Physical Separation of Features: A Survey with CPP Developers

Jacob Krüger
Harz University of Applied Sciences
Wernigerode, Germany
Otto-von-Guericke-University
Magdeburg, Germany
jkrueger@hs-harz.de

Kai Ludwig
Harz University of Applied Sciences
Wernigerode, Germany
kludwig@acm.org

Bernhard Zimmermann
Harz University of Applied Sciences
Wernigerode, Germany
bzimmermann@hs-harz.de

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

ABSTRACT

Several implementation techniques for software product lines have emerged over time. A common distinction of these techniques is whether features are annotated in the code base (virtually separated) or composed from modules (physically separated). While each approach promises different pros and cons, mainly annotations and especially the C PreProcessor (CPP) are established in practice. Thus, the question arises, which barriers prevent the adoption of composition-based approaches. In this paper, we report an empirical study among C and C++ developers in which we investigate this issue. Therefore, we ask our participants to describe how they use the CPP and how they assess the idea of moving annotated code into modules. More precisely, we use small examples based on our Feature Compositional PreProcessor (FeatureCoPP) that enables this separation while keeping annotations—avoiding divergences from the preprocessor concept. Overall, we identify different characteristics that indicate when physical separation can be useful. While most responses are skeptical towards the approach, they also emphasize its usability for source code analysis and for implementing specific use cases.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; • **General and reference** → *Empirical studies*;

KEYWORDS

Separation of concerns, product line, empirical study

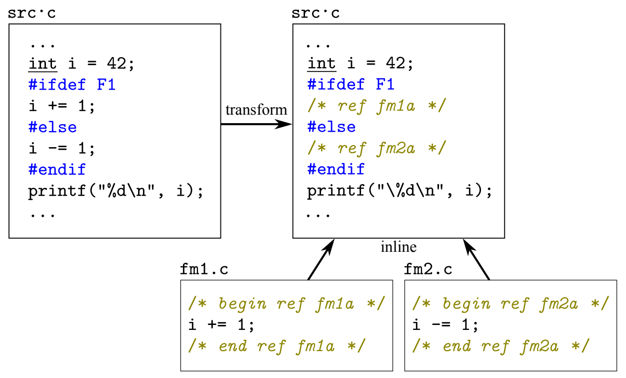
ACM Reference Format:
Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *SAC 2018: Symposium on Applied Computing, April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3167132.3167351

1 INTRODUCTION

Software product lines enable developers to systematically reuse artifacts, reducing costs and allowing to derive customized variants of a system [1, 6, 24]. These artifacts are consolidated into *features*, which represent a specific, user-visible behavior of the software product line [1]. Thus, features represent the commonalities and variabilities between the variants in a software product line. Based on a valid selection (a so-called configuration) of features, a tool automatically instantiates the specified variant.

Several techniques with different pros and cons can be used to implement software product lines [1, 9, 11], for example, runtime variability, components, preprocessor directives, or feature-oriented programming [25]. However, a common distinction of these techniques is whether they *physically separate* features from the code base or only annotate them (*virtual separation*) [1, 14, 16, 28]. In practice, annotation-based approaches, mainly the CPP [12], are used to implement variability [1, 10, 11, 21]. Here, all code is part of the same implementation and removed if the corresponding feature is not selected. Composition-based approaches, such as feature-oriented programming [25] and aspect-oriented programming [13], promise improved maintainability and traceability, due to the physical separation of features into modules. Despite these promises, they are hardly found in practice, especially because preprocessors are simpler, more flexible, and already included in some programming languages [1]. While preprocessors are often associated with the so-called *#ifdef hell* [17, 20, 29], it is also unclear if physical separation provides benefits in this regard [28].

To address this issue, experiments have been proposed to measure the impact of virtual and physical separation of features [8, 28]. First preliminary results show potential benefits but are not reliable, due to the small number of participants. In this work, we investigate the physical separation



```
...
int i = 42;
#ifdef F1
i += 1;
#else
i -= 1;
#endif
printf("%d\n", i);
...
```

```
...
int i = 42;
#ifdef F1
/* ref fm1a */
#else
/* ref fm2a */
#endif
printf("\%d\n", i);
...
```

```
/* begin ref fm1a */
i += 1;
/* end ref fm1a */
```

```
/* begin ref fm2a */
i -= 1;
/* end ref fm2a */
```

Figure 1: Example of the FeatureCoPP concept [16]. CPP code (left) can be transformed towards FeatureCoPP. Then, features are separated into modules (i.e., fm1, fm2) that are inlined at the defined reference points.

of features from another point of view: Instead of designing a controlled experiment, we conduct an online questionnaire with 34 experienced programmers that use the CPP in their daily work. We ask them to provide background information on their usage of the CPP and to give their opinion on different aspects of physically separating features. Thus, we do not focus on quantitative analysis but qualitative responses. To illustrate the concept of physical separation, we use examples based on FeatureCoPP [16], which integrates composition into a preprocessor to ensure uniformity while still relying on an annotation-based approach—with which the participants are familiar. As a result, we assess this integrative approach of FeatureCoPP in particular.

