The Road to Feature Modularity?

[Discussion Paper]

Christian Kästner
Philipps University Marburg

Sven Apel
University of Passau

Klaus Ostermann
Philipps University Marburg

ABSTRACT

Modularity of feature representations has been a long standing goal of feature-oriented software development. While some researchers regard feature modules and corresponding composition mechanisms as a modular solution, other researchers have challenged the notion of feature modularity and pointed out that most feature-oriented implementation mechanisms lack proper interfaces and support neither modular type checking nor separate compilation. We step back and reflect on the feature-modularity discussion. We distinguish two notions of modularity, *cohesion* without interfaces and *information hiding* with interfaces, and point out the different expectations that, we believe, are the root of many heated discussions. We discuss whether feature interfaces should be desired and weigh their potential benefits and costs, specifically regarding crosscutting, granularity, feature interactions, and the distinction between closed-world and open-world reasoning. Because existing evidence for and against feature modularity and feature interfaces is shaky and inconclusive, more research is needed, for which we outline possible directions.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; D.3.3 [**Software**]: Software Engineering—*Language Constructs and Features*

General Terms: Design, Languages

Keywords: modularity, interfaces, module systems, variability, granularity, crosscutting, feature interactions, feature modules, feature models

1. INTRODUCTION

Over the last years, we have frequently discussed what modularity means for feature-oriented software development. We and others have typically claimed that we want to modularize feature representations to make features explicit in design and implementation, to reason about their implementation locally, and to compose them flexibly. In contrast, others have repeatedly challenged that many contemporary forms of feature-oriented programming provide merely syntactic compositions, lack proper interfaces, and provide only closed-world reasoning.

We led numerous heated discussions and found that different people have very different assumptions of what to expect from modularity, which is also enforced by different notions of modularity in different communities. We distinguish a notion of cohesion and locality often found in the software-engineering community and a notion of information hiding using interfaces currently coming more from the programming-languages community. In our understanding, the major part of the community around feature-oriented software development focuses on cohesion and neglects information hiding and interfaces.

Behind the discussion of modularity is lurking another discussion about taking an open-world view or a closed-world view on features. In an open-world view not all features are necessarily known, whereas, in a closed-world view, we globally reason about a closed set of feature implementations. How suitable are open-world and closed-world views for applications such as software product lines?

In this discussion paper, we share and discuss our current understanding of this debate. Although we aimed at a neutral view, we cannot avoid bias from our own background, which certainly shaped much of this paper. Although not intended as retrospect justification, our discussion can be interpreted as such. Hence, we are open about our bias: The first two authors have a background rooted in the locality and cohesion notion, increasingly learning to appreciate other notions, whereas the third author has a background rather rooted in the programming-languages community and has a more critical view on lacking interfaces.

Although we cannot present a general solution for feature modularity—we are not even entirely convinced that feature interfaces should be a dominating goal—we discuss challenges (especially fine granularity and feature interactions) and possible research directions. Most insights are not new in isolation, but we hope to contribute a personal view on the debate that we gathered over several years.

2. FEATURE MODULARITY

So what is feature modularity? Already the term modularity is so overloaded with different definitions and interpretations that it seems infeasible to extract one general meaning. Hence, we explore two different notions of feature modularity (and probably bluntly forget others): one based on cohesion

SPLC'11 August 21-26, 2011, Munich, Germany.

and one based on information hiding. But first, we need to clarify what kind of feature representations we are discussing.

2.1 Representing features

There are different views on what a feature is and how it manifests in a program. In previous work on feature-oriented languages and tools, researchers view features mostly from a syntactic or structural perspective: When added to a program, a feature manifests in additions or modifications of development artifacts [5]. Hence, such work focuses on the representation of features in the sense of concerns in design documents and implementations. Interesting issues that arise from this view are how to arrange a code base such that features become explicit and composable.

At the same time, there is an alternative perspective on features, inspired by early work on feature-interaction detection in telecommunication systems: Features are viewed in terms of the behavior they induce. They are semantic units that interact and give rise to the observable behavior a user is interested in [28, 30, 34]. The code structures that implement a feature are of less interest in this perspective.

