Notice of Violation of IEEE Publication Principles

"Software Evolution with Feature-Oriented and Aspect-Oriented Programming,"
by Jian-hui Wang and Chun-zhang Bai,
in the 3rd International Conference on Innovative Computing Information and Control, 2008. ICICIC 2008.

After careful and considered review of the content and authorship of this paper by a duly constituted expert committee, this paper has been found to be in violation of IEEE's Publication Principles.

This paper contains significant duplication of original text from the paper cited below. The original text was copied without attribution and without permission.

Due to the nature of this violation, reasonable effort should be made to remove all past references to this paper, and future references should be made to the following paper:

“Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution”
by Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake.
in the Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE), pp 3–16. School of Computer Science, University of Magdeburg, July 2005.

Software Evolution with Feature-Oriented and Aspect-Oriented Programming

Jian-hui Wang
Shenyang Normal University
Department of Kexin Software
Northern Huanghe Str. 253, 110034 Shenyang
jwang116@hotmail.com

Chun-zhang Bai
Shenyang Normal University
Center of General Education
East Chongshan Rd. 46-2, 110032 Shenyang
czbai802@hotmail.com

Abstract

Common techniques of Feature-Oriented Programming (FOP) are important for software evolution and appropriate for implementing program families. It is discussed that the shortcomings in the crosscutting modularity cause problems in implementing unanticipated features. The discussed problems of FOP in this regards complicate the evolution of software. The integration of concepts of Aspect-Oriented Programming (AOP) into existing FOP solutions is proposed. Three approaches to solve these problems are presented: Multi Mixins, Aspectual Mixins and Aspectual Mixin Layers. Whereas, the first two approaches are only of conceptual nature, the third approach is implemented and FeatureC++ with the ability to express Aspectual Mixin Layers is enhanced. FeatureC++ supports FOP in C++ and solves several problems regarding the lacking crosscutting modularity by adopting AOP concepts.

1. Introduction

The idealized goal of software engineers is to reuse as much as possible code from previous development stages to build a new version of the software. To achieve this, software must be designed reusable, extensible, and customizable. A heavily discussed approach to implement software with such virtues to support software evolution are program families [1]. The key idea is to arrange the design and implementation as a layered stack of functionalities. Different programs consist of different layers. Thus, implemented layers can be reused in multiple programs. A fine-grained layered architecture leads to reusable, extensible, and customizable software. Representative studies in the domains of databases [2], middleware, avionics, and network protocols show that Feature-Oriented Programming (FOP) [3] and Mixin Layers [4] are appropriate to implement such layered, step-wise refined architectures. However, FOP yields some problems in expressing features and evolving software:

(1) FOP lacks adequate crosscutting modularity [5].

(2) Currently FOP is still an academic concept that is not widely accepted in the industry.

Consequently, our contribution is to solve both problems, supporting crosscutting modularity and using C++ as base language. We have developed FeatureC++, an extension to C++ that supports FOP. This article focuses primarily on the first problem and presents our investigations in solving the problem of insufficient crosscutting modularity. Our approach to improve the crosscutting modularity is to combine traditional FOP concepts with concepts of Aspect-Oriented Programming (AOP). We have elaborated three ways to integrate AOP concepts into FOP: Multi Mixins, Aspectual Mixins, Aspectual Mixin Layers.

2. Problems of FOP and Advantages of AOP

The purpose of FOP is to implement program families and separate crosscutting concerns. FOP yields promising results in this respect [6, 7]. However, problems occur in implementing unanticipated features: We argue that the frequently needed, unanticipated modifications and extensions of evolving software cause code tangling and code scattering. Mostly these new features are crosscutting concerns, and FOP is not able to modularize them all appropriately. From this point of view we perceive the solution to the problem of insufficient crosscutting modularity as an improvement for software evolution. The following paragraphs introduce the key problems and point to advantages of AOP in these respects.

