Access Control in Feature-Oriented Programming

Sven Apel[a], Sergiy Kolesnikov[a], Jörg Liebig[a], Christian Kästner[b], Martin Kuhlemann[c], Thomas Leich[d]

[a] *Department of Informatics and Mathematics, University of Passau, Germany*
{apel,kolesnik,joliebig}@fim.uni-passau.de
[b] *Department of Computer Science and Mathematics, University of Marburg, Germany*
kaestner@informatik.uni-marburg.de
[c] *School of Computer Science, University of Magdeburg, Germany*
kuhlemann@iti.cs.uni-magdeburg.de
[d] *Metop Research Center, Magdeburg, Germany*
thomas.leich@metop.de

Abstract

In *feature-oriented programming* (*FOP*) a programmer decomposes a program in terms of features. Ideally, features are implemented modularly so that they can be developed in isolation. Access control mechanisms in the form of access or visibility modifiers are an important ingredient to attain feature modularity as they allow programmers to hide and expose internal details of a module's implementation. But developers of contemporary feature-oriented languages have not considered access control mechanisms so far. The absence of a well-defined access control model for FOP breaks encapsulation of feature code and leads to unexpected program behaviors and inadvertent type errors. We raise awareness of this problem, propose three feature-oriented access modifiers, and present a corresponding access modifier model. We offer an implementation of the model on the basis of a fully-fledged feature-oriented compiler. Finally, by analyzing ten feature-oriented programs, we explore the potential of feature-oriented modifiers in FOP.

Keywords: Feature-Oriented Programming, Feature Modularity, Access Control, Access Modifier Model, FUJI

1. Introduction

The goal of *feature-oriented programming* (*FOP*) is to modularize software systems in terms of features [1, 2]. A *feature* is a unit of functionality of a program that satisfies a requirement, represents a design decision, or provides a potential configuration option [3]. A *feature module* encapsulates exactly the code that contributes to the implementation of a feature [4]. The goal of the decomposition into feature modules is to construct well-structured and customizable software. Typically, from a set of feature modules, many different programs can be generated that share common features and differ in other features.

Many feature-oriented languages and tools aim at feature modularity, e.g, AHEAD/Jak [2], FeatureC++ [5], and FeatureHouse [6]. Feature modules are supposed to hide implementation details and to provide access via interfaces [7]. The rationale behind such information hiding is to allow programmers to develop, type check, and compile features in isolation. However, contemporary feature-oriented languages do not perform well with regard to feature modularity [7]; they lack sufficient abstraction and modularization mechanisms to support independent development based on information hiding, modular type checking, and separate compilation. In a theoretical work, Hutchins has shown that, in principle, feature-oriented languages are able to attain this level of feature modularity [8]. However, there are many open issues, such as the interaction with other language mechanisms.

An important ingredient for feature modularity that is missing in contemporary feature-oriented languages is a proper mechanism for controlling the visibility of or access to individual program elements. Access modifiers allow a programmer to define the scope and visibility of program elements such that implementation details can be encapsulated. For example, in Java, programmers use access modifiers (e.g., `private` or `public`) to grant or prohibit access to classes, methods, and fields. Access control has not been considered so far in research on feature-oriented languages. Contemporary feature-oriented languages do not provide proper access control mechanisms that take feature-oriented abstractions such as feature modules into account. The absence of a well-defined access control model for FOP breaks the encapsulation of feature code and leads to unexpected program behaviors and inadvertent type errors, as we will demonstrate. To improve the situation, we make the following contributions:

- We analyze object-oriented modifiers used in FOP and identify several shortcomings that limit the expressiveness of feature-oriented languages and that may lead to unexpected program behaviors and inadvertent type errors.
- We explore the design space of feature-oriented access control mechanisms and propose three concrete access modifiers.
- We present an access modifier model that integrates common object-oriented modifiers with our novel feature-oriented modifiers.
- We offer an implementation of the proposed modifiers on top of the fully-fledged feature-oriented compiler FUJI.
- We analyze ten feature-oriented programs to demonstrate that there is a potential for feature-oriented access control modifiers.

Especially, the last two contributions are novel compared to an earlier version of this work presented at FOSD'09 [9]. In a nutshell, we found evidence that there is a need for feature-oriented modifiers in FOP. Another observation is that features are mostly self-referential, which supports the view of features being cohesive units rather than being stepwise refinements, as we will explain.

2. Feature-Oriented Programming

Often, a feature-oriented language extends an object-oriented base language by mechanisms for the abstraction and modularization of features.[1] Here we concentrate on languages that gained considerable attention in the past: AHEAD-/Jak [2], FeatureHouse [6], FeatureC++ [5], Classbox/J [12], CaesarJ [13], and OT/J [14]. To implement the additions and changes a feature makes, these and other feature-oriented languages introduce a mechanism for class refinement. We illustrate the capabilities of FOP by means of a brief example: we introduce a class (Figure 1) and refine it subsequently (Figure 2). Note that in real feature-oriented programs, a feature typically comprises multiple classes and refinements.

In Figure 1, we depict a class Stack written in the Jak language, which is an extension of Java and belongs to the AHEAD tool suite [2]. The class definition is identical to a definition in Java except for the layer declaration (Line 1), which defines the feature to which class Stack belongs—in our case, feature Base.

Feature Base

```
1  layer Base;
2  class Stack {
3    private LinkedList elements = new LinkedList();
4    public void push(Object element) {
5      elements.addFirst(element);
6    }
7    public Object pop() {
8      return elements.removeFirst();
9    }
10 }
```

Figure 1: A basic stack implemented in Jak.

Feature Undo

```
1  layer Undo;
2  refines class Stack {
3    private Object lastPush = null;
4    private Object lastPop = null;
5    public void push(Object item) {
6      lastPush = item; lastPop = null;
7      Super.push(item);
8    }
9    public Object pop() {
10     lastPop = Super.pop();
11     lastPush = null; return lastPop;
12   }
13   public void undo() {
14     if (lastPush != null) Super.pop();
15     else if (lastPop != null) Super.push(lastPop);
16   }
17 }
```

Figure 2: A refinement of class Stack implemented in Jak.

[1] We are aware of some feature-oriented languages that build on languages that are not object-oriented [10, 11]. These languages are outside the scope of the paper, as they do not provide access modifiers like the ones we consider here.

