# Effects of Explicit Feature Traceability on Program Comprehension

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany
jkrueger@ovgu.de

Gül Çalıklı
Chalmers | University of Gothenburg
Gothenburg, Sweden
calikli@chalmers.se

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden
bergert@chalmers.se

Thomas Leich
Harz University & METOP GmbH
Wernigerode & Magdeburg, Germany
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke University
Magdeburg, Germany
saake@ovgu.de

## ABSTRACT

Developers spend a substantial amount of their time with program comprehension. To improve their comprehension and refresh their memory, developers need to communicate with other developers, read the documentation, and analyze the source code. Many studies show that developers focus primarily on the source code and that small improvements can have a strong impact. As such, it is crucial to bring the code itself into a more comprehensible form. A particular technique for this purpose are explicit feature traces to easily identify a program's functionalities. To improve our empirical understanding about the effect of feature traces, we report an online experiment with 49 professional software developers. We studied the impact of explicit feature traces, namely annotations and decomposition, on program comprehension and compared them to the same code without traces. Besides this experiment, we also asked our participants about their opinions in order to combine quantitative and qualitative data. Our results indicate that, as opposed to purely object-oriented code: (1) annotations can have positive effects on program comprehension; (2) decomposition can have negative impact on bug localization; and (3) both techniques are considered beneficial. Moreover, none of the three code versions yields significant improvements on task completion time. Overall, our results indicate that lightweight traceability, such as using annotations, provides developers immediate benefits during software development without extensive training or tooling; and can improve current industrial practices that rely on heavyweight traceability tools (e.g., DOORS) and retroactive fulfillment of standards (e.g., ISO-26262, DO-178B).

## CCS CONCEPTS

• **General and reference** → *Empirical studies*; • **Software and its engineering** → *Software design tradeoffs*; *Maintaining software*.


*ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia*

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338968

## KEYWORDS

Program comprehension, Feature traceability, Software maintenance, Separation of concerns

**ACM Reference Format:**
Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338968

## 1 INTRODUCTION

Developers often need to understand the purpose and the details of specific parts of a codebase, which is a time-consuming and cognitively demanding activity during software engineering [32, 59, 60]. A developer performs this activity, known as *program comprehension*, when they are new to a program or forgot details that are required for their task [8, 30]. Consequently, to gain implicit knowledge about a program, developers need to read and comprehend the code, which can be facilitated by mentoring and by explanations from other developers. However, communicating knowledge in such a way requires considerable effort from other developers and interrupts their other activities.

To tackle such problems, several techniques have been proposed to reverse-engineer information or to improve program comprehension, often upon empirical studies. Contemporary techniques comprise, for instance, creating on-demand documentation [46], topic modeling [62], and visualizing execution traces [14]. Still, developers are known to mainly focus on the source code itself, rather than documentation and other artifacts [6, 32, 50, 56]. Consequently, bringing the source code itself into a more understandable form is crucial to support program comprehension and to improve the software design. Several concepts and techniques have been proposed for this purpose, such as programming paradigms (e.g., object-orientation [3], feature-orientation [21]), code recommendations (e.g., on identifier names [15, 33], decomposition strategies [26, 58]), and other supportive techniques (e.g., source code comments [38, 65], documentation traceability [1, 36]).

In this paper, we are concerned with a design decision that is often argued to positively impact software development and maintenance: **Explicit traceability of software features in the source code.** Explicit traceability refers to code styles that explicitly mark

what parts of the code belong to what feature. Such explicit locations help developers to faster identify relevant code and understand what the corresponding feature does. As we report in Section 2, some studies indicate a positive effect of explicit traces on program comprehension. However, these studies are usually conducted as controlled experiments with a small number of students and involve special implementation techniques, such as feature-oriented programming [43]. In contrast, we (i) conducted an experiment including 49 experienced, professional software developers; (ii) used feature traces that are independent of implementation techniques; and (iii) compared both types of traces not only to each other, but also to object-oriented code without any traces.

For this experiment, we randomly distributed all invited developers into three groups, each of which had to perform six tasks on Java code that comprised (1) no feature traces, (2) annotated features or (3) decomposed features. We refer to annotating (i.e., features were commented) and decomposing (i.e., features were implemented in separate classes) as *separation of features* [26, 52]. By using lightweight designs to incorporate feature traces, we did not need to teach our participants a new implementation technique. We did this to reduce learning efforts, which we argue to benefit the usability and introduction of explicit feature traces in practice. The results indicate that, compared to pure object-oriented code, annotations can have a positive impact on understanding features, while decomposition can potentially hamper bug localization. Still, due to our sample size, we have to be careful with interpreting these results, but qualitative responses also indicate a strong favor of most participants towards explicit feature traces. In combination with findings of other researchers on more specialized implementation techniques [23, 26, 49, 52], we argue that explicit feature traces and especially annotations can improve program comprehension and support automation without negatively impacting the time developers need to analyze code.

In summary, our contributions are as follows:

- We report and discuss quantitative data on the correctness and completion time of our participants for six program comprehension tasks.
- We discuss qualitative responses to shed further light into the benefits and problems that our participants faced.
- We provide a replication package that includes our experimental design, the source code of our subject system, and all anonymized responses in a repository.[1]

Our results provide empirical insights into the impact of explicit feature traces on developers' task performance. Especially, as we confront our participants with unfamiliar code, they have no previous knowledge about it and face the scenario of familiarizing with new code and the assigned tasks.

## 2 RELATED WORK

The notion of features has become a fundamental concept, not only to implement variability in software product lines [2, 7], but for software engineering in general—used to communicate, document, and structure systems [4, 27]. In particular, an extensive body of research investigates the task of locating features in the source code of a system [5], automatically [48] as well as manually [24, 61]. Feature location is a time-consuming and costly task that is necessary to maintain or fix—essentially, comprehend—a feature that is not made explicit in the code. The benefits of explicit feature traces are apparent, as they free the developer from locating features in the code, saving time and providing focus points for developers [19].

Research in the related area of requirements traceability is concerned with tracing requirements throughout various artifacts down to the source code of a system. To this end, several techniques have been proposed to recover traces to the source code [9, 39]. Moreover, empirical studies [10, 18, 36, 45] suggest that such traces can significantly facilitate developers' tasks. However, most of such techniques rely on external tools, and requirements are a different abstraction than features. Both can be in any relation to each other, for example, a feature needs to fulfill multiple requirements.

Due to the variety of techniques that can be used to enable feature traceability, an important question arises: *What technique is suitable in what situation to support developers in understanding source code?* In this regard, researchers have compared different feature characteristics and traceability techniques to gain insights. For instance, Liebig et al. [35], Passos et al. [41, 42], Melo et al. [37], and we [25] investigated the characteristics of feature implementations and how these impact maintainability, evolution, and the architecture of a system. Furthermore, Feigenspan et al. [12] analyze whether background colors instead of textual annotations facilitate program comprehension. In contrast, Parnas [40] discusses how to decompose a system into modules or components, another widely used technique to separate and trace features.

