FeatureCoPP: Compositional Annotations

Jacob Krüger
Harz University of Applied Sciences, University Magdeburg, Germany
jkrueger@hs-harz.de|ovgu.de

Ivonne Schröter
University Magdeburg, METOP GmbH Magdeburg, Germany
ivonne.schroeter@ovgu.de|metop.de

Andy Kenner
METOP GmbH Magdeburg, Germany
andy.kenner@metop.de

Christopher Kruczek
METOP GmbH Magdeburg, Germany
christopher.kruczek@metop.de

Thomas Leich
Harz University of Applied Sciences, Germany
tleich@hs-harz.de

Abstract

Software product lines can be implemented with different techniques. Those techniques can be grouped into annotation-based and composition-based approaches, with complementary strengths and weaknesses. A combination seems useful to utilize benefits of both groups but using two techniques in parallel may cause new problems. To our knowledge, there is no approach that integrates composition into an annotation-based approach or vice versa. We propose the use of an extended preprocessor to introduce physical separation of concerns similar to feature-oriented programming. In this paper, we *i)* present a preliminary implementation that integrates composition into annotation, *ii)* analyse its benefits and shortcomings, and *iii)* discuss implementation and tooling. Overall, we enable developers to keep on using familiar preprocessors but also to benefit from composition. Finally, we show the potential of our approach.

Categories and Subject Descriptors D.3.4 [*Processors*]: Preprocessors; D.3.3 [*Language Constructs and Features*]: Modules, packages; D.2.13 [*Reusable Software*]: Domain engineering

Keywords software product line, composition, annotation, feature orientation, preprocessor

1. Introduction

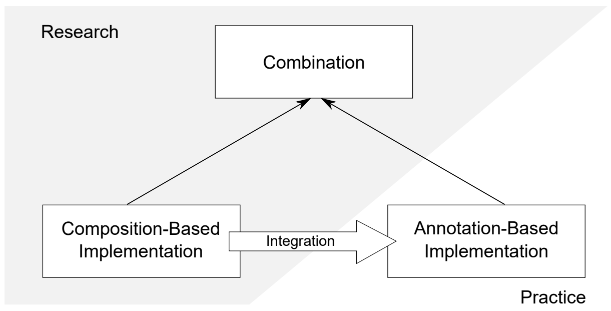
Software product lines are a concept to develop similar programs from a common code base, utilizing systematic reuse [2, 10, 39]. Different variants are described by their *features* that represent characteristics of a system [2]. In particular, features specify commonalities and variability within the product line. Developers select a valid set of features to configure and customize a variant. By combining reuse and variability, product lines enable companies to apply *mass-customization* and promise several benefits, such as reduced costs and faster time to market [2, 39].

For the implementation of product lines, several techniques exist [2, 18], for instance feature-oriented programming [40] or preprocessor directives [23]. They all have benefits and limitations that often complement each other [2, 22, 24]. The underlying mechanisms follow similar ideas and can be grouped in two classes. *Annotation-based* approaches separate features only *virtually*. Variability is annotated in the code base and removed during instantiation. In contrast, *composition-based* approaches *physically* separate features, using different modules that are composed during instantiation. To utilize the complementary benefits, Kästner and Apel [22] propose to combine both approaches. They focus on the idea of introducing an additional implementation technique on top of preprocessors. In the following, we refer to this as *combined approach* that adds a new *implementation layer*.

We illustrate the context of feature-oriented implementation techniques in Figure 1. In practice, composition-based approaches are rarely used [2, 22, 23]. Companies fear effort and risks, especially, as they already use annotations to enable variability [2, 19, 22, 34]. Because composition-based approaches will hardly establish in industry, combinations of both techniques also seem to be less interesting for practice. Hence, to introduce composition in practice it is more promising to improve annotations.

Our approach, ***Feature Co****mpositional* ***P****re****P****rocessor* (FeatureCoPP), enables physical separation of concerns for preprocessors. Such a technique is relevant for practice as it enables composition for a familiar approach. We are aware of several scenarios in which our approach can be useful. For instance, refactoring of annotated product lines to utilize,

FOSD'16, October 30, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4647-4/16/10...$15.00
http://dx.doi.org/10.1145/3001867.3001876


Research
Combination
Composition-Based Implementation
Integration
Annotation-Based Implementation
Practice
Figure 1: Feature-oriented implementation techniques.

or to migrate towards, composition. Another scenario, is to adopt product lines from legacy systems (the *extractive approach* [27]) [22]. Overall, we lower the *adoption barrier* [9] for composition-based approaches.

In this paper, we present an integrated implementation concept that introduces composition in an annotation-based approach. More precise, our proposal is to refine preprocessors to support physical separation of concerns similar to feature-oriented programming. With an integrated approach, developers are able to utilize benefits of annotation and composition, applying physical separation if suitable. In detail, we render the following contributions:

- We introduce concept and idea of integrating composition in an annotation-based implementation technique.
- We discuss previously reported quality criteria and analyse them with regard to our approach.
- We provide an overview of a preliminary implementation and challenges of our approach.

The remaining article is structured as follows. In Section 2, we introduce annotation-based and composition-based implementation approaches. Afterwards, we present our idea and the variability that it enables in Section 3. Then, we analyse quality criteria for our approach in Section 4 and discuss a preliminary implementation and its problems in Section 5. Thereafter, we provide a brief overview of related work in Section 6 and conclude in Section 7.

2. Implementation Approaches

Software product lines can be implemented with several techniques. In the following, we provide a brief introduction of annotation-based and composition-based approaches.