In detail, we contribute the following:

- We report and discuss how the CPP is used in practical settings. The results help to scope studies addressing preprocessors and their analysis. In particular, we find that most of our participants use the CPP to react to external factors, such as the platform, and face no restrictions, such as using only disciplined annotation.
- We investigate how our participants assess the potential of physically separating features in their systems. While the overall results indicate a negative opinion towards this concept, several participants consider it helpful in certain situations. Based on the responses, we derive parameters that can help to decide which features are suitable for physical separation. Furthermore, most of our participants think that physical separation can support code analysis, for example, to overview a feature or to fix bugs.

2 PROBLEM STATEMENT

Several techniques have emerged to implement software product lines [1, 9, 11], which can be separated in two types: With *annotation-based approaches*, the software product line is implemented as a single code base in which variation points are labeled with annotations. Code, which is encapsulated with annotations that correspond to unselected features, is removed from the code base to derive a variant. The most prominent and widely used annotation-based approach is the CPP [7, 12, 19, 21], other techniques being, for instance, Munge [1] or Spoon [22]. For the CPP, annotations are implemented with conditional compilation and defined directives. In contrast, *composition-based approaches* physically separate features from the code base into distinct modules. The resulting feature modules are composed into the base if the corresponding feature is selected.

Both types of techniques promise complementary benefits, due to the underlying concepts [1, 10, 11, 15]. For instance, annotation-based approaches are easier to adopt and allow fine-grained variability. In contrast, composition-based approaches can improve separation of concerns and information hiding. Despite the different benefits and several works investigating a combination of both types [3, 4, 11, 15, 16], only annotation-based approaches (mainly the CPP) are predominant in practice [1, 10, 11, 21].

Thus the question arises, *which reasons hamper the adoption of composition-based approaches in practice?* Empirical studies that investigate these reasons and would allow researchers to derive potential solutions are rare [28]. To address this issue, we conduct an online survey with 34 participants. All of them are experienced with the CPP and, thus, its variability mechanism. With our survey, we aim to assess how they use the CPP and to which extent they consider physical separation of variability to be useful. As our participants are familiar with annotation-based approaches, we rely on small examples based on FeatureCoPP [16] to illustrate the concept of separating features. We display a basic example in Figure 1, where we show how variable code implemented with the CPP (left side) is transformed towards FeatureCoPP (right side). For this purpose, the variable code is separated into modules and during instantiation inlined at the position of the corresponding reference—resembling aspect oriented programming [13]. The idea of FeatureCoPP is to provide a simple and uniform mechanism that integrates composition into the well-known preprocessor concept to facilitate its adoption. An alternative based on the CPP is to use macros to enable this mechanism as *conditional inclusion*, where specific parts of a header file are included depending on a condition. Overall, we contribute insights into obstacles of adopting composition-based approaches in practice and the usability of FeatureCoPP explicitly.

To address our goal, we investigate two research questions:

RQ$_1$ *How do our participants use the CPP?*
With this research question, we aim to identify our participants' main application scenarios of the CPP: Addressing internal (i.e., implementing application specific features) or external (e.g., adaptations to platform or compiler) factors. Furthermore, we want to assess whether they are forced to follow specific restrictions regarding the granularity of annotated code. Answering these questions helps us to put the results of our survey into context and to provide an overview on practices of using the CPP.

RQ$_2$ *How do our participants assess physical separation?* Regarding this question, we exemplify a small code example based on FeatureCoPP, similar to the one we show in Figure 1. To assess the potential of such an approach, we let the participants rate different aspects, such as the comprehensibility and maintainability of such structures. The results help us to identify practical problems and adoption issues of composition-based approaches, explicitly FeatureCoPP.

We remark that we expect more negative responses for the second research question as it introduces an additional mechanism that the developers may not be familiar with. Thus, they may not see benefits in applying physical separation and changing their own development approach. However, we especially focus on such developers as they have a good understanding of variability and can provide insights why they prefer preprocessors.

