An Orthogonal Access Modifier Model for Feature-Oriented Programming

Sven Apel and Jörg Liebig
Department of Informatics and Mathematics
University of Passau, Germany
{apel, joliebig}@fim.uni-passau.de

Christian Kästner and Martin Kuhlemann
School of Computer Science
University of Magdeburg, Germany
{kaestner, kuhlemann}@iti.cs.uni-magdeburg.de

Thomas Leich
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 is an important ingredient to attain feature modularity as it provides mechanisms to hide and expose internal details of a module's implementation. But developers of contemporary feature-oriented languages did not consider access control mechanisms so far. The absence of a well-defined access control model for FOP breaks the encapsulation of feature code and leads to unexpected and undefined program behaviors as well as inadvertent type errors, as we will demonstrate. The reason for these problems is that common object-oriented modifiers, typically provided by the base language, are not expressive enough for FOP and interact in subtle ways with feature-oriented language mechanisms. We raise awareness of this problem, propose three feature-oriented modifiers for access control, and present an orthogonal access modifier model.

1. INTRODUCTION

The goal of *feature-oriented programming* (*FOP*) is to modularize software systems in terms of features [19, 11]. A *feature* is a unit of functionality of a program that satisfies a requirement, represents a design decision, and provides a potential configuration option [2]. A *feature module* encapsulates exactly the code that contributes to the implementation of a feature [8]. The goal of the decomposition into feature modules is to construct well-structured software that can be tailored to the needs of the user and the application scenario. Typically, from a set of feature modules, many different programs can be generated that share common features and differ in other features, which is also called a software product line [12, 18].

Many feature-oriented languages aim at feature modularity, e.g, AHEAD/Jak [11], FeatureC++ [7], and FeatureHouse [6]. Feature modules are supposed to hide implementation details and to provide access via interfaces. 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 [16]; they lack sufficient abstraction and modularization mechanisms to support (1) independent development based on information hiding, (2) modular type checking, and (3) separate compilation. In a theoretical work, Hutchins has shown that, in principle, feature-oriented languages should be able to attain this level of feature modularity [14]. However, there are many open issues regarding the implementation on the basis of a mainstream programming language, such as the interaction with other language mechanisms, efficiency, and tool support.

An important ingredient for feature modularity that is missing in contemporary feature-oriented languages is a proper mechanism for access control. Access modifiers allow programmers to define the scope and visibility of their 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. However, there are no specific modifiers tailored to feature-oriented language mechanisms. Well, since a feature-oriented language usually extends an object-oriented language (e.g., Jak extends Java [11] and FeatureC++ extends C++ [7]), the object-oriented access modifiers are (re)used. But it is not possible to grant access, e.g., to a program element for all other program elements from the same feature and to disallow the access for all program elements of other features.

As said before, access control has not been considered so far in research on feature-oriented languages. In some sense access control mechanisms were for free when extending an existing object-oriented language. Of course, the object-oriented modifiers were not intended for the use in FOP, so one can say that they are misused. We contribute an analysis of object-oriented modifiers used in FOP and identify several shortcomings and problems that lead to a limited ex-

pressiveness of feature-oriented languages, unexpected and undefined program behaviors, and inadvertent type errors. We explore the design space of feature-oriented access control mechanisms and propose three concrete access modifiers. Furthermore, we present an orthogonal access modifier model, which integrates common object-oriented modifiers with our novel feature-oriented modifiers.

2. BACKGROUND

Often, a feature-oriented language extends an object-oriented base language by mechanisms for the abstraction and modularization of features.[1] In order to implement the additions and changes a feature makes, feature-oriented languages like Jak introduce a mechanism for class refinement.

In Figure 1, we depict a class `Stack` written in Jak, which is an extension of Java and belongs to the AHEAD tool suite [11]. The class definition is identical to a definition in Java except for the `layer` declaration, 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      if(elements.size() > 0) { return elements.removeFirst(); }
9      else { return null; }
10   }
11 }
```

Figure 1: A basic stack implemented in Jak.

In Figure 2, we depict a refinement of class `Stack`, declared by the keyword `refines`. The refinement is part of feature UNDO, which allows the clients of the stack to revert the last operation. When feature UNDO is composed with feature BASE, the refinement adds a new method `undo` and two new fields `lastPush` and `lastPop` to class `Stack`. Furthermore, it refines the methods `push` and `pop` (by overriding) in order to store the last item added to or removed from the stack. The keyword `Super` is used to invoke the method that has been refined.[2]

Typically, a feature comprises multiple class declarations and class refinements, which implement the feature in concert. We visualize a feature-oriented program design – like the design of our stack example – using a *collaboration diagram* [20, 23, 21]. 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 four classes $A - D$ (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

[1]We are aware that some feature-oriented tools build on languages that are not object-oriented [11, 1, 6]. These languages are outside the scope of the paper, as they do not provide access modifiers like the ones we consider here.

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

Feature UNDO