(1) Homogeneous vs. Heterogeneous Crosscuts. Homogeneous crosscutting concerns are distributed over several join points but apply every time the same code, e.g. logging; Heterogeneous crosscuts apply varying code, e.g. authentication [8]. Current AOP languages focus on homogeneous concerns whereas FOP languages deal with heterogeneous concerns. Both language paradigms can deal

The 3rd Intetnational Conference on Innovative Computing Information and Control (ICICIC'08)
978-0-7695-3161-8/08 $25.00 © 2008 IEEE

with both types of concerns but often this results in complicated code, code redundancy and inelegant workarounds. However, both are important for software evolution. Consequently, our objective is to enhance FOP with the possibility to deal with homogeneous concerns in an adequate way.

(2) Static vs. Dynamic Crosscutting. Both FOP and AOP deal with dynamic crosscutting. We argue, however, that the way AOP deals with dynamic crosscutting, namely by using pointcut expressions and advices, is more expressive. With regard to software evolution, we argue the more complex a software becomes (as this is the case of evolving software) the more the programmer needs to specify such complex feature bindings.

(3) Hierarchy-Conform Refinements. Using FOP, feature refinements depend on the structure of parent features. Usually, a feature refines a set of classes and extends methods. AOP is able to implement nonhierarchy- conform refinements by using wildcards in pointcut expressions. The problem of a rising abstraction level is serious to evolving software, because at the beginning of building software the abstraction of subsequent development phases cannot be foreseen.

(4) Excessive Method Extensions. The problem of excessive method extensions occurs if (1) a feature crosscuts a large fraction of existing implementation units and if (2) it is a homogeneous concern. For instance, if a feature wants to add multi-threading support, it has to extend lots of methods, and adds synchronization code. This code is in almost all methods the same and therefore redundant, e.g. setting lock variables. AOP deals with this problem by using wildcards in pointcut expressions to specify a set of target methods (join points). This prevents code redundancies and eases software evolution.

(5) Method Interface Extensions. The problem of method interface extensions frequently occurs in incremental designs. As an extended interface we understand an extended argument list. This problem occurs if refinements require additional parameters, e.g. an additional session id or a reference to a locking variable. Indeed, using some workaround this problem could be avoided. But AOP with its pointcut mechanism is much more elegant.

3. Overview of FEATUREC++

In order to implement FeatureC++, we have adopted the basic concepts of the ATS: Features are implemented by Mixin Layers. A Mixin Layer consists of a set of collaborating Mixins (which implement class fragments). Figure 1 depicts a stack of three Mixin Layers (1 - 3) in top down order.

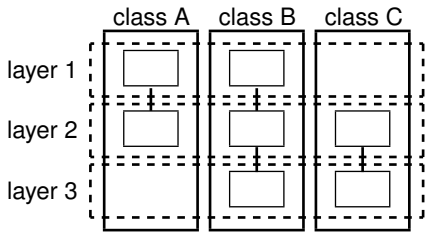

Figure 1. Stack of Mixin Layers

The Mixin Layers crosscut multiple classes (A - C). The boxes represent the Mixins. These Mixins that belong to and constitute together a complete class are called refinement chain. Solid lines represent refinement relations and connect refinement chains. Mixins that start a refinement chain are called constants; All others are called refinements. A Mixin A that is refined by Mixin B is called parent Mixin or parent class of Mixin B. Consequently, Mixin B is the child class or child Mixin of A. Similarly, we speak of parent and child Mixin Layers. In FeatureC++ Mixin Layers are represented by file system directories. Therefore, FeatureC++ represents them not explicitly (this follows the principle of AHEAD). Those Mixins, found inside the directories are assigned to be the members of the enclosing Mixin Layer.

FeatureC++ adopts the syntax of the Jak language [3]. The following paragraphs introduce the most important language features using an example, a buffer that serializes and stores objects.

