Integrating Compositional and Annotative Approaches for Product Line Engineering

Christian Kästner
School of Computer Science
University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

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

ABSTRACT

Software product lines can be implemented with many different approaches. However, there are common underlying mechanisms which allow a classification into compositional and annotative approaches. While research focuses mainly on composition approaches like aspect- or feature-oriented programming because those support feature traceability and modularity, in practice annotative approaches like preprocessors are common as they are easier to adopt. In this paper, we compare both groups of approaches and find complementary strengths. We propose an integration of compositional and annotative approaches to combine advantages, increase flexibility for the developer, and ease adoption.

1. INTRODUCTION

In recent years, *software product lines (SPLs)* have gained momentum [9, 37]. Instead of implementing each program from scratch, SPLs enable systematic reuse in a domain by generating a family of related programs – so called *variants* – from a common code base. In this context, features are domain abstractions to distinguish different variants. Typically, features implement increments in functionality. Developers who want to adopt SPL technologies for their product can choose from a wide range of different mechanisms to implement SPLs: from version control systems [42], over simple *#ifdef* statements [35], over frameworks and components [9], to various specialized languages or tools [19, 39, 17, 10]. Each of these approaches has different advantages and disadvantages and there is plenty discussion about which approach is suited best, e.g., [33, 34, 3, 24, 35].

In earlier work, we used various approaches to implement SPLs. We started with AHEAD [10] and AspectJ [28] and their integration Aspectual Feature Modules [7]. These languages can be used to compose variants from reusable code units (compositional approaches). In recent work, however, we found several limitations [23, 24], especially when adopting SPL technology for legacy applications. Therefore, we looked at more traditional approaches like *#ifdef* preprocessors and improvements thereof (annotative approaches). Without ever making it explicit in our research agenda, we pursued both paths – compositional and annotative approaches – in parallel.

While we addressed the specific problems like granularity, language independence, expressiveness, or type-safety of either group of approaches in earlier work [24, 8, 25, 7, 23, 22, 5], we noticed that both groups complement each other. There are several problems for which compositional approaches required severe overhead, but an annotative approach can solve straightforwardly, or the other way around. In this paper, we give an overview of both groups of approaches, discuss differences and synergies, and show how to integrate both.

We focus especially on SPL adoption. Adopting SPL technologies for a project is difficult, especially if the target application is not developed from scratch but derived from a legacy application. There is some discussion whether lightweight implementation approaches can lower the adoption barrier or whether more sophisticated approaches are needed for maintainability and long-term project success [13]. In this paper, we show how an integration of both groups of implementation approaches can ease SPL adoption, but still support long-term design qualities.

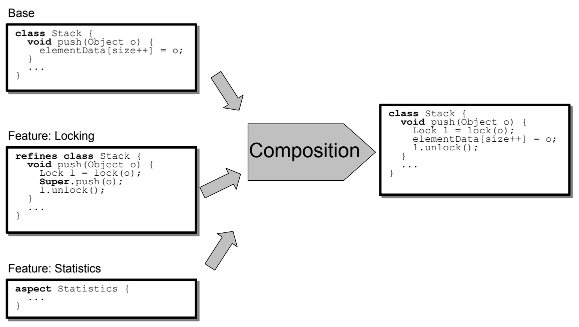
Specifically, we make the following contributions: (a) We put annotative and compositional approaches (two groups of common approaches for SPL implementation) in contrast and analyze advantages and disadvantages of each. (b) We discuss an integration of both. (c) We outline how the integration can lower the adoption barrier but still support long-term qualities.

2. SPL IMPLEMENTATION APPROACHES

There are many approaches to SPL implementation. Most of them can be grouped either as compositional or as annotative [24]. In this section, we briefly introduce both groups, before we compare them to discuss advantages and disadvantages in Section 3.

Compositional Approaches. Compositional approaches implement features as distinct (physically separated) code units. To generate a product line member for a feature selection, the corresponding code units are determined and *composed*, usually at compile-time or deploy-time. There is a large body of work on feature composition, usually employing component technologies [44], frameworks [21], feature-oriented programming with some form of feature modules [39, 41, 10, 7], subjects [19], multi-dimensional separation of concerns [45], and aspects [28]. Depending on the concrete approach or language, the composition mechanism varies from assembling plug-ins to complex code transformations, but the general idea of composition as illustrated in Figure 1 is the same.