3 SURVEY DESIGN

Conceptually, we conduct an online questionnaire, for which we describe *participants* and *questions* in this section.

3.1 Participants

Our main criteria when recruiting participants is that they are experienced with the CPP and have also real-life (working) experience. To invite participants, we rely on Google newsgroups, XING, and a local German software development mailing list, wherefore we received 22 responses from Germany. In the initial phase of our questionnaire, each participant is asked to fill out some information regarding their experience. Overall, we receive 35 responses of which we exclude 1, as the participant states to have no programming experience neither with C nor C++.

Of the remaining participants, 19 have a Master or Diploma, 10 a Bachelor, and 2 a PhD degree. Four of these participants' studies seem unrelated to programming as they stated to have no educational qualification in this area. Still, approximately 79.5% of our participants studied programming related courses. Also, while they may not have studied such courses, the remaining participants seem to have gained knowledge based on other education and their daily work.

Considering their experience, we find that on average our participants have approximately programmed for 19.8 years, use C or C++ for 13.9 years, and that programming is part of their job for 13 years. In Figure 2, we illustrate the distribution of our participants' experience. Here, we show the median value, which is close to the average in each case. We see that all of our participants are using C or C++ at least for some years.

Furthermore, we ask our participants to provide a coarse classification of their main tasks in software development. 29 (85.3%) of them state that they work mostly on implementation level, while 3 work in requirements engineering, and 2 on design. Thus, most of our participants should be familiar with the CPP as part of their daily work in programming.

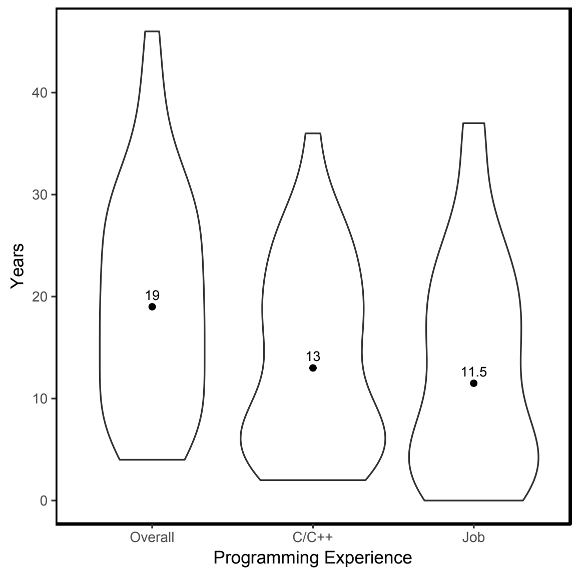


Figure 2: Distribution of the participants' experience in years.

Overall, we argue that our participants are experienced in implementing C and C++. Most of them studied programming related courses, are familiar with these languages for a long time, and are mainly working on implementation level. Thus, the participants' expertise in these programming languages and the CPP provides a reliable base for our analysis.

3.2 Questionnaire

To address our research questions, we use a questionnaire including the questions we show in Table 1. We remark that we abbreviate the displayed answers. In the questionnaire, they are often supported by more detailed descriptions based on small examples similar to the one we show in Figure 1.

The first part of our questionnaire aims at identifying how practitioners use the CPP. This includes questions regarding to which factors (i.e., internal or external) they react with the variability mechanism and which restrictions they face. Based on the first question, we want to identify if the CPP is actually used to implement application features and not just to customize a system to the underlying platform. Regarding the second question, we want to identify whether our participants can use undisciplined annotations.

The second part of our questionnaire addresses the actual physical separation of features. For this purpose, we first describe a simple example based on FeatureCoPP, in which an annotation is used to reference to feature code in a module that is inlined at this position (cf. Figure 1). Thus, the processed code will be exactly the same as before features are physically separated and requires no larger refactoring. Afterwards, we ask our participants several questions, regarding their opinions on comprehension, maintenance effort, and applicability of this techniques.

Table 1: Questionnaire and addressed research questions.

