}
```

Fig. 4. Parameterised version of simple caching aspect

Returning to our AspectJ implementation of a web cache (fig. 1) in section 2.3 we could apply parameterisation in numerous ways to enhance its reuse as shown in fig. 4, so that it can be used to store data other than `Document`, or be used with classes other than `Editor` or `Network` for example.

#### 3.4.2 Utilising Options, Adapts and Constraints

*Options* are used for conditional compilation of optional and alternative features. If an option is indicated in the specification frame then the code delineated by the option tags in the composition rules or the aspect code is included. *Adapts* provide the framework necessary for controlling the development of framed aspects and binding of the frames. Options and adapts are typically used in the composition rules module rather than in the aspect code itself. Although this is not a hard and fast rule, we believe that adding them directly to aspect code is a sign that parts of the aspect need to be refactored to another

frame. A new feature added by LFP is the ability to add *constraints* to the aspects. The form of these constraints might be lower and upper boundary limits or the requirement that the operation/value required is contained in a predefined set.

### 3.4.3 Refactoring Framed Aspects

Refactoring of aspect code is in order for the required joinpoints to be exposed. An example might be to refactor advice as a single or multiple method members, depending upon the scenario. Refactoring code in this way can improve aspect understandability.

## 4. Implementing a Generic Cache Using Framed Aspects

In this section we describe our experiences of implementing a server side generic caching component using framed aspects.

### 4.1 Cache Description

The cache is designed to be used in either database or web environments and can be configured easily to different requirements and situations.

#### 4.1.1 Database Cache

The database cache caches results of SQL queries sent from clients. The cache can be configured for situations where only read operations (SELECT queries) are in operation but optionally can also be used where write operations (UPDATE, INSERT, etc.) are used. When updates are received, there is the need for information in the cache to be refreshed, thus, there are two separate update strategies, namely:

- **Every Write:** Every write operation to the database triggers the cache update mechanism.
- **Time Based:** The cache is refreshed at time intervals.

#### 4.1.2 Web Cache

The web cache is responsible for storing web pages and has options for allowing the cache to be refreshed. The mechanisms that do this will be different from that of the database cache and so are implemented separately, they are:

- **Automatic:** Whenever a url from a client is received, the cache automatically sends a small query to check if the page the url points to is newer than the one in the cache. If so, the web page currently in the cache is replaced by the newer one.
- **Manual:** As above, but instead the client explicitly has to ask for the page in the cache to be refreshed.

#### 4.1.3 Deletion Scheme

Eventually the cache will become full and there has to be some mechanism in place to remove records currently held in the cache in order to free up space. The mechanism will be set to a particular strategy:

- **Access:** Delete least accessed records
- **Date:** Delete oldest records
- **Size:** Delete largest records.

## 4.2 The Feature Model

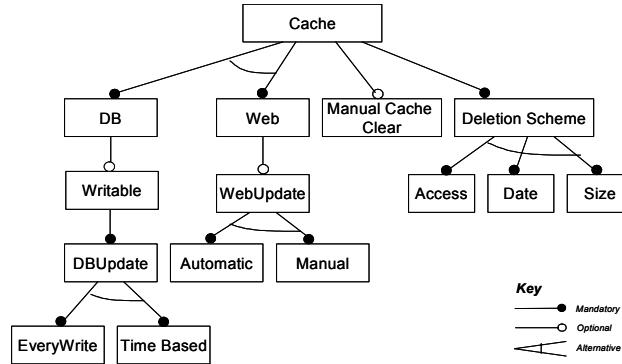


Fig. 5. Feature model for generic cache

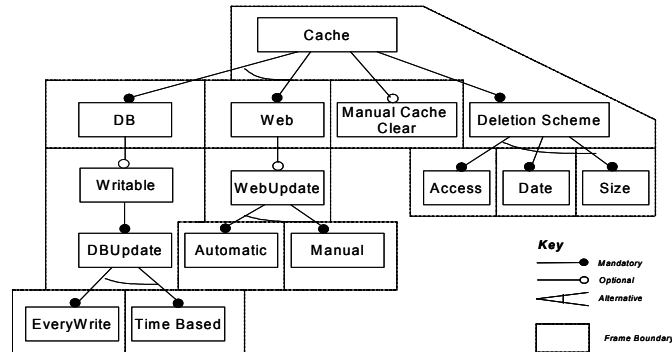


Fig. 6. Delineating frames in the generic cache

The generic cache is shown in the feature model depicted in fig. 5, which describes the possible compositions. From the feature diagram we can then delineate the framed aspect modules, shown in fig. 6, using the rules as discussed in section 3.3.

## 4.3 Framed Aspect Code Examples

In this section we illustrate examples of the framed aspect code contained in our generic cache. The examples contained herein focus mainly on the database caching variant, although the principles are entirely applicable to other features.

### 4.3.1 Cache Frame

The cache frame (fig. 7) contains code that is common to all variant forms of the generic cache. To enhance reusability and flexibility we used the framed aspect approach to parameterise the cache frame and lower order frames with the following variants:

- MAX\_CACHE\_SIZE: Sets the maximum size of records the cache will hold.
- PERC\_TO\_DEL: The amount of records to delete when the deletion mechanism is invoked.

- **CONN\_CLASS**: The class which contains the methods for sending the query to the database and also sending the results back to the client.
- **SEND\_QUERY**: The method which sends the query to the database.
- **REPLY\_CLIENT**: The method which sends the result back to the client.
- **DOC\_TYPE**: The type of information that is being stored in the cache (e.g. String, Document, CachedResultSet etc.).