In this paper, we use AHEAD [10] – respectively the compatible, newer, and language-independent FSTComposer [8] that uses the same mechanisms – as representative for compositional approaches. In AHEAD and FSTComposer, features are implemented in separate modules that modify the base code. In Figure 2, we show a



```
class Stack {
  void push(Object o) {
    elementData[size++] = o;
  }
  ...
}
```

```
refines class Stack {
  void push(Object o) {
    Lock l = lock(o);
    Super.push(o);
    l.unlock();
  }
  ...
}
```

```
aspect Statistics {
  ...
}
```

```
class Stack {
  void push(Object o) {
    Lock l = lock(o);
    elementData[size++] = o;
    l.unlock();
  }
  ...
}
```

Figure 1: Composing code units.

```
 1 class Stack {
 2   void push(Object o) { ... }
 3   Object pop() { ... }
 4 }
```

```
 5 refines class Stack {
 6   void backup() { ... }
 7   void restore() { ... }
 8   void push(Object o) {
 9     backup();
10     Super.push(o);
11   }
12 }
```

Figure 2: AHEAD example (compositional approach).

simple code example. The *base code* implements a stack, while in a separate module a feature *Undo* is implemented. The refines keyword indicates that the feature module extends an existing class. It introduces the new methods (*backup* and *restore*) and extends the existing method *push* similar to overriding using the keyword *Super*. The base code and different feature modules are composed with the AHEAD tool suite. Depending on which features are included in the composition process, different variants are generated.

Annotative Approaches. In contrast, annotative approaches implement features with some form of explicit or implicit annotations in the source code. The prototypical example, which is commonly used in industrial SPLs, are *#ifdef* and *#endif* statements of the C preprocessor to surround feature code. Such techniques are also common in commercial SPL tools as *pure::variants* [11] or *Gears* [31]. Other examples of annotative approaches are *Frames/XVCL* [20], *explict programming* [12], *Spoon* [36], *software plans* [14], *metaprogramming with traits* [47], and *annotation-based aspects* [29].

In this paper, we use our own tool CIDE as representative for annotative approaches. It is similar to *#ifdef* preprocessors in that code fragments are annotated and can be removed before compilation depending on the feature selection. However, CIDE improves over traditional preprocessors in several ways: (1) annotations are represented by background colors and do not obfuscate the source code with additional boilerplate code [24]; (2) annotations in CIDE are based on the underlying structure of the artifact and, thus, *disciplined* (e.g., it is not possible to annotate only an opening bracket but not the closing one) and ease the generation process [25, 24]; finally, (3) all annotations are managed by the tool infrastructure which allows virtual views on the source code (e.g., show all code that is annotated with feature *Backup*) and navigation support [27, 24].

In Figure 3, we show the previous example as implemented with CIDE. All feature code is located in the same code base. In the

```
class Stack {
  void push(Object o) {
    backup();
    ...
  }
  Object pop() { ... }
 void backup() { ... }
 void restore() { ... }
}
```

Figure 3: Conditional compilation example (annotative approach).

printed version of this paper, all code annotated with the *Backup* feature is underlined. In order to create a variant without this feature, all annotated code is removed before compilation. Nevertheless, this paper is not specifically on CIDE, but other annotative approaches could be used or visualized in the same way.

3. COMPARISON

After briefly introducing the two groups of approaches, we compare them based on several characteristics. We selected characteristics that emphasize the differences between these approaches or which arose during our prior research or case studies. The results are summarized in Table 1 with approximated grades. This evaluation is based on prior work in SPL research and our own experience.

Feature Traceability. Feature traceability is the ability to directly trace a feature from the feature model (domain space) to the implementation (solution space) [15, 2]. For example, feature traceability is important when developers want to debug an error that occurs in a specific feature and want to find all code related to this feature.

Compositional approaches directly support feature traceability as the code that implements a feature can be traced to a single code unit (component, plug-in, aspect, feature module, etc). For example, in Figure 2, all code of the backup feature can be found in the second module (Lines 5–12). In contrast, in annotative approaches feature traceability is poorly supported as feature annotations can be scattered over the entire code base (cf. Fig. 3). However, with special tools like CIDE it is still possible to provide feature traceability in an annotative approach at a tool level. As explained above, virtual views and navigation support can be used to explore all code that belongs to a feature (see [27] for details). Nevertheless, feature traceability in annotative approaches is a matter of tool support.