Nr	RQ	Question (Q) & Answers (A)
1	1	**Q:** The C preprocessor (henceforth CPP) allows the conditional lexical inclusion of source code, using so called conditional directives (#if[n]def, #if, #elif, #else). Besides using it as include guard for header files: What is the leading cause for using these directives within your project?
		A: **a)** external factors (platform, compiler); **b)** internal factors (application features); **c)** both (none prevails)
2	1	**Q:** Are there any restrictions in your projects, to which degree conditional directives should enclose syntactical units?
		A: **a)** none (anything can be enclosed); Permitted for **b)** ... statement level; **c)** ... function definition level; **d)** ... large cohesive units; **e)** other (open text)
3	2	**Q:** Do you think this modularized approach can improve the comprehensibility of source code?
		A: Yes, depending on **a)** ... the extracted code size; **b)** ... code complexity; **c)** ... the number of identifiers defined elsewhere; **d)** ... the functional cohesion of extracted code; **e)** no; **f)** other (open text)
4	2	**Q:** How would you assess the maintenance effort regarding the scenario compared to an integrated approach?
		A: Increasing **a)** ... if extracted portion is too small; **b)** ... if definitions of identifiers are scattered too much; **c)** ... since code units are removed from their context; Decreasing **d)** ... since code units are identifiable at first glance; **e)** ... since error-prone locations can be narrowed down quickly; **f)** ... since disposing personal resources on code units is simplified; **g)** other (open text)
5	2	**Q:** How would you categorize this separation approach?
		A: **a)** applicable in code editing; **b)** additional tool for code analysis; **c)** not applicable; **d)** other (open text)

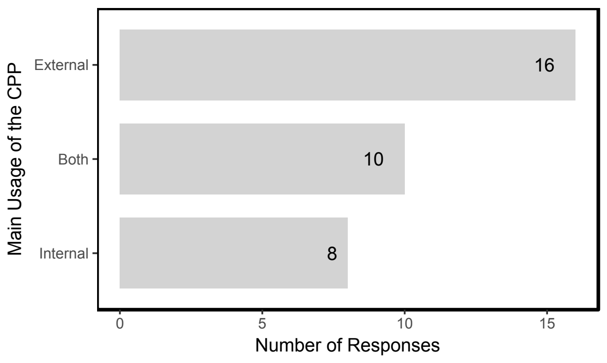


Figure 3: Usage of the CPP to react to factors.

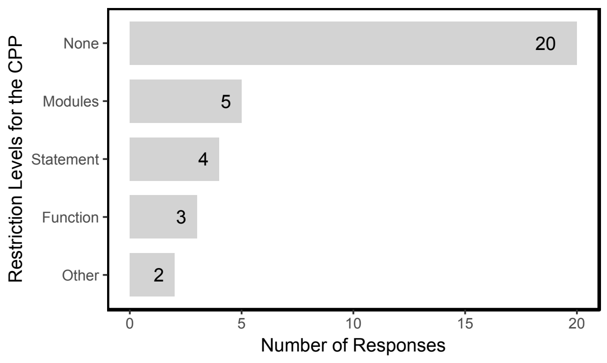


Figure 4: Restriction levels when using the CPP.

4 RESULTS

In this section, we describe the results of our survey. For this, we distinct the findings and discussion for each research question. Afterwards, we consolidate our findings by discussing additional remarks the participants' provide.

4.1 RQ$_1$ - Using the CPP

As we show in Figure 3, 16 participants report that they use the CPP mostly to react to external factors, such as the platform or compiler. For 8 participants the main cause are internal features of the designed system and for 10 neither of both factors prevails. Thus, for most of our participants the main reason for using the CPP is to adapt a system to influences from the outside.

In Figure 4, we display the responses regarding the annotation granularity our participants are allowed to use in their projects. We see, that 20 of them face no restrictions in this regard but can enclose any part of the code with conditional directives. Some participants are only allowed to enclose complete statements (4), functions (3), or even modules (5). Two participants stated different responses, one of them emphasizing that they dislike the CPP but not providing a concrete reason. The other participant faces surprising restrictions: In their case, enclosing code is not permitted at all. Instead, they implement variability in compositional fashion. They utilize a build system to compose features that are implemented in separate compilation units. This already resembles a solution for physical separation, but the participant also remarks their dislike for this solution.