In Figure 2, we depict a refinement of class Stack, declared by keyword refines (Line 2). The refinement is part of feature UNDO, which enables clients of the stack to revert the last operation. It adds a new method undo (Lines 13–16) and two new fields lastPush and lastPop (Lines 3 and 4) to class Stack. Furthermore, it refines the methods push and pop (by overriding; Lines 5–8 and 9–12) to store the last item added to or removed from the stack. Keyword Super refers to the class that has been refined (Lines 7 and 10).[2] Typically, a feature comprises multiple class declarations and class refinements that implement the feature in concert.

We visualize a feature-oriented program design—like the design of our stack example—using a *collaboration diagram* [15, 16, 17].[3] In Figure 3, we show a sample feature-oriented design, which decomposes the underlying object-oriented design into features. The design in Figure 3 consists of the five classes $A - E$ (represented by medium-gray boxes), which are located in the two packages P_1 and P_2 (represented by light-gray boxes). The diagram displays features ($F_1 - F_3$) as horizontal slices that cut across the core object-oriented design (represented by dark-gray boxes). Hence, a class is decomposed into several fragments, called *roles*, that belong to different features [16]; the set of roles belonging to a feature make up a feature module [4]. For example, class A consists of the roles A_1, A_2, and A_3; feature F_1 is implemented by the roles A_1, B_1, C_1, D_1, and E_1. The topmost role of a class is also called the *base class* (e.g., A_1) and the other roles are called *class refinements* (e.g., A_2 and A_3). The solid arrow denotes the refinement relationship between roles and the empty arrow denotes inheritance between full classes.

Note that the sample feature-oriented design of Figure 3 is minimal in the sense that it covers all situations that can occur in access control and it fits onto a single page. We created this design on the basis of the specifications of Java and Jak. Nevertheless, we checked that it applies also to the other languages we consider (see above). Despite its abstractness, we have implemented the design and used it for regression testing our compiler (see Section 5).

3. Problem Statement & Related Work

We explain the problems we encountered with feature-oriented languages by means of Jak—later we consider other feature-oriented languages. Jak, as a Java extension, has inherited the access modifiers of Java. Hence, programmers can control access to classes and members using the modifiers private, package, protected, and public.[4] But there are two problems with this:

[2] For brevity, we use a slightly less verbose notation than used in Jak; other feature-oriented languages use different keywords anyway.

[3] Note that the collaboration diagrams we refer to have their origin in collaboration-based design [16] and are not to be confused with UML collaboration diagrams.

[4] We assume a basic knowledge on Java's access modifiers. In Java, if a class, field, or method does not have an access modifier then only elements from the same package may access them. For sake of consistency, we assume modifier package for this case.

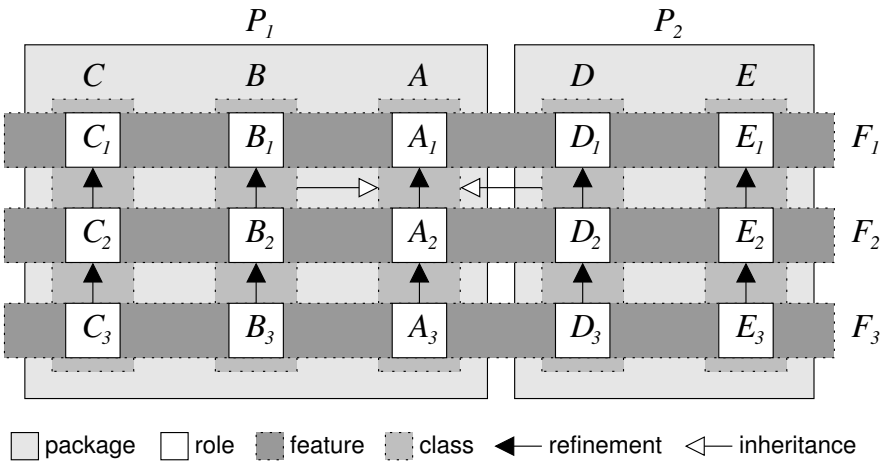

Figure 3: A sample feature-oriented design.

1. **Undefined semantics**: object-oriented modifiers interact in undefined ways with feature-oriented mechanisms such as class refinement.

2. **Limited expressiveness**: object-oriented modifiers are not expressive enough to control access in the presence of feature-oriented abstractions.

Undefined Semantics

Let us illustrate the problem of undefined semantics by means of our stack example. Suppose we refine our class Stack by applying a feature TRACE. Feature TRACE monitors the accesses to the stack and, as soon as the stack is changed, it writes all stack elements to the console. In Figure 4, we depict a corresponding refinement that refines the methods push (Lines 3–5) and pop (Lines 6–8), accesses the list storing the stack's elements (Lines 10 and 11), and prints them to the console (Line 11).

Feature TRACE

```
1  layer Trace;
2  refines class Stack {
3    public void push(Object item) {
4      Super.push(item); trace();
5    }
6    public Object pop() {
7      Object res = Super.pop(); trace(); return res;
8    }
9    private void trace() {
10     for (int i = 0; i < elements.size(); i++)
11       System.out.print(elements.get(i).toString() + " ");
12   }
13 }
```

Figure 4: A refinement of class Stack to trace accesses to a stack instance.

The question is whether the above example is correct. Is the class refinement allowed to access the private field elements of the refined class? The answer is not obvious since feature-oriented languages usually do not come with a specification and the behavior is de facto defined by the implementation of the compiler—a situation that we are going to change. Also, formally specified subsets of feature-oriented languages do not include modifiers [18, 19, 20, 21]. Compiling this code (or similar code) with the Jak compiler reveals that, depending on a compiler flag, the code is correct or not.

The background of this inconsistency is that the Jak compiler generates Java code in an intermediate step and it supports two options to do so [22]: in the first option, called Mixin, the compiler generates an inheritance hierarchy with one subclass per refinement (which is good for debugging); in the second option, called Jampack, the compiler generates a single class consisting of the elements of the base class and all of its refinements (which is good for performance). Comparing the two options, it becomes clear that we get different results in our example, which we illustrate in Figure 5.[5] Using Mixin (top of Figure 5), private field elements cannot be accessed because the refinement is translated to a subclass, which cannot access private members of superclasses. Using Jampack (bottom of Figure 5), private field elements can be accessed because all code of all refinements is moved to the class that is refined. So, we have two different results when compiling a single program depending on a flag that is intended for debugging and optimization.

