Supporting Product Line Evolution with Framed Aspects

Neil Loughran¹, Awais Rashid¹, Weishan Zhang² and Stan Jarzabek²

¹Computing Department, Lancaster University, Lancaster LA1 4YR, UK
{loughran | awais} @comp.lancs.ac.uk
²Department of Computer Science, National University of Singapore, Singapore 117543
(zhangws | stan) @comp.nus.edu.sg

Abstract. This paper discusses how the evolution of software product lines can be supported using framed aspects: a combination of aspect-oriented programming and frame technology. Product line architectures and assets are subject to maintenance and evolution throughout their lifetime due to the emergence of new user requirements, new technologies, business rules and features. However, the evolution process can be compromised by inadequate mechanisms for expressing the required evolutions. It maybe possible to anticipate future evolutions and, therefore, prepare and design the architecture to accommodate this, but there will eventually come a time when a certain feature or scenario appears which could not have been foreseen in the early stages of development. We argue that frames and aspects when used in isolation cannot overcome these weaknesses effectively. However, they can be addressed by using the respective strengths of both technologies in combination. The amalgamation of framing and aspect-oriented techniques can help in the integration of new features and thus reduce the risk of architectural erosion.

1 Introduction
Software systems evolve over time as new requirements and functionality emerge. It has been estimated that up to 80% of lifetime expenditure on a system will be spent on the activities of maintenance and evolution. However, software product line (SPL) evolution is a much more complex problem than traditional single system evolution due to the differing configuration requirements and possibilities for different systems within the product family. Product lines, particularly those in volatile business domains such as banking, will constantly be subject to maintenance and evolution throughout their lifetime due to the emergence of new requirements, new technologies, business rules and features. Clearly, tools and paradigms which manage this complexity and facilitate modification of the architecture or ease the introduction of new features are needed if a product line architecture is to have any longevity. In this paper we discuss how a combination of two such techniques namely, frame technology [1] and aspect-oriented programming (AOP) [2], can be used to improve evolution of SPLs and their assets. We argue that both techniques offer complementary support for software product line evolution and, hence, improved support can be derived by using them in combination.

The next section provides some background on evolution needs in SPLs. Section 3 introduces frames and AOP and discusses their respective strengths and weaknesses in
supporting SPL evolution. Section 4 describes our approach: framed aspects and demonstrates its effectiveness in supporting evolution in comparison with the frame-based and AOP implementations discussed in section 3. Section 5 discusses some related work while section 6 concludes the paper.

The discussion in sections 3 and 4 is based on the development of an SPL for electronic city guide systems such as GUIDE [3]. Variation points in this SPL range from the customisation of GUI components and stylings to the capability for the system to run on devices with limited resources (such as PDAs and mobile phones). The feature used as the basis for the discussion in this paper is the implementation of a cache which stores previously visited pages. Variants in this instance are maximum size of cache, deletion strategy (i.e. delete least accessed records, oldest records, etc.), percentage of records to delete and the ability for systems to be configured to be cached or uncached.

2 Evolution Issues in Software Product Lines

Software evolution is difficult to predict and rarely uniform over time. During software development requirements can change by up to 30% [4]. Managing this volatility is difficult because the changes can have major impacts on the design of the architecture. Therefore, effective mechanisms are required which can handle requirement changes through all stages of SPL development as well as evolution of the architecture throughout its life. Traditional generative approaches parameterise components and leave hooks in the architecture for most likely evolutions. The problem with this approach is that complex changes not thought of cannot be effectively handled and often give rise to the need to reorganise existing modules. Some of these issues have been highlighted by [5] in the context of evolution of SPLs for middleware.

Traditional approaches also mainly focus on the classic categories of evolution [6] namely, corrective (fixing of bugs), adaptive (adding a new feature), perfective (improving performance) and preventative (preventing problems before they occur). While this categorisation is useful in showing the type of evolution to be performed, it does not demonstrate how the change affects the software architecture itself. In order to support this, it is more useful to think of crosscutting and non-crosscutting evolution. When a proposed evolution requires changes to more than one module it is said to be crosscutting, while non-crosscutting evolutions can be localised. The need to address crosscutting evolution is crucial in SPLs as a change can affect different variants and branches.