2.1 Annotation-Based Implementation

Annotation-based approaches mark code that belongs to a feature accordingly [2]. Code mapped to non-selected features is either removed during compilation or ignored at runtime. This way, only a single code base is necessary and includes all variable code. In Figure 2 we illustrate an annotated `Hello World` example provided in *FeatureIDE* [30]. Figure 2a shows the implementation of different features. The variable code is encapsulated in `if-end` comments of

```
public class Main {
  public static void main(String[] args){
    /*if[Hello]*/
    System.out.print("Hello");
    /*end[Hello]*/
    /*if[Beautiful]*/
    System.out.print(" beautiful");
    /*end[Beautiful]*/
    /*if[Wonderful]*/
    System.out.print(" wonderful");
    /*end[Wonderful]*/
    /*if[World]*/
    System.out.print(" world!");
    /*end[World]*/
  }
}
```

(a) Basic implementation of annotation-based variability.

```
public class Main {
  public static void main(String[] args){
    System.out.print("Hello");
    System.out.print(" beautiful");
    System.out.print(" world!");
  }
}
```

(b) Processed variant for features `Hello`, `Beautiful`, and `World`.

Figure 2: Annotated code a) in implementation and b) after instantiation using Munge.

the *Munge*[1] preprocessor. We can generate different variants by removing unselected features (e.g., `Hello`, `Beautiful`, `Wonderful`, or `World`) [18]. The result is a single code base as we display in Figure 2b.

There are several annotation-based implementation techniques, such as the `C` preprocessor, *XVCL* [20], or *Spoon* [38]. Especially, preprocessors are commonly used in industrial product lines [2, 22, 34]. This can be explained with their simplicity, flexibility, and that some programming languages include them [2]. For instance, the `C` preprocessor is one of the most successful variability mechanisms in open-source and industry [19]. However, there are several disadvantages, such as *scattering* and *tangling* of feature code, or missing, since not intended, physical separation of concerns [2, 19, 22, 34]. In our work, we aim to extend the existing idea of preprocessors to support composition.

We remark that we use Munge for our examples due to availability and simplicity. In contrast, our further analysis is based on the `C` preprocessor mainly for two reasons. First, it provides far more functionality. Second, the `C` preprocessor and its usage are already discussed in several works.

2.2 Composition-Based Implementation

Composition based approaches separate feature code physically, excluding them from the basic implementation [2]. Thus, several modules exist that specify more detailed functionality of a program. Those modules are composed to instantiate a customized variant. In Figure 3 we show a compo-

[1] `https://github.com/sonatype/munge-maven-plugin`, 30.07.2016

```java
public class Main {
  protected void print() {
    System.out.print("Hello");
  }
  public static void main(String[] args){
    new Main().print();
  }
}
public class Main {
  protected void print(){
    original();
    System.out.print(" beautiful");
  }
}
public class Main {
  protected void print(){
    original();
    System.out.print(" wonderful");
  }
}
public class Main {
  protected void print() {
    original();
    System.out.print(" World!");
  }
}
```

(a) Basic implementation of composition-based variability.

```java
public class Main {
   private void print__wrappee__Hello() {
    System.out.print("Hello");
  }
   private void print__wrappee__Beautiful() {
    print__wrappee__Hello();
    System.out.print(" beautiful");
  }
  protected void print() {
    print__wrappee__Beautiful();
    System.out.print(" World!");
  }
  public static void main(String[] args) {
    new Main().print();
  }
}
```

(b) Composed variant for features `Hello`, `Beautiful`, and `World`.

Figure 3: Compositional code a) in implementation and b) after instantiation using AHEAD.

sitional `Hello World` example from FeatureIDE. Figure 3a illustrates the implementation of different features. Variable code is physically separated in classes that refine others using the *AHEAD* approach [4] for feature-oriented programming [40]. Selecting a valid configuration results in a composed program as we show in Figure 3b. Even then, most approaches separate the variability in some kind, for instance methods as in this example.

Besides feature-oriented programming, several other approaches exist, for instance aspect-oriented programming [25] or frameworks [2]. In this paper, we aim to utilize the idea of feature orientation: physically separate and compose variability. Still, there are some disadvantages compared to annotations. For example, the possible granularity of variable code is coarser, programming languages require extensions, and composition-based approaches are rarly used in practice [2, 22, 23].

3. Compositional Annotations

Combining annotation-based and composition-based approaches promise to utilize advantages of both [22]. While some studies suggest to introduce composition upon annotations [5, 22, 24], our goal is to develop an integrated technique. In this section, we argue that an extension of a preprocessor to support composition is reasonable and provides several benefits. Furthermore, we describe how variability can be implemented.

3.1 Motivation and Idea

Composition can be introduced into preprocessors by using an additional implementation technique. However, this solution implies several challenges, for example:

- A non-integrated compositional technique is used and, thus, adds a new implementation layer. This introduces additional complexity and requires adapted tooling [22].
- Introducing a new approach for compositional variability is error-prone and challenging [5].
- Developers and companies are familiar with preprocessors but not with composition. Hence, they often fear to introduce such approaches despite their benefits [24].

The motivation of our idea is to enable composition without introducing a new implementation technique to overcome those challenges. To the adoption of our approach we define four goals. First, our approach has to be *minimal-invasive* on implementation level. As a result, companies can reuse all of their code. They only have to separate features and map them to their desired positions. Second, *IDE integration* shall be straight-forward. Without tools, companies can hardly introduce new development processes in a structured way. Third, our implementation technique is *language independent*. Thus, we can apply our preprocessor on any programming language or use it on top of existing approaches. Finally, we aim to use a *simple implementation*, such as preprocessors. As a result, companies can decide if and when they want to use new annotations while retaining their current implementation. More important, we do not want to add a third implementation layer as a combination would do.