Despite such techniques and studies, it is still an open issue how to separate features most effectively. Several authors argue about potential advantages and disadvantages of annotating features in a single codebase versus separating them into modules [22, 29, 34]. Due to the complexity of comparing such implementation techniques and due to psychological biases [54], only few researchers report empirical studies. Siegmund et al. [52] conducted a controlled experiment in which they compare preprocessor annotations and feature-oriented programming. However, this experiment includes only eight students, limiting more general interpretation. In a follow-up experiment on bug fixing [49], 33 students have been involved, but the results show no significant benefits of either technique. Our previous works on this topic include a survey with 34 developers [26], a preliminary analysis of developer communities [23], and a case study [28]. During these works, we have been concerned with annotating and decomposing features to provide insights into the opinions and experiences of developers, but they do not provide experimental evidence on pros and cons of either technique. All these studies focus on specialized implementation techniques for variability, adding complexity and effort for practical usage. Moreover, none of these studies analyzes pros or cons of using any of these techniques compared to not using it.

**Overall, it is still not clear to what extent separating features with one of the basic techniques—annotations or decomposition—impacts a developer's ability to understand a program.** Our goal is to improve the empirical evidence concerning the impact of such explicit features traces. In contrast to previous works, we are not concerned with implementation techniques that

[1] https://doi.org/10.5281/zenodo.3264974

allow variability, but rely on comments and classes that do not require developers to learn new concepts. This also excludes the usage of external tools, as these add further abstractions and developers may be reluctant to use them [26]. Moreover, we are interested in understanding the pros and cons compared to code that does not comprise any feature traces. Nonetheless, our study design is partly inspired by previous studies and guidelines [11, 52, 53].

# 3 EXPERIMENTAL DESIGN

In this section, we describe the *goal*, *subject system*, *implementation*, *distribution* of participants, and *tasks* of our experiment.

## 3.1 Goal & Research Questions

We aimed to empirically assess the impact of explicit feature traces in source code. To this end, we have been concerned with two established techniques: annotations and decomposition into components (cf. Section 2). Arguably, both techniques facilitate feature location, as features are separated and can be easily found through searching their identifiers in annotations or file names, respectively. In order to investigate their impact on program comprehension, we considered the different code versions as *independent variables*, comprising the three levels object-oriented, annotated, and components. Moreover, we aimed to control the participants' programming experience, meaning that we considered the experience as independent and not as confounding variable.

To address our goal, we defined three **research questions**:

**$RQ_1$ To what extent does feature traceability impact the effectiveness of program comprehension?**
We investigated whether annotations or decomposition improve our participants' ability to *correctly* understand code (i.e., effectiveness). To this end, we used the number of faults as metric (*dependent variable*) and compared the ratios of correct solutions between all three code versions.

**$RQ_2$ To what extent does feature traceability impact the efficiency of program comprehension?**
We investigated whether annotations or decomposition facilitate our participants' ability to understand code *faster* (i.e., efficiency). To this end, we measured their completion time (*dependent variable*) for each task.

**$RQ_3$ What is our participants' perception of feature traceability on the performed tasks?**
Besides quantitative measures, we were concerned with our participants' perception of explicit feature traces. In particular, we wanted to understand what problems or benefits they experienced while understanding the source code. Consequently, we addressed this research question based on qualitative responses and mapped those to our quantitative data.

Based on existing studies [49, 52], we **hypothesized** that annotations and decomposition perform comparable to each other. In contrast, we assumed that the explicit traceability of features would facilitate all tasks compared to pure object-oriented code, while the correctness should remain similar. Overall, we defined our *null-hypotheses* that we aimed to refute with our data as follows for the corresponding research questions:

$H_1$ The correctness of our participants' task solutions does not differ between groups.

$H_2$ The efficiency of our participants to complete tasks does not differ between groups.

We tested each hypothesis by comparing two groups to each other (pair-wise) for all of our six tasks (cf. Section 3.4), resulting in a total of 18 tests for each hypothesis (e.g., object-oriented compared to annotations, annotations compared to composition). We corrected our test results to address multiple hypothesis testing (cf. Section 4).

## 3.2 Subject System

As our subject system, we selected Mobile Media, which has been developed by researchers of the software-product-line community [66] and was later extended with feature annotations (using the C preprocessor) that we used as baseline [52]. Due to its careful design and usage of standard coding techniques, it is an appropriate subject system that has been used in several studies [49, 51, 52]. Moreover, it is implemented in Java, which is one of the most common programming languages.

The software provides a content management system for media files on mobile devices. In our experiment, we used a single file, namely *MediaControler.java*, that implements ten features of the software. These features are related to storing and managing photos, music, and videos. To avoid biases, we removed all existing comments in the file. Moreover, we removed library imports, which contribute to approximately 10% of the total lines of code, and an SMS feature, of which only a small part is implemented in this file. We did this to limit the code size that our participants had to read, which was around 400 lines, in the end.

Finally, we refactored the file into three different versions:

(1) *Object-Oriented:* In this version, we only removed the existing preprocessor annotations (e.g., `#ifdef`) to provide the source code without any feature traces. Thus, we obtained pure object-oriented code that we used for the control group.
(2) *Annotated:* For annotation-based feature traces, we replaced existing preprocessor annotations with traceability annotations based on existing studies (i.e., `// &begin [ feature ]`, `// &end [ feature ]`) [19, 25, 27]. We decided to do this, *(i)* as C preprocessor annotations are rarely used in Java programs, *(ii)* to avoid confusion over potential variability we are not interested in, and *(iii)* to not introduce new concepts.
(3) *Components:* To obtain the decomposed version, we extracted each feature into a class and added static methods that comprise the feature's code. For each class, we used the corresponding feature's name as file name and removed all existing annotations.

Due to these code designs, the participants did not need to learn any new concepts for any version. Knowledge about comments and object-orientation are sufficient to understand such feature traces after a short introduction.

## 3.3 Distribution of Participants

We personally invited 144 software developers from different countries and asked them to share the invitation with others. Our goal was to include developers with industrial experiences and increase their motivation to participate. After accepting the invitation, each

**Table 1: Questions to quantify programming experience.**

| ID | Question (Q) \| Answering Options (A) |
|---|---|
| $Q_1$ | How do you estimate your programming experience?<br>A: 1 (very inexperienced) – 10 (very experienced) |
| $Q_2$ | How experienced are you with the Java programming language?<br>A: 1 (very inexperienced) – 10 (very experienced) |
| $Q_3$ | For how many years have you been programming?<br>A: ○ <2; ○ 2-5; ○ 6-10; ○ 11+ |
| $Q_4$ | For how many years have you been programming for larger software projects (e.g., in companies)?<br>A: ○ <2; ○ 2-5; ○ 6-10; ○ 11+ |
| $Q_5$ | What is your highest degree of education that is related to programming?<br>A: Multiple choice (optional text) |

developer had to fill in a survey to assess their programming experience. We provide an overview of the survey questions and possible answers in Table 1.

