Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake

Department of Computer Science
University of Magdeburg, Germany
email: {apel,leich,rosenmue,saake}@iti.cs.uni-magdeburg.de

Abstract. Starting from the advantages of using Feature-Oriented Programming (FOP) and program families to support software evolution, this paper discusses the drawbacks of current FOP techniques. In particular we address the insufficient crosscutting modularity that complicates software evolution. To overcome this tension we propose the integration of concepts of Aspect-Oriented Programming (AOP) into existing FOP solutions. As study object we utilize FeatureC++, a proprietary extension to C++ that supports FOP. After a short introduction to basic language features of FeatureC++, we summarize the problems regarding the crosscutting modularity. In doing so, we point to the strengths of AOP that can help. Thereupon, we introduce three approaches that combine FOP and AOP concepts: *Multi Mixins*, *Aspectual Mixins*, and *Aspectual Mixin Layers*. Furthermore, we discuss their benefits for software evolution.

1 Introduction

Nowadays software is subject to frequent changes in order to react to altering and evolving requirements. The process of continuous adaptation, extension, and customization is known as *software evolution*. This article focuses on the evolution of the design and the implementation base. 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 [18]. Program families group programs with similar functionalities in families. 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 [18]. Representative studies in the domains of databases [4], middleware [1], avionics [3], and network protocols [4] show that *Feature-Oriented Programming (FOP)* [5] and *Mixin Layers* [21] are appropriate to implement such layered, step-wise refined architectures. However, FOP[1] yields some problems in expressing features and evolving software:

1. FOP lacks adequate crosscutting modularity. During the evolution, software have to be adapted to fit unanticipated requirements and circumstances. This results in modifications and extensions that crosscut many existing implementation units in numerous ways [13].
2. Currently FOP is still an academic concept that is not widely accepted in the industry. We argue that is because of the focus on Java that is not acceptable in many domains, e.g. operating systems, databases, middleware, realtime embedded systems, etc. Even these domains demand for appropriate support of software evolution. Currently, C++-based solutions are too complex and hard to use [21, 19]. Moreover, an adequate tool support is missing.

Consequently, our contribution is to solve both problems, supporting crosscutting modularity and using C++ as base language. We have developed FeatureC++[2], an extension to C++ that supports FOP [2]. This article focuses primarily on the first problem and presents our investigations in solving the problem of insufficient crosscutting modularity. FeatureC++ serves as study object and representative FOP language. A detailed introduction to FeatureC++ is given in [2]. Our approach to improve the crosscutting modularity is to combine traditional FOP concepts with concepts of *Aspect-Oriented Programming (AOP)* [13]. AOP focuses on the separation and modularization of crosscutting concerns and is therefore best qualified to improve FOP. We have elaborated three ways to integrate AOP concepts into FOP: *Multi Mixins, Aspectual Mixins, Aspectual Mixin Layers*. This article introduces and compares them, as well as discusses their pros and cons with regard to software evolution.

The remaining article is structured as follows: Section 2 gives some background information about FOP and AOP. Section 3 introduces the basic language concepts of FeatureC++. Thereupon, Section 4 reviews the problems of FOP in modularizing crosscutting concerns. In this regard, we point to the advantages of AOP to solve these problems. Section 5 introduces our three approaches to combine AOP and FOP, and discusses theirs pros and cons. Afterwards, Section 6 reviews a selection of related work. Finally, Section 7 gives a conclusion.

2 Background

Pioneer work on software modularity was made by Dijkstra [11] and Parnas [18]. They have proposed the principle of *separation of concerns*. The idea is to separate each concern of a software system in a separate modular unit. They argue

[1] In the remaining article we presume that Mixin Layers are used to implement feature-oriented programs.

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

that this lead to maintainable, comprehensible software, which can be easily reused, customized, and evolved.

AOP was introduced by Kiczales et al. [13]. The aim of AOP is to separate crosscutting concerns. Common object-oriented methods fail in this context [13, 10]. The idea behind AOP is to implement so called orthogonal features as *Aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using join point specifications (*pointcuts*), an aspect weaver brings aspects and components together. Due to the ability to implement unanticipated features in a modular way AOP is an important technique to ease software evolution [12]. *AspectJ*[3] and *AspectC++*[4] are prominent AOP extensions to Java and C++.