One can argue for one or the other semantics of private in FOP, and certainly it is possible to fix either Mixin or Jampack such that both provide equal behavior. But we would like to stress that the semantics of access modifiers in general and their interaction with feature-oriented mechanisms such as class refinements are not well-defined in contemporary feature-oriented languages.

The difference between Mixin and Jampack is furthermore not only a matter of tool support because it can affect the program semantics beyond type errors, as illustrated in Figure 6.[6] Which value is returned by method bar? Again, it depends on a compiler flag: using Jampack, bar returns 42; using Mixin, bar returns 23. The return value of method bar depends on the composition mechanism (Mixin or Jampack) and Java's overloading rules. Method foo in class Foo is overloaded, the first variant with a parameter of type A and the access modifier protected (Line 5) and the second variant with a parameter of type B and the access modifier private (Line 6). Calling foo from bar returns either 23 or 42, depending on which variant of foo was selected during method dispatch, which in turn depends on whether we use Mixin or Jampack. Using Jampack, the compiler merges class Foo and its refinement into a single class. In this case, both variants of foo are visible to bar and the second is called

[5] Figure 5 illustrates also how method overriding is implemented with Mixin and Jampack. A discussion of the differences is outside the scope of this paper; more details are presented elsewhere [22, 6].

[6] For brevity, we merged the definitions of A, B, and Foo in a single listing.

Code generated by Mixin

```
1  class StackBase {
2    private LinkedList elements = new LinkedList();
3    public void push(Object element) {
4      elements.addFirst(element);
5    }
6    public Object pop() {
7      return elements.removeFirst();
8    }
9  }
10 class Stack extends StackBase {
11   public void push(Object item) {
12     super.push(item); trace();
13   }
14   public Object pop() {
15     Object res = super.pop(); trace(); return res;
16   }
17   private void trace() {
18     for (int i = 0; i < elements.size(); i++)
19       System.out.print(elements.get(i).toString() + " ");
20   }
21 }
```

Code generated by Jampack

```
1  class Stack {
2    private LinkedList elements = new LinkedList();
3    public void push(Object element) {
4      elements.addFirst(element); trace();
5    }
6    public Object pop() {
7      Object res = elements.removeFirst(); trace(); return res;
8    }
9    private void trace() {
10     for (int i = 0; i < elements.size(); i++)
11       System.out.print(elements.get(i).toString() + " ");
12   }
13 }
```

Figure 5: Code generated by Mixin (top) vs. code generated by Jampack (bottom).

(returning 42) based on Java's overloading rules (its parameter type is more specific with respect to the argument supplied by bar). Using Mixin, a subclass is created for the refinement of class Foo, such that bar cannot access the second variant of foo and has to resort to the first variant (returning 23).

In Table 1, we compare different feature-oriented languages with respect to their semantic rules for accessing fields from a refinement and the program behavior with respect to our example of Figure 6. We argue that the differences between the individual feature-oriented languages are not intended but stem solely from the fact that research on FOP has not considered access modifiers so far.

We hope that the examples make clear that we need well-defined semantics of feature-oriented languages including access modifiers as well as a scientific discussion that motivates the choices of the semantics definition. We argue that internal implementation details of compilers or the use of debugging and optimization flags should not decide arbitrarily program semantics.

	Jak[1] *(Mixin)*	*Jak*[1] *(Jampack)*	*FeatureHouse*[2]	*FeatureC++*[3]	*Classbox/J*[4]	*CaesarJ*[5]	*OT/J*[6]
private	×	✓	✓	✓	✓	×	✓
package	✓	✓	✓	—	✓	—	✓
protected	✓	✓	✓	✓	✓	✓	✓
public	✓	✓	✓	✓	✓	✓	✓
bar() (Figure 6)	23	42	42	42	42	23	42

[1] http://www.cs.utexas.edu/~schwartz/ATS.html
[2] http://www.fosd.de/fh/
[3] http://www.fosd.de/fcc/
[4] http://scg.unibe.ch/research/classboxes/
[5] http://caesarj.org/
[6] http://www.objectteams.org/

Table 1: Comparison of different feature-oriented languages with regard to which members of a class can be accessed by a refinement (× access prohibited; ✓ access granted; — not supported). The bottom row shows the results of calling method bar of Figure 6 in different feature-oriented languages.

Feature Base

```
1  layer Base;
2  class A {}
3  class B extends A {}
4  class Foo {
5    protected int foo(A a) { return 23; }
6    private int foo(B b) { return 42; }
7  }
```

Feature Ext

```
8  layer Ext;
9  refines class Foo {
10   public int bar() { return foo(new B()); }
11 }
```

Figure 6: Abstract example that illustrates the difference between Jampack and Mixin compiler flag. Method bar returns 23 using Jampack and 42 using Mixin.

Limited Expressiveness

We have also observed that object-oriented modifiers are not expressive enough for feature-oriented mechanisms (see page 4), as illustrated by the following example. Suppose we refine our class Stack such that accessing the stack's methods is thread-safe (feature Sync). A refinement of feature Sync shown in Figure 7 adds a new field lock and overrides the methods push and pop to synchronize access via the methods lock and unlock. Furthermore, suppose that feature Sync also refines many other classes to attain thread safety (e.g., Queue, Map, and Set) and that a central registry keeps track of all locks in use. To grant the lock registry access to the locks of the synchronized stack (queue, map, set, etc.) objects, we have to change the access modifier in Line 3 from private to public (similarly for the other synchronized classes). However, this also means that *every* class of the *entire* program has access to the lock (not only the lock registry), which is certainly not desired. Other modifiers such as package and protected are also not sufficient for similar reasons, and none of the languages compared in Table 1 provide proper support. Instead, we envision a modifier that states that all roles of a given feature may access a member within the same feature. In our case, we would like to grant access to the locks from the lock registry, which is introduced in the same feature as the locks, but not from other classes of other features. The synchronization example illustrates that the access modifiers available in contemporary feature-oriented languages are not sufficient for fine-grained, feature-based access control.

Summary

Our discussion shows that we need access modifiers that are specific to the needs of FOP. Programmers would like to restrict access to a program element to only certain features. Furthermore, we would like to define how the feature-oriented modifiers interplay with the object-oriented modifiers to avoid inadvertent interactions. To this end, in the next section, we define an access modifier model for feature-oriented languages.