(1) Constants and Refinements. Each constant and refinement is implemented as a Mixin inside exactly one source file. Each constant is the root of a chain of refinements:

```
class Buffer {
  char * buf ;
  void put (char *s) {}
};
```

Refinements refine constants as well as other refinements. They are declared by the keyword refines:

```
refines class Buffer {
  int length ;
  int getLength () {}
};
```

Usually, they introduce new members attributes and methods:

```
refines class Buffer {
  void put (char *s) {
    if (strlen(s)+getLength()<MAX_LEN)
      super::put(s); }
};
```


The 3rd Intetnational Conference on Innovative Computing Information and Control (ICICIC'08)
978-0-7695-3161-8/08 $25.00 © 2008 IEEE

(2) Extending Methods. Refinements can extend methods of their parent classes. To access the extended method the super keyword is used. Super refers to the type of the parent Mixin. It has a similar semantic to the Java super keyword and is related to the proceed keyword of AspectJ and AspectC++.

4. Enhancing FOP with AOP Concepts

This section presents our first results in integrating AOP concepts into FOP to support software evolution. The presented approaches show that there are numerous possibilities for such symbiosis.

4.1. Multi Mixins

Our first idea to prevent a programmer from excessive method extensions, hierarchy-conform refinements, and to support homogeneous crosscuts were Multi Mixins. The key idea, instead of refining one Mixin by another one Mixin only, is to refine a whole set of parent Mixins. Such sets are specified by wildcards ('%') adopted from AspectC++. Both Multi Mixins, depicted in the following, use wildcards to specify the Mixins and methods they refine.

```
refines class Buffer% {};
refines class Buffer {
  void put%(...)  {}
};
```

The first refines all classes that start with "Buffer". The second refines all methods of Buffer that start with "put". The meaning of the first type of refinement is straight forward: The wildcard Buffer% has the same effect as one creates a set of new refinements for each found Mixin that matches the pattern (Buffer%). This type of Multi Mixin eases the implementation of static homogeneous features in FOP.

The second type of Multi Mixins, which refines methods, eases the expression of dynamic homogeneous features. Similar to pointcuts and advices in AOP languages, one code fragment can be assigned to multiple methods. However, with Multi Mixins it is not possible to implement execution.

4.2. Aspectual Mixin Layers

The idea behind Aspectual Mixin Layers is to embed aspects into Mixin Layers. Each Mixin Layer contains a set of Mixins and a set of aspects. Doing so, Mixins implement heterogeneous and hierarchy-conform crosscutting, whereas aspects express homogeneous and nonhierarchy-conform crosscutting. In other words, Mixins refine other Mixins and depend, therefore, on the structure of the parent layer. These refinements follow the static structure of the parent features and encapsulate heterogeneous crosscuts. Aspects refine a set of parent Mixins by intercepting method calls and executions as well as attribute accesses. Therefore, aspects encapsulate homogeneous and non-hierarchy-conform refinements. Furthermore, they support advanced dynamic crosscutting.

Figure 2 shows a stack of Mixin Layers that implement some buffer functionality, in particular, a basic buffer with iterator, a separated allocator, synchronization, and logging support.

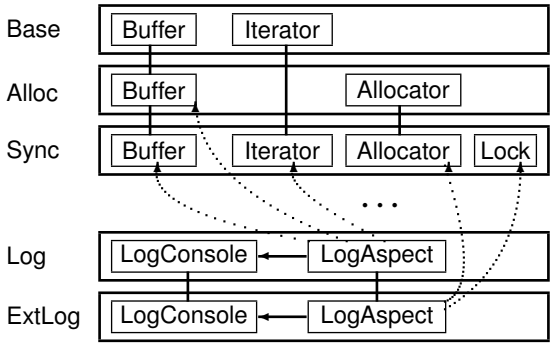

Figure 2. Implementing and refining a logging feature using Aspectual Mixin Layers.

Whereas the first three features are implemented as common Mixin Layers, the Logging feature is implemented as an Aspectual Mixin Layer. It consists of a logging aspect and a logging console. The logging console prints out the logging stream and is implemented using a common Mixin. The logging aspect captures a whole set of methods that will be refined with logging code (dotted arrows). This refinement is homogeneous, non-hierarchy-conform, and depends on the runtime control flow (dynamic crosscutting). Moreover, the use of wildcards prevents the programmer of excessive method extensions. Without Aspectual Mixin Layers the programmer has to extend all target methods.