The two perspectives are two sides of the same coin, both encourage abstraction, but both emphasize different properties. Here we take side with the syntactical and structural perspective, as it guided our research for many years. Of course that does not mean that the semantic perspective is less important; nevertheless, we address the issue of modularity from our perspective.

2.2 Locality and cohesion

The core idea of feature-oriented programming was to use features as an additional dimension of decomposing a program [60] (as addition, not as replacement for other forms of decomposition using packages, classes, methods, abstract data types, functions, and so forth). When decomposing a program into features, the individual parts are typically called *feature modules* [11]. Seminal papers on feature-oriented programming claim that feature-oriented programming "allows to compose objects from individual features or abstract subclasses in a fully flexible and *modular* way" [60] and that "feature refinements are *modular*, albeit unconventional, building blocks of programs" [11].

In this context, a feature module is regarded as a unit of composition (and often as a transformation that applies changes to other features [11, 51]). Decomposing a program into feature modules has the goal of making features explicit in design and implementation. Modularity in this context means *locality* and *cohesion*. The idea is to place everything related to a feature into a separate structure (file or folder), which is then called feature module. For example, in the simple feature-oriented program in Figure 1, everything related to the implementation of feature *Undo* is localized in a separate file.

Most concepts and tools in feature-oriented software development follow this notion of modularity, focusing on locality and cohesion. We found that this notion is pervasive in large parts of the aspect-oriented-programming community and the software-engineering community as well.[1]

[1]We avoid the term separation of concerns, because it is overloaded and used inconsistently. Intuitively, we would equate separation of concerns with the cohesion notion of modularity, but it can equally be regarded as rooted in information hiding, encapsulation, and abstraction.

Feature module *Base*

```
1 class Stack { ...
2   void push(int v) {/*...*/}
3   int pop() {/*...*/}
4 }
```

Feature module *Top*

```
1 refines class Stack {
2   int top() {/*...*/}
3 }
```

Feature module *Undo*

```
1 refines class Stack { ...
2   int backup;
3   void undo() {/*...*/}
4   void push(int v) {
5     backup=top();
6     original(v); //calling push of Base
7   }
8 }
```

Figure 1: AHEAD-style feature-oriented implementation of a stack with three feature modules.

We believe that, for many tasks, locality and cohesion provide a significant advantage compared to implementations of a feature in which feature code is scattered across the code base and tangled with code of other features. Locality and cohesion help developers to get an overview and to focus their attention when maintaining source code. For example, in annotative approaches [39]—such as using #ifdef directives of the C preprocessor to mark scattered feature implementations—we need to perform a global search to find all fragments of a feature. (It is this context in which we argue that views as in FeatureMapper [31], C-CLR [65], CIDE [38], and others provide a virtual separation of concerns and emulate locality and cohesion, cf. [38, 39].)

2.3 Information hiding

Information hiding and encapsulation, enforced with *interfaces* and typically associated with *module systems*, are another side of modularity, typically emphasized in the programming-language community.[2] The idea behind information hiding and encapsulation is to distinguish between an internal and an external part of a module. The internal part is hidden (encapsulated) from other modules. The external part is called an *interface* (described more or less formally in different contexts) and describes a contract with the rest of the world. The contract *enforces* the desired abstraction and corresponding invariants. To understand a module, we need to look only at the module itself and at (imported) interfaces of other modules. Furthermore, interfaces can also be detached from specific modules, also known as abstract interface [18].

In that sense, interfaces enable *modular reasoning:* We can understand a module in isolation without looking at internal parts of other modules. Hence, we reduce complexity with a divide and conquer strategy. Note that the locality

[2]When linking notions to communities, we refer to where we currently and subjectively observe the active discussants in this debate. Modularity as means for information hiding originated from the software-engineering community; but more recently, many developments from this community take a more syntactic, cohesion-based path. In contrast, many outspoken members of the programming-language community have adopted the interface view point and strongly argue for it.

and cohesion notation may support local reasoning to some degree, but without interfaces there are no hidden parts and abstraction is not enforced, so global whole-program analysis might still be necessary. For example, to understand feature *Undo* in Figure 1, we might need to look at the implementation of both *Top* and *Base*, because there are no interfaces that would enforce abstraction from their implementations. We frequently experienced the need to look at code from other features in various case studies. (Interestingly, the notion of modular reasoning has stirred quite some debate in the aspect community, discussing to what degree locality is still useful for reasoning without formal interfaces—without apparent consensus [2, 42, 57, 66, and others].)