These questions are based on an empirically derived proposal [53]. We based the answer classifications for $Q_3$ and $Q_4$ on a large user survey of Stack Overflow.[2] In this survey, approximately one quarter of the participants has been in each of the classes we show in Table 1. We mapped the classes to a scale from one to ten (i.e., 2, 4, 7, 9), aligning them to the first two questions. Considering the degree, we only identified whether a developer received one (8) or not (3), as it is hardly possible to say which ones may indicate "better" developers. For the experience value, we computed the average of all scales and considered a developer as novice if the result was below or equal to 5.5—or as expert, otherwise. We randomly distributed our participants into three groups with equal ratios of novices and experts, one for each code version (i.e., object-oriented, annotated, components), and sent the actual experiment.

## 3.4 Tasks & Questions

For the first part of our experiment, we selected six tasks that involve, but are not directly concerned with, feature location for two reasons:

- Participants of the *annotated* and *component* groups can quickly locate features by searching the names.
- We aimed to limit learning effects that may impact our subjects' performances in completing their tasks.

In contrast to the straight-forward task of feature location, we were interested in the impact of feature traceability on tasks that require actual comprehension. Therefore, we designed two sections with three tasks, each.

In the first section, we were concerned with comprehending features and their interactions, which does not only require to locate the corresponding code, but to also understand it. Feature interactions represent different system functionalities that interact and may influence each other. Thus, feature interactions are an important challenge that can easily result in problems during program comprehension and bug fixing [2]. The tasks that we defined for the first section were:

[2] https://insights.stackoverflow.com/survey/2016#developer-profile-experience

**Table 2: Questions to evaluate the participants' experiences with the tasks and on feature traceability.**

| ID | Question (EQ) \| Answering Options (A) |
|---|---|
| $EQ_1$ | Did you have any problems in answering the survey, e.g., understanding the questions or concepts?<br>A: ○ yes; ○ no |
| $EQ_2$ | What was your strategy for comprehending the code in order to do the tasks?<br>A: Free text |
| $EQ_3$ | What have been your main problems or challenges during the tasks?<br>A: Free text |
| $EQ_4$ | *(Annotated)* Do you think that the annotations provided for each feature helped you understand the code?<br>*(Components)* Do you think that the separation of features into classes helped you understand the code?<br>*(Object-Oriented)* Do you think that a different code design concerning the features (e.g., annotating their begin and end, implement them in separate classes) would have facilitated your program comprehension?<br>A: Free text |
| $EQ_5$ | Did you face an interruption (more than 5 minutes) for any of the 6 tasks?<br>A: Checkbox for each task |
| $EQ_6$ | Do you have any comments on the survey?<br>A: Free text |

(1) Out of four feature pairs, select those that interact;
(2) Select the lines where two described features interact; and
(3) Out of four statements about this feature interaction, select those that are correct.

In the second section, we asked our participants to locate bugs, which we inserted:

(4) Into a feature (cannot capture photos);
(5) Into a feature interaction (wrong counter for videos); and
(6) Into the base code (cannot delete photos).

These bugs resemble simple faults (i.e., copy-paste errors, increments), similar to mutations [20]. Each task was about a different feature to mitigate learning biases.

At the end of our experiment, we asked our participants to elaborate on their experiences and to describe whether they faced any problems. We show the corresponding questions in Table 2. $EQ_1$ was a simple check question to verify if there were any misunderstandings, which could also be elaborated on in $EQ_6$. We used $EQ_5$ to verify whether a participant was interrupted during any task, meaning that we considered the corresponding results differently. With the remaining three questions, we were concerned with gathering qualitative data to answer **$RQ_3$**. We remark that $EQ_4$ exists in three different versions, one for each code version.

## 3.5 Implementation & Testing

Due to the tasks we defined (i.e., marking lines of code) and the design of our experiment (i.e., accessible via internet), we were not able to reuse existing survey tools without considerable costs and adaptations. For these reasons, we decided to implement our own solution that was based on a simple server-client architecture and fulfilled the requirements of our experiment. We tested our

**Table 3: Experience values of our participants.**

| Version | Experience | | | | Participants |
|---|---|---|---|---|---|
| | Min | Median | Mean | Max | |
| Annotated | 5.60 | 7.00 | 6.96 | 8.80 | 18 |
| Components | 4.00 | 7.20 | 6.88 | 9.20 | 15 |
| Object-Oriented | 6.20 | 7.60 | 7.61 | 9.20 | 16 |
| Total | 4.00 | 7.40 | 7.15 | 9.20 | 49 |

implementation extensively with own test runs. Code reviews and additional tests of our implementation were performed by three colleagues consisting of a software developer, a system administrator, and a PhD student. Moreover, two of these colleagues tested the actual survey to evaluate formulations and the tasks' complexity. None of the three colleagues participated in the actual experiment.

We decided to conduct our experiment via the internet to increase our range, provide the opportunity to conduct the tasks at any time, and have access to developers all around the world without extensive traveling. Thus, this is not a fully controlled experiment, but an unsupervised one that was conducted in real-world settings in which developers may be distracted or switch tasks. With this design, we aimed to increase the external validity of our results.

# 4 RESULTS AND DISCUSSION

In this section, we report details about the participants of our experiment and the results. We separately derive and discuss observations for each of our research questions.

## 4.1 Participants

Unsurprisingly, not all developers that we invited participated in our experiment. Overall, we received 49 responses from around the world, mostly from Turkey (20), Germany (13), and the Unites States (7). In Table 3, we show the distribution of our participants' experience values based on our rating scale (cf. Table 1). Only two participants stated that they have worked for less then two years on large-scale projects. As we can see, the median and mean values are close to each other and among the groups. While the distribution of participants for each program version is not identical, the differences are small. Moreover, most participants are considered experts according to our analysis with only three of them in the components group not achieving this rating.

Overall, we can see small differences between the groups of participants. Nonetheless, we had at least 15 participants and 12 experts for each code version of our experiment. Considering this information, the responses we analyzed represent a diverse and experienced set of practitioners. Thus, we argue that none of the differences threatens the results of our study, but we have to be cautious with the interpretation.

