Supporting Product Line Evolution with Framed Aspects

Neil Loughran[1], Awais Rashid[1], Weishan Zhang[2] and Stan Jarzabek[3]

[1]Computing Department, Lancaster University, Lancaster LA1 4YR, UK
(loughran | awais) @comp.lancs.ac.uk
[2]School of Software Engineering, Tongji University, Shanghai, China 200311
zhangws@mail.tongji.edu.cn
[3]Department of Computer Science, National University of Singapore, Singapore 117543
stan@comp.nus.edu.sg

Abstract. This paper discusses how evolution in 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 changes. 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%[16] 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, facilitate modification of the architecture or ease the introduction of new features are needed if we are to reduce the risk of architectural erosion [21]. 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 *preventive* (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 Hashtable 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).

```
class Editor extends JEditorPane implements HyperlinkListener
  {
  private Network network;
  private Hashtable cache = new Hashtable();
  // .. methods for adding and retrieving data to/from cache
  //.. constructor and editor initialisation

  public void hyperlinkUpdate(HyperlinkEvent e)
    {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
      String url = e.getURL().toString();
      Document cachedPage = (Document)getFromCache(url);
      if(cachedPage == null)
        {
        network.requestInfo(this, url);
        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 a cache into a simple editor pane. Using a frame processor such as XVCL [7] we can tag this code to ease its retraction from the codebase (cf. fig. 2).

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.

```
class Editor extends JEditorPane implements HyperlinkListener
  {
  private Network network;
  <option cache>
  private Hashtable cache = new Hashtable();
  // .. methods for adding and retrieving data to/from cache
  </option>
  //.. constructor and editor initialisation
  public void hyperlinkUpdate(HyperlinkEvent e)
    {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
      String url = e.getURL().toString();
      <option cache>
      Document cachedPage = (Document)getFromCache(url);
      if(cachedPage == null)
        {
      </option>
        network.requestInfo(this, url);
        <option cache>
        addToCache(url, this.getDocument);
        }
       else
        {
        // get record from cache and display it
        this.setDocument((Document)cachedPage.getContent());
        }
        </option>
      }
    }
  }
```

Fig. 2. Using frame option tags to identify caching code

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], Hyper/J [9] and emerging frameworks such as AspectWerkz [10], JBoss AOP [17] and Nanning [18], are now gaining considerable support as a means for managing the separation of concerns and features which would traditionally lead to unmanageable code tangled across multiple classes in OO systems. Examples of concerns in 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). To illustrate this, consider the AspectJ implementation of the cache in fig. 3, which 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 without 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.

```
aspect CacheAspect
  {
  private Hashtable cache = new Hashtable();
  // ..code
  void around(Editor g, String url): args (g,url) &&            } 1
      call (public void Network.requestInfo(Editor, String))    }
      {                                                          } 2
      PageContent cachedPage=(PageContent) cache.get(url);      }
      if(cachedPage==null)                                       }
          {                                                      }
           proceed(g,url);                                       }
           PageContent page=new PageContent(g.getDocument());    }
           addToCache(url,page);                                 }
           }                                                     } 3
      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. In deeper inheritance structures this can lead to inheritance anomalies [11] 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 [12].

3.3 Comparing Frames with AspectJ

The strengths and weaknesses of frames and aspects are summarised in table 1.

Table 1. Comparing frames and AOP

Capability	Framing	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 and code tangling.
Templates	Allows code to be generalised to aid reuse in different contexts	Not supported
Code Generation	Allows static autogeneration of code and refactoring.	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 this will change as AOP gains wider acceptance
Use on Legacy Systems	Limited at present	Supports evolution of legacy systems at source and byte code level
Dynamic Runtime Evolution	Not supported	Possible in JAC and JMangler. Future versions of AspectJ will have support.

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.

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 reduce 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.

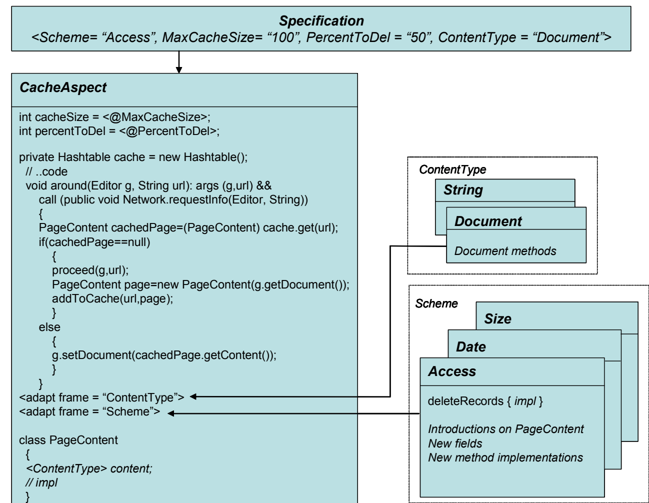

Fig. 4. Using parameterised <adapt> to provide variations in the cache aspect

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 cleaner 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 (i.e. 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.

