Features as First-class Entities – Toward a Better Representation of Features

Sagar Sunkle, Marko Rosenmüller,
Norbert Siegmund,
Syed Saif ur Rahman, Gunter Saake
School of Computer Science
University of Magdeburg, Germany
{sagar.sunkle,rosenmue,norbert.siegmund,
srahman,saake}@iti.cs.uni-
magdeburg.de

Sven Apel
Dept. of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

ABSTRACT

Features are distinguishable characteristics of a system relevant to some stakeholder. A product line is a set of products that differ in terms of features. Features do not have first-class status in contemporary programming languages (PLs). We argue that various problems related to features are a result of this abstraction and representation mismatch and that features should be elevated to a first-class status. We propose an extension to Java that implements features as first-class entities. We give examples of the syntax and semantics of this extension and explain how the new representation can handle features better.

General Terms

Languages, Design

Keywords

Feature-Oriented Programming, JastAdd, Separation of Concerns

1. INTRODUCTION

Separation of concerns is one of the most important principles in software engineering [16]. Abstractions like features and classes are viewed as dimensions in concern space [36]. Separation of concerns means decomposing software into manageable pieces along a dimension in concern space. It consists of *identification*, *encapsulation*, and *integration*. *Identification* means a software is decomposed into entities that represent the abstraction, e.g., classes and features, *encapsulation* means some mechanism is provided so that these entities can be manipulated as a first-class entities [32], *and integration* means that some composition mechanism is provided that integrates concerns represented as first-class entities[36]. The first-class status of an entity in a programming language (PL) indicates the degree to which one can address or manipulate concepts in a given domain arranged along the dimensions of a concern space and the ease with which this is made possible in a given PL [32].

A feature is defined as an end-user-visible characteristic of a system, or a distinguishable characteristic of a concept (system, component, and so on) that is relevant to some stakeholder [14]. A product line contains different products that vary in features. Consequently, features are used to understand the commonalities (shared features) and variabilities (optional or unshared features) between the products of a product line.

Many technologies have been used to implement features [3, 4, 9, 28, 31, 38]. The main kind of concern supported by them is one of functions, classes, aspects, hyperslices, mixins, and frames, etc. Features, which are themselves a kind of concern, are essentially implemented in terms of entities that basically represent some other kind of concern. Instead of thinking only about features, the developer has to organize them in terms of the modular structure of the approach he is using and see to it that the intent of features is precisely represented by the entities in this approach. We take the position that this abstraction and representation mismatch causes problems [26, 29] such as, e.g., hierarchical misalignments, limitations in feature composition and order, and inexpressive program deltas, etc. Our claim is that such problems can be addressed and various possibilities for features can be achieved more easily if features were represented not in terms of other entities, but as first-class entities themselves.

In this position paper, we propose to represent features as first-class entities. We discuss what it means when certain programmatic entities have first-class status in a given PL. We review various feature implementation approaches and enumerate related problems. We argue that these problems arise due to an inadequate representation of features. We put forward an agenda that establishes features as first-class entities. Finally,

we propose an implementation of the extension and discuss how features implemented as first-class entities can be used to address the problems.

2. BACKGROUND

2.1 First-class Entities in PLs

There is no specific definition for first-class status of entities in a given PL. Certain properties have been observed that indicate a first-class status of a given programmatic entity [13, 35]. We deem the following five properties as the defining properties that must be exhibited by entities in a given PL to be called first-class entities.

1. First-class entities can be instantiated at compile-time or run-time and possibly other stages of program execution.
2. First-class entities can be stored in variables and data structures.
3. First-class entities can be statically or dynamically typed, thus allowing compile-time or run-time structural manipulation.
4. First-class entities can be passed as parameters to other program elements such as methods and returned from methods.
5. First-class entities can be part of various expressions and statements in this PL, giving a program developer ample options to represent his intent in representing the problem domain.