Another goal of many module systems is to enable *modular type checking* (and other modular checks), prevent name clashes, and support *separate compilation* with binary linking [20]. Just as humans should not need internal parts of other modules for understanding, type systems should check a module only relying on imported interfaces, but not on implementation details of other modules (not possible in Figure 1). Whereas modular reasoning has a human component and allows different interpretations, modular type checking and separate compilation can be defined and proven formally. Many module systems developed in the programming-language community formally underpin the separation between internal and external part and guarantee properties such as modular type checking [15, 20, 49, 54, and others].

Enforcing interfaces encourages an *open-world view* on modules. We can reason about or check a module (potentially against imported interfaces) without knowing about the remainder of the program. The result will not change when we add a new module to the system—a property we cannot guarantee just with cohesion.

Many other advantages are associated with this notion of modularity, including black-box reuse, independent multi-person development, and modular testing. In all cases, interfaces provide contracts (more or less formal, more or less mechanically enforced) between independent stakeholders.

In feature-oriented software development, the information-hiding notion of modularity seems underrepresented. Except for attempts that use languages with a strong module system to encode the style of feature-oriented programming [4, 21, 22, 32, 33, 43, and others], we are not aware of any languages designed for feature-oriented programming that support explicit formal or informal interfaces for feature modules. Instead, previous feature-oriented type checkers performed closed-world checks on all feature modules together [6, 25, 40, 70]. Already Java, which many feature-oriented approaches extend, is often rather frowned upon for lacking a proper module system [12, 24, and others].

Hence, researchers with a background in programming languages and module systems (especially those with a focus on formal guarantees) sometimes look rather critically at current feature-oriented-programming approaches. We believe they have a point, and it is a point worth discussing.

3. CHALLENGES OF FEATURE MODULARITY

So, is there a reason not simply to define interfaces for feature modules and hide internal implementations of a feature? That seems feasible and we could start right away with encodings in existing module systems. However, there are additional challenges that are specific to features.

3.1 Modularity costs

Before we come to feature-specific challenges let us keep in mind that modularity always causes costs in the form of development effort, maintenance effort, and so forth. We need to invest additional effort into design, we need to decide between hidden and exposed parts, we need to design suitable interfaces, and we need to write and maintain the interfaces. In addition, modularity may introduce architectural overhead. Deciding on an interface is a difficult decision that should take prospective future developments into account. When we decide on what to hide, we commit to this decision and fix the interface. Changing this interface later on is nonmodular; it will affect other modules as well. In this sense, information hiding can hinder evolution (explored in more detail elsewhere [58]).

During development, a nonmodular solution is typically cheaper. Modularity is beneficial if the expected benefits—such as modular reasoning, easier maintenance, and the possibility to develop open systems—outweigh the additional costs. Hence, when developing a mechanism for feature modularity, we have to consider costs and benefits.

3.2 Crosscutting and granularity

Crosscutting and fine granularity challenge feature modularity. First, features often crosscut other dimensions of modularization. The implementation of a feature may scatter across multiple classes and methods, and implementations of multiple features may be tangled in the same underlying module. For example, the feature *transaction system* in a database will extend and interact with many other parts of the system. Especially, the view of feature-oriented programming as code transformations [11]—each feature transforms the program and produces a new program—lets it appear quite naturally that features have crosscutting effects.

Second, in scenarios that we typically looked at, features often implement fine-grained extensions. It is quite common for features to change source code at expression, statement, or parameter level [39]. For example, feature *Undo* in Figure 1 adds a statement to an existing method. In contrast, traditional module systems rather focus on coarse-grained structures. Contemporary interfaces usually describe functions, modules, classes, packages, events, and so forth, so that we hide large parts of the implementation behind a comparably small interface—the goal of information hiding.

We could argue that fine-grained scattered feature implementations are poor style (often originating from decomposing legacy applications) and that we should redesign such programs. However, we argue that this is not realistic in practice: Although we might be able to avoid some fine-grained crosscutting with a suitable architecture, we believe that implementers want the flexibility and that introducing (or extracting) variability in existing systems is an important use case.