There are numerous ways of utilising the framed aspect approach. In the previous example the aspect code was affected directly with frame tags, however we have found an alternative approach for use in more complex scenarios where there is a need for more control of how different modules (alternative and optional features) can be merged together in terms of constraints and rules for configuration (cf. fig. 5).

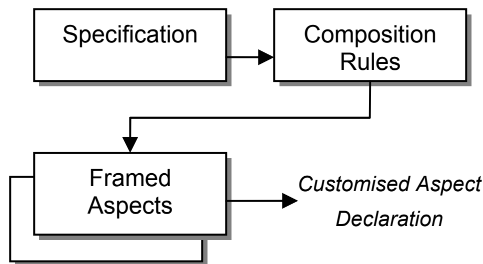

Fig 5. Alternative approach to using framed aspects

We have found that this approach offers a very powerful mechanism for removing even more of the invasive frame code (mainly due to the moving of option and adapt tags from the framed aspect code to the composition rules) and have developed a methodology which allows a feature diagram using FODA [19] for a given reusable aspect component to be created and mapped directly to framed aspects. A future paper [20] will demonstrate this approach in more detail.

5. Related Work

The framed aspect approach displays many similarities with feature oriented programming (FOP, Genvoca et al) [13], where modules are created as a series of layered refinements, SALLY [14], where introductions can be parameterised and Aspectual Collaborations [15] where modular programming and AOP techniques are combined. In FOP, composition is performed by layers stacked upon one another, with upper layers adding refinements to the lower ones via parameterisation, however, the technique is limited at present to static crosscutting feature refinements. With regards to SALLY, only its special style of introductions can be parameterised whereas in framed aspects any AOP construct can be in any AOP language. Aspectual Components have a similarity to framed aspects as they allow for external composition and black box reuse. Emerging AOP frameworks such as AspectWerkz, JBoss AOP and Nanning Apsects allow for aspects to be created as standard classes and configured via XML files which contain advice and other AO details. The main difference with framed aspects over the aforementioned is in the language independence of frames and the flexibility of parameterisation where any programming construct can be a parameter.

6. Conclusion

In this paper we have shown how aspects can benefit from the parameterisation and generalisation support 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. Product line engineering benefits from the configurational power that framed aspects bring and helps to improve the integration of features that would normally crosscut multiple modules in OO and traditional framing technologies. Utilising AO and Frames allows crosscutting concerns to be localised thus improving system comprehensibility and minimising design erosion of architectures.

References

[1] Bassett, P.: Framing Software Reuse - Lessons from the Real World, Prentice Hall (1997).

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.: Aspect Oriented Programming, Proc. ECOOP '97.

[3] Davies, N et al.: Lancaster GUIDE Project homepage http://www.guide.lancs.ac.uk/

[4] Cusumano, M.A. and Selby R.W.: Microsoft Secrets, Simon & Schuster, New York (1998).

[5] Colyer, A., Blair, G., Rashid, A.: Managing Complexity in Middleware. Workshop on Aspect Components and Patterns, AOSD 2003.

[6] Lientz, B., Swanson, E., and Tompkins, G.: Characteristics of Application Software Maintenance, CACM 21(6) June 1978.

[7] XVCL homepage, http://fxvcl.sourceforge.net

[8] AspectJ Team, "AspectJ Project", http://www.eclipse.org/aspectj/, 2003.

[9] IBM Research, Hyperspaces, http://www.research.ibm.com/hyperspace/

[10] AspectWerkz homepage, http://aspectwerkz.codehaus.org/

[11] Mikhajlov, L. and Sekerinski, E.: A Study of The Fragile Base Class Problem. Proc. ECOOP '98, Lecture Notes in Computer Science, 1445, (Springer-Verlag 1998), pp. 355-382.

[12] Fayad, M. E. and Schmidt, D. C.: Object-Oriented Application Frameworks. CACM 40(10), pp. 32-38, (1997).

[13] Batory, D., Sarvela, J. N., Rauschmayer, A.: Scaling Step-Wise Refinement. ICSE 2003.

[14] Hanenberg, S. and Unland, R.: Parametric Introductions. Proc. of AOSD 2003, pp. 80-89.

[15] Lieberherr, K., Lorenz, D. H., Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. The Computer Journal, 46(5) 2003.

[16] Lehman M. M., Ramil J. F. and Kahen, G. A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics. Technical report Imperial College London 2001.

[17] JBoss AOP homepage, http://www.jboss.org/developers/projects/jboss/aop.jsp

[18] Nanning Aspects homepage, http://nanning.codehaus.org

[19] Kang, K. C. et al. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21,Software Engineering Institute, Carnegie Mellon University, 1990.

[20] Loughran, N., Rashid, A. Framed Aspects: Supporting Configurability and Variability for AOP, *submitted to ICSR 2004.*

[21] Van Gurp J. and Bosch J., Design Erosion: Problems and Causes, Journal of Systems & Software, vol 61, issue 2, 2002.