*Validity of Responses.* As aforementioned, we aimed to attract experienced software developers and intended to focus on external validity. Due to our study design, there have been several participants who reported disruptions while they worked on a task or problems in understanding some details (however, most elaborated about code issues, rather than the experiment itself). To address this issue, we first performed a *sanity check* in the context of $\mathbf{RQ_1}$ for developers stating comprehension problems. Considering $\mathbf{RQ_2}$, we removed all completion times for which interruptions were reported, as these measures would not accurately represent the required effort.

Due to technical issues, single data points for some participants are missing. First, three participants reported problems with task 1, or were just missing the entry. We decided not to count these responses, wherefore task 1 for the annotated group comprises only 15 responses (cf. Figure 1). Second, one participant of the components group did answer all questions except the elaboration (cf. Table 2). We decided to include this response, but to put it into the group with comprehension problems for the sanity check (assuming that there have been misunderstandings). Except for these four, we excluded all other unfinished or incomplete responses.

## 4.2 $\mathbf{RQ_1}$: Effectiveness

In Figure 1, we show how many of our participants were able to correctly solve each task. We distinguish between three groups according to our first independent variable, the version of the system the participants investigated (**A**: Annotated; **C**: Components; **OO**: Object-Oriented). Moreover, we considered whether the participants indicated problems in understanding any part of the experiment (**CP**) or not (**NCP**). We applied hypothesis testing to test whether our observations may be significant. In particular, we tested observation 1 that represents our sanity check and based on which we scoped our remaining observations, analyses, and tests.

*Observation 1: Difficulties in understanding the survey had no impact on the results.* Comparing the correct and incorrect answers of the participants with and without comprehension problems for each task, we can see that the distributions are similar. Moreover, in some tasks the ratio of correctly solved tasks with comprehension problems is identical compared to those without problems (e.g., for task 2 of the annotated group, both have eight correct and one incorrect answer). Thus, it seems that problems in understanding our experiment had only limited impact on our participants' ability to correctly solve a task. This is reasonable, as the code was unknown to our participants, meaning that they had to understand it anew anyway. As in daily life, they can still understand code, even if facing a potentially vague assignment. In addition, most participants stated that the code was the problem for understanding (e.g., too long), rather than the tasks themselves. However, the code and its design were the subject we aimed to understand, meaning that the results should be comparable.

*Hypothesis testing.* Based on our observation, we hypothesized that *there are no threatening differences between the participants who did have and who did not have problems in understanding our experiment.* To test our hypothesis, we applied Fisher's exact test [13], as implemented in the R statistics software [44]. We used Fisher's exact test, because it can be applied on small sample sizes, but we still have to be careful with interpreting the results. To account for multiple hypothesis testing, we relied on a Bonferroni-Holm correction [16] with a global confidence interval of 0.95. In the remaining paper, we report the p-values of all significant results and also state

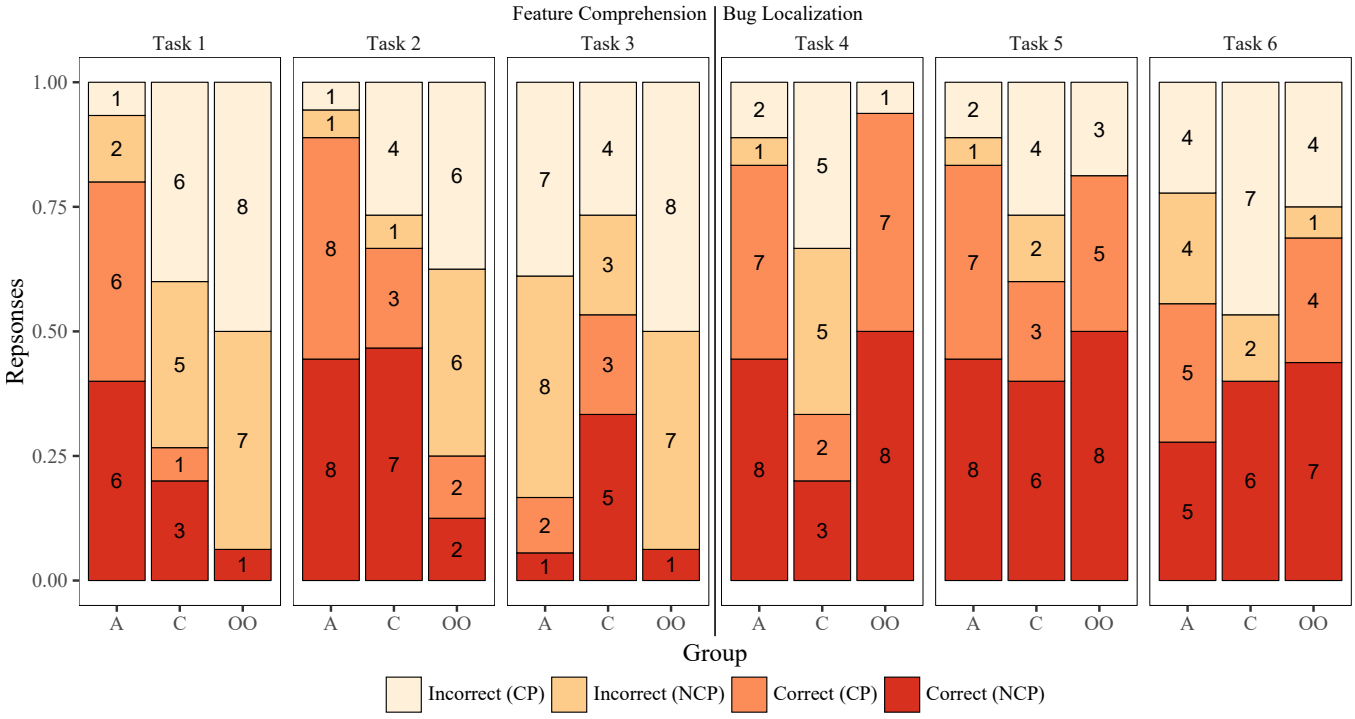


**Figure 1: Distribution of correctly and incorrectly solved tasks for each program version (A: Annotated; C: Components; OO: Object-Oriented) and task. Moreover, we display whether the corresponding participants stated comprehension problems or not (CP: Comprehension Problems; NCP: No Comprehension Problems).**

the approximated, Bonferroni-Holm corrected threshold that had to be fulfilled. The *null-hypothesis* is that correct and wrong responses are equally distributed.

In total, we tested 18 hypotheses, one for each pair-wise combination of groups for each of the six tasks. None of the tests indicated that the differences are significant, wherefore we cannot reject our null-hypothesis So, we continued under the assumption we derived from our observation, namely that participants who had problems in understanding the survey did not perform worse than those who did not. Due to this observation, we focused on all participants and did not separate those that had comprehension problems in our remaining analyses.