Various PLs that claim first-class status for a kind of concern, support different subsets of these properties differing in their semantic treatment. The degree of manipulation of first-class entities may depend on the kind of typing and the kind of composition supported by given PLs. Runtime manipulation of such entities creates new possibilities. In this case, such entities can have identity and be aware of other entities of the same kind. This makes it possible to represent and manipulate interactions among these entities more naturally. Also, such entities can store context and be aware of the state of a program, thus making possible changes at wider range of stages in the program. These two properties are indicative of reflective and meta programming support for the first-class entities. They depend on the reflection and meta programming support of a given PL and may increase the degree of manipulation substantially for the first-class entities.

2.2 Features

In feature-oriented domain analysis (FODA) [22], features are organized in feature diagrams. A feature diagram is a tree with the root representing a concept and its descendant nodes being features. These features can be mandatory, optional, or alternative. Feature-oriented decomposition is a feature modeling activity used to capture commonalities and variabilities in terms of features, of systems in a domain [14]. It is used to model a domain in terms of features from the ground up. Feature-oriented refactoring is the process of decomposing an already existing system to a system exposing features [25].

2.3 Feature Implementations

Features as a programming model was first conceived by Prehofer [33], citing the rationale behind using features to be the flexible composition of objects possible from a set of features. The implementation technique for FODA is broadly referred to as feature-oriented programming [9, 10], but as asserted earlier, there are many ways in which features can be implemented. Kästner et al. [24] distinguish between compositional and annotative approaches. The same distinction can also be applied to various feature implementation approaches. Compositional approaches for implementing features represent features as distinct modules, which are composed at compile time or deployment time or similar. Examples of compositional approaches are mixin layers [5], HyperJ hyperslices [32], and Scala traits [31]. The *ifdef* statements in C, frames in XVCL [38] and color annotations in CIDE [38] are, on the other hand, examples of annotative approaches. Annotative approaches implement features by identifying code belonging to a feature in the source and annotating it, so that variants may be created by including or removing annotated code from the source [24].

The compositional approaches generally allow coarse-grained refinements to programs due to the fact that naming schemes of container entities such as classes and methods are required to be kept invariant as they are used in identifying parts of the program to which refinement must be applied. They are not suitable for fine-grained refinements in which order of statements or expressions added by features needs to be controlled [24]. Fine-grained refinements are possible with the annotative approaches. Annotative approaches allow refinements of arbitrary granularity since the important concerns of compositional approaches like naming schemes and order of composed code do not matter as all code fragments belonging to features are at their final position and only need to be annotated [24].

We propose that, by using a combination of compositional and annotative approaches, we can create a better representation of features. In the following section, we address the problems faced by feature-oriented approaches in general and then state the proposed solution which uses elements from compositional and annotative approaches to tackle these problems.

3. THE PROBLEM

Mezini and Ostermann [29] identified weaknesses of various current feature-oriented approaches in managing the variability in product lines. Similarly, Lopez-Herrejon et al. [26] evaluated support for features in advanced modularization technologies and concluded that despite the crucial importance of features, features are rarely completely modularized. The shortcomings described below are not necessarily present in all the approaches considered, but none of the approaches provides a uniform treatment of the various shortcomings either. The weaknesses of various current feature-oriented approaches identified in [5, 26, 29] follow:

- ***Hierarchical refinements*** – Features are implemented as refinements to base classes. Mezini and Ostermann [29] claim that this is a shortcoming, because the hierarchical modularity of the refinements to the base classes imposes a structure on features which are not in hierarchical relationship to each other. The problem with this is that for a given feature, there may not be a class in one-to-one relation to which this feature may be mapped. For further details refer to [3], [5], and [29]. Similarly, because features are refinements, a feature that is in fact reusable, would need to be encoded separately as a refinement to each class that needs it. This makes reuse of common features hard [29].
- ***Feature composition and feature order*** – Feature modules should be composable in different orders and should follow the commutativity or pseudo-commutativity of features [1]. Feature composition should be closed under composition, which means that features may be grouped to larger features and such a composite feature is valid wherever the constituents features are used [26] as this increases the reuse of features. Different approaches support either or both of these properties. Even in those approaches that support both closed composition and feature order, actually implementing it can be a nontrivial task [26].
- ***Program deltas*** – Various program refinements are deltas with respect to the base program [26]. New classes, interfaces, fields, method statements, and method arguments, etc., are examples of program deltas. Considering features as semantic blocks of code, preferably any statement or expression, or group of statements and expressions in a given programming language can be part of the refinements a feature makes. The order of blocks of code to be inserted into a method for example, cannot be controlled in simple method refinement approaches.
- ***Type support for features*** – Feature modules and composite modules should be well defined via type support. Types for features can be extremely beneficial not only in safe composition of features, but also in controlling interactions among features and also between features and the regular types in a given programming language [23, 37]. But types for features have been treated in isolation, e.g., it is not known how features represented as types will fare in dynamic composition. Other treatments of types for features consider only an extension to a subset of Java [2]. Type checking or similar concepts are difficult to apply to features because it requires some way of identifying and localizing feature code and representing features as types that interact with programming language types. By expressing a type checking mechanism for features as a calculus language that interfaces with a feature description language, type checking may be more clearly applied to an entire programming language [2, 23].
- ***Dynamic composition and separate compilation*** – It should be possible to alter the configuration of features which have already been instantiated [29]. Similarly, it should be possible to bind features dynamically based on conditions related to specific expressions [21] and this must happen considering the performance of application that uses such dynamic reconfiguration of features. Some approaches have been suggested for dynamic composition of features [29, 34], but dynamic composition remains a largely unexplored issue in other feature-oriented approaches. Separate compilation of features is also desirable for better debugging of feature implementation and distribution of byte code [26].

Feature implementations also lack a common ground with feature modeling concepts. In order to use features for creating program variants, some sort of structure has to be imposed on them. Such structure indicates the relationship between features, their grouping into different collections and imposes certain constraints about which choices of features are valid. Though, in current feature-oriented approaches, no programmatic or language level mechanisms are provided for it. Below we summarize the problem and then describe the proposed solution in the next section.

Problem Summary

Feature implementations either weave refinement code based on a naming scheme or employ some sort of redirection or delegation mechanism for executing feature related code. Features cannot be stored in variables or data structures, neither can they be used in pure Java code. Features are not aware of their or other features'

contents via some sort of interface, consequently their interactions cannot be easily modeled. The general shortcomings of various feature-oriented approaches indicate in a way also the desirable properties of a feature implementation which should be considered in concert instead of providing support for only some of them. We propose to rectify this situation by providing a better representation of features as well as combining the feature modeling concepts of product lines, for a complete feature based software solution.

4. SOLUTION PROPOSED

We propose a feature extension to a programming language under consideration. This extension will have two parts, one as a feature and feature models description language and the other as the feature development and refactoring language used to manipulate code and program fragments. In case of Java, the first part can be expressed as an embedded domain specific language [18]. The second part, i.e, the feature development and refactoring language can be implemented as an extension of the Java syntax and semantics in accordance with the first part. In the following, we establish an agenda for features as first-class entities.

1. The kind of features (such as mandatory and optional), parent-child relations (such as AND,OR, and alternative) about features and constraints between features should be expressible in the extension.

2. Features should be represented as types and interaction between features and regular types should be controlled. The mechanisms of feature normalization and conversion to disjunctive normal form for finding valid feature instances [15] could be coupled with the meta information about features in the programs to compose safe variants. We propose to implement the composition core based on feature algebra [6].

3. It should be possible for features to contain classes and various class members. It should also be possible for classes to contain feature annotations. Such a representation would gain from both compositional and annotative syntax. Coarse-grained program deltas can be represented in a compositional manner while fine-grained deltas can be represented by annotative syntax.

4. Feature models should be expressed adequately in the extension. A feature model should be modifiable at runtime, reflecting in a changed program variant. Reification, i.e., storing information about a feature such as the container entity of a code fragment, can be used to create altered feature variants at runtime. Changing a feature model may entail removing a child feature from a parent feature, relocating it elsewhere or remove it entirely. All such changes need to be supported with the above mechanism.