FOP studies feature modularity in program families [5]. The idea of FOP is to build software by composing *features*. Features are basic building blocks that satisfy intuitive user-formulated requirements on the software system. Features refine other features incrementally. This *step-wise refinement* leads to a layered stack of features. *Mixin Layers* are one appropriate technique to implement features and layered designs [21]. The basic idea is that features are often implemented by a collaboration of class fragments (a.k.a. *roles*). A Mixin Layer is a static component encapsulating fragments of several different classes (*Mixins*) so that all fragments are composed consistently. Advantages are the high degree of modularity and the easy composition [21]. *AHEAD* is an architectural model for FOP and a basis for large-scale compositional programming [5]. It extends the concept of FOP to all software artifacts, e.g. UML diagrams, documentation, etc. It makes a broad consistent software evolution possible. The *AHEAD Tool Suite (ATS)*[5], including the *Jak* language, implements AHEAD for Java.

3 Overview of FeatureC++

This section gives a short overview of FeatureC++. For a more detailed introduction we refer to [2].

3.1 Introduction to Basic Concepts

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. The Mixin Layers crosscut multiple classes ($A-C$). The rounded boxes represent the Mixins. These Mixins that belong to and constitute together a complete class are called *refinement chain*. Solid lines represent refinement relationships and connect refinement chains (Fig. 1). Roots of a refinement chain are called *constants*; All

[3] http://eclipse.org/aspectj/
[4] http://www.aspectc.org/
[5] http://www.cs.utexas.edu/users/schwartz/Hello.html

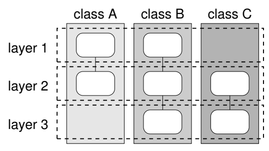

Fig. 1. Stack of Mixin Layers.

other Mixins 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.

3.2 Syntax of Basic Language Features

FeatureC++ adopts the syntax of the Jak language [5]. The following paragraphs introduce the most important language features by example, a buffer that serializes and stores objects.

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 (see Fig. 2).

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

Fig. 2. Defining a basic buffer.

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

Fig. 3. Adding a length attribute and an access method.

```
1  refines class Buffer {
2    void put(char *s) {
3      if(strlen(s) + getLength() < MAX_LEN)
4        super::put(s);
5    }
6  };
```

Fig. 4. Limiting the buffer length.

Refinements refine constants as well as other refinements. They are declared by the keyword *refines* (see Fig. 3). Usually, they introduce new members attributes and methods (Lines 2-3).

Extending Methods. Refinements can extend[6] methods of their parent classes (see Fig. 4). To access the extended method the *super* keyword is used (Line 4). *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++.

Further Language Features. Due to the space limitations, we omit a discussion of the below listed language feature of FeatureC++. A detailed introduction can be found in [2].

– FeatureC++ supports multiple inheritance, templates for generic programming, accessing overloaded methods from extern, as well as refinements of static methods, structs, and destructors.
– FeatureC++ solves several problems regarding class hierarchy extensions that are caused by the divergence of variations and extensions.
– FeatureC++ solves the constructor problem that occurs in incremental designs and results in unnecessary constructor redefinitions (cf. [20]).

4 Problems of FOP and how AOP could help

This section reviews problems of FOP regarding crosscutting modularity and software evolution. The purpose of FOP is to implement program families. Commonly, their design and implementation is well planned. FOP yields promising results in this respect (see [4, 3, 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 (as we will see soon). From this point of view we perceive the solution to the problem of insufficient crosscutting modularity as an improvement for software evolvability. The following paragraphs introduce the key problems and point to strengths of AOP in these respects. The discussion of the problems extends [17, 2].

[6] With 'extend' we refer to overriding and to call the overridden method.

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]. Common AOP languages focus on homogeneous concerns whereas FOP languages deal with heterogeneous concerns. Indeed, both language paradigms can deal 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 opportunity to handle homogeneous concerns in an adequate way.

Static vs. Dynamic Crosscutting. Both FOP and AOP deal with dynamic crosscutting[7]. Dynamic crosscutting affects the runtime behavior and depends on the control flow. Static crosscutting affects the static structure of a base feature. We argue, however, that the way AOP deals with dynamic crosscutting, namely by using pointcut expressions and advices, is more expressive. Feature binding specifications as "bind feature A to all calls to method m that are in the control flow of method c and only if expression e is true" are difficult to express in FOP languages. 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.