Overall, our idea is similar to *macros* in the `C` preprocessor and aspect-oriented programming [25]. However, both have shortcomings and conflict our goals:

- Macros
  - *Code analysis*: With macros it is challenging to enable code analysis [13]. While a specific variant can be tested, family-based analyses on domain artefacts are problematic. Some examples for validations are syntax and type checking, static analysis, or model checking [48]. With our approach we can still validate source code.
  - *Textual replacement*: Macros work on textual level, replacing labels with a defined text. This leads to several

problems, for instance cascading calls or the usage of undefined statements [13]. Additional language dependent tooling is required to address those points.

- Aspect-oriented programming
  - *Language independence*: Aspects require adopted extensions for each programming language [2]. In contrast to our goals, this technique is not language independent.
  - *Fragile-pointcut problem*: Fragile pointcuts are a major problem of aspect-oriented implementations [26]. For instance, if we rename a base method without addressing the according advice, it will not be refined. Also, adding a method with the same name as another one, it will also be refined even if not intended. Our approach can overcome those problems.
- Both
  - *Minimal-invasive refactoring*: Macros are an invasive technique, as we have to switch from code to textual level. Aspect-oriented programming requires new pointcuts, advices, and code structure, for instance consistent renaming. In contrast to both, our approach can extract existing code as it is and label an according position. No further changes are necessary.

We illustrate a basic example of our idea in Figure 4. There, we provide a preliminary implementation but no final solution. In this example, we use naming to identify features in different modules. As we display in Figure 4a, we are able to apply preprocessor annotations as before. This way, fine-grained adaptations, which are hard or even impossible to separate physically, are supported [22–24]. While fine granularity in extensions is rare, it still occurs. Even some coarser extensions are difficult to address with compositions [31]. Also, not all features are suited for physical separation [11].

Despite those points, it is still beneficial to enable composition by modularization [36]. For instance, developers implement similar ideas by separating functionality in different files as alternative to preprocessor annotations [34]. In Figure 4a we show a basic implementation example for our approach. We are able to extract and later compose the features `Beautiful` and `Wonderful`. In this preliminary example, we use the key word `include` to specify the correct context of the variable code. This is similar to other approaches that use point-cuts or refine specific methods. Hence, we enable physical separation of concerns using preprocessors.

An important point of our idea is to apply inlining of variable code [49]. Thus, the processed code, as we illustrate in Figure 4b, is identical to preprocessors, without additional statements or method calls. Compositional approaches that still separate refinements after composition also cause problems with scope-sensitive statements [5]. Inlining enables us to extract not only methods that are later refined but also small code fragments. Hence, we can resolve problems caused by scope-sensitive statements. Another benefit of our concept is the ability to reuse the same variable code at different positions in the code. Our example in Figure 4 only illustrates refinements of a method. However, due to inlining we can place the variable code anywhere for example, to introduce a variable.

```
public class Main {
  public static void main(String[] args){
    /*if[Hello]*/
    System.out.print("Hello");
    /*end[Hello]*/
    /*if[Beautiful] include hook_beautiful*/
    /*if[Wonderful] include hook_wonderful*/
    /*if[World]*/
    System.out.print(" world!");
    /*end[World]*/
  }
}
public class Beautiful {
  public void hook_beautiful() {
    System.out.print(" beautiful");
  }
}
public class Wonderful {
  public void hook_wonderful() {
    System.out.print(" wonderful");
  }
}
```

(a) Idea of compositional annotations.

```
public class Main {
  public static void main(String[] args){
    System.out.print("Hello");
    System.out.print(" beautiful");
    System.out.print(" world!");
  }
}
```

(b) Processed variant for features `Hello`, `Beautiful`, and `World`.

Figure 4: Preliminary implementation of integrating composition into an annotation-based approach a) in implementation and b) after instantiation.

In contrast, Batory et al. [4] found that bug fixing in *Jampack*, which is similar to inlining [49], is error-prone. Bugs must be manually propagated backwards to the original implementation because layer boundaries are not preserved. Still, for our approach it seems feasible to support fixes with tooling. Based on annotations, mapping towards the original implementation can be eased and bug fixing partly automated. While inlining is rarely used in compositional approaches, we argue that its advantages are useful for our approach.

In this section, we described and illustrated our basic idea on a preliminary implementation. Next, we address how we aim to combine annotation-based and composition-based variability on implementation level.

3.2 Implementing Variability

In this section, we discuss how to integrate composition into preprocessors. We argue, that our approach is capable to combine and implement variability mechanisms of both techniques. In addition, we are able to utilize variable code similar to object-oriented methods and integrate the same code several times. This is not supported, neither by existing preprocessors nor feature-oriented programming.

Variability of Preprocessors Preprocessors enable developers to use fine-grained adaptations on the level of single statements (compare with Figure 2) or even letters [2, 23]. The annotations are normally referred to as `#ifdef` statements. However, this summarizes several possible statements, for instance to include new files (`#include`), introduce macros (`#define`), or provide alternatives (`#else`) in the `C` preprocessor [19]. Some of those concepts cannot be implemented the same way with composition-based approaches. As we keep using a preprocessor to implement variability, we are able to apply all of those functionalities without introducing new concepts.

Still, those advantages come at a price. Especially, tangling, scattering, or nesting of variable code can be challenging and easily result in errors [15]. There are several approaches to escape this "*#ifdef hell*" [15, 33], for instance by hiding variability [3]. Another idea is to refactor the features into compositional modules [32]. However, this is error-prone and not a suitable approach for all annotations due to limitations of existing composition-based approaches [5, 32]. We follow the same idea but aim to extend the functionality of preprocessors to support composition directly. Hence, we ease refactorings and spare an additional implementation layer.