5. A program delta that is refined by some feature may be required by other features. This information should be expressible at language level so that the choice of creating a variant with altered code or creating variants in which one variant contains the original code and the other variant contains altered code remains with the user.

The above indicates that a mechanism for encapsulating various code fragments that constitute a feature in a programmatic entity should be available. If operations were available on such an entity for code composition as well as product line customization, then a direct correspondence can be established from features at the modeling level and implemented features and both could be manipulated with precise control.

5. IMPLEMENTATION DETAILS

In the following we show how a feature extension can be implemented in Java.

5.1 JastAdd

We propose to use JastAdd[1] which is a Java based compiler construction system [19]. We choose JastAdd because it implements Java 1.4 and 1.5 in terms of modular compiler analyses [17]. JastAdd considers an object-oriented abstract syntax tree (AST) as the basis for language design. It uses AspectJ introductions to add behavior to various classes representing language constructs [17, 19]. Behavior can be added to AST nodes both in a declarative and imperative manner using the extended versions of synthesized and inherited attributes. The declarative specification ensures internally that attributes and analyses need not be ordered by the programmer. Different transformations can be applied to an AST in terms of attributes and an AST can be prepared as required [30].

5.2 Syntax and Semantics of the Proposed Extension

In the following, we give some examples of syntax and semantics of the proposed extension. Consider the feature diagram for a Graph Product Line (GPL)[27], shown in Figure 1.

A feature model representing a product line is declared using the keyword productLine (Figure 2). Figure 2 shows the feature description for the feature model shown in Figure 1. The one, more and all operators in Figure 2 indicate the alternative, the OR and the

[1]http://jastadd.org/the-jastadd-extensible-java-compiler

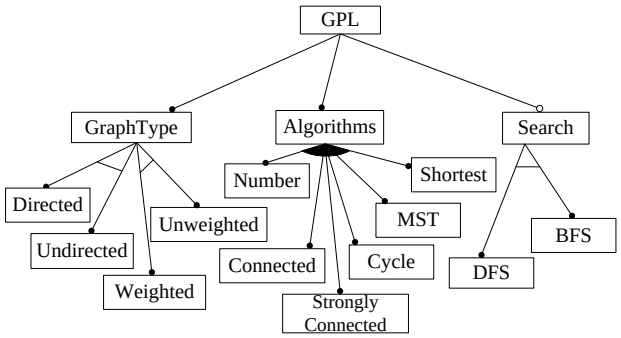

Figure 1: Graph Product Line

```
1 productLine GPL {
2  GraphType : all(one(Directed, Undirected),
3                  one(Weighted, Unweighted))
4  Search     : one(DFS, BFS)
5  Algorithms: more(Number, Connected,
6                   StronglyConnected, Cycle, MST,
7                   Shortest)
8 }
9 all(GraphType, Algorithms,Search?)
```

Figure 2: Feature Description of Graph Product Line

AND features respectively. Optionality is denoted by **?**. This feature description is sufficient to create feature types, so that it is semantically expressible that, e.g., the feature Number is of feature type Algorithms and feature Weighted is of the type GraphType. This feature description provides the language level specification of what to do with features in the software system, i.e., how to group them, how to compose them with respect to any constraints if present. The type representation for features consists of representation for feature specific properties such as whether they are mandatory or optional. Various advanced feature modeling concepts such as feature attributes, groups, and cardinalities can be implemented in the type representation for features in the extension compiler. Once the features are defined in the program, different feature models may be associated with these features. This allows creating not only different products per product line but also different product lines per set of features.

A specific variant graphProduct of the product line

```
1 variant GPL.graphProduct {
2   GraphType = Directed and Unweighted ,
3   Algorithms= StronglyConnected,
4   Search    = DFS
5 }
```

Figure 3: Creating a program variant for GPL

```
1 public variant alterGraphProduct(
2  variant graphProduct) {
3   removeFeature graphProduct DFS,
4   addFeature graphProduct BFS,
5   modifyVariant graphProduct
6   Algorithms=MST;
7   return graphProduct ;
8 }
```

Figure 4: Typed modification, addition and removal of features

GPL is created using the keyword variant (Figure 3). A program variant can be modified by altering the choice of features that constitute it. In Figure 4, feature modification, addition and removal are shown. Specific type for a feature need not be given as in Algorithms=MST, the type of feature is inferred from the feature description. The keywords addFeature, removeFeature and modifyVariant are used in the Java method alterGraphProduct() as operators, to alter the configuration of previously instantiated variant graphProduct.