Alternatively, we can always use coarse-grained interface mechanisms to hide internals of fine-grained implementations, but such encodings introduce overproportional overhead [39]. When a feature makes fine-grained changes in many positions, we need to expose extension points at all positions, resulting in a potentially large interface that is difficult to understand. Specifically, the interface can become so large that it outweighs the size of the original implementation. So, is the additional effort for the interface really worth the achieved

hiding? We may reach a point at which the interface contains the whole information of the feature implementation, thus rendering information hiding expensive but pointless, because we expose all internal information anyway.

Investigating possible interface mechanisms for fine-grained crosscutting extensions is an interesting avenue for research. Those should hide internals of fine-grained extensions at low costs. Work on interfaces for aspect-oriented programming [2, 23, 67, 68, and others] is highly related to this issue and has yielded some interesting mechanisms; however, such solutions can limit flexibility and impose costs that may make using aspects less attractive. For example, instead of matching on events in the whole program (quantification), some approaches require that join points are announced locally [2, 67]. How such interface mechanisms affect code comprehension and modular reasoning by humans needs to be evaluated.

There seems to be a trade-off between expressiveness and modularity mechanisms in current solutions. The more we restrict the kind of extensions a feature can make (by restricting granularity or crosscutting) the easier it becomes to design interfaces and module systems. It remains an interesting research question whether we can also build useful interfaces for highly expressive feature mechanisms or whether we should strive to find a sweet spot between expressiveness and modularity costs.

3.3 Feature interactions

Feature interactions, both intended and accidental ones, are a second significant challenge for feature modularity. Features should often influence the behavior of other features. For example, a statistics feature in a database engine should collect statistics about all parts of the database, but the extent of collecting statistics will vary depending on whether other features such as transactions and persistence are included. In addition, we want to avoid accidental feature interaction, when two features work fine without the other, but behave unexpectedly when composed, as in the famous call waiting/call forwarding examples of open telecommunication systems [19] and the flood control/fire control example [37].

Feature interactions are interesting for module systems, because they raise the question of how to modularize interacting feature implementations and the question to what degree interfaces can protect from accidental interactions (and ensure intended ones).

Implementing intended interactions: For illustration, we pick up a simple but well-known example of an expression product line providing different terms (plus, power, logarithm, etc.) and different operations (eval, print, simplify, etc.). We want to modularize all terms and operations as features and compose them flexibly to derive different expression libraries. This scenario is based on the well-known expression problem, but has the additional challenge that all features should be freely composable (i.e., we not only want to add new terms and operations, but also be able to omit existing terms and operations) [53]. The expression product line is an example of a simple *structural* form of feature interactions, far less complex and easier to recognize than the typical interactions in telecommunications systems [19]. Yet, these structural interactions already let us illustrate modularity problems.

In Figure 2a and b, we illustrate the typical strategy to modularize terms or operations. In each scenario, we can hide information about the internal implementation, but we

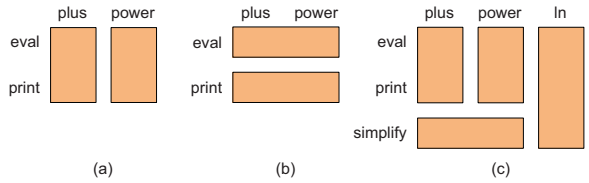

Figure 2: Unsuitable modularizations of the expression product line

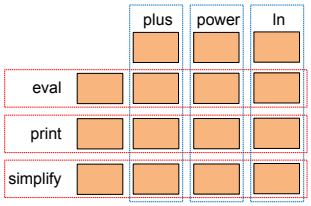

Figure 3: Micromodularization of the expression product line

cannot flexibly select which terms (in Fig. 2b) or operations (in Fig. 2a) to compose. This problem of interacting features is related to the tyranny of the dominant decomposition [69]. Common solutions to the expression problem typically arrive at implementations, in which we can *add* additional terms or operations modularly to existing implementations, as illustrated in Figure 2c, but this still does not allow freely selecting and composing features (e.g., we cannot compose all features *except* feature print).