```
12 layer Undo;
13 refines class Stack {
14   private Object lastPush = null;
15   private Object lastPop = null;
16   public void push(Object item) {
17     lastPush = item; lastPop = null;
18     Super.push(item);
19   }
20   public Object pop() {
21     lastPop = Super.pop();
22     lastPush = null; return lastPop;
23   }
24   public void undo() {
25     if(lastPush != null) { Super.pop(); }
26     else if(lastPop != null) { Super.push(lastPop); }
27   }
28 }
```

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

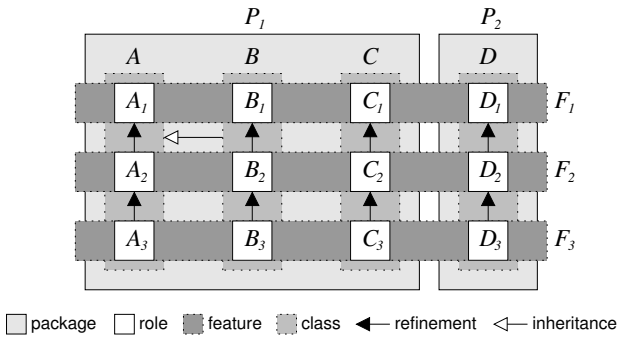

Figure 3: A sample feature-oriented design.

features ($F_1 - F_3$) as 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 [23]; the set of roles belonging to a feature is called a *collaboration* [21] and is encapsulated by a feature module [8]. 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, and D_1. The top most 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) [11]. The solid arrow denotes the refinement relationship between roles and the empty arrow denotes inheritance between full classes.

3. PROBLEM STATEMENT

We explain the problems we encountered with feature-oriented languages by means of the Jak language. Jak, as a Java extension, has inherited the access modifiers of Java. Hence, like in Java, programmers can control the access to classes and members in Jak using the modifiers `private`, `protected`, `package`, and `public`.[3] But there are two problems with this:

[3]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 symmetry with the other modifiers, we introduce modifier `package` for this case.

1. **Undefined semantics**: object-oriented modifiers interact in undefined ways with feature-oriented mechanisms such as class refinements
2. **Limited expressiveness**: object-oriented modifiers are not expressive enough to control the access to elements introduced by features.

Undefined Semantics

Let us illustrate the first problem 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, which refines the methods `push` and `pop`, accesses the list storing the stack's elements, and prints them to the console.

Feature Trace

```
1  layer Trace;
2  refines class Stack {
3    public void push(Object item) {
4      Super.push(item);
5      trace();
6    }
7    public Object pop() {
8      Object res = Super.pop();
9      trace(); return res;
10   }
11   private void trace() {
12     for(int i = 0; i < elements.size(); i++) {
13       System.out.print(elements.get(i).toString() + " ");
14     }
15   }
16 }
```

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

The question is whether the above example is correct. Is it allowed for the class refinement 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 (the behavior is de facto defined by the implementation of the composition engine) and formally specified subsets of feature-oriented languages do not include modifiers [5, 13, 4]. Compiling this code (or similar code) with the Jak compiler reveals that it depends on certain compiler flags whether this code is considered correct.

The background is that the Jak compiler generates Java code in an intermediate step and it supports two options to do so [15]: in the first option, called Mixin, the compiler generates an inheritance hierarchy with one subclass per refinement; 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. Comparing the two options it becomes clear why they show different behaviors in our example, which we illustrate in Figure 5. In the first option, private field `elements` cannot be accessed because the refinement is translated to a subclass, which cannot access private members of superclasses. In the second option, 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 behaviors of a single program depending on a compiler flag that is intended for optimization.