Feature Sync

```
1  layer Sync;
2  refines class Stack {
3    private Lock lock = new Lock();
4    public void push(Object item) {
5      lock.lock();
6      Super.push(item);
7      lock.unlock();
8    }
9    public Object pop() {
10     lock.lock();
11     Object res = Super.pop();
12     lock.unlock(); return res;
13   }
14 }
```

Figure 7: A refinement of class Stack to synchronize accesses to a stack instance.

4. An Orthogonal Access Modifier Model

We explore the design space of possible and potentially useful modifiers for feature-oriented language mechanisms. First, we introduce three feature-oriented modifiers and, second, we explain how they can be combined with the modifiers commonly found in object-oriented languages.

4.1. Feature-Oriented Modifiers

We introduce three modifiers that control the access to members of roles. The motivation for the modifiers comes directly from the fact that features cut across the underlying object-oriented design (see Figure 3).

Modifier feature

The modifier feature is motivated by our previous example, in which we added synchronization support to a stack and other data structures. There, we had the problem that object-oriented modifiers are not able to express that only elements introduced by the synchronization feature—possibly of different classes and packages—may access the locks of the synchronized objects. The modifier feature grants exactly this access and forbids the access from other features, as we illustrate for our stack example in Figure 8. Modifying a member with feature allows every other role of the same feature to access the member in question, in our example, including the lock registry (not shown), and prohibits access from the roles of all other features.

Feature Sync

```
1  layer Sync;
2  refines class Stack {
3    feature Lock lock = new Lock();
4    ...
5  }
```

Figure 8: Using modifier feature to grant access to field lock from all members of feature Sync.

Modifier subsequent

Modifier subsequent is motivated by FOP approaches that treat features as stepwise refinements. That is, starting from a base program, features refine the existing program code gradually and produce a new version in each step [2, 23]—some researchers even draw a connection to functions that map programs to programs [2, 23, 24]. In the stepwise refinement scenario, a feature (represented by a function) must not "know" about program elements that are introduced by subsequent features. The positive effect of such a disciplined programming style is that features can be composed incrementally [23].

In Figure 9, we show a feature StreamLine that refines class Stack such that it writes the stack's state to a stream after each operation (details omitted). Further suppose that another feature ByteOrder, shown also in Figure 9, allows a client to switch the byte order of the stream. If we would like to be able to compose the stack's features incrementally, we have to compose feature StreamLine before feature ByteOrder because the latter refers to the former. To guarantee incremental compositionality, we propose a modifier subsequent that grants access to a program element from all elements of the same feature or of features added subsequently. Features that have been added previously cannot access the program element in question. So, for our example, we modify field objectStream with subsequent (Figure 9, Line 3) and thus require that elements that access this field are added subsequently.

Feature StreamLine

```
1  layer StreamLine;
2  refines class Stack {
3    subsequent ObjectStream objectStream = ...
4    public void push(Object item) {
5      Super.push(item);
6      objectStream.write(this);
7    }
8    ...
9  }
```

Feature ByteOrder

```
10 layer ByteOrder;
11 refines class Stack {
12   public void littleEndian() {
13     objectStream.setByteOrder(LITTLE_ENDIAN);
14   }
15   ...
16 }
```

Figure 9: Using modifier subsequent to prohibit access to field objectStream from all features applied before feature StreamLine; ByteOrder may access objectStream only if it is composed after StreamLine.

Another use case for modifier subsequent is in languages that support a pattern-based selection of extension points such as advice and implicit invocation [25, 26], which have been discussed recently in the context of FOP [23, 4]. With pattern-based selection, we can extend, for instance, all methods whose name begins with set. Modifier subsequent can be used to prevent methods from

being extended by previous features. It has been shown that, in this way, programmers can avoid certain kinds of inadvertent interactions [23, 26].

Modifier program

Modifier program broadens the scope of accessing a program element to all other program elements of all features of the program (previous and subsequent). This is equivalent to the current situation in feature-oriented languages, in which programmers have no fine-grained access control regarding to feature-oriented abstractions. Our motivation to add modifier program is to achieve backward compatibility with programs written without feature-oriented modifiers. If no feature-oriented modifier is specified, we use program as the default modifier (like package in Java).

Discussion

A question that arises is whether the new modifiers are expressive enough or whether we need an even more fine-grained access control mechanism. The smallest modularization unit in feature-oriented design is the feature. With our three feature-oriented modifiers, we are able to precisely control the access on a per-feature basis: individual, subsequent, or all features. So there is no need for a more fine-grained or more coarse-grained access.

It would be possible to grant access only to a special feature or a subset of features by name. We did not consider this possibility *so far* because of three reasons. First, in our previous work on feature orientation, we did not encounter situations in which such a mechanism would have been useful. Second, granting the access to special features would be a departure from the object-oriented modifier model. For example, in Java, a programmer cannot declare that a member is visible only for certain classes or packages, so we do not support this mechanism at the level of features. Third, we would like to minimize the coupling between the features' implementation and their mutual relationships. Apart from the layer declaration (which is even optional) at the beginning of each Jak file, there is no information about the actual feature in the program text. Instead, in languages such as Jak, the link between feature and program text is implicit and managed externally by the tool infrastructure. We believe that this separation of concerns (feature implementation vs. feature management) is one of the reasons of the success of contemporary feature-oriented languages and tools [3].

4.2. Object-Oriented and Feature-Oriented Modifiers in Concert

The three feature-oriented access modifiers interact with common object-oriented modifiers in different ways. In Table 2, we illustrate the interplay between object-oriented and feature-oriented modifiers with respect to the sample feature-oriented design of Figure 3. For each combination of an object-oriented modifier and a feature-oriented modifier, the table shows the roles that may access the members of role A_2 in our sample design. That is, each cell of Table 2 contains the roles that are allowed to access role A_2's members, which have the

A_2	feature	subsequent	program
private	A_2	A_2, A_3	A_1, A_2, A_3
package	A_2, B_2, C_2	A_2, A_3, B_2, B_3, C_2, C_3	A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3
protected	A_2, B_2, C_2, D_2	A_2, A_3, B_2, B_3, C_2, C_3, D_2, D_3	A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3, D_1, D_2, D_3
public	A_2, B_2, C_2, D_2, E_2	A_2, A_3, B_2, B_3, C_2, C_3, D_2, D_3, E_2, E_3	A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3, D_1, D_2, D_3, E_1, E_2, E_3

Table 2: Overview of the roles that are allowed to access a member that has been introduced in role A_2 (according to the design shown in Figure 3).

combined modifiers corresponding to the cell's column and row. For example, a member of role A_2 with the modifiers package and feature can be accessed by the roles A_2, B_2, and C_2 (first column, second row) because they are located in the same package and feature; a member of role A_2 modified with private and program can be accessed by the roles A_1, A_2, and A_3 (third column, first row) because they are of the same class.