The typical solution to modularizing such interacting features in feature-oriented programming is to decompose the problem even further as shown in Figure 3, called *micromodularization* and also known as *lifters* [60], *tiles* [45], *origami* [10], or *derivatives* [51].[3] That is, we create a module for each feature and an additional module for each interaction. Each additional module can be seen as adapter or binding that adds glue code between two or more features. With a suitable module system, technically, we could create interfaces and hide internal implementations for each of these modules. Essentially, micromodularization separates the core of a feature from (intended) interactions with other features.

Although we can (micro-)modularize interactions, again, we need to weigh costs and benefits. We add quite a number of additional modules (one for each interaction), causing increasing development effort. In addition, modules may become so small that, again, there remains little to hide and little to reason about modularly. In many cases, we have observed that most complexity of a feature's implementation is in the interactions with other features, whereas the implementation of the feature's core is almost trivial. The expression example above illustrates this in an extreme form: A feature such as *print* without interactions has a minimal implementation; but also each interaction module introduces only a small behavior extension (evaluating additions, simplifying multiplication, etc.), of which we cannot hide much and about which we cannot reason much in isolation.

[3]We credit the term *micromodularization* for this problem to Shriram Krishnamurthi, coined at Dagstuhl #11021. We adopt this term over previous ones, because it better reflects the problem of many small modules in our discussion.

Micromodularization is a technical necessity and usually not a desired property. We are interested in larger structures, such as all behavior of an operation or term (dotted line in Figure 3); however, as shown in Figure 2, we cannot modularize one feature without scattering the other. Although hierarchical module systems enable composing and nesting of smaller modules into larger modules [15], it is not obvious how to build larger building blocks in the presence of variability, when modules are often composed in different combinations and belong to multiple larger concepts. In several case studies, we found that intended feature interactions between optional features are quite common, and we experienced significant overhead of micromodularity and usually opted for nonmodular solutions to avoid that effort [41].

It seems that the root of the problem is that we cannot simply partition code hierarchically into features since features interact. Module relationships do not form a tree structure, but a directed acyclic graph (aligning with knowledge representations that are also structured as graphs and not as trees) or an n-dimensional matrix (cf. Fig. 3).

We conjecture that developers would be more interested in aggregated views on multiple modules, with different tasks requiring different (overlapping) views. An on-demand zooming and aggregation environment could provide suitable views (or projections) [35, 38, 56, 61]. This raises the questions (a) whether reasoning about features is really a modularity issue or rather a tooling issue and (b) whether feature modules should be designed for developers or whether they should be the hidden underlying building blocks of a sophisticated development environment.

Still, investigating how to support micromodularity with lightweight (possibly inferred) interfaces, investigating the costs and benefits of micromodularity in practice, and searching for alternatives to micromodularity (e.g., with architectures that avoid or channel interactions uniformly) remains an interesting research avenue.

Preventing accidental interactions: In addition to the question of how to modularize features despite intended interactions, there is also the challenge of how to detect unintended feature interactions. Feature modularity and feature interfaces seem promising for this task. Ideally, all interactions are specified in interfaces, so that we can reason about modules locally using only imported interfaces; in a modular setting there should be no chance for accidental interactions to occur.

To detect accidental feature interactions automatically, however, we need more powerful interfaces, beyond structural information known from mainstream programming languages. We need to give semantic (behavioral) guarantees in interfaces in order to reason automatically about behavior modularly.

This raises the question of what a feature must know about other features to work properly (i.e., what information must be exposed in an interface, formally or informally), or alternatively what a feature is allowed to do. May a feature intercept input and output, change variables, suppress actions, modify the control flow, or add states? The more we allow a feature to do, the more expressive our interfaces must be to detect or prevent feature interactions modularly.

Detecting or preventing feature interactions in feature-oriented programming is a prime example for the need of semantic interfaces in software analysis, as explored in design by contract [46, 55].

Exploring suitable interfaces to automatically detect feature interactions, again balancing costs and benefits, can be the killer application for feature modularity. For example, several researchers have extensively explored compositional model checking to detect feature interactions [16, 47, 50, and others].

3.4 Feature model

What is the role of the feature model in all this? A feature model describes a set of features and their relationships [36]. Most approaches expect a global feature model describing all features. The global feature model decides how to compose feature modules; hence, features in a global feature model are typically linked directly to feature modules. Because a global feature model describes all features of the product line, it is inherently antimodular (closed-world view). Although there has been work on composing feature models from smaller feature models [1, 14], there has been little discussion of what information hiding and interfaces would mean for a feature model and how feature models should relate to feature modules.