Note that an SPL can be subject to a variety of changes over its lifetime ranging from addition, retraction, restructuring and replacement of a feature to introduction of a new product or an entirely new product line (in instances when variability becomes too large). The example in this paper focuses on evolutions pertaining to a particular feature. Introduction of new products or product lines will form the subject of a future paper.

3 Frames vs AOP

3.1 Frame Technology

Frame technology was conceived during the 1970s as a means to providing a mechanism for creating generalised components that can be easily adapted or modified to different reuse contexts. Frame technology is essentially a language independent textual pre-processor that creates software modules by using code templates and a specification from the developer. Examples of typical commands in frames are <set> (sets a variable), <select> (selects an option), <adapt> (refines a module with new functionality) and <while> (creates a loop around repeating code).
To illustrate the use of frames, consider the object-oriented (OO) implementation of the cache feature for the guide SPL. Using OO alone we implemented the cache by creating a Cache instance in the Editor class and then wrapped calls to a requestInfo with a check to see if records existed in the cache before proceeding with the requestInfo method call (cf. fig. 1).

```java
class Editor extends JEditorPane implements HyperlinkListener
{
  private Network network;
  private Cache cache = new Cache();
  //.. constructor and editor initialisation

  public void hyperlinkUpdate(HyperlinkEvent e)
  {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
    {
      String url = e.getURL().toString();
      Document cachedPage = (Document)cache.getFromCache(url);

      if (cachedPage == null)
      {
        network.requestInfo(this, url);
        cache.addToCache(url, this.getDocument);
      }
      else
      {
        // get record from cache and display it
        this.setDocument((Document)cachedPage.getContent());
      }
    }
  }
}
```

Fig. 1. OO implementation of the Cache feature

The code shown in bold in fig. 1 is the code added by the integration of the cache. Using a frame processor such as XVCL [7] we can tag this code to ease its retraction from the codebase (cf. fig. 2).

```xml
<option cache>
  private Cache cache = new Cache();
</option>
//.. etc
```

Fig. 2. Using frames to tag caching code

While the framing solution helps to clearly identify the caching concern, it is not a particularly elegant solution to the problem as the class now becomes cluttered with tags which can make the code difficult to read, understand and therefore evolve. Another solution might involve making a copy of the hyperlinkEvent method and having separate frames for the two variants. While this would be a neater solution, it fragments the module and future requirements pertaining to the hyperlinkEvent method would require that the code is updated in both frames, therefore inducing unneeded duplication.

### 3.2 AOP

AOP mechanisms such as AspectJ [8] and Hyper/J [9] are now gaining considerable support as a means for managing the separation of concerns and features which can lead to unmanageable code tangled across multiple classes in traditional OO systems. Examples
of concerns in traditional OO systems that exhibit this fragmentation of context are logging, profiling and tracing. AOP languages such as AspectJ allow multiple modules to be refined statically using introductions or through injection of additional behaviour in the control flow at runtime via advices.

AOP can alleviate the problem of tangled caching code (or tags in case of frames). As shown in the AspectJ implementation of the cache in fig. 3, the aspect can simply be plugged into the Editor. The key part of the aspect is the around advice which encapsulates the following sequence of operations:

1. Whenever the requestInfo method within the Editor class is called, grab the argument URL.
2. Search the cache for the URL.
3. If the URL doesn’t exist, proceed with the call and add the content of the editor to the cache. If it does exist then simply update the editor pane with the content proceeding with the call to requestInfo.

Note that PageContent is a data structure used to store the editor content along with other data (i.e. number of accesses) in the cache.

```java
aspect CacheAspect
{
    private Hashtable cache = new Hashtable();
    // ..code
    void around(Editor g, String url): args (g,url) &&
        call (public void Network.requestInfo(Editor, String))
        { 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());
            }
        }
    // inner class for data structures
}
```

**Fig. 3.** AOP implementation of the cache using AspectJ

While the AOP implementation cleanly modularises the caching code, no parameterisation support is available. Consequently, the aspect needs to be modified to vary the caching behaviour. Alternatively, an abstract aspect needs to be provided with concrete aspects specifying the specific caching variants required by a particular product. However, this can lead to inheritance anomalies [10] and also require that the developer or maintainer possesses an understanding of the operations encapsulated by the abstract aspect as is the case for hot spots exposed in such a white-box fashion [11].