Looking closer at Table 2, it is interesting to observe that the individual modifier combinations constitute a lattice with scope inclusion as partial order, 'public program' as top element, and 'private feature' as bottom element, as illustrated in Figure 10. The lattice guided us in the development of a corresponding type system (see Section 5), which is concerned with the question whether a member is in the scope of an access. For example, when accessing a field of an unrelated class (i.e., not from a subclass) introduced in another package of the same feature, the field must be declared at least public feature (public subsequent and public program would be sufficient, too).

Finally, since all feature-oriented languages of Table 1 rely on collaboration-based design, they can be seamlessly enhanced by our modifier model.

5. Implementation and Evaluation

To evaluate and experiment with our model, we have implemented a compiler that incorporates our access modifier model and we have conducted an empirical study on ten feature-oriented programs exploring the potential of feature-oriented modifiers in FOP.

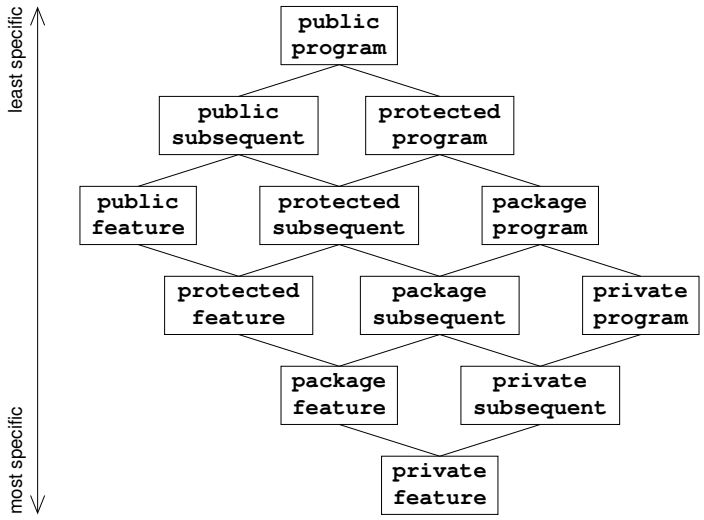

Figure 10: A lattice formed by modifier combinations.

5.1. FUJI

To evaluate and experiment with our model, we implemented a compiler for a feature-oriented language based on Java called FUJI.[7] FUJI's syntax is based on the syntax of FeatureHouse and Jak.[8] We implemented the FUJI compiler as an extension of the Java compiler JastAddJ [27], which in turn fully implements the Java 5 language specification. Specifically, we made three extensions. First, we extended the grammar of JastAddJ to include the new keywords (feature, subsequent, and program). Second, we extended the frontend of JastAddJ with support for features and feature composition (the FUJI compiler expects the input features in separate directories). Third, we extended the type system of JastAddJ to implement our access modifier model; the extended type system ensures that member accesses respect the rules defined by our model, as explained in the previous sections. The powerful extension mechanism of JastAddJ (which is based on aspect weaving) allowed us to implement all extensions modularly. The compiler with examples and the results of our analysis is available on the Web.[9]

For evaluation, we implemented a systematic test application following the example shown in Figure 3 (available at FUJI's website). Role A_2 declares twelve fields, each of which is modified by a unique combination of an object-oriented

[7] Feature-Oriented Java Compiler

[8] The reason for implementing a new compiler, rather than extending FeatureHouse or Jak, is that FeatureHouse and Jak do not provide a type system, which is necessary to implement access control mechanisms.

[9] http://www.fosd.de/fuji/

and a feature-oriented modifier. Each role, including A_2, accesses all of the fields of role A_2.[10] The FUJI compiler reports errors for all accesses that do not conform with the specification in Table 2. As the test application covers all situations shown in Table 2, we are confident that the compiler implements our specification correctly and completely, although this cannot be a definitive proof.

5.2. Data of Ten Feature-Oriented Programs

Beside technical correctness and completeness, we evaluate the practicality of our model. Rather than implementing one or two feature-oriented programs from scratch using the novel modifiers, we aim at a more realistic and less biased evaluation. We are interested in the question of whether there is a need for feature-oriented modifiers in practical FOP. The approach we have taken to answer this question is to analyze the members and accesses in current (third-party) feature-oriented programs and to explore whether object-oriented modifiers alone are sufficient or whether additional feature-oriented modifiers are necessary.

To determine whether a declared object-oriented modifier is sufficient for a given member, we analyze all accesses to the member and determine the most-specific combination of an object-oriented and a feature-oriented modifier with which all accesses of the member are valid (i.e., the combination that is bottommost in the lattice shown in Figure 10). If this combination is more specific than the declared object-oriented modifier,[11] this *may* indicate that the programmer was not able to define the scope more precisely (see Section 5.3 for a discussion). For example, if we encounter a field which is declared `public` but all accesses stem from subclasses introduced by the same feature, then we know that the most-specific scope is `protected feature` and the scope chosen by the developer is less specific.

In our evaluation, we analyzed ten feature-oriented programs including several non-trivial programs developed by others (see Table 3). We extended the FUJI compiler to collect for each member the following information:

- The *declared modifier* of the member (an object-oriented modifier).
- A list of *access scopes*, each of which corresponds to an access of the member and which is most-specific such that the corresponding access is still valid (a pair of object-oriented and feature-oriented modifiers).
- The *minimal modifier* of the member, with which all accesses to the member are valid (a pair of object-oriented and feature-oriented modifiers). The minimal modifier is the least upper bound of all access scopes in the access modifier lattice.

[10] To be specific, roles of the classes A, B, and D access the fields of A_2 via `this`; roles of C and E access the fields of A_2 via `new p1.A()`.