Introducing Compositional Variability With feature-oriented programming, existing code is refined with feature modules (compare with Figure 3) [2]. Current implementation techniques refer to a specific class and method that they extend and specify which way they are integrated, for instance by overriding. Similar to this idea and the `#include` instruction of preprocessors, we propose to use statements that enable fine-grained extensions. In contrast to existing approaches, we inline physically separated code (compare with Figure 4), which enables us to introduce more detailed variability. This has several benefits in contrast to introducing an additional implementation layer:

- We are able to separate and later inline feature code of any length at any position. Thus, we are independent of predefined extension points but can introduce them ad-hoc when necessary.
- We do not need to introduce a new concept but use preprocessor annotations to physically separate features. This decreases the complexity of the implementation and is easier to use.
- We can reuse the same feature module at several points by including it again. This is similar to object-oriented methods and can decrease the code size if the same functionality is required multiple times.
- We provide the ability to further separate feature interactions and inline them. Therefore, compared to compositional approaches, we can reduce the number of derivatives and duplicates that are otherwise necessary to implement interactions. This does not solve all problems but provides additional solutions.
- We enable new alternatives to replace complicated or error-prone code constructions. For instance, we can replace `goto` statements that are often considered harmful [12] and are challenging to extract into composition [5].

Summarized, we are able to implement variability in several granularities and styles. In Figure 5 we conceptually illustrate some examples. Centralized, we display a basic code artefact, that is refined with three different features, `Orange`, `Green`, and `Blue`, marked accordingly. Code that is not variable is displayed in grey. `Green` only implements an additional block within the existing code. Identical to existing preprocessors, we encapsulate its variability with a start and end statement. During instantiation the code is removed if `Green` is not selected. Feature `Orange` implements additional code in two separated feature modules. It refines existing code at the position it is called and provides a new class that did not exist before but is required for the feature. During instantiation, the first segment is inlined while the second one is additionally included into the variant. `Blue` illustrates some more capabilities of our approach. It defines a refinement that is used at different positions with the same implementation. Most existing approaches require separated code for each call or workarounds. However, we are able to inline the code at the desired position, utilizing a similar idea as methods in object-oriented programming. Thus, we can refine other features, which we illustrate by the call of `Blue` in `Orange`. We could also separate this feature interaction into an additional module and define calls to this one. Those are only conceptual examples of the possibilities we can utilize with our approach.

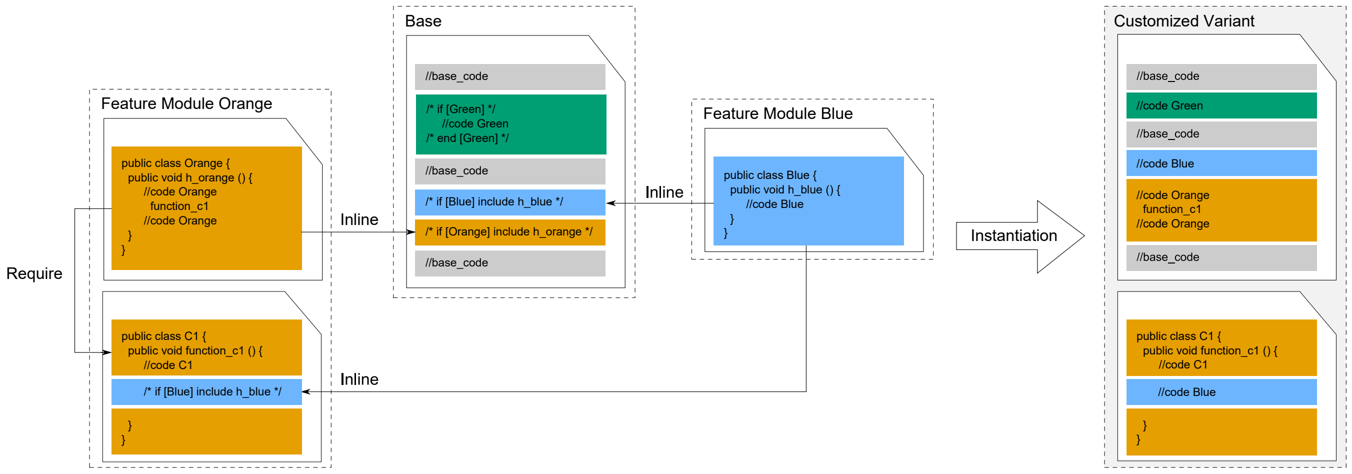
In this section, we presented our idea to introduce composition into a preprocessor. We illustrated several benefits that can ease feature-oriented development. Within the next section, we analyse our proposed approach with regard to several quality criteria.

4. Discussion

In this section, we discuss different characteristics of our implementation approach. Our selection is based on previous discussions [22, 45] and quality criteria [2, 18] that focus on feature-oriented software development. We address *preplanning*, *adoption*, *separation of concerns*, *traceability*, *information hiding*, *granularity*, *uniformity*, and *language independence*. Furthermore, we discuss different aspects of *understandability* regarding our approach. Finally, we provide a brief overview to summarize our analysis.

4.1 Analysis

Several characteristics can be used to evaluate and compare feature-oriented implementation techniques [2]. In the following, we focus on preprocessors and feature-oriented program-



```
public class Orange {
  public void h_orange () {
    //code Orange
    function_c1
    //code Orange
  }
}
```

```
public class C1 {
  public void function_c1 () {
    //code C1
    /* if [Blue] include h_blue */
  }
}
```

```
//base_code
/* if [Green] */
   //code Green
/* end [Green] */
//base_code
/* if [Blue] include h_blue */
/* if [Orange] include h_orange */
//base_code
```

```
public class Blue {
  public void h_blue () {
    //code Blue
  }
}
```

```
//base_code
//code Green
//base_code
//code Blue
//code Orange
 function_c1
//code Orange
//base_code
```

```
public class C1 {
  public void function_c1 () {
    //code C1
    //code Blue
  }
}
```

Figure 5: Examples of refinements with our approach on a conceptual level.

ming. Based on previous discussions [22], we also compare a combination of both with our integrated approach.