The feature modeling constraints can be expressed explicitly in the embedded DSL. A feature model can be converted to a constraint satisfaction problem and various Java CSP solvers can be used to obtain valid configurations of features [8, 11, 15]. This can be implemented as a part of the type checking the AST nodes representing productline and variant types. Both mutual inclusion and mutual exclusion constraints between features can be represented as attributes of AST nodes representing the feature type. The implicit implementation constraints between the features are similarly taken care of in type checking the features, e.g., calls between methods of two features making these features dependent on each other. This type of constraints can be handled as a specialized checking of relations between related program elements [23], e.g., a feature that adds a call to a method must ensure that the method itself already exists.

Features can contain not only the class definitions and the class bodies, but also be part of classes and various statements and expressions. In Figure 5, feature Weighted contains definitions of feature specific introductions to separate classes in one place. JastAdd operates with the AST as the only repository of program information. The AST in JastAdd can be copied, extended, and rewritten based on conditions as well as compiled to byte code [17, 19, 30]. This provides a unique opportunity to modify the AST noninvasively both at compile time and run time. Therefore, we can implement features in such a way that the feature definitions need not be complete and additional code fragments can be added to features at runtime as well. A program element like a method can be part of many fea-

```
feature Weighted {
 class Graph {
  public void addEdge(Vertex begin,Vertex
  end, int weight) {
   addEdge(new Edge(begin,end, weight));
  }
 }

 class Edge {
  public int weight;
  public Edge(int the_weight){
   weight = the_weight;
  }
 //constructor with three arguments.
 ...
 }
}
```

Figure 5: feature containing various classes

tures, thus restricting duplication of code. Assume that addEdge() is part of features Weighted and Shortest. This can be achieved as shown in Figure 6. Currently, we intend to provide support for modularizing classes on the basis of features, but in future, we can include aspects in our extension. This is possible in JastAdd because aspects related extensions to their base Java compiler have already been added [7]. In the following,

```
public class Graph {
  feature Weighted,Shortest {
   public void addEdge(Vertex begin,
   Vertex end,int weight) {
    addEdge(new Edge(begin,end, weight));
   }
  }
...
  feature Directed {
   public static final boolean
   isDirected = true;
  ...
  }
...
}
public class Edge {
  feature Weighted {
   public int weight;
   public Edge(int the_weight) {
    weight = the_weight;
   }
 //constructor with three arguments.
 ...
  }
}
```

Figure 6: Classes containing features

we briefly explain how we propose tackle the problems of features mentioned before.

5.3 Solving Problems Related to Features

The combination of feature descriptions and first-class status for features in the extension compiler provides a clearcut way to approach feature-based software development.

- ***Hierarchical refinements*** – Features are no longer related to the class hierarchy as seen in Figures 5 and 6. The feature definitions, whether occurring inside classes/methods or themselves containing definitions of specific elements, are reconciled in one coherent collection when instantiating a product. Once features are reified internally, different transformations can be applied easily to the AST so that version of classes without feature annotations or feature definitions can be generated.
- ***Program deltas*** – Features in this extension use both compositional and annotative syntax as shown in Figures 5 and 6. Not only classes, methods and fields, but method parameters, various statements and expressions in Java can be assigned to features. Because parsing and semantic specifications in JastAdd are modular, our feature extension to Java can be modified easily to support deltas of only the required granularity.
- ***Feature composition and feature order*** – Features can be composed based on feature types. For example, Weighted and Directed features from Figure 1 may be composed to obtain a feature WeightDirected, based on the fact that both of them are of the type GraphType. Order may be specified between features and feature groups whenever required.
- ***Type support for features*** – We represent features as a reference type in the compiler. Various consistency checks for safe compositions can be straightforwardly implemented as lookups and Java typechecks which are implemented as inherited and synthesized attributes respectively in JastAdd.
- ***Dynamic composition and separate compilation*** – For implementing dynamic composition, we intend to use the capability of obtaining transformed copies of the AST as well as the possibility to reify feature code to byte code which can be used via variety of byte code manipulation packages. We intend to explore the use of contextual information for separate compilation of individual features.