[11] The absence of feature-oriented modifiers in the analyzed programs is equivalent to uses of the modifier `program`.

program name	# lines of code	# feature modules	description
Berkeley DB[1]	45 000	100	Oracle's transactional storage engine
EPL[2]	149	11	arithmetic expression evaluator
GPL[3]	1 929	28	graph and algorithm library
GUIDSL[4]	11 527	29	graphical configuration tool
MobileMedia[5]	4 227	47	multimedia management for phones
Notepad[6]	1 751	13	graphical text editor
PKJab[7]	3 305	8	instant messaging client for Jabber
Prevayler[8]	5 270	6	persistence library
TankWar[9]	4,845	38	shoot 'em up game
Violet[10]	7 151	88	graphical model editor

[1] refactored by C. Kästner (U Magdeburg)
[2] developed by R. Lopez-Herrejon (UT Austin)
[3] developed by R. Lopez-Herrejon (UT Austin)
[4] developed by D. Batory (UT Austin)
[5] refactored by C. Kästner (U Magdeburg)
[6] developed by A. Quark (UT Austin)
[7] developed by P. Wendler (U Passau)
[8] refactored by J. Liu (UT Austin)
[9] developed by L. Lei et al. (U Magdeburg)
[10] refactored by A. Kampasi (UT Austin)

Table 3: Overview of the analyzed programs.

For example, consider a class A with a public field foo, shown in Figure 11 (Line 3), which is accessed once from inside the same role (Line 5) and twice from another role of the same class but in a subsequent feature (Lines 11 and 13). In this example, the derived access scopes are: private feature (Line 5), private subsequent (Line 11), and private subsequent (Line 13). The derived minimal modifier (with which all three accesses are valid) is private subsequent, so the corresponding member could be redeclared safely using this modifier.

Feature Base

```
1  layer Base;
2  class A {
3    public boolean foo = false;
4    private void bar() {
5      foo = true;
6    }
7  }
```

Feature Ext

```
8  layer Ext;
9  refines class A {
10   private void foobar() {
11     foo = true;
12     ...
13     foo = false;
14   }
15 }
```

Figure 11: A class with a public field foo and a method bar that accesses foo (feature Base), and a refinement with a method foobar that accesses foo, as well (feature Ext).

In Table 4, we show for each of the ten programs the overall number of members and accesses and the occurrences of declared modifiers, derived access scopes, and derived minimal modifiers. The raw data are available in the electronic appendix and on Fuji's website (along with the analyzed programs).

5.3. Analysis and Discussion

We analyze and discuss the data with regard to two questions: First, which declared modifiers, derived access scopes, and derived minimal modifiers occur most frequently? Second, why did developers not declare more-specific modifiers in the case this would have been possible?

Access Scopes and Minimal Modifiers

From the data presented in Table 4, we infer that $86.69 \pm 13.59\,\%$ of all member accesses are from within the same feature as the corresponding members (only $10.08 \pm 10.49\,\%$ are from subsequent features and $3.23 \pm 3.5\,\%$ are from previous features).[12] Furthermore, $76.43 \pm 20.42\,\%$ of the derived minimal modifiers limit the scope of a member access to the same feature (only $14.76 \pm 13.32\,\%$ limit the scope to subsequent features and $8.81 \pm 8.09\,\%$ to the entire program). These results show that most accesses occur within individual features, rather than between different features, and, in many cases, the scope of the corresponding members can be limited accordingly. So, there is a potential for information hiding at the level of features, which cannot be expressed properly by object-oriented modifiers.

Actually, we were surprised that we could not find more features that access members of previous features, which we expected to be the natural programming style in stepwise refinement (i.e., subsequent features build on previous features) [2]. In the ten programs, features mainly access their own program elements (87 %)—that is, they are very cohesive—rather than referring to elements previously introduced by other features (13 %). So, our results support the philosophy of viewing features as cohesive units that should be developed independently [7, 28, 29], rather than viewing features as transformations that have access to all program elements [30, 31].

Given that not all feature-oriented modifiers appear to be of equal usefulness, the question arises whether we really need all of them. Modifier `feature` appears to be more useful in the analyzed programs than `subsequent` and `program`. However, we argue that it is too early to answer this question. So, we leave the modifiers in the model and the implementation, so that time will show which modifiers are used in practice.

Specificity of Modifiers

Based on the collected data, we computed for each program the fraction of the minimal modifiers that are more specific than their declared modifiers. In Table 5, we show the fractions for the analyzed programs. On average, $91.19 \pm 8.09\,\%$ of the minimal modifiers are more specific than the corresponding declared modifiers. This fraction raises the question of why developers did not declare more-specific modifiers. We can think of three possible reasons:

[12] We write $a \pm s$ denoting the average value a and the standard deviation s.

		Berkeley DB	*EPL*	*GPL*	*GUIDSL*	*MobileMedia*	*Notepad*	*PKJab*	*Prevayler*	*TankWar*	*Violet*
# members		3 257	9	117	2 301	250	86	534	731	539	335
# accesses		9 978	20	829	114 435	700	923	13 524	13 516	9 120	4 733
# declared modifiers	private	1 282	0	15	433	54	26	241	450	44	140
	package	271	8	4	628	44	9	24	23	37	21
	protected	1 305	0	2	89	120	15	72	67	284	26
	public	399	1	96	1 151	32	36	197	191	174	148
# access scopes	private feature	9 977	17	500	86 391	700	576	10 667	11 110	5 658	3 951
	private subsequent	0	0	81	5 246	0	177	193	298	476	220
	private program	0	0	17	635	0	44	63	125	98	4
	package feature	1	3	72	8 238	0	55	1 040	713	560	325
	package subsequent	0	0	132	8 680	0	55	118	53	1 483	192
	package program	0	0	27	5 245	0	16	81	28	845	25
	protected feature	0	0	0	0	0	0	17	9	0	0
	protected subsequent	0	0	0	0	0	0	0	0	0	0
	protected program	0	0	0	0	0	0	0	0	0	0
	public feature	0	0	0	0	0	0	916	946	0	16
	public subsequent	0	0	0	0	0	0	205	165	0	0
	public program	0	0	0	0	0	0	224	69	0	0
# minimal modifiers	private feature	3 256	8	45	1 280	250	48	307	497	217	202
	private subsequent	0	0	20	92	0	3	15	16	67	20
	private program	0	0	5	22	0	8	4	13	17	1
	package feature	1	1	7	93	0	11	42	64	63	48
	package subsequent	0	0	26	367	0	11	23	12	95	49
	package program	0	0	14	447	0	5	20	2	80	13
	protected feature	0	0	0	0	0	0	1	1	0	0
	protected subsequent	0	0	0	0	0	0	0	0	0	0
	protected program	0	0	0	0	0	0	0	0	0	0
	public feature	0	0	0	0	0	0	61	77	0	2
	public subsequent	0	0	0	0	0	0	32	33	0	0
	public program	0	0	0	0	0	0	29	16	0	0

Table 4: Numbers of members, accesses, declared modifiers, derived access scopes, and derived minimal modifiers of ten programs.