Discussion. As the results show, our participants are using the CPP more often to react to external factors. This seems not surprising, as C and C++ are not platform independent and several compilers as well as standards exist. Variably adapting a system to these factors is necessary to deploy it for different customers. Still, 8 of our participants use the CPP mainly to implement system-specific features and for 10 both factors are of equal importance. Overall, we find

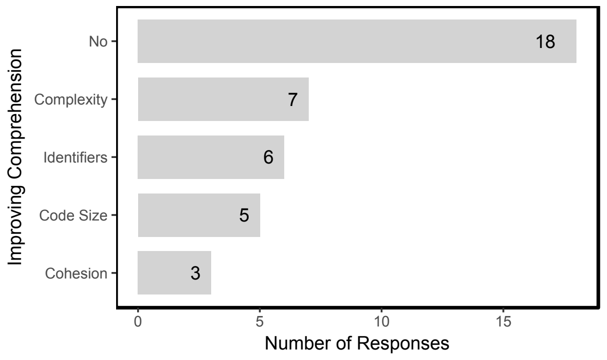


Figure 5: Assessed effect of physical separation on program comprehension.

that the CPP is actually used to react to external factors but also to implement user-visible features of a system. Totally, this suggests that at least 54.4 % of our participants should have a certain awareness of variability in the means of features, instead of considering variability solely as a means for platform adaptation. However, such adaptations seem to be the main cause of using the CPP.

Furthermore, it is important to consider the granularity and discipline with which the developers can apply conditional compilation [10, 18, 19]. Allowing a fine granularity can result in small fractions of variable code that are scattered and tangled among different files. As we see in Figure 4, most of our participants face no restrictions in applying the CPP. This can impact a developer's opinion on physical separation, due to two situations: First, they may emphasize the importance of the code's context as it represents only a small portion of a greater unit. Then, physical separation may complicate the comprehension of code, for example, because identifiers or operations on variables are unknown. Second, they may emphasize the importance of gathering these code parts into a module to get an overview on a feature. This way, traceability and analysis of specific features can be facilitated based on physical separation. Which of these two situations is more likely for a feature depends on different factors that we investigate with our second research question.

4.2 RQ$_2$ - Physical Separation of Features

To answer our second research question, we ask our participants to assess the effect of physical separation on three different aspects. Note that in the following figures the number of responses can add up to more than 100% as we allowed multiple selections. Firstly, *program comprehension* effects how fast a developer understands a piece of code and is the main activity of software developers [27, 30, 31]. Thus, it is important to assess if physical separation can improve the comprehension of a program [8, 28]. As we show in Figure 5, approximately half (52.9%) of our participants does not think that physically separating features can improve comprehensibility. Still, other participants see improvements depending on the code's complexity (7), the number of identifiers that

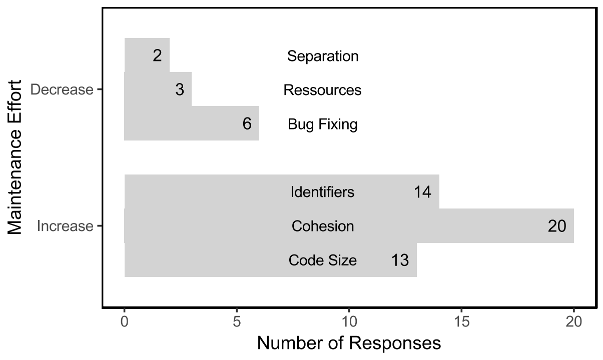


Figure 6: Assessed effect of physical separation on maintenance efforts.

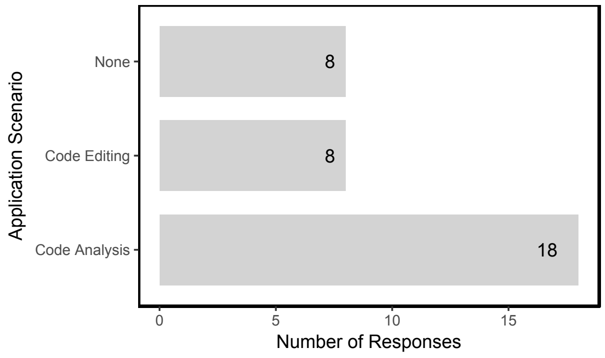


Figure 7: Assessed application scenarios of physically separating features.

are defined elsewhere (6), the actually extracted code size (5), and the overall cohesion of a feature (3).

Secondly, as maintenance is the main cost driver in software development [5, 26, 30], we ask our participants to assess whether the *maintenance effort* would increase or decrease. We display the corresponding results in Figure 6. Here, we see that only 11 participants think that the effort could actually decrease, mainly (6 responses) because error-prone locations could be narrowed down more quickly. Faster resources assignment (3) and separated features (2) are rarely considered to decrease such efforts. In contrast, most responses state that maintenance efforts would increase if the code is too small (13), if identifiers are defined elsewhere (14), and if the code is removed from its context (cohesion, 20).

Finally, we ask the participants to assess if they see any *application scenario* for physical separation. We illustrate the responses in Figure 7. 8 participants state that they see no application scenario for this approach at all, while the same number can imagine it for code editing. The most considered application scenario for physical separation, with 18 responses, is as an additional tool to analyze a system.