6. RELATED WORK

Deursen and Klint [15] propose a language for describing feature models, but they implement features

using UML and Java code generation. In Caesar [29], classes can act as crosscutting layer modules containing many classes or types contributing to features. But it does not provide any interface for feature descriptions, or programmatic means of changing feature configurations. In Object Teams [20], a team is a container for classes and also at the same language level as class. Although it can be used to implement features, it is non-trivial to do so, as teams have a complex inheritance model in which features must be accommodated. Classbox/J [12] provides support for localized refinements, such that original and refined classes co-exist and can be referred to separately. But classboxes have the same problems as other compositional approaches that use redirection mechanisms in implementing features [12]. Like these approaches, we propose to use a more flexible containment for features with respect to classes. At the same time, we combine both feature descriptions and feature-oriented programming concepts together in features represented as first-class entities. Unlike the above mentioned approaches, a developer need not concern himself of how to represent features in terms of underlying technologies, e.g., how to represent features in terms of layers and bidirectional interfaces [29], teams with bindings [20], or classboxes [12]. Features have a structure set by a feature model expressed as feature descriptions and no extra representation is required to relate different code fragments to specific features.

7. CONCLUSION

We have proposed to raise the implementation level of features to first-class status by representing them as types with crosscutting containment in the extension. We have identified various properties that such an implementation should have in order to tackle various problems related to features. In future, we intend to work on extending the Java implementation of the JastAdd extensible compiler framework to include features.

ACKNOWLEDGEMENT

We thank Christian Kästner and Mario Pukall for comments on an earlier draft of this paper.

8. REFERENCES

[1] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.

[2] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.

[3] 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*, pages 125–140. Springer, 2005.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering*, pages 122–131. ACM Press, 2006.

[5] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, 2008.

[7] P. Avgustinov, T. Ekman, and J. Tibble. Modularity first: a case for mixing aop and attribute grammars. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 25–35. ACM, 2008.

[8] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[9] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.

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

[11] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortes. Using Java CSP Solvers in the Automated Analyses of Feature Models. *LECTURE NOTES IN COMPUTER SCIENCE*, 4143:399, 2006.

[12] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: controlling the scope of change in Java. *ACM SIGPLAN Notices*, 40(10):177–189, 2005.

[13] R. Burstall. Christopher Strachey - Understanding Programming Languages. *Higher-Order and Symbolic Computation*, 13(1):51–55, 2000.

[14] K. Czarnecki and U. Eisenecker. *Generative*

Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[15] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[16] E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[17] T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[18] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 855–865, 2006.

[19] G. Hedin and E. Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[20] S. Herrmann. Object Confinement in Object TeamsReconciling Encapsulation and Flexible Integration. *Aspect-Oriented Software Development*, 2003.

[21] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 2007.

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[23] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Computer Society, 2008.

[24] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *In Proceedings of the International Conference on Software Engineering*, May 2008.

[25] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering*, pages 112–121. ACM Press, 2006.

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

[27] R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Conference on Generative and Component-Based Software Engineering*, pages 10–24, 2001.

[28] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM Press, 2001.

[29] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 127–136. ACM Press, 2004.

[30] A. Nilsson, A. Ive, T. Ekman, and G. Hedin. Implementing java compilers using rerags. *Nordic Journal of Computing*, 11(3):213–234, 2004.

[31] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. *LAMP-EPFL*, 2004.

[32] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art InSoftware Development*. Kluwer, 2000.

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

[34] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008. to appear.

[35] C. Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1):11–49, 2000.

[36] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. IEEE Computer Society, 1999.

[37] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007.

[38] H. Zhang and S. Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, 53(3):381–407, 2004.