We believe that many interesting design decisions remain to be explored, including but certainly not limited to: Do we want to modularize feature models in an open-world fashion? Should features and their attributes and dependencies be part of an interface of feature modules? Should feature modules and feature models be linked (e.g., so that feature modules import a feature model or vice versa)?

3.5 Other concerns

We believe that granularity, crosscutting, interactions, and feature models are the most important challenges for feature modularity, but there are many more design decisions to investigate.

In the context of feature-oriented programming, we and several others have explored *language-independent* solutions (or solutions that can easily be extended toward other languages, including noncode languages such as grammars or HTML) [4, 7, 11, 26]. Was this merely a syntactic exercise or can we translate that effort to language-agnostic interfaces or to module systems that can be applied to multiple languages? In this context, the gDeep calculus [4], which divides the type system into a language-independent core and language-dependent extensions, may guide the way for various kinds of interfaces.

Another interesting question is whether we should *declare* or *infer* interfaces. Already for current feature-oriented programming languages, we can infer structural and certain semantic interfaces (using whole-program analysis) as done internally by Thaker's safe composition approach [70], Schaefer's compositional type checking of delta modules [63], Ribeiro's emergent feature modularization [61], and the verification approaches by Fisler, Li, et al. [27, 47]. Whereas inferring interfaces lifts some burden from developers, explicitly declared interfaces provide *more local* checks, guarantees, and error messages at the costs of additional specification effort.

4. FEATURE MODULARITY FOR AN OPEN WORLD

After discussing potential modularity costs, let us come back to the open-world versus closed-world discussion to judge

potential benefits of feature modularity.

Feature-oriented software development is often seen as a technique for implementing software product lines. Software product lines are traditionally planned centrally and developed in coordination [9, 59]. The entailing closed-world view might justify focusing on whole-program analysis (or more precisely whole-product-line analysis) and neglecting feature interfaces.

In the product-line context, it makes sense that feature implementations are tied to the base code or to other features of the product line. Product lines reuse a feature's implementation in many products derived from the product line, whereas reusing a feature outside the product line (e.g., using feature *Undo* from Figure 1 in a different context) is rarely a goal.

Of course, developers can significantly benefit from interfaces also in closed systems. But, we conjecture that in a closed-world product-line setting, the benefits of feature modularity (modular reasoning, modular type checking, separate compilation, and so forth) appear less urgent. This may explain, in retrospect, why many researchers rather neglected interfaces for features so far.

The picture changes, when considering an open world. For example, we might want to develop a feature-based library, such as the notorious graph-library example [52], that is supposed to be used in yet unknown contexts. We might even want to reuse a product line inside another separately developed product line [44,62]. Finally, we can consider extensions in open platforms as features, such as plugins in Eclipse, apps in Android, or packages in Debian; these features are independently developed in a software ecosystem [13, 64]. Also in the product-line community, such open-world scenarios gradually gain attention [17, 64, and others]. For example, software ecosystems can also encourage a community process and third-party contributions of features [17].

In all these open-world scenarios, developers deviate from central planning and attempt to reuse features in a potentially distributed setting. Here, feature modularity promises significant benefits, especially when whole-program analysis is no longer acceptable: Even when features are developed independently and unaware of each other, we want to check them modularly. We want guarantees that independently developed features do not change the behavior of other features. We even want to detect inadvertent feature interactions modularly, which are especially problematic in such open-world settings. In this context, research on feature interfaces can pay off.

In an open world, some form of feature modularity seems essential. Whether we need novel fine-grained semantic feature interfaces or whether simpler mechanisms are sufficient remains open. These questions are an interesting avenue for research. We believe that our community should not hide behind scenarios of closed-world product lines. We should investigate benefits and costs also in the light of the closed-world versus open-world debate; eventually we will arrive at different approaches for different scenarios.

5. SHAKY EVIDENCE

So far we have only shaky evidence that could provide insights about costs and benefits and that could guide the design of a module system and of feature interfaces.