Discussion. Considering program comprehension, it seems not unexpected that approximately half of our participants cannot see any improvement when applying physical separation. Especially fine-grained separation on the level of single statements or expressions would cause a loss of contextual information regarding identifiers, data flow, and control flow, thus hampering understandability. Furthermore, our participants are familiar with annotating code to implement variability and physical separation is a less known concept to them. However, 18 of them can imagine physical separation to be beneficial, depending on different factors. Surprisingly, the overall cohesion of the extracted code parts was considered seldom to improve program comprehension. As this is one of the strong points of annotation-based approaches, we assumed that it would be considered to be more important.

For the maintenance effort, even fewer participants see any benefit in physical separation. Most responses substantiate the importance of the aforementioned factors: Too small code units and identifiers that are defined somewhere else are associated with increasing maintenance effort. Interestingly, missing cohesion was rated fewest to affect program comprehension but is considered to be the main factor for increasing maintenance effort. Unfortunately, we cannot derive a meaningful explanation from our study regarding this apparent contradiction. Still, the responses indicate that our participants believe that to improve the understandability of a feature, it does not need to be cohesive unit. However, due to reduced cohesion, it can be unclear how maintenance activities, such as bug fixing or refactoring, affect other parts of the system. Thus, while comprehensibility is not threatened, the maintenance effort could increase. Further studies are necessary to assess this issue in more detail.

Regarding the previous responses of our participants—indicating a more negative opinion towards physical separation of features—we are surprised by their assessment of its application scenarios. Despite the stated problems, 18 participants think that this approach is suitable for additional source code analysis. For example, this includes the options we exemplify in the questionnaire: To summarize the code of specific features or narrow down bugs. This indicates that developers are indeed aware of the potential benefits of physical separation. However, only few of them see this approach as a feasible implementation technique. Based on the responses, we argue that physical separation can be integrated more into practice by first utilizing it for source code analysis. Tools to summarize features can support developers and may improve the awareness for compositional implementation techniques.

4.3 Participants' Remarks

At the end of our questionnaire, we provide a free text field in which our participants can document any additional remarks concerning our study. In the following, we review those responses for a more complete overview on our participants' opinions on physical separation. We remark, that we do not ask for specific projects but that our participants come up with any example by themselves.

First, some participants state remarks that resemble our previous findings. For example, one of them writes:

> "It looks like it could be for small thin[g]s (like the example) a bit too much, but for a large one it could be practical to maintain that in a different file/location."

Furthermore, another participant states:

> "[T]here must be an restriction that no declaration is included and the [...] exported code should have a minimum size, [...] otherwise you have too much structure for too small code size."

These two responses substantiate our previous findings. If physical separation shall be useful, it is necessary that the extracted source code fulfills certain characteristics. For this, the code size, complexity, and content of the module should follow specified rules.

Second, one of the participants provides detailed responses to many of our questions and, thus, shared their concerns with us. For example, considering program comprehensibility (compare with our second research question), they state:

> "It's my opinion that comprehensibility of source code is inversely related to the number of tools required to build it, and to the number of source files which need to be read to understand a particular functional unit. That said, #ifdef-hell kills comprehensibility as well. This is a toss-up for me."

As a general conclusion they write:

> "[...] I'm generally in favor of the process, but I am slightly concerned that excessive automation may lead to proliferation of #ifdef-hell. To pick on one particular project, Vim's code is bad. It should be fixed, not papered over with a tool."

The first response describes one concern we identify for most participants: Scattering the source code among several files looses context and reduces cohesion, which can complicate program comprehension. In addition, the participant states that additional tools make it more challenging for software developers to implement code. Thus, it seems necessary for tools, especially those that aim to address comprehensibility, to be as integrated as possible. We argue that this is also in favor of FeatureCoPP, as it integrates composition into annotations instead of solely combining two different techniques. Overall, this participant sees potential benefits in physical separation. This may also be related to the fact that they apply a similar concept based on build systems, as we describe for our first research question.

Third, one participant sees no benefit in physical separation outside of existing language capabilities and emphasizes to utilize these:

> "C and C++ already have a clean separation of code at function level, adding new separation layers just makes the code less manageable."

Of course this is possible and, in connection with the CPP, resembles the same concept we utilize with FeatureCoPP. This remark indicates that instead of developing language extensions or new paradigms it may be practically more relevant to utilize the capabilities that already exist. However, extending such tools becomes more and more challenging and follows standards, limiting practically useful extensions.