```

<frame name = "CACHE">

private int MAX_CACHE_SIZE = <@MAX_CACHE_SIZE>;
private int PERCENTAGE_TO_DEL = <@PERC_TO_DEL>;
private Hashtable cache = new Hashtable(MAX_CACHE_SIZE); } ①

public void addToCache(String key, <@DOC_TYPE> data)
{ //..impl.. }

pointcut QUERY_PCT(String key, <@CONN_CLASS> c) : this(c) && args
(key) && call(public void<@CONN_CLASS>.<@SEND_QUERY>(String)); ②

pointcut RESULT_PCT(<@DOC_TYPE> data, <@CONN_CLASS> c):
!within(Cache) && this(c) && args(data) &&
call (public void @CONN_CLASS"/>.<@REPLY_CLIENT>(<@DOC_TYPE>)); ③

class CacheDS
{
private <@DOC_TYPE> data;
public CacheDS(<@DOC_TYPE> d)
{
data = d;
}
public <@DOC_TYPE> getData()
{
return data;
}
} } ④

```

**Fig. 7.** Cache frame

*Code Description*

1. Sets the size of the cache and percentage to be deleted as set by the parameters in the specification.
2. Creates a pointcut for intercepting the call to the method which executes SQL queries on the database.
3. Creates a pointcut for intercepting the results sent back to the client.
4. CacheDS is a data structure for storing the cache results.

**4.3.3 Writable Frame**

The writable frame (fig. 8) is an optional variant, which is used when the cache is to be used in situations where insertions and updates are made to the database. The cache will thus require to be updated in some manner, although this functionality will be implemented fully in the lower frames, this frame contains functionality common to all variants below it in the hierarchy.

*Code Description*

1. Pointcut used to trap new instances of CacheDS (data structure for holding the result data to be cached).
2. Pointcut to capture ResultSet from currently executing query.
3. Advice which adds tables contained within the executing query by a particular client to the CacheDS data structure
4. Advice which captures the ResultSet to obtain the ResultSetMetaData and, therefore, the tables used in the resulting query.

5. Introductions into the CacheDS data structure which adds new fields and methods.
6. Introductions into the current CONN\_CLASS to store tables for the current executing query.

This frame demonstrates the strength of the framed aspect approach over frame technology and AOP alone, by showing how parameterisation and crosscutting refinements can be encapsulated within a single frame.

```

<frame name = "WRITABLE">

pointcut DS_INSTANCE_PCT(<@CONN_CLASS> c) : cflow (this(c)) &&
call(CacheDS.new(..) );

pointcut RESULTSET_PCT(<@CONN_CLASS> c) : this(c) &&
call (ResultSet Statement.executeQuery(String));

after(<@CONN_CLASS> c) returning(CacheDS cds): DS_INSTANCE_PCT(c)
{
  cds.setTables(c.getTables());
}

after(<@CONN_CLASS> c) returning(ResultSet rs): RESULTSET_PCT(c)
{
  try
  {
    ResultSetMetaData rsmd = rs.getMetaData();
    c.setTables(getTablesFromMetaData(rsmd));
  }
  catch(SQLException sqle) {}
}

private boolean CacheDS.isValid = true;
private Vector CacheDS.tables;
public void CacheDS.setTables(Vector v)
{
  tables = v;
}
public void CacheDS.containsTable(String s)
{
  if(tables.contains(s)) isValid = false;
}
public boolean CacheDS.isValid()
{
  return isValid;
}
public Vector CacheDS.getTables()
{
  return tables;
}

private Vector <@CONN_CLASS>.tables = new Vector();
public void <@CONN_CLASS>.setTables(Vector v)
{
  tables=v;
}
public Vector <@CONN_CLASS>.getTables()
{
  return tables;
}
// methods for getting table names from an SQL query and metadata

```

**Fig. 8.** Writable frame

#### 4.3.4 Specification Frame

The developer can write a specification, separate from the aspects, that will adapt the framed aspects with the required functionality. Fig. 9 demonstrates a typical specification for a database cache.