On the one hand, we have experienced fine-grained and scattered feature implementations in many product lines [39, 48], and we observed numerous intended structural feature interactions that would require micromodularization [41]. Many further studies have similarly shown the problematic nature of feature interactions [29, 47, 51]. Such studies indicate that feature modularity might produce substantial or even overwhelming costs.

On the other hand, when analyzing the structure of existing feature-oriented implementations (using whole-program analysis), we found that most code fragments are not referencing code from other features [8] and found that feature implementations are mostly cohesive [3]. Both studies can be carefully interpreted as encouragement for feature modularity: Despite all problems, we might be able to hide certain parts of a feature implementation behind an interface.

Furthermore, there are several open systems, such as Eclipse, the Android platform, or the Debian package system, that can be interpreted as further encouragement for feature modularity. Although rather coarse grained and with comparably few interactions, individual plugins, apps, or packages can be regarded as features that are separately developed and modularly compiled and that have explicit interfaces (e.g., extension points, intentions) [64]. At least in this coarse-grained context, modular open systems seem to pay off.

In summary, evidence about potential costs and benefits of feature modularity is not only shaky, but not even conclusive. That makes research about feature modularity even more important. In addition, it shows that we should not focus only on designing module systems and proving properties about them. We need to emphasize empirical research (case studies, data collection, controlled experiments) on the costs and benefits of modularity and need to look at practice.

6. CONCLUSION AND FUTURE DIRECTIONS

Feature modularity has been a long-standing goal of our community. Many approaches have been developed to separate the representation of features and to compose them back again. Yet, feature interfaces, modular type checking, and separate compilation have usually not been the focus—a reason why several researchers have challenged our claims of modularity.

We have learned that conflicts arise due to different notions of modularity. Many researchers are satisfied with locality and cohesion of feature modules and do not see the need for enforcing explicit interfaces. As discussed, feature interfaces come at a cost, especially if fine-grained crosscutting extensions and feature interactions are involved. On the other hand, interfaces promise significant benefits for detecting feature interactions and handling open-world scenarios. Evidence exists both for and against feature interfaces, but is too shaky to draw sound conclusions.

We would like to encourage research toward feature modularity. In summary, we propose the following directions:

- Module systems for feature-oriented programming.
- Low-cost feature interfaces for fine-grained crosscutting extensions.
- Implementation patterns for intended feature interactions.
- Working with micromodularization: lightweight language mechanisms vs. tooling.
- Semantic feature interfaces to detect accidental feature

interactions.
- Modularity for feature models.
- Empirical evaluation of costs and benefits of feature modularity mechanisms in a closed/open world.

Acknowledgements. We thank all the colleagues and reviewers who challenged our view on modularity over the years. In particular, we thank Shriram Krishnamurthi and the anonymous FOSD reviewers for their constructive feedback on this submission. Most recently, a heated discussion in a session about modularity at Dagstuhl #11021, which went completely haywire, served as a catalyst to clarify the root of many arguments and misunderstandings and paved the way for constructive discussions; we thank all participants. This research was supported by the ERC grant #203099 and DFG grants AP 206/2 and AP 206/4.

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. France. Comparing approaches to implement feature model composition. In *Proc. European Conf. Modelling Foundations and Applications (ECMFA)*, volume 6138 of *Lecture Notes in Computer Science*, pages 3–19. Springer-Verlag, 2010.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer-Verlag, 2005.

[3] S. Apel and D. Beyer. Feature cohesion in software product lines: An exploratory study. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 421–430. ACM Press, 2011.

[4] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 32(5):1–33, 2010.

[5] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[6] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[7] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231. IEEE Computer Society, 2009.

[8] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access control in feature-oriented programming. *Science of Computer Programming (Special Issue on Feature-Oriented Software Development)*, 2012. in press.

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

[10] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 48–57. ACM Press, 2003.

[11] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng. (TSE)*, 30(6):355–371, 2004.

[12] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in Java. *Software: Practice and Experience*, 33(5):461–480, April 2003.

[13] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wąsowski, and S. She. Variability in software ecosystems: How to manage and how to encourage? 2011. *under review*.

[14] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 73–82. ACM Press, 2010.

[15] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 21(4):813–847, 1999.