Hierarchy-Conforming Refinements. Using FOP, feature refinements depend on the structure of parent features. Usually, a feature refines a set of classes and extends methods. For each implementation unit we want to refine, we have to introduce a new unit. In fact, the programmer is forced to express new features in terms of structural elements of the existing features. This becomes problematic if new features are implemented at a different abstraction level. AOP is able to implement non-hierarchy-conforming refinements by using wildcards in pointcut expressions [17]. The problem of a raising abstraction level is serious to evolving software because at the beginning of building software the abstraction of subsequent development phases cannot be foreseen. If the programmer is forced to express new features using abstractions of former features the code becomes unnecessary complicated, bloated, and difficult to understand.

We clarify this by an example (adopted from [17]). As basic feature we consider a stock information broker. This feature should be refined by a pricing feature. Whereas the broker is expressed in terms of stock information, requests, brokers, clients and database connections, the pricing feature is expressed using the intuitive product-consumer-pattern. FOP is not able to change the abstraction level accordingly. Instead, AOP is able to implement non-hierarchy-conforming refinements by using wildcards in pointcut expressions [17].

Excessive Method Extensions. The problem of excessive method extensions occurs (1) if a feature crosscuts a large fraction of existing implementation units and (2) if it is a homogeneous concern. For instance, if a feature wants to add

[7] Note that dynamic crosscutting is not dynamic weaving.

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.

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 [17].

Unpredictable Aspect Composition. This problem regards AOP languages only. Nevertheless it is of importance because we want to integrate AOP mechanisms into FOP. The problem of current AOP languages is that the binding of aspects is independent of the current development stage. That means an aspect may affect subsequent integrated features. This can lead to unpredicted effects, e.g. an aspect is unintentionally bound to new features. In [15] an alternative composition mechanism is proposed. They argue that with regard to software (program family) evolution, features should only affect features of prior development stages. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. This decreases aspect reuse and complicates incremental design. Consequently, our approaches satisfy this principle.

5 Enhancing FOP with AOP concepts

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

5.1 Multi Mixins

Our first idea to prevent a programmer from excessive method extensions, hierarchy-conforming 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 Figure 5, use wildcards to specify the Mixins and methods they refine. The first refines all classes that start with "*Buffer*" (Line 1). The second refines all methods of Buffer that start with "*put*" (Line 3-5). 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.

```
refines class Buffer% { /* ... */ };

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

Fig. 5. Two Multi Mixins that refine sets of Mixins and methods.

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* or *cflow* pointcuts.

5.2 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. Hence, Mixins implement heterogeneous and hierarchy-conforming crosscutting, whereas aspects express homogeneous and non-hierarchy-conforming 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-conforming refinements. Furthermore, they support advanced dynamic crosscutting.

Figure 6 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.

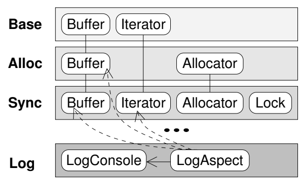

Fig. 6. Implementing a logging feature using Aspectual Mixin Layers.

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 set of methods that will be refined with logging code (dashed arrows).

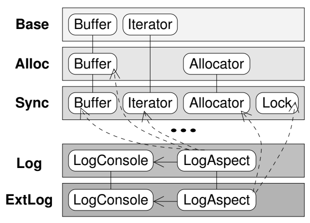

Fig. 7. Refining an Aspectual Mixin Layer.

```
1 refines aspect LogAspect {
2   void print() {
3     changeFormat();
4     super::print();
5   }
6   pointcut log() = call("%␣%::get()") || super::log();
7 };
```

Fig. 8. An aspect embedded into a Mixin Layer.

This refinement is homogeneous, non-hierarchy-conforming, 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 manually.

A further highlight of Aspectual Mixin Layers is that aspects can refine other aspects. Figure 7 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). Note that refining/extending aspects is conceptually different than applying aspects themselves. Whereas the former case results in a transformation of the aspect code before applying them to the target program, the latter case applies the aspects in two steps which leads to two independent aspect instances.

To express aspects in Aspectual Mixin Layers we adopt the syntax of AspectC++. Figure 8 depicts an aspect refinement that extends a logging feature including a logging aspect. It overrides a parent method in order to adjust the output format (Line 2-5) and refines a parent pointcut to extend the set of target join points (Line 6). Both is done using the *super* keyword.