### 3.3 Comparing Frames with AspectJ

The strengths and weaknesses of frames and aspects are summarised in table 1. We can observe that the strengths of one technique are the weaknesses of the other and vice versa. A hybrid of the two approaches can provide essentially all the combined benefits thus increasing configurability, modularity, reusability, evolvability and longevity of product line assets.
Table 1. Comparing frames and AOP

<table>
<thead>
<tr>
<th>Capability</th>
<th>Framing</th>
<th>AOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Mechanism</td>
<td>Very comprehensive configuration possible</td>
<td>Not supported natively, dependent on IDE</td>
</tr>
<tr>
<td>Separation of Concern</td>
<td>Only non crosscutting concerns supported</td>
<td>Addresses problems of crosscutting concerns and code tangling</td>
</tr>
<tr>
<td>Templates</td>
<td>Allows code to be generalised to aid reuse in different contexts</td>
<td>Not supported</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Allows static autogeneration of code and reflecting.</td>
<td>Generates code which (in the case of advice) is bound at runtime</td>
</tr>
<tr>
<td>Language Independence</td>
<td>Supports any textual document and therefore any language</td>
<td>Constrained to implementation language although this will change as AOP gains wider acceptance</td>
</tr>
<tr>
<td>Use on Legacy Systems</td>
<td>Limited at present</td>
<td>Supports evolution of legacy systems at source and byte code level</td>
</tr>
<tr>
<td>Dynamic Runtime Evolution</td>
<td>Not supported</td>
<td>Possible in JAC and JMangler. Future versions of AspectJ will have support</td>
</tr>
</tbody>
</table>

4 Framed Aspects

Our approach to framed aspects is based on using aspects to encapsulate otherwise tangled features in the SPL and use frames to provide parameterisation and reconfiguration support for the feature aspects. The approach has been realised in the form of the Lancaster Frame Processor which is a trimmed down implementation of the functionality offered by XVCL. It only takes certain frame constructs and forces the programmer to use AOP techniques for the remainder. This balance of AOP and frames helps cut down on the template code clutter induced by frames alone and at the same time provides effective parameterisation and reconfiguration support through the ability to create meta variables and options which can be bound to a specification from the developer when the frame processor is executed.

Returning to our caching example, in the guide SPL, it should be possible for the cache to be configured to different specifications. Utilising framed aspects we have developed a cache that can be configured with the following parameters: <Scheme, MaxCacheSize, PercentToDel, ContentType> where Scheme = Access or Date or Size, MaxCacheSize = any integer, PercentToDel = any value between 1 and 100, and ContentType = Document, String, etc.

![Fig. 4. Using parameterised <adapt> to provide variations in the cache aspect](image)

The choice of different scheme strategies has an impact on the data structure within the cache as well as the deletion method. We can capture this within an aspect very easily by using the introduction mechanism where new fields and methods are inserted on defined
objects. We could then use inheritance to inherit these properties when we need them. However, a much neater approach is to frame these properties and use a parameterised adapt to incorporate them into our aspect (cf. fig. 4). To make the aspect more reusable across different platforms (ie J2SE and J2ME) we could generalise parts of the cache aspect so that they can store information without being constrained to the J2SE Document. The use of a framed aspect for the cache has effectively created a reusable and simpler to manage component, which would have been difficult to realise in AOP or frames alone without inducing some degree of complexity. We believe that the same technique can be applied to ease the introduction of other features into product lines.

5 Related Work

Our approach shares many similarities with AHEAD [12], where modules are created as a series of layered refinements, SALLY [13], which allows AspectJ style introductions to be parameterised and Aspectual Collaborations [14] where modular programming and AOP techniques are combined. The key difference in our approach is in the language independence of frames which means that there are no constraints as to which programming language or AOP technique is used.

6 Conclusion

In this paper we have shown how aspectual modules can benefit from parameterisation and generalisation that frame technology brings. We have demonstrated how the integration of new features into a product line can be simplified and believe the same technique can be applied to different concerns. We believe that our approach offers an effective approach to achieve the best of what both technologies have to offer in terms of flexibility, reusability and evolvability. Utilising AO and Frames allows crosscutting concerns to be localised thus improving system comprehensibility and minimising erosion of the product line architecture.

References