A further highlight of Aspectual Mixin Layers is that aspects can refine other aspects. Figure 2 shows an Aspectual Mixin Layer that refines the logging aspect by additional join points to extend the set of intercepted methods. Additionally, the logging console is refined by additional functionality, e.g. a modified output format. Aspects can refine the methods of parents aspect as well as the parent pointcuts. This allows to easily reuse and extend of existing join point specifications (as in the logging example).

4.3. Aspectual Mixins

The idea of Aspectual Mixins is to apply AOP language concepts directly to Mixins. In this approach, Mixins refine

other Mixins as with common FeatureC++ but also define pointcuts and advices (see the following).

```
refines class Buffer {
  int length () {}
  pointcut log()=call("% Buffer::%( )");
};
```

In other words, Aspectual Mixins are similar to Aspectual Mixin Layers but integrate pointcuts and advices directly into its Mixin definition. In the following, we discuss only the important differences:

The set of pointcuts, advices, and aspect-specific attributes and methods is called aspectual subset of the overall Mixin. This mixture of AOP concepts and Mixins reveals some interesting issues: Using Aspectual Mixins the instantiation of aspects is triggered by the overall Mixin instances. Regarding the above presented example, the buffer Mixin and its aspectual subset are instantiated as many times as the buffer. This corresponds to the perObject qualifier of AspectJ. However, in many cases only one aspect instance is needed. To overcome this problem, we think of introducing a perObject and perClass qualifier to distinguish these cases. This introduces a second problem: If an aspect, part of an Aspectual Mixin, uses non-static members of the overall Mixin it depends on the Mixin instance. In this case, it is forbidden to use the perClass qualifier. FeatureC++ must guarantee that perClass Aspectual Mixins, especially their aspectual subset, only access static members of the overall Mixin instance. In case of perObject Aspectual Mixins this is not necessary.

Finally, we want to emphasize that all three approaches are not specific to FeatureC++. All concepts can be applied to other AOP/FOP languages.

5. Solutions and Discussion

All three approaches provide solutions for problems of FOP with crosscutting modularity discussed in Section 2. Table 1 summarizes the improvements to FOP with respect to the above presented problems.

Table 1. Comparison of approaches

approach	(1)	(2)	(3)	(4)	(5)
Multi Mixins	√	-	√	√	(√)
Aspectual Mixin Layers	√	√	√	√	√
Aspectual Mixins	√	√	√	√	√

6. Conclusion

In this paper we have argued that common FOP techniques are important for software evolution and appropriate for implementing program families. However, we have discussed the present drawbacks regarding crosscutting modularity and the missing support of C++. We have stated that the shortcomings in the crosscutting modularity cause problems in implementing unanticipated features. Often, these features are wide-spread crosscutting concerns. The discussed problems of FOP in this regards complicate the evolution of software. Consequently, we have presented our approach: FeatureC++ supports FOP in C++ and solves several problems regarding the lacking crosscutting modularity by adopting AOP concepts.

In this paper, we have focused on solutions to these problems to ease evolvability of software. We have summarized the problems of FOP, advantages of AOP in these respects, and presented three approaches to solve these problems: Multi Mixins, Aspectual Mixins and Aspectual Mixin Layers. Whereas, the first two approaches are only of conceptual nature, we have implemented the third approach and enhanced FeatureC++ with the ability to express Aspectual Mixin Layers.

References

[1] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *TSE*, IEEE, SE-5(2):106-109, 1979.

[2] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *TOSEM*, ACM, 1(4):447-451, 1992.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, IEEE, 30(6):321-330, 2004.

[4] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *TOSEM*, ACM, 11(2):203-208, 2002.

[5] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):979-983, 2003.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Germany, 2000.

[7] R. Filman. *Aspect-Oriented Software Development*. Addison-Wesley, Germany, 2004.

[8] V. Singhal and D. Batory. A Language for Large-Scale Reusable Software Components. *Workshop on Software Reuse*, IEEE, 133-137, 1993.

978-0-7695-3161-8/08 $25.00 © 2008 IEEE