```

<frame name = "CACHE SPEC">
  <select option = "CACHE_TYPE" value = "DATABASE_CACHE" />
  <set var = "MAX_CACHE_SIZE" value = "1000" />
  <select option = "DELETION_SCHEME" value = "ACCESS" />
  <set var = "PERC_TO_DEL" value = "50" />
  <set var = "CONN_CLASS" value = "DBConnection" />
  <set var = "SEND_QUERY" value = "sendQuery" />
  <set var = "REPLY_CLIENT" value = "replyToClient" />
  <set var = "DOC_TYPE" value = "String" />
  <select option = "WRITABLE" value = "TRUE" />
  <select option = "DB_UPDATE_SCHEME" value = "EVERYWRITE" />
  <adapt frame = "CACHE_RULES"/>
</frame>

```

**Fig. 9.** Typical specification frame for a database cache

*Description*

1. The database cache option is selected for CACHE\_TYPE, 1000 query resultsets can be stored by setting MAX\_CACHE\_SIZE, DELETION\_SCHEME is set to the least accessed option, and PERC\_TO\_DEL is set to 50%.
2. CONN\_CLASS targets a class called DBConnection, the methods for sending queries (sendQuery) to the database and sending the query results back to the client (replyToClient) are bound to SEND\_QUERY and REPLY\_CLIENT respectively, while the type of data to be stored in the cache, DOC\_TYPE, is bound to String.
3. The WRITABLE option is selected and the EVERYWRITE update scheme is chosen.
4. Finally the specification is processed by the composition rules defined for the cache component to bind the components together.

### 4.3.5 Composition Rule Frame

The composition rules shown in fig. 10 bind the framed aspects together and also define constraints as to what can be bound.

*Description*

The rules shown here consist of:

1. Constraining meta variables to sets or ranges of possible values.
2. Adapting mandatory features as defined by the specification.
3. Adapting optional features if selected.
4. Adaptation rules for the database cache.