Preplanning Effort Product-line engineering always requires preplanning, independently of the used approach and implementation [2]. Still, most techniques try to reduce the required effort by easing later introduction of variability. Using preprocessor annotations, our approach allows to introduce and add features at any time with low effort [2, 22]. For instance, we can easily implement a new feature `Amazing` in our example in Figure 4. There would be no preplanning required to add the necessary code with annotations. Still, large features that introduce several adaptations require planning, which cannot be overcome with any implementation technique. In addition, feature interactions must always be designed carefully. However, our approach enables ad-hoc separation of interacting code and, thus, can ease their design.

Adoption Adoption summarizes the necessary effort and industrial motivation to introduce variability. Therefore, this characteristic is related to preplanning effort. Preprocessors are already well-known in some programming languages and are easy to introduce. In contrast, feature-oriented programming (and other composition-based techniques) are challenging and seldom used in industry [2, 23]. Therefore, combining or integrating both approaches can pay off [22]. Companies can use lightweight annotations to initiate variability at lower risks [9]. Later, the code might be refactored into separated modules using a compositional approach. We see an additional benefit of our integrated approach as it utilizes an already existing technique. The code must only be extracted into modules and marked accordingly but does not require additional concepts, which eases the transition and lowers adoption efforts.

Separation of Concerns Separation of concerns describes a fundamental principle of software design [37]. Features are the concerns of primary interest in product-line engineering [2]. Implementing multiple features in one module is sometimes seen as beneficial [18]. However, their separation into different code artefacts can improve maintenance and evolution of a program [2]. This modularization is intended in most composition-based but not in annotation-based approaches [22]. Feature separation in our approach depends on the implementation used by developers. They can either use the well known `#ifdef` statements or introduce feature modules that are later inlined. Our concept allows the separation of all variable code (even if not always useful). As a result, modularization is even stronger than in many composition-based approaches. For instance, *crosscutting concerns* can additionally be separated by modularizing feature interactions. Still, scattering and tangling are not removed and are still present as labels remain at the desired positions.

Traceability Feature traceability describes the mapping between a feature and its code artefacts [2]. Composition-based approaches support tracing directly as they separate feature code into different modules [22]. Annotations by themselves do not provide traceability but this can be enabled with tooling as all features are marked accordingly [2, 22]. In our approach, variable code does not have to be physically separated. Instead, annotations can be inlined, increasing scattering and tangling. Thus, traceability is not enforced as in compositional approaches. Still, our approach can perform better than combinations of annotation-based and composition-based implementations. For such cases, two different separation styles must be mapped. In contrast, we only use annotations to mark features accordingly. Hence, tracking is simple and existing tools might be able to support traceability of variable code without further extensions.

Information Hiding Information hiding means to separate a module's implementation by only providing an externaly visible part, an *interface* [2]. While some implementation techniques, such as frameworks or components, support information hiding well, others do not. Schröter et al. [43] propose *feature-context interfaces* as implementation of interfaces for

	FOP	PP	Combined	**Our approach**
Preplanning Effort	High	Low	Low	Low
Adoption	◐○	●●	●●	●●
Separation of Concerns	●◐	○○	●○	●◐
Traceability	●●	●◐	●◐	●◐
Information Hiding	●○	○○	●○	◐○
Granularity	◐○	●●	●●	●●
Uniformity	●●	●●	○○	●◐
Language Independence	●◐	●◐	●◐	●◐

●●very good, ●◐good, ●○medium, ◐○poor, ○○no support

Table 1: Comparing feature-oriented programming (FOP), preprocessors (PP), and their combination [2, 22], with our approach.

features. This enables information hiding for composition-based implementation to a certain degree. Our approach uses inlining to include variable code and, hence, does not support information hiding in such cases. Especially, the proposed fine-grained separation partly prevents the usage of interfaces. Still, we are able to implement interfaces and include them if necessary but this heavily depends on the developer and is not enforced. In conclusion, information hiding is weaker than in compositional approaches.

Granularity Granularity describes the level on which variability is implemented, either *coarse-grained* (on top of a hierarchical structure) or *fine-grained* (any lower level) [2]. Introducing fine-grained adaptations with compositional approaches is possible but normally includes replication of code. Our approach can implement fine-grained variability with low effort using `#ifdef` statements. For instance, we can implement the feature `Amazing` with annotations, instead of using a module as before. Later, fine-grained changes can be separated and migrated towards a composition-based approach [22]. Therefore, *disciplined* annotations [32], for example on method granularity, may be helpful [44]. Nevertheless, finer and undisciplined annotations are used in practice. That is why our approach does not enforce disciplined annotations and supports all levels of granularity.

Uniformity The implementation of a feature results in a set of associated software artefacts. Uniformity describes the concept that all those artefacts are similar encoded, regardless of which technique is used [2, 4]. Both, preprocessors and feature-oriented programming, define a set of rules and enforce a common style. However, an approach that combines those two techniques enables developers to use different encodings, `#ifdef` statements and modules, at the same time. Hence, uniformity is not well supported. In contrast, our approach integrates composition into annotations. We only use a single encoding similar to preprocessors. Therefore, we argue that our approach supports uniformity on a similar level as existing preprocessors. Due to the introduction of composition, an additional implementation layer, uniformity is slightly weakened.

Language Independence Preprocessors work directly on textual level and are independent of a specific language. Still, supporting tools, for instance for syntactical testing, require knowledge on the basic implementation and must be adapted. Same accounts for feature-oriented programming that depends on its implementation language but can be adopted with little effort [1]. For a combination of both approaches, the same level of language independence can be achieved but may require more effort [22]. Thus, our approach still utilizes preprocessors and is also completely independent.