Modularity. Modularity as needed for modular reasoning or even separate compilation is possible in some compositional approaches. For example, when using components [44], plug-ins [21], subjects [19] or hypermodules [45] this is well supported. However, many more advanced compositional approaches like many aspect languages or the analyzed approaches AHEAD or FSTComposer are based on source code transformations and provide only limited modularity, e.g., separate compilation is not supported. There are no interfaces for feature modules in these approaches, thus to understand a feature, it is often necessary to look also at the base code or even other features[1].

In annotative approaches a modularization is not intended. Modular reasoning can be simulated with tool support (views and navigation support as in CIDE) to some degree, but separate compilation is not possible.

[1]In the context of aspect-oriented programming, modularity has been addressed intensively, e.g., [43, 18], and can potentially be adapted for other compositional approaches.

```
class Stack {
  void push(Object o) {
    if (o==null) return;
    hook();
    ...
  }
  Object pop() { ... }
  void hook() {}
}
```

```
refines class Stack {
  void backup() { ... }
  void restore() { ... }
  void hook() {
    backup();
  }
}
```

Figure 4: Fine-grained extension with AHEAD.

Granularity. The granularity of implementation mechanisms provided by an approach is closely related to its expressiveness. Very coarse-grained approaches only assemble files in a directory, while fine-grained approaches allow modifications on the level of methods, statements, parameters or even expressions [24].

Annotative approaches support even fine-grained extensions well. As many are line-based or character-based, they scale from annotating entire files to even small code fragments of statements (cf. Fig. 3). Even when the underlying structure is used as in CIDE, annotations on the level of AST nodes allow even fine-grained annotations on statements, parameters, or expressions [24].

In contrast, compositional approaches only provide a coarse granularity composing usually only components or – in some approaches like aspects, AHEAD, or FSTComposer – down to introducing or extending methods in existing classes (cf. Fig. 2). However, manipulation of statements inside the middle of a method, of parameters or of expressions is not possible in any compositional approach due to conceptual limitations [24].[2] Instead, workarounds like hook methods are needed. For example, a slight modification in the original example makes an implementation with AHEAD difficult. Imagine that the *backup* call in Figure 2 is not the first statement executed in the *push* method, but located after some sanity checks. In this case workarounds as the hook method in Figure 4 are required, because – in contrast to annotative approaches – introducing a statement in the middle of a method is not supported.

Safety. For both compositional and annotative approaches recent research has provided solutions that ensure that all (potentially millions) variants of the SPL are syntactically correct and well-typed. While compositional approaches ensure syntactical correctness with their composition mechanism, many annotative approaches are line-based or character-based and can easily generate syntactically incorrect variants. However, CIDE – our annotative approach which enforces disciplined annotations – provides such safety using the underlying structure [25] and can thus achieve the same level of safety.

Although they use different implementations, there are several approaches to type-check entire SPLs for both compositional [46, 5] and annotative approaches [16, 22, 30]. Beyond type-safety, we know of no approach to verify behavior of all SPL variants that would scale to mid-sized SPLs. There are several approaches on SPL testing [38]; however, they work on the level of generated variants and are thus independent of the implementation mechanism.

[2]Some aspect languages like AspectJ can intercept join points in the body of methods, which can be used for extending statements to some degree (*'call && withincode'*). However, there are several limitations as discussed elsewhere [23, 24].

Language independence. Language independence is another characteristic where both approaches perform similarly well. While, many annotative approaches are line-based or character-based and thus completely language-independent, even the more disciplined CIDE which uses the underlying structure can be extended for arbitrary languages (generated from the language's grammar) [25]. In contrast, compositional approaches are usually depending on a particular host language, where AHEAD and FSTComposer provide notable exceptions. Especially in FSTComposer, there is a general composition mechanism which can be easily extended for a new language with only little manual effort [8].

SPL Adoption. Industry is very careful on adopting compositional approaches because it influences their existing code base and development process too much. At most, after careful planning, frameworks or components are used [9]. In contrast, annotative approaches can be adopted much quicker, because they introduce only lightweight tools which do not change the code or development process too much at first [13]. Annotative approaches, thus, make adoption of SPL technologies easier and more likely in the initial steps of evaluation and early development. In this context, CIDE's concept of storing annotations separate from the source code is worth mentioning, because it allows annotating a legacy application without changing its source code representation. This makes CIDE well-suited for evaluating SPL technologies.