*Observation 2: Explicit feature traces result in higher effectiveness for comprehending feature interactions.* Considering the first three tasks, we can see in Figure 1 that the pure object-oriented code performs worse compared to annotations and components. For the first and third task, only one participant who worked on the object-oriented version was able to correctly identify and understand the feature interactions. Moreover, concerning task 2, only three more participants have been able to correctly solve the task. This result seems unsurprising, as explicit feature locations facilitate understanding interactions considerably: Developers can focus on certain parts of the code and do not have to identify the code that implements the feature first, which is time consuming and can be faulty due to different notions of what a feature comprises [4, 5, 24].

However, it is surprising that participants who analyzed components had considerably more problems in identifying features that interact (task 1). The data shows that participants selected multiple wrong interactions. In contrast, they were more often able to correctly explain how features interact (task 3). For this task, the faulty responses usually show that it seems unclear for the annotated group, to what extent features interact: Which feature does modify which feature in what way? Both groups performed comparable for locating a single feature interaction (task 2).

Potentially, it is easier to identify that features interact at all if their code is close to each other (annotated code), rather than separated into different classes—resulting in the code loosing its surrounding context and potentially leading to the anti-pattern *action at a distance* [23, 26]. In contrast, this loss of context may be better to identify how features interact in the data-flow: Method calls can already indicate whether a functionality is only used or whether variables are changed. By inspecting the separated features, developers can more easily identify globally accessible and potentially interacting variables. Annotated code may complicate this analysis as all context, even irrelevant one, is connected to the feature. Identifying and understanding the actual data-flow interactions [47] of features remains challenging, even with explicit feature traces, as was explicitly stated by some participants (cf. Section 4.4).

*Observation 3: Decomposition results in less effectiveness for bug localization.* For the last three tasks, we can see that participants who faced the decomposed code identified fewer bugs correctly

compared to other groups. Surprisingly, they even performed worse for a faulty named label within a feature (task 4). As this bug is connected to, and thus directly placed in, a feature, we expected that the participants could easily identify the bug. The wrongly selected answers show that most participants identified the correct class, but selected a wrong line after the actual bug. As is also highlighted by participants' feedback (cf. Section 4.4), this problem seems connected to the fact that the classes represented features and not logical objects, as intended in object-oriented programming. Again, the same issue of lost context and understanding data-flow we described for our previous observation seems to have impacted our participants' ability to locate bugs. However, to understand these effects in more detail, we require additional studies on locating and fixing bugs in decomposed code.

*Observation 4: Annotations do not result in more effective bug localization.* For annotations, we observe that the ratio of correctly localized bugs is similar compared to the object-oriented version. Thus, annotations seem to have no negative impact on bug localization. As the bugs are rather simple, the annotations may not be helpful in this scenario, or the analysis of object-oriented code may have resulted in better knowledge of our participants. More extensive studies of these factors are needed to better understand how they influence developers' program comprehension.

*Hypothesis testing.* For each task, we compared all groups against each other (pair-wise tests). To account for learning effects of our participants, we conducted all 18 tests simultaneously and corrected them together—instead of testing each task individually. Again, we used Fisher's exact test and Bonferroni-Holm correction. To this end, we always assumed as *null-hypothesis* that the ratio of correct and incorrect answers between two groups is equal (cf. $H_1$).

The test results include three significant outcomes for which we can refute our null hypothesis. For tasks 1 and 2, we found significant differences between the annotated and object-oriented group ($p < .0001$ and $p < .001$ with corrected thresholds of $p = .0028$ and $p = .0029$, respectively). This supports our second observation that explicit feature traces support comprehending feature interactions. However, this is solely limited to annotations and does not significantly apply to the decomposed code version. In addition, we found significant differences between components and object-oriented code for task 4, supporting our third observation ($p < .001$ and a corrected threshold of $p = .0031$).

To summarize **$RQ_1$**, our results indicate that explicit feature traceability can have positive, but also negative, effects on program comprehension. Still, we have to be careful with interpretations and must conduct industrial studies, for which annotations seem to be more promising.

- $O_1$ Understanding problems do not bias the results. *Not rejected.*
- $O_2$ Explicit traces improve interaction comprehension. *Accepted twice for annotated code.*
- $O_3$ Decomposition hampers bug localization. *Accepted once.*
- $O_4$ Annotations have no effect on bug localization. *Not rejected.*

## 4.3 $RQ_2$: Efficiency

In Table 4, we display statistics about the times our participants in each group needed to complete a task, regardless of correctness. We only considered participants that did not state interruptions for a task (undisturbed). Nonetheless, we found few extreme outliers where some participants worked for several hours on a single task. These outliers indicate that the corresponding participant had been interrupted, but did not state so. In order to address such extreme cases, we removed entries that were more than twice above the third quartile of each task and group. This led to the exclusion of 22 data points from our analysis, the difference between undisturbed and included participants in Table 4.

*Observation 5: Explicit feature traceability does not influence efficiency.* The results do not vary heavily between different versions of the code. Moreover, only the first task required considerably more time compared to the others. This is rarely surprising, as our participants had to get familiar with the code and its structure. For all other tasks, all groups needed between 1.19 and 3.2 minutes to complete a task, on average. Likewise, the minimum and maximum times are similar throughout all tasks. Thus, explicit feature traceability seems to have neither a positive nor a negative impact on the analysis time, especially compared to the time needed to familiarize with the code.

*Hypothesis testing.* We compared the completion time distributions of our groups within each task with the Kruskal-Wallis test [31]. This test does not require normal distributions and can compare multiple groups against each other. The *null-hypothesis*, which we aimed to refute, was that there are no significant differences between the completion times (cf. $H_2$). As none of the tests resulted in a p-value below 0.3, we cannot reject our null-hypothesis and argue that our observation is reasonable.

To summarize **$RQ_2$**, the results show no impact of explicit feature traceability on the completion times. Considering that annotations seem to improve program comprehension, this indicates an overall positive effect of these.

- $O_5$ Explicit feature traces do not influence efficiency. *Not rejected.*

## 4.4 $RQ_3$: Participants' Judgment

In Table 5, we summarize the qualitative responses we received from our participants. Partly, the numbers do not accumulate to the total number of participants, as they were allowed not to elaborate, if they wanted. We read all their comments and summarized the mentioned analysis strategies, challenges, and opinions on code design for each group. Our summarizing strategy followed the idea of open-card sorting [57].

*Analysis strategies.* Concerning their analysis strategy, many participants in each group stated that they started with a general exploration of the source code (25). They aimed to understand the structure of the code and its behavior on an abstract level. How these tasks have been performed is quite different among the participants: Some simply skimmed through the code to get a rough understanding, while others focused on specific code constructs, such as labels and methods.