We summarize and compare our results in Table 1. A combination of annotation and composition provides several benefits [22]. In particular, we can benefit from a better separation of concerns, fine-grained extensions, and easier adoption. Still, the selection of the most promising approaches and how to combine those has also impact. As we illustrated, we argue that our integrated approach may perform slightly better in some characteristics than a pure combination. For instance, we further improve physical separation or uniformity. We also enable new possibilities in reusing variable code and remove an implementation layer compared to a combination of annotation and composition. However, our approach also has shortcomings compared to annotation, composition, and combinations. For instance, information hiding is challenging.

4.2 Understandability

Understandability is an important factor because developers spend most time during maintenance with understanding code [47], which is causing most costs in software development [8]. In particular, correct identification and removal of bugs becomes problematic with increasing variability [35]. While understanding is often discussed, it is also challenging to assess. For example, a common assumptions is that separation of concerns improves comprehension [45]. However, this is not empirically evaluated, mainly due to the fact that such investigations are difficult and only possible in a small scope, especially for feature-oriented software development [14]. In the following, we preliminary discuss and compare our approach with existing techniques. We focus on understandability of annotations and *code smells*.

Understanding Annotations Preprocessors are often considered as a bad concept in research. Annotations are scattered, tangled, and wrongly used. Therefore, the readability and understandability of code suffers [18, 29, 34, 46]. Sometimes, this situation is even called "*#ifdef hell*" [15]. Still, those problems can partly be resolved by using suited tools [22, 29]. For instance, CIDE [21] enables decomposition and colouring of annotations to ease understanding. It is not clear, to which extent feature-oriented programming can improve this situation. However, some studies found that virtual and physical separation of concerns can improve understandability and development tasks [15, 29, 45]. Thus, while preprocessors by themselves do not support understandability, compositional approaches are slightly improved in this regard, due to physical separation.

We argue that a combined but not integrated approach may decrease the understandability compared to solely using either preprocessors or feature-oriented programming. As we illustrated before, developers have to handle two different implementation techniques. This additional layer requires adapted tooling and disciplined implementation. Overall, it might be challenging to understand and analyse code that is separated in different ways.

In contrast, our approach may improve the understanding compared to pure preprocessors and a combined technique. The compositional layer is an addition that allows fine-grained extraction of variability. We are able to ease the usage of annotations by physically separate features and, thus, better manage the "*#ifdef hell*". This is an improvement to pure preprocessors. Feature-oriented programming may perform better as we also introduce new complexity due to the introduction of composition. As we integrate both layers into a single implementation technique, our approach should also perform better than a combination.

Code Smells Another point to address in regard of understandability are flaws in design and code, so called code smells [17]. Fenske and Schulze [16] analyse those code smells in the context of variable systems. For preprocessors they introduce the *annotation bundle*. This smell describes a large and tangled number of variable statements (annotations) that belong to different features, the "*#ifdef hell*". For example, such implementations are difficult to understand, require knowledge about several features, and complicate maintenance. A counterpart is defined for feature-oriented programming: the *long refinement class*. In this case, an implementation is refined several times, resulting in additional method calls. Therefore, introducing new variability is challenging as developers must analyse all existing refinements and gain knowledge about their structure.

Combined approaches may improve the handling of those code smells. We can freely switch between annotation and composition, depending on the required granularity. Hence, both introduced smells can be partly resolved by replacing annotation with composition and the other way around. Still, as for not combined approaches the design and implementation of code highly depends on the discipline of developers.

Overall, it is unclear how the understandability is influenced by our new instructions. Therefore, further investigations in this regard are necessary. In particular, it might be interesting whether our approach helps to avoid the "*#ifdef hell*". Compared to our integrated approach, a main benefit of using annotation or composition solely, or a combination of both, is the tooling. Existing approaches can rely on well suited and tested tools that can be used and freely combined. As we will describe in the next section, we have to improve and develop new techniques to support our technique.

5. Implementation

The most challenging part for now is the implementation of our approach. Especially, developing appropriate tools beyond existing ones, for instance FeatureIDE [30] or CIDE [21], is a concern. In the following, we categorize our approach along three dimensions of variability implementation to sketch the starting point. Afterwards, we briefly analyse the required tooling and conclude with an analysis of shortcomings of our approach.

5.1 Dimensions of Variability Implementation

Apel et al. [2] describe three dimensions to classify variability implementations. Those are *binding time*, *language versus tool*, and *annotation versus composition*. However, it is difficult to clearly classify our approach for all categories. We summarize our classification in Table 2.

Binding time describes at which time-point features are selected for a variant. We can separate between *static*, selection before the program is executed, and *dynamic*, selection at runtime, binding [41]. By default, preprocessors support static binding. However, they can be adopted to support dynamic binding [42]. We can imagine to enable a preprocessor to refactor annotated features to be selectable at runtime, transforming them to parameter-based variability. Still, this requires additional implementation effort and planning. Overall, we categorize our approach to elementary support static binding. It might be possible to implement code transformation to enable dynamic binding to a certain degree.

Language-based approaches host a programming language that is able to implement features. In contrast, tool-based techniques require external tools for the implementation and configuration of features [2]. Our approach is clearly tool-based. Preprocessors are already an additional tool and require more support to adequately manage and map variability.

The whole idea of our approach is to combine annotation and composition. On the basis, we use an annotation-based technique and extend it to enable composition. It might be arguable to still categorize it as annotation-based approach because we use preprocessors. However, we made strong points that we support composition on several granularities and could even apply it completely. In this case, annotations

Dimension	Classification
Binding time	Static
Language vs. tool	Tool-based
Annotations vs. composition	Integrated combination

Table 2: Implementation dimensions [2].

would only describe points at which features are introduced, similar to other composition-based techniques. Hence, we argue that we provide an integrated combination of both approaches.