	Berkeley DB	*EPL*	*GPL*	*GUIDSL*	*MobileMedia*	*Notepad*	*PKJab*	*Prevayler*	*TankWar*	*Violet*
# more-specific	3 257	9	98	1 832	250	73	481	700	442	321
# declared	3 257	9	117	2 301	250	86	534	731	539	335
%	100.00	100.00	83.76	79.62	100.00	84.88	90.07	95.76	82.00	95.82

Table 5: The fractions of minimal modifiers that are more specific than their corresponding declared modifiers.

1. **Expressiveness**: the developer was not able to express the scope more precisely due to the absence of feature-oriented modifiers.
2. **Extensibility**: the developer chose the scope deliberately unspecific to facilitate extensibility [32].
3. **Code smell**: the developer simply did not care about access control.

The first reason for declaring less-specific modifiers (expressiveness) is supported by the fact that most accesses occur within individual features, rather than between different features—a situation that cannot be expressed properly with object-oriented modifiers. So there is an untapped potential for feature-oriented modifiers to improve information hiding at the level of features.

The second reason for declaring less-specific modifiers (extensibility) is motivated by software design. If it is known that a program is going to be extended, developers should prepare the program accordingly. In object-oriented programming, this means that developers sometimes declare less-specific modifiers than possible (e.g., `protected` instead of `private`) to enable subsequent extension. However, even when looking only at the cases in which the declared object-oriented modifiers are as specific as possible (and dismissing all cases in which the declared object-oriented modifiers are less specific than possible as intentionally loosened for extensibility), feature-oriented modifiers can reduce the scope of members in $97 \pm 4.33\,\%$ of the considered cases ($37 \pm 23.27\,\%$ of all cases).

The third reason (code smell) is difficult to recognize in the analyzed programs. Hence, we choose a conservative approach and consider programs that do not contain object-oriented modifiers (or only very few modifiers) as programs in which the developers simply did not care about access control. Fortunately, only EPL falls in this category (the author of EPL confirmed our suspicion). This is not too surprising because it is rather small and designed to illustrate a fundamental problem of object-oriented programming rather than being a fully-fledged program in its own right. Since EPL is small compared to the other analyzed programs, we can neglect its effect on our conclusions.

To summarize, even when considering reasons two (extensibility) and three (code smell), most declared modifiers are too unspecific and can be specialized by feature-oriented modifiers.

5.4. Threats to Validity

There are some threats to construct, internal, and external validity.

Construct Validity

To determine the potential of feature-oriented modifiers, we counted the number of all declared modifiers and related them to their minimal modifiers. A minimal modifier is calculated by comparing the corresponding declared modifier with its list of access scopes. To this end, we process and analyze information on references and types provided by the FUJI compiler. That is, the construct validity of our study relies on the correctness of the FUJI compiler. Although a correctness proof is certainly outside the scope of the article, our test application (following the design of Figure 3) makes us confident that our data are reliable.

Internal Validity

A key idea of our study is to measure the use of object-oriented modifiers and to draw conclusions about the need for feature-oriented modifiers. In Section 5.3, we already discussed the soundness of the conclusions one can draw from the fact that most object-oriented modifiers are less-specific than possible. We discussed the likeliness of three possible reasons (expressiveness, extensibility, and code smell) because we lack data to identify the reasons definitively. Hence, a threat to internal validity emerges from the soundness of our arguments. As said in Section 5.3, there are good reasons to assume that object-oriented modifiers are too coarse-grained to control access in feature-oriented design (91 % is a quite large fraction) but, at the end, missing data pose a threat to validity.

External Validity

FUJI is a feature-oriented extension of the Java programming language. Other feature-oriented languages are similar, but may differ in their concrete syntax and may also support other language constructs (e.g., virtual classes). A threat to external validity emerges from that fact that the results of our study may be specific to FUJI and may differ significantly for other feature-oriented languages. Fortunately, our access modifier model relies only on the principles of collaboration-based design, which abstracts from specific details of the language. So we are confident that we can generalize our conclusions to other languages that are based on collaboration-based design. FeatureC++ is the only exception because it is based on C++, and the modifier model of C++ has a different semantics.

Another issue is to what extent the external validity of our study relies on the selection of sample programs. Can we generalize to other programs and application domains? To increase external validity, we collected as many feature-oriented programs as possible, deliberately excluding our own projects and artificial examples. Although a larger sample size would increase the external validity—ideally including industrial case studies, we argue that the selected programs represent the state of the art in FOP because they are of substantial

size, of different domains, and have been developed by others for different purposes. Furthermore, there are not many more studies on this area—a situation that has to be changed in the future. Note that the fourth author was involved in refactoring two of the sample programs (not in the initial development), but this was well before we began investigating access control.

6. Conclusion

The notion of access control has not gained much attention in feature-oriented language design, which has led to a suboptimal modularity and expressiveness and to unintuitive semantics and inadvertent errors in feature-oriented programs. Based on our experience with contemporary feature-oriented languages, we proposed three modifiers specifically targeting feature-oriented language mechanisms. We developed an access modifier model that seamlessly integrates object-oriented and feature-oriented modifiers. The model can serve as a reference for compilers to avoid inadvertent program behavior and type errors and to provide expressive means to control access in the face of feature-oriented abstractions.

We provide an implementation of the model based on a fully-fledged feature-oriented compiler and found evidence that feature-oriented modifiers can improve the situation in practical FOP. An analysis of ten non-trivial feature-oriented programs revealed that common object-oriented modifiers are not able to define the scope of member accesses sufficiently and therefore give away a potential for information hiding. On average, $91.19 \pm 8.09\,\%$ of all declared modifiers can be specialized with feature-oriented modifiers. Even when considering extensibility issues and code smells, the essence of this result remains the same.

A further interesting observation is that features are mainly self-referential and thus are very cohesive. This observation supports the philosophy of viewing features as cohesive units that should be developed independently, rather than viewing features as transformations that have access to all program elements.