```

<frame name = "CACHE_RULES">
  <constrain var = "CACHE_TYPE" toSet = "DATABASE_CACHE,WEB_CACHE"/>
  <constrain var = "PERC_TO_DEL" toBoundary = "25,100"/>
  <constrain var = "DELETION_SCHEME" toSet = "ACCESS,DATE,SIZE"/>
  <constrain var = "DB_UPDATE_SCHEME" toSet = "EVERYWRITE,TIME_BASED"/>
  <constrain var = "WEB_UPDATE_SCHEME" toSet = "AUTOMATIC,MANUAL"/>
  <adapt frame = "CACHE_TYPE"/>
  <adapt frame = "DELETION_SCHEME"/>
  <option name = "MANUAL_CACHE_CLEAR" value = "TRUE">
  <adapt frame = "MANUAL_CACHE_CLEAR" />
  </option>
  <option name = "CACHE_TYPE" value = "DATABASE_CACHE">
  <option name = "WRITABLE" value = "TRUE">
  <adapt frame = "DB_UPDATE_SCHEME"/>
  </option>
</option>
</frame>

```

**Fig. 10.** Composition rules for generic cache

The separation of the composition rules from the main aspect code allows different rules to be created and enhances the possibility for reuse of the framed aspects in different contexts.

## 5. Related Work

Template programming, as used in languages such as C++, is a means of creating generic classes (or functions) and allowing customisations to be made based on parameters provided by the programmer when they instantiate instances of the required class. However, templates are not supported within languages such as Java and C# and therefore the vast majority of AOP languages. Generics are a new addition to the Java language (and also C#) and, like templates, allow the programmer to instantiate customised classes. However, aspect languages using AspectJ like instantiation models cannot take advantage of generic support due to the fact that aspects, unlike classes, are not directly instantiated. Moreover, in our opinion, AOP really needs a different generic model to the aforementioned due to the fact that the programmer may want to apply parameterisation to, for instance, supply a method name as a parameter for use in a pointcut definition or advice. In this respect our approach offers the only current way to generalise aspects.

Our approach also shares many similarities with feature oriented programming (FOP, GenVoca et al) [6], where modules are created as a series of layered refinements, SALLY [7], which allows AspectJ style introductions to be parameterised and Aspectual Collaborations [8] where modular programming and AOP techniques are combined. In FOP layers are stacked upon one another, with each layer containing any number of classes. Upper layers add refinements (new functionality, methods, classes, etc.) to lower layers by parameterisation. There are commonalities between FOP, AOP and in particular our framed aspect approach. However, presently, only static crosscutting (introductions) is currently supported within the FOP model, in contrast to the power of dynamic crosscuts via advice in our framed aspect model. Our approach brings a FOP style variability to AOP hence, facilitating potential for AOP implementation of product lines. In contrast to SALLY, which allows introductions to be parameterised, framed aspects allow any AOP construct (pointcut, advice, introduction, members, etc.) or technique (Hyper/J [9], AspectJ, etc.), to be parameterised. Framed aspects are in some ways similar to Aspectual Collaborations as they help build aspects for black-box reuse and also support external composition and binding between aspects and base. Recent developments focus around AOP frameworks such as AspectWerkz [10], JBoss AOP [11] and Nanning Aspects [12]. AspectWerkz allows for parameterisation and uses XML for defining aspects, advice and introductions. JBoss AOP also uses XML for defining advice, introductions and pointcuts but lacks a comprehensive join point model. Nanning has been designed as a simple to use AOP framework but, like JBoss AOP, also lacks a rich joinpoint model. The key difference in our approach, compared with any of the models previously mentioned, is in the language independence of frames, which means that there are no constraints as to which AOP technique or programming language platform (e.g. Java, C# etc.) is used.

## 6. Conclusions

This paper has demonstrated how AOP can benefit from parameterisation, generation and generalisation that frame technology brings. We have demonstrated how frames can enhance reuse and ease the integration and creation of new features and believe the same technique can be applied to different concerns. Framed aspects improve upon traditional

framing methods by removing a great deal of the meta code that frames suffer from and allow crosscutting features to be modularised in a non-invasive manner. We can utilise the technique in the creation of reusable component libraries or domains which require high levels of reuse, such as product line engineering [13], which can benefit from the parameterisation and configurational power that framed aspects can bring. Framed aspects improve the integration of features that would normally crosscut multiple modules, thus causing severe problems with evolution, and resulting in architectural erosion [14]. The technique can also be utilised to allow configuration of reusable aspects that can be woven into existing systems where the original code may or may not be available, thus allowing frame techniques to be used in legacy systems to some degree.

In this paper we demonstrated how framed aspects could be used with production aspects (code included in the final product) in order to increase their reusability. However, we also have found that development aspects (e.g. tracing and testing aspects) can benefit from framing in order for them to be reused in other contexts or across different domains. Future work will involve improving the framing technique by adding semantic checks and more static type checking, utilising IDE support and also demonstrate how framed aspects can be used within existing frame based technologies such as XVCL. Java 1.5 will bring generics as standard and it will be interesting to view in more detail how this will contrast with the parameterisation support available in framed aspects. Due to language independence, the framed aspect technique can be used in combination with new AOP languages as they emerge and also with existing techniques as they evolve. Utilisation of frames and AOP allows features and concerns to be modularised and adapted to different reuse contexts thus improving comprehensibility and improving the possibilities for evolution.

## 7. Acknowledgements

This work is supported by a UK Engineering and Physical Sciences Research Council Studentship. The authors would like to thank Stan Jarzabek and Weishan Zhang at the National University of Singapore for previous discussions regarding framing languages.

## References

- [1] [http://www.theserverside.com/news/thread.tss?thread\\_id=24698](http://www.theserverside.com/news/thread.tss?thread_id=24698)
- [2] AspectJ Team, "AspectJ Project", <http://www.eclipse.org/aspectj/>, 2003.
- [3] Bassett, P.: Framing Software Reuse - Lessons from the Real World, Prentice Hall (1997).
- [4] XVCL homepage, <http://fxvcl.sourceforge.net>
- [5] Kang, K. et al, Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21, Nov. 1990.
- [6] Batory, D., Sarvela, J. N., Rauschmayer, A.: Scaling Step-Wise Refinement. ICSE 2003.
- [7] Hanenberg, S. and Unland, R.: Parametric Introductions. Proc. of AOSD 2003, pp. 80-89.
- [8] Lieberherr, K., Lorenz, D. H., Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal, 46(5) 2003.
- [9] IBM Research, Hyperspaces, <http://www.research.ibm.com/hyperspace/>
- [10] AspectWerkz homepage, <http://aspectwerkz.codehaus.org/>
- [11] JBoss AOP homepage, <http://www.jboss.org/developers/projects/jboss/aop.jsp>
- [12] Nanning Aspects homepage, <http://nanning.codehaus.org>
- [13] Loughran, N., Rashid, A., Zhang, W. and Jarzabek, S., Supporting Product Line Evolution with Framed Aspects, ACP4IS Workshop, AOSD 2004.
- [14] van Gurp J. and Bosch J., Design Erosion: Problems & Causes, Journal of Systems & Software, vol 61, issue 2, 2002.