```
1  class StackBase {
2    private LinkedList elements ...
3  }
4  class Stack extends StackBase {
5    ...
6    private void trace() {
7      ... elements.size() ...
8      ... elements.get(i) ...
9    }
10 }
```

```
1  class Stack {
2    private LinkedList elements ...
3    ...
4    private void trace() {
5      ... elements.size() ...
6      ... elements.get(i) ...
7    }
8  }
```

Figure 5: Mixin vs. Jampack.

One can argue for one or the other behavior, and certainly it is possible to fix either Mixin or Jampack such that both obey an equal behavior, but what we would like to stress is that the semantics of access modifiers and their interaction with feature-oriented mechanisms such as class refinements is not well-defined. This fact is not only a matter of tool support since it can affect the program semantics beyond type errors. Have a look at the example shown in Figure 6.[4] Which value is returned by method `bar`? Again, it depends on the composition mechanism: using Jampack, `bar` returns 23; using Mixin, `bar` returns 42. A comprehensive discussion of the reason of difference is outside the scope of the paper and we leave it as "homework" for the reader. A hint is that it depends again on the underlying composition mechanism (Mixin-like or Jampack-like) and that it has to do with Java's overloading mechanism.

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: Which value is returned by method `bar`?

In Table 1, we compare different (variants of) feature-oriented languages with respect to their 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 (variants of) feature-oriented languages are not intended but stem solely from the fact that research on FOP did not consider access modifiers so far. The language developers got modifiers for free from the base language and the implementation of the composition in a preprocessing step decides over the semantics of the composed program.

We hope that the above 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. What we

[4] For brevity we have merged the definitions of the classes `A`, `B`, and `Foo` in a single listing.

	Jak[1] (Mixin)	Jak[1] (Jampack)	FeatureHouse[2]	FeatureC++[3]	Classbox/J[4]	CaesarJ[5]	OT/J[6]
`private`	×	✓	✓	✓	✓	×	✓
`protected`	✓	✓	✓	✓	✓	✓	✓
`package`	✓	✓	✓	—	✓	—	✓
`public`	✓	✓	✓	✓	✓	✓	✓
`bar()` (Fig. 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://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/`
[4] `http://scg.unibe.ch/research/classboxes/`
[5] `http://caesarj.org/`
[6] `http://www.objectteams.org/`

Table 1: Which members of a class can be accessed by a refinement? What is the return value of `bar`? (× access prohibited; ✓ access granted; — not supported)

do not want is that internal implementation details of compilers or the use of compiler flags, which target at optimization [15], decide arbitrarily over the program semantics.

Limited Expressiveness

With regard to the second problem (object-oriented modifiers are not expressive enough for feature-oriented mechanisms), consider the following example. Suppose we refine our class `Stack` such that accessing the stack's methods is thread-safe. The refinement shown in Figure 7 adds a new field `lock` and overrides the methods `push` and `pop` in order to synchronize access via the methods `lock` and `unlock`. Furthermore, suppose that feature Sync also refines many other classes in order to attain thread safety (e.g., `Queue`, `Map`, and `Set`) and that a central registry keeps track of all locks in use. In order to grant the lock registry access to the lock fields of the synchronized stack (queue, map, set, ...) 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 not sufficient as well, which is easy to see and omitted for brevity. 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 are. 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 previous discussion shows that we need access modifiers that are specific to the needs of FOP. Programmers would like to provide access to a program element from certain features. Furthermore, we would like to define how the feature-oriented modifiers interplay with the object-oriented modifiers in order to avoid inadvertent interactions. To this end, in the next section, we define an orthogonal 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

Next, 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

Using the sample feature-oriented design of Figure 3, we explain three possible 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.

Modifier `feature`

The idea for modifier `feature` is motivated by our example, in which we added synchronization support to a stack and other data structures. There we had the problem that with object-oriented modifiers we were not able to express that only elements introduced by the synchronization feature may access the lock fields of the refined classes. 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.

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`

The proposal of modifier `subsequent` is motivated by the fact that some FOP approaches treat features as stepwise refinements. That is, starting from a base program, features gradually refine the existing program code and produce in each step a new version [11, 17]. Some researchers even draw a connection to functions that map programs to programs [10, 11, 17]. In the stepwise refinement scenario, it has been argued that a feature (represented by a function) does never "know" about program elements applied by feature that have been applied subsequently. The positive effect of such a disciplined programming style is that inadvertent interactions cannot occur with program elements that are not known at the development time of a feature [17]. This is especially important for languages that support a pattern-based selection of extension points such advice and implicit invocation [22, 3], which have been discussed recently in the context of FOP [17,8]. In order to support this view, 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.