In an earlier case study, we experienced ourselves that refactoring a legacy application (in this case Oracle's Berkeley DB) into separate code units that can later be composed is by far more difficult and tedious than just annotating code [24].

4. COMPOSITIONAL AND ANNOTATIVE APPROACHES IN CONCERT

The comparison in the previous section showed that compositional and annotative approaches are quite different and have different – often complementary – strengths and weaknesses, which are also differently important in various phases of SPL adoption. Research on SPL implementation focuses almost exclusively on compositional approaches, ignoring the advantages of annotative approaches. Instead, the demand from industry has been answered mostly by commercial vendors like *BigLever* and *Pure Systems* which (among others) provide tools to annotate code. In this paper, we show that integrating compositional and annotative approaches is beneficial. In the following, we propose a simple integration and discuss its impact on the characteristics above.

4.1 Integration

Conceptually, an integration is straightforward. Using a compositional approach, features can be physically separated into code units (e.g., components, aspects, feature modules). Inside these physically separated code units, an annotative approach can be used to additionally annotate code fragments.

In Figure 5, we show a possible different implementation of the extended example of Figure 4. The method declarations for *backup* and *restore* are implemented in a physically separated AHEAD feature module, while some code (that would be hard to extract) is left as an annotation (underlined) in the base code. To generate a variant, the code units are composed and the annotations are evaluated to remove unneeded code fragments.

Also technically, such integration is straightforward. The annota-

	AHEAD/FSTComposer (compositional)	CIDE (annotative)	Integrated Approach	Importance in the life cycle
Traceability	++	+	+	all phases
Modularity/Separate comp.	+	−−	+/−	maintenance
Granularity	−	++	++	early adoption, implementation
Safety	+	+	+	all phases
Language independence	+	+	+	all phases
Adoption	−	++	++	project start

++ very good support, + good support, +/− medium support, − poor support, −− no support

Table 1: Comparison

```
class Stack {
  void push(Object o) {
    if (o==null) return;
    backup();
    ...
  }
  Object pop() { ... }
}
```

```
refines class Stack {
  void backup() { ... }
  void restore() { ... }
}
```

Figure 5: Fine-grained extension with AHEAD.

tive approach – in our case CIDE – must only be extended to support the additional language constructs of the compositional approach (e.g., *refines* keyword), and the generation process must be adapted to handle both composition and evaluating annotations. If annotations are evaluated before the actual composition, the composition process itself does not even need to be adapted. On the tool level, an integration is more difficult as both tools must be integrated. Views and navigation support for the annotative approach must be extended for physically separated feature implementations, and existing support for the compositional language as in FeatureIDE [32] (e.g., syntax highlighting, code completion, outline view) must be extended to understand the annotations. However, integrating tools is not a conceptual challenge, but merely an engineering task.

Interestingly, *automated refactorings* that transform annotated code into physically separated AHEAD feature modules [26] and automated refactorings that transform AHEAD feature modules into annotated CIDE code [30] have been developed. While the result of these refactorings might be difficult to read (e.g., the transformation from CIDE to AHEAD heavily requires hook methods as in Figure 4, which makes generated code hard to read in the presence of fine-grained extensions) or unambiguous (transforming AHEAD to CIDE, there are many different possible annotated programs that express the same behavior), these automated refactorings still help developers to deal with the different possible representations of the source code. Note, refactorings can also be used to convert only parts of the SPL, e.g., convert individual features or only code from certain classes or methods.

4.2 Comparison

An integration of compositional and annotative approaches does not automatically dissolve all disadvantages of either approach. For example, when using annotations in physically separated feature modules, modularity is lost as if only annotations were used in the first place. However, the main advantage is that developers can always decide when to use which approach and when to use a combination of both, e.g., to achieve fine granularity or ease adoption. In the following, we discuss the criteria listed in Section 3 for the integrated approach. Approximated grades are shown in Table 1.