**Table 4: Statistics on the completion times (in minutes) of our participants.**

| | Task 1 | | | Task 2 | | | Task 3 | | | Task 4 | | | Task 5 | | | Task 6 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | C | OO | A | C | OO | A | C | OO | A | C | OO | A | C | OO | A | C | OO |
| **Und. Part.** | 10 | 10 | 9 | 13 | 12 | 15 | 16 | 14 | 15 | 18 | 13 | 16 | 18 | 13 | 15 | 16 | 10 | 16 |
| **Incl. Part.** | 10 | 8 | 9 | 12 | 11 | 13 | 14 | 14 | 13 | 16 | 11 | 15 | 16 | 12 | 14 | 15 | 10 | 14 |
| **Times (mins)** | | | | | | | | | | | | | | | | | | |
| **Min** | 2.91 | 2.23 | 2.72 | 0.44 | 1.14 | 0.91 | 0.70 | 0.67 | 0.52 | 0.38 | 0.66 | 0.61 | 1.63 | 1.47 | 0.57 | 0.61 | 1.30 | 0.76 |
| **Mean** | 13.07 | 5.51 | 12.27 | 1.72 | 3.26 | 3.30 | 2.73 | 2.26 | 1.84 | 1.19 | 2.40 | 1.58 | 3.03 | 2.90 | 2.91 | 3.23 | 2.59 | 1.49 |
| **Median** | 11.23 | 4.03 | 9.75 | 1.06 | 2.63 | 2.09 | 2.04 | 2.11 | 1.68 | 1.07 | 1.79 | 1.21 | 2.66 | 2.54 | 2.28 | 3.20 | 2.50 | 1.23 |
| **Max** | 25.02 | 12.73 | 22.92 | 4.90 | 8.48 | 11.96 | 7.29 | 4.70 | 3.90 | 2.33 | 6.37 | 4.09 | 6.84 | 5.95 | 7.55 | 8.82 | 5.05 | 3.48 |
| **SD** | 8.34 | 3.59 | 7.54 | 1.43 | 2.34 | 3.14 | 1.78 | 1.30 | 0.89 | 0.52 | 2.01 | 1.00 | 1.45 | 1.37 | 2.01 | 2.16 | 1.19 | 0.75 |

Part.: Participants; Und.: Undisturbed; Incl.: Included; SD: Standard Deviation

**Table 5: Summary of our participants' qualitative responses concerning analysis strategies, challenges, and code design ("–" means not applicable).**

| Response | # Mentioned | | |
|---|---|---|---|
| | Annotations | Components | Object-Oriented |
| Participants | 18 | 15 | 16 |
| | **Analysis strategy** | | |
| Get picture of code | 7 | 6 | 12 |
| Look for keywords | 4 | 2 | 8 |
| Use search function | 0 | 1 | 3 |
| Follow annotations | 8 | – | – |
| Follow class names | – | 7 | – |
| | **Challenges** | | |
| Code quality | 9 | 6 | 6 |
| Code length | 7 | 0 | 1 |
| Missing IDE | 4 | 4 | 3 |
| Feature location | 2 | 0 | 3 |
| Missing knowledge | 1 | 3 | 1 |
| | **Code design** | | |
| Positive | 14 | 9 | – |
| Unsure | 2 | 2 | – |
| Negative | 2 | 3 | – |
| Components | 1 | – | 5 |
| Comments | – | 0 | 4 |
| Explicit locations | – | – | 3 |

Unsurprisingly, 15 participants relied on the explicit feature traces to address their tasks, if these were available. In some cases, the participants mentioned that they also focused on keywords (14), mostly to understand details, and used their browser's search function (4). Keywords and searches were also explicitly mentioned and used by participants that worked on the object-oriented code. This behavior aligns with the results of previous studies on manual feature location [24, 61].

*Challenges.* Considering challenges, 21 participants mentioned quality issues of the code. Most concerns were connected to design decisions of our experiment that they did not like, for example, the (long) code length (8), missing comments, or inappropriate identifiers. We specifically removed comments to avoid biases and reduced the code size, but the code had to be large enough for feature traces to be useful.

Other general concerns were the intentionally missing IDE support (11), avoiding too many biases that would make any meaningful assessment impossible. Five participants also mentioned their missing knowledge about the system as a problem that hampered their comprehension. However, this was also intended to have equal preconditions for every participant. Interestingly, not only two participants of the object-oriented group, but also two participants of the annotated group had problems to identify the feature locations. For example, in the annotated group, one participant indicated the need for decomposing features to avoid cluttering:

> "[T]he biggest challenge for me was that all of the features are in a single place, just written one after another."

*Opinions.* Concerning the feature traceability techniques on their own, most of our participants stated a positive perception after the experiment. For example, 14 out of 18 participants in the *annotated* group argue that the annotations helped, some stating that they were elementary to locate and understand features—conflicting some scientific beliefs about annotations:

> "Yes, they did. In fact, without the annotations (provided that they are correct), it would have been significantly more difficult to understand which part of the code does what."

The few critics of annotations were not focusing on the actual annotations, but argue that comments indicate poor code:

> "[N]o, adding comments in the code is a bad sign, it screams that code is not self explanatory enough."

Similarly, nine of 15 participants stated a positive effect of *decomposing* the system into features. Most participants stated that it helped to faster trace features:

> "It helps [to] logical[ly] aid to decide where to start."

The negative experiences were connected to identifying which feature to look at. Such issues mainly arose because our participants had not been familiar with the system:

> "Yes, I understood the intent [...] with this sorting, naming and separation. It was still unfamiliar and took more time than it would have with familiar code."

This indicates that decomposition has to be used carefully: The right separation strategy is important and especially to new developers we have to explain how it is used. For developers who are familiar with the structure of the code and its features, this problem will arguably diminish. One participant specifically explained their experienced pros and cons of decomposition and may best summarize our overall results (i.e., **RQ$_1$**):

> "On the one hand, it made the classes small and locating possibly relevant code easy. On the other hand, interactions were more difficult to spot because I had to switch between different classes."

For the *object-oriented* group, we did not ask about the anticipated impact of the code design, but if annotations or decomposition would have been helpful:

> "Features could have been implemented in a more organized way. [W]e clearly need more than one class here."

More precisely, four participants were in favor of decomposing the code and five were in favor of adding comments (i.e., annotations) to indicate feature locations:

> "More comments and better restructuring of the code should be more helpful."

Overall, 11 out of 16 participants mentioned that any explicit feature traces in the code would have been helpful.

To summarize **RQ$_3$**, the results show that most of our participants have a positive perception of explicit feature traces. Thus, introducing traces in practice may not be a problem and especially annotations are simple to adopt. Condensing the qualitative responses, we can derive three observations:

- O$_6$ Explicit features extend general analysis strategies.
- O$_7$ Feature traces themselves are unproblematic to use.
- O$_8$ Making features explicit has a positive perception.