Modifier `program`

Modifier `program` broadens the scope of access to a member from program elements of all features. This is like the current situation in feature-oriented languages where programmers have no fine-grained access control with regard to feature-related code, except that in our novel proposal the semantics of object-oriented modifiers and their interplay with feature-oriented mechanisms is well-defined, which we explain in Section 4.2.

Discussion

A question that arises is whether the new modifiers are expressive enough or whether we need even a more fine-grained access control mechanism. The smallest modularization unit in feature-oriented designs is the role. With our three feature-oriented modifiers, we are able to precisely control the access of individual roles to the elements of another role. So there is no need for a more fine-grained access. At the other end of the spectrum, it is possible to grant universal access, which is like leaving out feature-oriented access modifiers at all. The modifier `subsequent` is in the middle and motivated by previous work on program design. One can imagine a further modifier `previous`, which would be the inverse of `subsequent`, but we argue that such a modifier is not of practical value. Although it has been observed that there are situations, in which a feature access elements that have been introduced later, this is not the rule [3]. In these situations, a programmer can use modifier `program` because it is certainly not meaningful full to forbid the access from subsequent features.

A_2	`feature`	`subsequent`	`program`
`private`	A_2	A_2, A_3	A_1, A_2, A_3
`protected`	A_2, B_2	A_2, A_3, B_2, B_3	A_1, A_2, A_3, B_1, B_2, B_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
`public`	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

Table 2: Overview of the roles that may access a member that has been introduced in role A_2.

A further possibility would be to grant access only to a special feature or a subset of features. We did not consider this possibility *so far* because we would like to minimize the coupling between feature implementation and feature management. Apart from the `layer` declaration at the beginning of each Jak file, there is no information about the actual features. Instead, the relation between features and code 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 success factors for contemporary feature-oriented languages and tools [2]. But the last word is not spoken on this issue.

Some feature-oriented languages support to modify the access to individual roles, e.g., `public refines class A { ... }`. Using such a modifier in such a position we can subsequently broaden the access to a class. That is, we can make a private class protected or public but not vice versa. Thus, a modifier in such a position does not control the access to program elements of feature-related code, but it overrides an existing object-oriented modifier. This mechanism can also be used to broaden the access to the members of a class.

Finally, it remains open how modifiers like `abstract` and `final` fit into the picture and how they can be combined gainfully with feature-oriented modifiers. We shall address this issue in further work.

4.2 Object-Oriented and Feature-Oriented Modifiers in Concert

We have proposed three feature-oriented access modifiers, which interact with object-oriented modifiers in different ways. In Table 2, we depict the interplay between object-oriented and feature-oriented modifiers with respect to the sample feature-oriented design of Figure 3. For each combination of object-oriented and feature-oriented modifiers, the table shows the roles that may access the members of role A_2 in our sample design of Figure 3. That is, each cell of Table 2 contains the roles that are allowed to access role A_2's members, which have the combined modifiers corresponding to the cell's column and row. For example, a member of role A_2 with the modifiers `protected` and `feature` can be accessed by the roles A_2 and B_2 (first column, second row); 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).

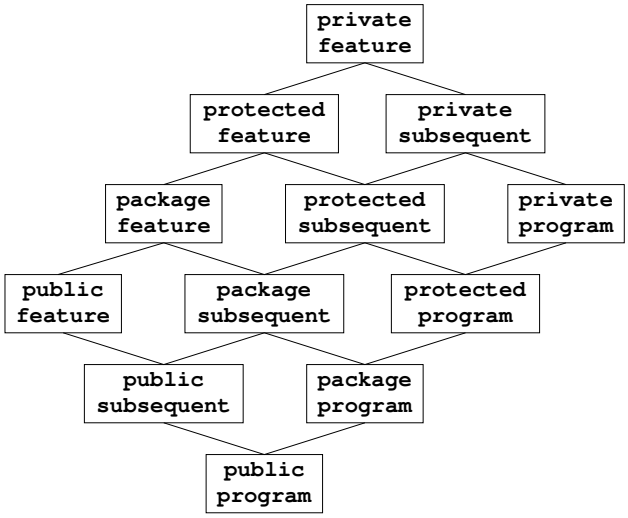

Figure 9: A lattice formed by modifier combinations.

Looking closer at Table 2, it is interesting to observe that the individual modifier combinations constitute a lattice with '`private feature`' as bottom element and '`public program`' as top element, as illustrated in Figure 9. The lattice can guide the formalization and implementation of a corresponding type system, which is concerned with the question whether the scope of the requested access is smaller or larger than the one of the accessed element. When a programmer overrides a member, as in the case of method overriding, the scope of the member's access may stay unchanged, can be extended, but cannot be limited, which means the modifier itself or any modifiers below the original modifier.

5. FORMALIZATION AND IMPLEMENTATION ISSUES