Feature Traceability. First, feature traceability is weaker than in pure compositional approaches. A feature must not be physically separated, but can instead or *additionally* be implemented by some scattered (annotated) code fragments, as shown in the example in Figure 5. However, still the same views and navigation support from CIDE can be used, so feature traceability is not worse than in the CIDE solution. Moreover, if a full (or partial) physical separation is desired it can be achieved and even improve feature traceability. In that case, all (or most) relevant feature code can be found in one corresponding code unit.

Modularity. When integrating compositional and annotative approaches modularity depends on how the developers implement a feature. They can choose between a modular implementation (using classes, modules, aspects, feature modules) or a non-modular implementation with (at least some) scattered code fragments as in Figure 5. In general, modularity is weakened and separate compilation is no longer possible. However, using gradual refactorings it is possible to achieve modularity in the long run.

Granularity. The integrated approach benefits from the fine granularity of the annotative approach. Features can still be implemented with physically separated code units as far as reasonable or possible with the low available granularity. However, additionally fine-grained extensions can still be added to the base code or other features and marked with annotations. For example, instead of using workarounds as the hook method in Figure 4, such fine-grained extensions can be implemented with an annotation as shown in Figure 5. Again, the integrated approach allows a quick solution using annotations at first, while further refactorings are possible to change the implementation to avoid fine-grained extensions and use only the compositional approach in the long run.

Safety and language independence. As discussed above, there are striking commonalities in the solutions for type-checking and language-independence developed in recent research for both, compositional and annotative approaches. As the mechanisms are already related, and integration is mostly an engineering task. Thus, the same level of safety and language independence can be achieved as in the original isolated approaches.

Adoption.

The most interesting results from an integration affect the process of adopting SPL technologies for a project. While industry is very careful about adopting compositional approaches, they are usually seen as superior in academia because of modularity, separations of concerns and thus promised improvements during maintenance and evolution in the later life cycle phases. Nevertheless, companies often use lightweight annotative approaches for faster results and lower initial risk [13].

In this scenario, the integration of compositional and annotative approaches pays off. In early evaluation and adoption stages, developers can simply annotate legacy code. They can use the lightweight capabilities of annotative approaches without having to change their code base. As annotations are stored separately in CIDE, these annotations do not even affect the code base at all.

In later stages, when the idea of developing SPLs is established, and annotations already provide variability they can gradually change from the annotated code base to separated feature modules by automated or manual refactoring. Still, it is not necessary to refactor all annotated code fragments at once, but developers can start with the obvious coarse-grained ones (separate entire classes or method introductions) and gradually prepare the code (e.g., by introducing explicit extension points) to avoid even the annotated fine-grained extensions. This way it is possible to adopt a compositional approach gradually and eventually achieve long-term goals of modularity and maintainability.

4.3 Discussion

Instead of being forced to choose between an annotative or compositional approach, an integration allows to start with one and gradually refactor to the other. For each problem, developers can choose the mechanism that suits best at first and only later refactor to a different version if reasonable. Though this allows developers to break modularity, it also gives them expressive power to express fine-grained extensions. Goals that have been achieved in either approach like traceability, safety, or language independence can be adapted also for the integrated approach.

Instead of a one-step effort with uncertain costs and risks, the integration allows a lightweight adoption with annotative approaches and gradual refactoring to compositional approaches. This eases the initial adoption barrier significantly, while long-term goals are explicitly supported.

5. RELATED WORK

Several related publications compare different approaches to SPL implementation. First, Lopez-Herrejon et al. [33] and Mezini and Ostermann [34] compare several compositional approaches. Both focus in detail on modularization support and do not cover annotative approaches. Next, Anastasopoulos and Gacek [1] briefly compare 11 concrete implementation approaches, Muthig and Patzke [35] also compare 6 approaches. Both comparisons include conditional compilation and frames as annotative approaches. However, in both works analysis is focused on expressiveness and several details of analyzed languages and does not consider an integration of approaches. We provide a broader comparison on a higher level of abstraction, where we each subsume all compositional approaches and annotative approaches. For our analysis the subtle distinctions between different compositional approaches is not relevant, but the importance lies in the bigger picture achieved with an integration.

The idea of adopting SPL technology slowly and stepwise emerged from Spinczyk in a discussion at the Dagstuhl seminar 'Software Engineering for Tailor-made Data Management' [4]. The participants proposed a migration path from 'thinking in product lines', over using advanced preprocessors, over object-oriented decomposition, toward more advanced compositional approaches of aspect-oriented programming and feature-oriented programming, and eventually toward model-driven development or a decoupling using service-orientation. We follow this migration path on the lower levels and actually combine annotative and compositional approaches to be able to evolve gradually from one approach to the other.