```
1  refines class Buffer {
2    int length() { /* ... */ }
3    pointcut log() = call("% Buffer::%(...)");
4  };
```

Fig. 9. Combining Mixins and AOP elements.

5.3 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 Fig. 9). 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 (Fig. 9, Lines 1-4) and its aspectual subset (Line 3) 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.

5.4 Discussion

All three approaches provide solutions for problems of FOP with crosscutting modularity discussed in Section 4:

– support homogeneous and heterogeneous crosscuts (1)
– extended dynamic crosscutting (pointcuts, etc.) (2)
– non-hierarchy-conforming refinements (3)
– prevent excessive method extensions (4)
– handling method interface extensions (5)

Table 1 summarizes the improvements to FOP with respect to the above presented problems.

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

Table 1. Evaluation of approaches.

5.5 Bounding Quantification.

A further highlight of all three AOP extensions is a specific bounding mechanism that supports a better incremental design and that prevents unpredictable aspect composition (cf. Sec. 4). This mechanism bounds aspects and their effects on the target program. To implement this bounding mechanism the user-declared join point specifications must be restructured: Type names in wildcards are translated in order to match only these types that are declared by the current and the parent layers. Each wildcard expression that contains a type name is translated into a set of new expressions that refer to all type names of the parent classes. Figure 10 shows a synchronization aspect that is part of an Aspectual Mixin Layer. It has two parent layers (*Base*, *Log*) and several child layers. Using this novel bounding mechanism, FeatureC++ transforms the aspect and the pointcut as depicted in Figure 11. This transformation works similar for Aspectual Mixins. In case of Multi Mixins we have to add a mechanism for combining wildcard expression logically.

```
aspect SyncAspect {
  pointcut sync() : call("% Buffer::put(...)");
};
```

Fig. 10. A synchronization aspect with a simple pointcut expression.

```
aspect SyncAspect_Sync {
  pointcut sync() : call("% Buffer_Sync::put(...)")
            || call("% Buffer_Log::put(...)")
            || call("% Buffer_Base::put(...)");
};
```

Fig. 11. Bounding quantification by transforming pointcuts.

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

6 Related Work

Several approaches aim to combine AOP and FOP. Mezini et al. argue that using AOP as well as FOP standalone lacks feature modularity [17]. They propose *Caesar* as combined approach. Similar to FeatureC++, Caesar supports dynamic crosscutting using pointcuts. Instead of FeatureC++, the focus of Caesar is on aspect reuse and on-demand remodularization. *Aspectual Collaborations* proposed by Lieberherr et al. [14] encapsulate aspects into modules, with expected and provided interfaces. The rationales behind this approach are similar to Caesar. Colyer et al. propose the *principle of dependency alignment*: a set of guidelines for structuring features in modules and aspects with regard to program families [9]. They distinguish between orthogonal and weak-orthogonal features/concerns. Loughran et al. support the evolution of program families with *Framed Aspects* [16]. They combine the advantages of frames and AOP, to serve unanticipated requirements.

7 Conclusion

In this paper we argued that common FOP techniques are important for software evolution and appropriate for implementing program families. However, we discussed the drawbacks regarding crosscutting modularity and the missing support of C++. We 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 these 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. A first prototype can be found at the FeatureC++ web site[8]. In ongoing work we will apply all three approaches to real-world case studies.

References

1. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *Proceedings of the ASE Workshop on Software Engineering and Middleware (SEM)*, volume 3437 of *Lecture Notes on Computer Science*. Springer, 2005.

[8] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

2. S. Apel et al. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical report, Deptartment of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
3. D. Batory et al. Creating Reference Architectures: An Example from Avionics. In *Proceedings of the Symposium on Software Reusability (SSR)*, 1995.
4. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4), 1992.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6), 2004.
6. D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2), 1997.
7. R. Cardone et al. Using Mixins to Build Flexible Widgets. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2002.
8. A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.
9. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University, 2004.
10. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
12. R. Filman et al. *Aspect-Oriented Software Development*. Addison Wesley, 2004.
13. G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.
14. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5), 2003.
15. R. E. Lopez-Herrejon and D. Batory. Improving Incremental Development in AspectJ by Bounding Quantification. In *Software Engineering Properties and Languages for Aspect Technologies*, 2005.
16. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *Proceedings of the AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.
17. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Foundations of Software Engineering (FSE)*, 2004.
18. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2), 1979.
19. V. Singhal and D. Batory. P++: A Language for Large-Scale Reusable Software Components. In *Proceedings of the Workshop on Software Reuse*, 1993.
20. Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, 2000.
21. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), 2002.