5.2 Tooling

An important part of software-product-line engineering is the selection of suitable tools. Managing and mapping variability from requirements over modelling to implementation and testing is a challenging task. In the following, we discuss different steps that are necessary and may require new tools to use our approach. Especially, we focus on variability *implementation*, *modelling*, and *testing*.

The first step towards tooling for our approach is to implement a usable preprocessor. Therefore, we can extend an existing, which might be impractical due to standardisations and complexity, or develop a new preprocessor. For example, companies can utilize existing knowledge and just introduce new annotations. For each new annotation, we have to define algorithms that compose and inline code accordingly. Working on textual level seems to be the most suitable approach. As a result, our approach is independent of a specific language and can also be used on top of other techniques.

Afterwards, we can enable modelling and mapping of variability. Preprocessors are a familiar approach and several tools support them. Still, in practice annotations are fine-grained, scattered, and nested [15, 31, 32]. Our approach may increase those factors as we enable further decomposition. While existing tools, such as FeatureIDE [30], BigLever GEARS [28], or pure::variants [7], are able to handle modelling and mapping for preprocessors, we still have to introduce our newly defined statements. However, this task is straight forward. More challenging might be to visualize all dependencies and interactions that exist within the system. For instance, we may require adopted views [32] or use colouring to separate non-modularized feature code.

Another point to address is correct instantiation and testing of our approach. One specific tool that can be used for preprocessors is CIDE [21]. For instance, it checks syntactical correctness and configurations. While CIDE already supports fine granularity, we argue that it still needs adoption for our approach. In particular, we enable physical separation of concerns even of syntactical wrong code artefacts. For instance, we can extract a context-sensitive statement, such as `return`, without its context. Normally, this would result in error messages by the IDE. However, due to inlining, the final product will still work. Thus, we have to introduce new testing procedures.

Overall, it is clear that we at least must adopt existing IDEs to support our approach. The first step is to define and implement a suited preprocessor. While modelling and mapping seem fairly manageable, we see huge challenges in testing. In the regard of tooling, existing approaches and their combinations have a clear advantage.

5.3 Reflection

Considering the implementation, we also reflected on negative aspects of our approach. In this section, we briefly discuss some obstacles that we identified. We already discussed others, such as understandability or the usage of preprocessors, previously.

Code Validation Many IDEs are capable to validate the correctness of implemented source code on syntactical level. Existing preprocessors have no problem with this because they only provide additional annotations, similar to comments. The code can still be validated. Our approach enables the extraction of modules of any length. For most analysis existing validations can be used but some are more difficult. Depending on the implementation, type checking might be partly achieved but we have to address local variables and scope-sensitive statements. For instance, we can extract a single case in a switch. Still, physically separated `switch`, `case`, and `break` commands are not aware of their scope until composition. Thus, IDEs will throw error messages. A first idea we can imagine is some kind of *virtual inlining* that enables an IDE to correctly validate the code.

Code Fragmentation Our idea is to enable fine-grained physical separation of concerns. This can improve several aspects of feature-oriented software development. However, utilizing our approach can also result in a huge number of small code fragments. The overall number of separated modules can be higher compared to compositional techniques. An increasing amount of small modules may cause several problems. First, testing all combinations becomes more challenging. Second, the fragments must be stored and mapped to assign them to a specific feature. Third, it might be problematic to identify loops or recursion within the fragments. Such problems can be addressed with tooling, consistent development styles, or disciplined annotations. Still, it might be challenging to apply our approach to its fullest extent.

Preprocessor Usage We are aware, that preprocessors and, thus, our approach are seen as an unclean approach. Composition-based implementations are often seen as the better solution because they are integrated into the desired language. However, such techniques are not established in industry. Our approach has a far better chance as we extend a widely used and familiar approach. Hence, we might be able to increase the industrial awareness of composition-based software development.

In this section, we addressed the implementation of our approach. Mainly, we have to address tooling to enable the usage in practice. Still, there are some challenges we have to solve.

6. Related Work

There are some works that focus on the combination of annotation and composition to some extent. In the following, we briefly overview a selection of those.

Batory et al. [4] introduce AHEAD, an approach for feature-oriented development. The proposed *jampack* composes all refinements into a single code base. This idea is similar to inlining but the code is not merged [49]. Our approach does not only compose a single code base but inlines variable code.

A closely related concept to our idea are *classboxes* [6]. Classboxes represent modules that are only visible to specific clients and are inlined on instantiation. However, due to the used separation only classes and methods can be refined or added. In contrast to our approach, fine-grained adaptations are not possible. Furthermore, classboxes are not language independent.

Kästner and Apel [22] discuss the idea of combining annotation-based and composition-based approaches. They also provide examples using AHEAD and CIDE. In contrast to them, we do not aim to combine two approaches but integrate composition into a preprocessor. Our approach may improve some characteristics of feature-oriented software development compared to their idea.

A formal model for refactorings from annotation to composition and vice versa is provided by Kästner et al. [24]. Their case study uses AHEAD and CIDE to illustrate their model and refactorings on different product lines, for example Berkeley DB. We aim to extend an existing implementation technique, the preprocessor, instead of enabling migrations to different approaches.

Finally, Benduhn et al. [5] provide a case study that utilizes the idea proposed by Kästner and Apel [22]. They migrated Berkeley DB, annotated with the `C` preprocessor, towards partial composition. While this was possible, they emphasized that the task is challenging, error-prone, and that not all physical separations can be achieved easily. Again, our idea is to remove the third layer of implementation techniques. As we discussed, we can also solve some of the problems described by Benduhn et al. [5]. In particular, scope-sensitive statements are easier to extract.

7. Conclusion

Implementing software product lines can be done in different ways [2, 18]. While several independent approaches, such as feature-oriented programming [40] or preprocessors [23] exist, a combination of annotation and composition can provide benefits of both [22]. In particular, we can ease the introduction of composition and extraction of product lines from legacy applications. Work focusing on this approach mainly discusses combinations of both approaches using separated implementation techniques. We argue that integrating composition into an annotation based approach can provide additional benefits and solve some problems.