## 5 THREATS TO VALIDITY

The goal of our study was to provide empirical insights into a fundamental design decision based on studying experienced software developers in the real world. Due to the trade-off between internal and external validity [55] and the magnitude of interacting factors that impact program comprehension, we can hardly address all biases—resulting in more internal threats. In the following, we report threats to the validity of our study based on the guidelines of Wohlin et al. [64].

*Construct Validity.* Concerning the construct validity of our study, some participants indicated that they had problems understanding the survey or the concepts of annotations and decomposition to separate features. To mitigate this threat, we provided small examples and used check questions to identify if any confusions occurred. Moreover, we performed a sanity check on the correctness of tasks and found no differences for participants who stated comprehension problems. So, we argue that this threat is properly addressed in our design. In addition, most participants stated that they had problems with the code and not the experimental design, meaning that the construct validity would not be threatened.

*Internal Validity.* We aimed to reduce the impact of different development environments by using a web-interface to display the code. Still, we kept identifier names as well as syntax highlighting, and did not control for tool usage (e.g., searches). While we cannot ensure that our participants conducted the experiment with the exact same set-ups (e.g., noise level, using additional tools, web searches), we argue that developers in real-world settings also have a multi-fold of tools, environments, and different comprehension patterns. Thus, the set-up may bias our results, but reflects practice.

Our code examples comprise different techniques to trace features, namely annotations and decomposition. We relied on the existing preprocessor directives in the original Mobile Media system to add our own annotations. For the decomposition, we separated the corresponding code into different classes. Both techniques are inspired by the usage of preprocessors in open-source and industrial systems, which are similarly structured [17]. Together with the additional changes that we applied to the code (i.e., removing one feature, deleting imports and comments), the nature of our code examples changed. Such changes may have influenced the results. We did all changes in order to keep our participants motivated and to control biases. Still, we cannot fully avoid this threat to our study and, for example, another decomposition may have resulted in better results for our participants in the corresponding group.

A concerning internal threat are learning effects of our participants, meaning that they may got more familiar with the code. We addressed this threat in two ways: First, while we used a single code example, we asked about different features for each task. This way, our participants may have achieved better understanding about the overall code, but not the specific feature. Most of the participants also indicated that they did neither focus on nor did achieve an understanding of the overall code, besides a general overview. Second, we decided against a random order of the survey tasks. So, for each task, the experience with the source code should be comparable between our participants. Based on this, we argue that learning effects are mostly impacted by the different traceability techniques for features, which is the concern of our research questions.

*External Validity.* Software developers have various backgrounds, expertise with a programming language, and experiences with certain tasks. To address these threats, we invited a group of experienced software developers from several countries and organizations. Besides most of them working on larger projects for a long time, we also evaluated their programming experience, based on which we randomly sampled them into equally distributed groups. While the responses resulted into three novices being part of the same group, they were close to expert level. Overall, our participants are a rather homogeneous group considering their experiences, wherefore we argue that such threats are diminished, but may have occurred.

Several background factors, such as age, gender, or motivation may have an impact on the results. Moreover, program comprehension comprises cognitive processes that highly depend on the individual developer, as they learn and understand based on different patterns and rates. We aimed to address such factors partly by measuring them (i.e., programming experiences) and by personally inviting participants (e.g., increasing motivation). Still, we cannot control all of these factors perfectly. Consequently, they remain a threat to the external validity of our study.

Several studies used Mobile Media to provide code examples for empirical studies. The code is also designed to reflect a real-world system, and thus we argue that our examples can be considered as realistic. Nonetheless, Mobile Media is an academic system, which is why our results may not be completely transferable to industrial practice. Still, our participants' activities during program comprehension will most likely not have changed, since such systems are similarly structures in industrial, open-source and academic contexts [17, 52]. In addition, all participants faced the same system (independent variable), meaning that our analysis of feature traces (dependent variable) remains valid.

*Conclusion Validity.* We have to be careful with the conclusions we derived from our observations. While they are interesting, our statistical tests revealed only few significant correlations. However, due to the problems of such tests [63], we only used them as supportive means and focused more on our actual observations. To this end, we carefully investigated different variables and analyzed their impact on program comprehension. This way, we aimed to mitigate threats to the conclusion validity.

Despite the discussed threats, we argue that our study is valid and provides reliable and interesting insights into an important design decision. We used quantitative and qualitative methods, combining measured data with subjective responses and tested our observations statistically. Still, we encourage other researchers to conduct further studies in this direction to strengthen the empirical evidence and gain insights into the impact of explicit feature traces. In this regard, we argue that our study can be replicated.

## 6 CONCLUSION

In this paper, we reported an online experiment with 49 experienced software developers concerning explicit feature traceability, which we implemented based on annotations and decomposition. We based our design on existing studies and recommendations, with a particular focus on increasing the external validity of our results. To this end, we invited especially practitioners from various countries and organizations. We relied on quantitative and qualitative analyses to find indications for the following four conclusions:

(1) Annotations positively impact the effectiveness of developers when comprehending features and their interactions, while not negatively impacting bug localization.
(2) Decomposition into components has no significant impact on the effectiveness of developers when comprehending features, but resulted in less correct bug localization. However, this is arguably connected to the structure, size, and cohesion of the decomposed features.
(3) Explicit feature traces do not impact the efficiency of developers during program comprehension.
(4) Explicit feature traces do not result in comprehension problems and practitioners have a positive perception of such explicit traces.

We remark that there are several threats to our results and we highly encourage further studies. However, we argue that especially for annotations our results indicate that they can be a helpful means to support program comprehension. In particular, this may be the case if developers are facing unfamiliar code. As they are also simple to introduce, annotations may be a good way for organizations to implement and test feature traceability as well as for researchers to conduct further studies in this direction.

In future work, we aim to extend our analysis and focus on additional variables, particularly extending our investigations to programmers' memory. Moreover, we plan to design different experiments and observational studies that maximize internal or external validity, including collaborations with industrial partners. This way, we can consolidate the empirical knowledge about explicit feature traces, provide more precise recommendations to practitioners, and identify open research problems. Furthermore, we argue that different tracing techniques should be compared to identify what impact they may have. Similarly, our study was focused on program comprehension tasks. In the future, we also aim to analyze the impact of explicit feature traces on other activities that we did not consider, for example, on maintaining and evolving a system (e.g., introducing new features).

## ACKNOWLEDGMENTS

Jacob Krüger would like to thank ACM SIGSOFT for supporting the presentation of this paper with a CAPS award. Gül Çalıklı's work is supported by the SEFIS project funded by Chalmers Area of Advance (ICT-SEED-2018). Thorsten Berger's work is supported by the ITEA project REVaMP$^2$ funded by Vinnova Sweden (2016-02804), and by the Swedish Research Council Vetenskapsrådet (257822902). Thomas Leich's (LE 3382/2-1, LE 3382/2-3) and Gunter Saake's (SA 465/49-1, SA 465/49-3) work is supported by the German Research Foundation (DFG) project EXPLANT.