FeatureC++ [6], a compositional approach for C++ based on feature-oriented and aspect-oriented programming mechanisms, still allows to use the C preprocessor (*#ifdef*). FeatureC++ can therefore be considered as an existing implementation that already integrates compositional and annotative approaches to some degree. Nevertheless, this integration was not planned and never made explicit. The used annotative approach is line-based, does not use the underlying structure and there are neither views nor navigation support. In prior work using FeatureC++, developers explicitly avoided to use the preprocessor to achieve variability [40].

6. CONCLUSION

There are many different approaches to implement SPLs. Most can be classified as either compositional or annotative. While annotative approaches like simple *#ifdef* directives are common in industry, research focuses mostly on compositional approaches like aspect-oriented or feature-oriented programming.

In this paper, we compared both groups of approaches and found that there are many differences in strengths and weaknesses, but that it is also possible to base both on similar foundations based on the artifact's structure. To combine the advantages of each and address the shortcomings we propose an integration. With this integration a developer can separately choose which implementation mechanism to use for each problem. This eases SPL adoption. When adopting SPL technologies, it is possible to first use the lightweight annotative approaches and then gradually refactor toward compositional approaches as far as possible and reasonable.

In future work, we want to implement the integration and experimentally merge our tools CIDE and FSTComposer, and evaluate our approach empirically in an industrial case study or experiment.

7. REFERENCES

[1] M. Anastasopoules and C. Gacek. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3), 2001.

[2] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui. On Feature Traceability in Object Oriented Programs. In *Proc. ASE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. 2005.

[3] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2006.

[4] S. Apel, D. Batory, G. Graefe, G. Saake, and O. Spinczyk, editors. *Software Engineering for Tailor-made Data Management*. Dagstuhl Seminar Proceedings. 2008. to appear.

[5] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2008.

[6] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2005.

[7] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2), 2008.

[8] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. ETAPS Int'l Symposium on Software Composition*, 2008.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[10] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.

[11] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program.*, 53(3), 2004.

[12] A. Bryant et al. Explicit programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. 2002.

[13] P. Clements and C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4), 2002.

[14] D. Coppit, R. Painter, and M. Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2007.

[15] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press, 2000.

[16] K. Czarnecki and K. Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2006.

[17] M. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference (SPLC)*. 2000.

[18] W. Griswold et al. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 2006.

[19] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, 28(10), 1993.

[20] S. Jarzabek et al. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2003.

[21] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[22] C. Kästner and S. Apel. Type-checking Software Product Lines - A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 2008.

[23] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*. 2007.

[24] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2008.

[25] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report 2, School of Computer Science, University of Magdeburg, Germany, 2008.

[26] C. Kästner, M. Kuhlemann, and D. Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *Poster presented at Europ. Conf. Object-Oriented Programming*, 2007.

[27] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, 2008.

[28] G. Kiczales et al. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2001.

[29] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2005.

[30] C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2008.

[31] C. Krueger. Easing the Transition to Software Mass Customization. In *Proc. Int'l Workshop on Software Product-Family Eng.* 2002.

[32] T. Leich, S. Apel, and L. Marnitz. Tool Support for Feature-Oriented Software Development: featureIDE: an Eclipse-based Approach. In *OOPSLA workshop on eclipse technology eXchange*. 2005.

[33] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2005.

[34] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6), 2004.

[35] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proc. Net.ObjectDays*. 2003.

[36] R. Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distrib. Sys. Onl.*, 7(11), 2006.

[37] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[38] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49(12), 2006.

[39] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 1997.

[40] M. Rosenmüller et al. FAME-DBMS: Talor-made Data Management Solutions for Embedded Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, 2008.

[41] Y. Smaragdakis and D. Batory. Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 2002.

[42] M. Staples and D. Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proc. Asia-Pacific Software Engineering Conf.* 2004.

[43] K. Sullivan et al. Information Hiding Interfaces for Aspect-Oriented Design. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering*. 2005.

[44] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[45] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 1999.

[46] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2007.

[47] A. Turon and J. Reppy. Metaprogramming with Traits. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2007.