In further work, we shall provide further evidence on the practicality and soundness of our access modifier model. This includes analyzing further programs with regard to the need of feature-oriented modifiers, using feature-oriented modifiers systematically in developing feature-oriented programs from scratch, and interviewing developers about their rationales in using object-oriented modifiers in existing projects and their expectations of and experiences with feature-oriented modifiers.

Finally, access control is an important ingredient for feature modularity. Limiting the scope to certain features aids modular type checking and verification in that the compiler can guarantee that certain members cannot be accessed from outside a feature or set of features. Our model and implementation is a step in this direction. However, further ingredients are necessary for feature modularity (e.g., modular type checking and linking [8]), which are outside the scope of this paper and shall be addressed in further work.

Acknowledgments

This work is being supported in part by the German Research Foundation (DFG), project number AP 206/2-1 and by the Metop Research Center.

References

[1] C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 1241 of LNCS, Springer-Verlag, 1997, pp. 419–443.

[2] D. Batory, J. Sarvela, A. Rauschmayer, Scaling Step-Wise Refinement, IEEE Transactions on Software Engineering (TSE) 30 (6) (2004) 355–371.

[3] S. Apel, C. Kästner, An Overview of Feature-Oriented Software Development, Journal of Object Technology (JOT) 8 (5) (2009) 49–84.

[4] S. Apel, T. Leich, G. Saake, Aspectual Feature Modules, IEEE Transactions on Software Engineering (TSE) 34 (2) (2008) 162–180.

[5] S. Apel, T. Leich, M. Rosenmüller, G. Saake, FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), Vol. 3676 of LNCS, Springer-Verlag, 2005, pp. 125–140.

[6] S. Apel, C. Kästner, C. Lengauer, FeatureHouse: Language-Independent, Automated Software Composition, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE Computer Society, 2009, pp. 221–231.

[7] R. Lopez-Herrejon, D. Batory, W. Cook, Evaluating Support for Features in Advanced Modularization Technologies, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 3586 of LNCS, Springer-Verlag, 2005, pp. 169–194.

[8] D. Hutchins, Pure Subtype Systems: A Type Theory For Extensible Software, Ph.D. thesis, School of Informatics, University of Edinburgh (2009).

[9] S. Apel, J. Liebig, C. Kästner, M. Kuhlemann, T. Leich, An Orthogonal Access Modifier Model for Feature-Oriented Programming, in: Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD), ACM Press, 2009, pp. 26–32.

[10] F. Anfurrutia, O. Díaz, S. Trujillo, On Refining XML Artifacts, in: Proceedings of International Conference on Web Engineering (ICWE), Vol. 4607 of LNCS, Springer-Verlag, 2007, pp. 473–478.

[11] S. Apel, C. Kästner, A. Größlinger, C. Lengauer, Feature (De)composition in Functional Programming, in: Proceedings of the International Conference on Software Composition (SC), Vol. 5634 of LNCS, Springer-Verlag, 2009, pp. 9–26.

[12] A. Bergel, S. Ducasse, O. Nierstrasz, Classbox/J: Controlling the Scope of Change in Java, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2005, pp. 177–189.

[13] I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, An Overview of CaesarJ, Transactions on Aspect-Oriented Software Development (TAOSD) 1 (1) (2006) 135–173.

[14] C. Hundt, K. Mehner, C. Pfeiffer, D. Sokenou, Improving Alignment of Crosscutting Features with Code in Product Line Engineering, Journal of Object Technology (JOT) – Special Issue: TOOLS EUROPE 2007 6 (9) (2007) 417–436.

[15] T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, P. Stenslet, OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems, Journal of Object-Oriented Programming (JOOP) 5 (6) (1992) 27–41.

[16] M. VanHilst, D. Notkin, Using Role Components in Implement Collaboration-based Designs, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 1996, pp. 359–369.

[17] 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) 215–255.

[18] S. Apel, C. Kästner, C. Lengauer, Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2008, pp. 101–112.

[19] B. Delaware, W. Cook, D. Batory, Fitting the Pieces Together: A Machine-Checked Model of Safe Composition, in: Proceedings of the International Symposium on Foundations of Software Engineering (FSE), ACM Press, 2009, pp. 243–252.

[20] S. Apel, D. Hutchins, A Calculus for Uniform Feature Composition, ACM Transactions on Programming Languages and Systems (TOPLAS) 32 (5) (2010) Article 19.

[21] S. Apel, C. Kästner, A. Größlinger, C. Lengauer, Type Safety for Feature-Oriented Product Lines, Automated Software Engineering—An International Journal 17 (3) (2010) 251–300.

[22] M. Kuhlemann, S. Apel, T. Leich, Streamlining Feature-Oriented Designs, in: Proceedings of the International Symposium on Software Composition (SC), Vol. 4829 of LNCS, Springer-Verlag, 2007, pp. 168–175.

[23] R. Lopez-Herrejon, D. Batory, C. Lengauer, A Disciplined Approach to Aspect Composition, in: Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM Press, 2006, pp. 68–77.

[24] D. Batory, Program Refactoring, Program Synthesis, and Model-Driven Development, in: Proceedings of the International Conference on Compiler Construction (CC), Vol. 4420 of LNCS, Springer-Verlag, 2007, pp. 156–171.

[25] F. Steimann, T. Pawlitzki, S. Apel, C. Kästner, Types and Modularity for Implicit Invocation with Implicit Announcement, ACM Transactions on Software Engineering and Methodology (TOSEM) 20 (1) (2010). Accepted for publication.

[26] S. Apel, C. Kästner, D. Batory, Program Refactoring using Functional Aspects, in: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), ACM Press, 2008, pp. 161–170.

[27] T. Ekman, G. Hedin, The JastAdd Extensible Java Compiler, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2007, pp. 1–18.

[28] D. Hutchins, Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes, in: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2006, pp. 1–19.

[29] R. Lopez-Herrejon, Understanding Feature Modularity, Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin (2006).

[30] D. Batory, From Implementation to Theory in Product Synthesis, in: Proceedings of the International Symposium on Principles of Programming Languages (POPL), ACM Press, 2007, pp. 135–136.

[31] P. Ebraert, First-Class Change Objects for Feature-Oriented Programming, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE Computer Society, 2008, pp. 319–322.

[32] P. Bouillon, E. Großkinsky, F. Steimann, Controlling Accessibility in Agile Projects with the Access Modifier Modifier, in: Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE), Vol. 11 of LNBIP, Springer-Verlag, 2008, pp. 41–59.