In this work, we described the fundamentals of such a technique based on preprocessors and feature-oriented programming. We discussed the basic idea and illustrated benefits of fine-grained annotation-based composition. Furthermore, we analysed quality criteria and categorizations to put our idea into context. We overviewed necessary tooling and its extension. Finally, we concluded that our integrated approach has some additional benefits but also opens challenges.

In future work, we aim to define and implement a suited preprocessor. Based on its structure, we will adopt tool support. While modelling and mapping are already well supported and can be reused, other parts might be more challenging. For example, testing becomes more challenging foremost because we can extract code into syntactically incorrect modules. Overall, our future work focuses on tool support for all steps in the software life-cycle. In addition, we aim to conduct studies to evaluate the characteristics of our approach.

Acknowledgments

This research is partly supported by DFG grant LE 3382/2-1, BMBF grant 01IS14017A, and Volkswagen Financial Services AG.

References

[1] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *SC*, pages 20–35. Springer, 2008.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[3] D. L. Atkins, T. Ball, T. L. Graves, and A. Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. Softw. Eng.*, 28(7):625–637, 2002.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[5] F. Benduhn, R. Schröter, A. Kenner, C. Kruczek, T. Leich, and G. Saake. Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven Process. In *SOFTENG*, pages 102–109. IARIA, 2016.

[6] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. *SIGPLAN Not.*, 40(10): 177–189, 2005.

[7] D. Beuche. Modeling and Building Software Product Lines with Pure::Variants. In *SPLC*, page 255. ACM, 2012.

[8] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[9] P. Clements and C. W. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Softw.*, 19(4):28–30, 2002.

[10] P. C. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[11] M. V. Couto, M. T. Valente, and E. Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *CSMR*, pages 191–200. IEEE, 2011.

[12] E. W. Dijkstra. Go To Statement Considered Harmful. *Commun. ACM*, 11(3):147–148, 1968.

[13] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170, 2002.

[14] J. Feigenspan, C. Kästner, S. Apel, and T. Leich. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *FOSD*, pages 55–62. ACM, 2009.

[15] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empir. Softw. Eng.*, 18(4):699–745, 2013.

[16] W. Fenske and S. Schulze. Code Smells Revisited: A Variability Perspective. In *VaMoS*, pages 3–10. ACM, 2015.

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[18] C. Gacek and M. Anastasopoules. Implementing Product Line Variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3):109–117, 2001.

[19] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empir. Softw. Eng.*, 21(2):449–482, 2016.

[20] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: Xml-Based Variant Configuration Language. In *ICSE*, pages 810–811. IEEE, 2003.

[21] C. Kästner. CIDE: Decomposing Legacy Applications into Features. In *SPLC*, pages 149–150. IEEE, 2007.

[22] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40. University of Passau, 2008.

[23] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[24] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *GPCE*, pages 157–166. ACM, 2009.

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242. Springer, 1997.

[26] C. Koppen and M. Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *EIWAS*. 2004.

[27] C. W. Krueger. Easing the Transition to Software Mass Customization. In *PFE*, pages 282–293. Springer, 2002.

[28] C. W. Krueger. BigLever Software Gears and the 3-tiered SPL Methodology. In *OOPSLA*, pages 844–845. ACM, 2007.

[29] D. Le, E. Walkingshaw, and M. Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software variation. In *VL/HCC*, pages 143–150. IEEE, 2011.

[30] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool Support for Feature-Oriented Software Development - FeatureIDE: An Eclipse-Based Approach. In *eclipse*, pages 55–59. ACM, 2005.

[31] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*, pages 105–114. ACM, 2010.

[32] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *AOSD*, pages 191–202. ACM, 2011.

[33] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. In *EuroSys*, pages 191–204. ACM, 2006.

[34] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *ECOOP*, pages 495–518. Schloss Dagstuhl, 2015.

[35] J. Melo, C. Brabrand, and A. Wąsowski. How Does the Degree of Variability Affect Bug Finding? In *ICSE*, pages 679–690. ACM, 2016.

[36] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating Features in Source Code: An Exploratory Study. In *ICSE*, pages 275–284. IEEE, 2001.

[37] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[38] R. Pawlak. Spoon: Annotation-Driven Program Transformation - the AOP Case. In *AOMD*, pages 1–6. ACM, 2005.

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

[40] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, volume 1241, pages 419–443. Springer, 1997.

[41] M. Rosenmüller. *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines*. PhD thesis, University of Magdeburg, 2011.

[42] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *GPCE*, pages 3–12. ACM, 2008.

[43] R. Schröter, N. Siegmund, T. Thüm, and G. Saake. Feature-Context Interfaces: Tailored Programming Interfaces for Software Product Lines. In *SPLC*, pages 102–111. ACM, 2014.

[44] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. *SIGPLAN Not.*, 49(3):65–74, 2013.

[45] J. Siegmund, C. Kästner, J. Liebig, and S. Apel. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *FOSD*, pages 17–24. ACM, 2012.

[46] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *USENIX Conference*, pages 185–198. USENIX Association, 1992.

[47] T. A. Standish. An Essay on Software Reuse. *IEEE Trans. Softw. Eng.*, SE-10(5):494–497, 1984.

[48] T. Thüm. *Product-Line Specification and Verification with Feature-Oriented Contracts*. PhD thesis, University of Magdeburg, 2015.

[49] T. Thüm, S. Apel, A. Zelend, R. Schröter, and B. Möller. Subclack: Feature-oriented Programming with Behavioral Feature Interfaces. In *MASPEGHI*, pages 1–8. ACM, 2013.