Although Table 2 captures the idea of our modifiers nicely, in further work, a formal definition of the operational semantics and type system of a feature-oriented language that supports these modifiers is desirable. This way, we will be able to define the semantics of our modifiers unambiguously and to guide the implementation of feature-oriented compilers. As a formal system, we will use the Feature Featherweight Java (FFJ) calculus [5], which extends a minimal core of Java with feature-oriented mechanisms. The formalization of the orthogonal access modifier model should be straightforward and we believe that we will be able to prove the soundness of the corresponding type system.

Furthermore, we intend to implement a compiler on the basis of an existing feature-oriented language, preferably Jak or FeatureHouse, which can be used for an empirical evaluation. The problem of current language implementations is that they do not provide a type system that takes the feature-oriented abstractions into account. Merely, feature-oriented code is translated to object-oriented code, and an object-oriented compiler type checks the translated code. Since our feature-oriented modifiers do not have corresponding constructs in the generated object-oriented code, the object-oriented compiler is not able to detect access violations offhand. Hence, we need a feature-oriented compiler with feature-oriented type system. Whereas there are some formalizations of subsets of feature-oriented type systems, there are no fully-fledged compilers that have been developed with feature orientation in mind. Another possibility is to adapt existing compilers of related languages such as CaesarJ [9].

Once we have a feature-oriented compiler, case studies should explore the practicality of feature-oriented modifiers and reveal potential problems but also potential benefits for the mission of attaining real feature modularity.

6. CONCLUSION

Based on our experience with contemporary feature-oriented languages, we have proposed three modifiers targeting specifically at feature-oriented languages mechanisms. Furthermore, we have developed an orthogonal access modifier model that seamlessly integrates object-oriented and feature-oriented modifiers. The background is that the notion of access control has not gained much attention in feature-oriented language design, which leads to a suboptimal modularity and expressiveness and unintuitive semantics and inadvertent errors in feature-oriented programs.

A question that remains is whether the novel modifiers will prove of value in practical software development. Certainly, in order to attain real modularity, further ingredients are necessary (e.g., declarative completeness and modular linking), which are outside the scope of this paper (see the work of Hutchins for details [14]). Also it is not clear whether our names of the modifiers match the intuition of the programmers well. In the case a program element has no modifiers, which modifiers should we assume as default? We intend to initiate a discussion about these and other open issues and inspire further research that evaluates the benefits and drawbacks of our model and its successors.

Furthermore, it is open which further mechanisms are necessary to attain the properties necessary for real modularity (information hiding, modular type checking, and separate compilation) and how they interact with our orthogonal access modifier model.

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.

7. REFERENCES

[1] F. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proceedings of International Conference on Web Engineering (ICWE)*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer-Verlag, 2007.

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

[3] S. Apel, C. Kästner, and D. Batory. Program Refactoring using Functional Aspects. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*,

pages 161–170. ACM Press, 2008.

[4] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type-Safe Feature-Oriented Product Lines. Technical Report MIP-0909, Department of Informatics and Mathematics, University of Passau, 2009.

[5] S. Apel, C. Kästner, and 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)*, pages 101–112. ACM Press, 2008.

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

[7] S. Apel, T. Leich, M. Rosenmüller, and 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)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.

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

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

[10] D. Batory. Program Refactoring, Program Synthesis, and Model-Driven Development. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, 2007.

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

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Proceedings of the International Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. ACM Press, 2009.

[14] D. Hutchins. *Pure Subtype Systems: A Type Theory For Extensible Software*. PhD thesis, School of Informatics, University of Edinburgh, 2008.

[15] M. Kuhlemann, S. Apel, and T. Leich. Streamlining Feature-Oriented Designs. In *Proceedings of the International Symposium on Software Composition (SC)*, volume 4829 of *Lecture Notes in Computer Science*, pages 168–175. Springer-Verlag, 2007.

[16] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer-Verlag, 2005.

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

[18] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

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

[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):215–255, 2002.

[22] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and Modularity for Implicit Invocation with Implicit Announcement. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009.

[23] M. VanHilst and 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)*, pages 359–369. ACM Press, 1996.