Finally, two interesting remarks are related to the usage of the CPP in general:

"[...] the need for using the preprocessor is at least 10x less in C++ than in C."

"[t]here's very little need to use the preprocessor anymore. It's obsolete."

The first response indicates that further investigations may be necessary to assess to which extent the CPP is more important in C than in C++. Such analyses could help to assess the necessity of preprocessors (and variability mechanisms) in languages with (C++) and without (C) object-oriented mechanisms. While we do not know the participant's actual reason for this assumption, it may indicate that features can be more easily implemented as objects. The second response goes even further and states that there is no need for preprocessors at all. Unfortunately, we do not know why the participant believes the CPP is obsolete nor which programming language (C or C++) they rely on. Potentially, they may face the same situation as the previous participant when using C++ or may rely on another variability mechanism.

5 THREATS TO VALIDITY

In this section, we provide an overview on threats to validity, following the classification described by Perry et al. [23].

Construct Validity. A threat to the construct validity are the used questions and terms. Potentially, the participants may have misunderstood something, due to language barriers and different background knowledge in the domain. However, we ask our participants at the end of the survey to state if they had any problems in understanding but none of them reported ambiguous questions. Thus, we argue that the construction of our survey is not threatened.

Internal Validity. There are other usage scenarios and factors in which physical separation may be more helpful than in these we assess. Due to this limitation, our findings can be biased. We focus especially on program comprehension and maintenance effort as these are crucial in software development. To this end, our survey provides reliable insights.

Another threat to the internal validity is that we solely rely on subjective assessments. We did not derive any data from an experiment or interviews but ask our participants for their opinion. Thus, there may be a discrepancy between individual opinions, measurable effects, and qualitative interviews. This threat also includes that we have to trust the opinions of a small set of participants. We argue that this is not a big threat, as considering the opinions of representative, experienced developers is a reliable source of information. Still, we cannot generalize the findings to a global level.

External Validity. The main threat to the external validity of our survey is the sample of participants: We focus only on C and C++ but no other software developer. This biases our findings, which are only applicable in this context. Still, our goal is to receive qualitative responses on the usability of physical separation of features. We argue that our participants are familiar with variability mechanisms and can at least imagine this concept. Also, our goal is to identify problems in adopting physical separation in practice and such developers are suitable participants for this purpose.

Reliability. While there are some threats to this work, we think that any researcher can replicate it. The results may change depending on the participants and questionnaire. However, this is the case for all empirical studies and is not a threat for our survey. Replications and extensions can help to consolidate our findings.

6 RELATED WORK

A closely related study to ours is reported by Siegmund et al. [28]. The authors propose an experiment to compare physical and virtual separation of features. Still, due to the small number of participants, the authors could not derive meaningful results. In contrast to Siegmund et al. [28], we rely on qualitative analysis in the form of a questionnaire.

Several works address combinations of composition-based and annotation-based implementation techniques. Kästner and Apel [11] compare different factors of both approaches and identify pros and cons. They propose to combine composition and annotations and, thus, aim to utilize the benefits of these approaches. Our work is complementary to this, as our results can help to refine such assessments and provide practitioners knowledge as input. Based on this work, some authors investigated how to implement such combinations [3, 15]. They face different problems and challenges, connected to combining composition and annotations, as well as applying this concept in practice. Here, our work can help to scope such works to practitioners' needs.

Behringer [2] follows another idea than the aforementioned works. Instead of combining different implementation techniques, he uses projectional editing, which enables developers to switch between an annotated and a modularized representations of the system without changing the underlying code. While this promises additional benefits, it is questionable how fast this approach can be adopted by practitioners and if they see any benefit in it. To this end, our work may help to refine the proposed approach and improve its applicability.

Some works address the granularity and discipline of CPP directives [18, 19, 21]. The authors compare different systems and how annotations are applied, identifying commonalities between most systems and proposing improvements on using the CPP. Thus, these works can help in further investigations to analyze the factors our participants consider important.

7 CONCLUSIONS

In this paper, we report an empirical study conducted with 34 experienced developers. We investigate how the CPP is used in practice and if our participants see benefits in physically separating features. Overall, we conclude the following:

- The CPP is used more often to react to external factors, such as underlying platforms or compilers.
- Most developers face no restrictions on how they use the CPP. Accordingly, they can use undisciplined and

fine-grained annotations that are often considered to be problematic.