[16] C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. Parameterized interfaces for open system verification of product lines. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society, 2004.

[17] J. Bosch. From software product lines to software ecosystems. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 111–119. Carnegie Mellon University, 2009.

[18] K. H. Britton, R. A. Parker, and D. L. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 195–204. IEEE Computer Society, 1981.

[19] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[20] L. Cardelli. Program fragments, linking, and modularization. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 266–277. ACM Press, 1997.

[21] W. Chae and M. Blume. Building a family of compilers. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 307–316. IEEE Computer Society, 2008.

[22] W. Chae and M. Blume. Language support for feature-oriented product line engineering. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 3–10. ACM Press, 2009.

[23] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 539–554. ACM Press, 2010.

[24] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: A rational module system for Java and its applications. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 241–254. ACM Press, 2003.

[25] 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)*, pages 211–220. ACM Press, 2006.

[26] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1), 2011. to appear.

[27] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 152–163. ACM Press, 2001.

[28] K. Fisler and S. Krishnamurthi. Decomposing verification around end-user features. In *Proc. Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 74–81. Springer-Verlag, 2005.

[29] R. J. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[30] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 110–119. ACM Press, 2000.

[31] F. Heidenreich, I. Şavga, and C. Wende. On controlled visualisations in software product line engineering. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 303–313. Lero, 2008.

[32] S. S. Huang and Y. Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 33(2):6:1–44, 2011.

[33] D. Hutchins. *Pure Subtype Systems: A Type Theory For Extensible Software*. PhD thesis, University of Edinburgh, 2009.

[34] M. Jackson and P. Zave. Distributed feature composition: A

virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng. (TSE)*, 24(10):831–847, 1998.

[35] D. Janzen and K. De Volder. Programming with crosscutting effective views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 195–218. Springer-Verlag, 2004.

[36] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Pittsburgh, PA, 1990.

[37] K. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Software*, 19:58–65, July 2002.

[38] C. Kästner and S. Apel. Virtual separation of concerns – A second chance for preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009.

[39] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[40] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2011. accepted for publication.

[41] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 181–190. Carnegie Mellon University, 2009.

[42] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–58. ACM Press, 2005.

[43] K. Klose and K. Ostermann. Modular logic metaprogramming. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 484–503. ACM Press, 2010.

[44] C. W. Krueger. New methods in software product line development. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 95–102. IEEE Computer Society, 2006.

[45] T. Kühne. *A Functional Pattern System for Object-Oriented Design*. PhD thesis, Darmstadt University of Technology, 1999.

[46] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[47] H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 195–204. IEEE Computer Society, 2002.

[48] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

[49] B. Liskov. Modular program construction using abstractions. In *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 354–389. Springer-Verlag, 1979.

[50] J. Liu, S. Basu, and R. R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18:39–76, March 2011.

[51] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.

[52] R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proc. Int'l Conf. Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer-Verlag, 2001.

[53] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer-Verlag, 2005.

[54] D. MacQueen. Modules for Standard ML. In *Proc. Conf. LISP and Functional Programming (LFP)*, pages 198–207. ACM Press, 1984.

[55] B. Meyer. Applying "design by contract". *Computer*, 25:40–51, October 1992.

[56] H. Ossher and P. Tarr. On the need for on-demand remodularization. In *ECOOP Workshop on Aspects and Dimensions of Concerns*, June 2000. published online.

[57] K. Ostermann. Reasoning about aspects with common sense. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 48–59. ACM Press, 2008.

[58] K. Ostermann, P. G. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2011. to appear.

[59] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin/Heidelberg, 2005.

[60] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[61] M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent feature modularization. In *Companion Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 11–18. ACM Press, 2010.

[62] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. University of Duisburg-Essen, Jan. 2010.

[63] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 43–56. ACM Press, 2011.

[64] K. Schmid. Variability modeling for distributed development: A comparison with established practice. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2010.

[65] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A tool for navigating highly configurable system software. In *Proc. AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, page 9. ACM Press, 2007.

[66] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 481–497. ACM Press, 2006.

[67] 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)*, 20(1):Article 1; 43 pages, 2010.

[68] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 166–175. ACM Press, 2005.

[69] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society, 1999.

[70] 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)*, pages 95–104. ACM Press, 2007.