- Physically separating features is rarely desired for actually implementing source code. If this separation shall help to improve comprehension or reduce maintenance efforts, different factors, such as the code size, complexity, and cohesion, must be assessed.
- In contrast, many of our participants can imagine physical separation to support code analysis, for example to collect and investigate all code belonging to a feature.
- Some surprising statements indicate that the CPP may be replaced by developers with other approaches, for example, in the object-oriented C++. Here, the question arises, due to which reasons and how the preprocessor is replaced.

To summarize the overall outcome and problems, a response by one of our participants seems to hit the mark:

> "[...] the example was too long and there was no convincing argument why someone should do that. Maybe your idea is good for a specific use case, but this use case is not obvious for me."

Based on this statement, it seems necessary to improve the understanding and practical applicability of composition, as it can help to resolve problems of annotation-based approaches.

Based on our findings, several further works are interesting. In the future, we aim to conduct additional studies and develop concepts on physically separating features from source code. Here, we will identify the extent and scenarios for which this can be useful. In addition, analysis on how developers use the CPP in C and C++, focusing on differences and potential workarounds, will be part of our work.

ACKNOWLEDGMENTS

This research is supported by DFG grant LE 3382/2-1 and Volkswagen Financial Services AG.

REFERENCES

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.
[2] Benjamin Behringer. 2017. *Projectional Editing of Software Product Lines - The PEOPL Approach*. Ph.D. Dissertation. University of Luxembourg.
[3] Fabian Benduhn, Reimar Schröter, Andy Kenner, Christopher Kruczek, Thomas Leich, and Gunter Saake. 2016. Migration from Annotation-Based to Composition-Based Product Lines: Towards a Tool-Driven Process. In *International Conference on Advances and Trends in Software Engineering*. IARIA, 102–109.
[4] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. 2005. Classbox/J: Controlling the Scope of Change in Java. *SIGPLAN Notices* 40, 10 (2005), 177–189.
[5] Barry W. Boehm. 1981. *Software Engineering Economics*. Prentice-Hall.
[6] Paul C. Clements and Linda M. Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.
[7] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering* 28, 12 (2002), 1146–1170.
[8] Janet Feigenspan, Christian Kästner, Sven Apel, and Thomas Leich. 2009. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *International Workshop on Feature-Oriented Software Development*. ACM, 55–62.
[9] Critina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. *ACM SIGSOFT Software Engineering Notes* 26, 3 (2001), 109–117.
[10] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2016), 449–482.
[11] Christian Kästner and Sven Apel. 2008. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*. University of Passau, 35–40.
[12] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall.
[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*. Springer, 220–242.
[14] Jacob Krüger. 2017. Lost in Source Code: Physically Separating Features in Legacy Systems. In *International Conference on Software Engineering*. IEEE, 461–462.
[15] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. Composing Annotations Without Regret? Practical Experiences Using FeatureC. *Software: Practice and Experience* (2017).
[16] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development*. ACM, 74–84.
[17] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Symposium on Visual Languages and Human-Centric Computing*. IEEE, 143–150.
[18] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *icse*. ACM, 105–114.
[19] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *International Conference on Aspect-Oriented Software Development*. ACM, 191–202.
[20] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. 2006. A Quantitative Analysis of Aspects in the eCos Kernel. In *SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 191–204.
[21] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *European Conference on Object-Oriented Programming*. Schloss Dagstuhl, 495–518.
[22] Renaud Pawlak. 2005. Spoon: Annotation-Driven Program Transformation - the AOP Case. In *Workshop on Aspect Oriented Middleware Development*. ACM, 1–6.
[23] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. 2000. Empirical Studies of Software Engineering: A Roadmap. In *Conference on The Future of Software Engineering*. ACM, 345–355.
[24] Klaus Pohl, Günter Böckle, and Frank van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.
[25] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming*. Springer, 419–443.
[26] David Sharon. 1996. Meeting the Challenge of Software Maintenance. *IEEE Software* 13, 1 (1996), 122–125.
[27] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 13–20.
[28] Janet Siegmund, Christian Kästner, Jörg Liebig, and Sven Apel. 2012. Comparing Program Comprehension of Physically and Virtually Separated Concerns. In *International Workshop on Feature-Oriented Software Development*. ACM, 17–24.
[29] Henry Spencer and Collyer Geoff. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *USENIX Conference*. USENIX Association, 185–198.
[30] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* 5 (1984), 494–497.
[31] Rebecca Tiarks. 2011. What Maintenance Programmers Really Do: An Observational Study. In *Workshop on Software Reengineering*. 36–37.