We thank Christian Lausberger for testing and administrating the experiment; Wardah Mahmood and Yüceer Çalıklı for testing our tasks; and Sebastian Krieter for helping us troubleshoot our R code. Moreover, we thank the anonymous reviewers for their valuable feedback, and especially for pointing out a bug in our data transformation. Finally, we would like to thank all participants of our experiment.

## REFERENCES

[1] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering Traceability Links Between Code and Documentation. *IEEE Transactions on Software Engineering* 28, 10 (2002), 970–983.
[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer.
[3] Deborah J. Armstrong. 2006. The Quarks of Object-Oriented Development. *Communications of the ACM* 49, 2 (2006), 123–128.
[4] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Conference on Software Product Line (SPLC).* ACM, 16–25.
[5] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1993. The Concept Assignment Problem in Program Understanding. In *International Conference on Software Engineering (ICSE).* IEEE, 482–498.
[6] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. In *Conference on Human Factors in Computing Systems (CHI).* ACM, 557–566.
[7] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley.
[8] Joseph W. Davison, Dennis M. Manci, and William F. Opdyke. 2000. Understanding and Addressing the Essential Costs of Evolving Systems. *Bell Labs Technical Journal* (2000).
[9] Alexander Delater and Barbara Paech. 2013. Tracing Requirements and Source Code During Software Development: An Empirical Study. In *International Symposium on Empirical Software Engineering and Measurement (ESEM).* IEEE, 25–34.
[10] Alexander Egyed, Florian Graf, and Paul Grünbacher. 2010. Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments. In *International Requirements Engineering Conference (RE).* IEEE, 221–230.

[11] 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 (FOSD)*. ACM, 55–62.

[12] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering* 18, 4 (2013), 699–745.

[13] Ronald A. Fisher. 1936. *Statistical Methods For Research Workers.* Oliver and Boyd.

[14] Florian Fittkau, Santje Finke, Wilhelm Hasselbring, and Jan Waller. 2015. Comparing Trace Visualizations for Program Comprehension through Controlled Experiments. In *International Conference on Program Comprehension (ICPC)*. IEEE, 266–276.

[15] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2019. Shorter Identifier Names Take Longer to Comprehend. *Empirical Software Engineering* 24, 1 (2019), 417–443.

[16] Sture Holm. 1979. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics* (1979), 65–70.

[17] 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.

[18] Khaled Jaber, Bonita Sharif, and Chang Liu. 2013. A Study on the Effect of Traceability Links in Software Maintenance. *IEEE Access* 1 (2013), 726–741.

[19] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *International Conference on Software Product Line (SPLC)*. ACM, 61–70.

[20] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *Transactions on Software Engineering* 37, 5 (2011), 649–678.

[21] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 1 (1998), 143.

[22] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The Road to Feature Modularity?. In *International Software Product Line Conference (SPLC)*. ACM, 5:1–5:8.

[23] Jacob Krüger. 2018. Separation of Concerns: Experiences of the Crowd. In *Symposium on Applied Computing (SAC)*. ACM, 2076–2077.

[24] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and How to Find Them: A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems: Foundations and Applications*. LLC/CRC Press, 153–172.

[25] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM.

[26] Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *Symposium on Applied Computing (SAC)*. ACM, 2042–2049.

[27] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.

[28] Jacob Krüger, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2018. Composing Annotations Without Regret? Practical Experiences using FeatureC. *Software: Practice and Experience* 48, 3 (2018), 402–427.

[29] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *International Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 74–84.

[30] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember this Source Code?. In *International Conference on Software Engineering (ICSE)*. ACM, 764–775.

[31] William H. Kruskal and W. Allen Wallis. 1952. Use of Ranks in One-Criterion Variance Analysis. *J. Amer. Statist. Assoc.* 47, 260 (1952), 583–621.

[32] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *International Conference on Software Engineering (ICSE)*. ACM, 492–501.

[33] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering* 3, 4 (2007), 303–318.

[34] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150.

[35] 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 *International Conference on Software Engineering (ICSE)*. ACM, 105–114.

[36] Patrick Mäder and Alexander Egyed. 2015. Do Developers Benefit from Requirements Traceability when Evolving and Maintaining a Software System? *Empirical Software Engineering* 20, 2 (2015), 413–441.

[37] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *International Conference on Software Engineering (ICSE)*. ACM, 679–690.

[38] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2018. Commenting Source Code: Is it Worth it for Small Programming Tasks? *Empirical Software Engineering* (2018), 1–40.

[39] Nan Niu, Wentao Wang, and Arushi Gupta. 2016. Gray Links in the Use of Requirements Traceability. In *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 384–395.

[40] David L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15, 12 (1972), 1053–1058.

[41] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *International Conference on Modularity (MODULARITY)*. ACM, 81–92.

[42] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2018. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* (2018). Preprint.

[43] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 419–443.

[44] R Core Team. 2018. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. https://www.R-project.org

[45] Patrick Rempel and Parick Mäder. 2016. Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *IEEE Transactions on Software Engineering* 43, 8 (2016), 777–797.

[46] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-Demand Developer Documentation. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 479–483.

[47] Iran Rodrigues, Márcio Ribeiro, Flávio Medeiros, Paulo Borba, Baldoino Fonseca, and Rohit Gheyi. 2016. Assessing Fine-Grained Feature Dependencies. *Information and Software Technology* 78 (2016), 27–52.

[48] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer.

[49] Alcemir Rodrigues Santos, Ivan do Carmo Machado, Eduardo Santana de Almeida, Janet Siegmund, and Sven Apel. 2019. Comparing the Influence of Using Feature-Oriented Programming and Conditional Compilation on Comprehending Feature-Oriented Software. *Empirical Software Engineering* 24, 3 (2019), 1226–1258.

[50] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending Studies on Program Comprehension. In *International Conference on Program Comprehension (ICPC)*. IEEE, 308–311.

[51] Kanwarpreet Sethi, Yuanfang Cai, Sunny Wong, Alessandro Garcia, and Claudio Sant'Anna. 2009. From Retrospect to Prospect: Assessing Modularity and Stability from Software Architecture. In *Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA)*. IEEE, 269–272.

[52] 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 (FOSD)*. ACM, 17–24.

[53] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and Modeling Programming Experience. *Empirical Software Engineering* 19, 5 (2014), 1299–1334.

[54] Janet Siegmund and Jana Schumann. 2015. Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Software Engineering* 20, 4 (2015), 1159–1192.

[55] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *International Conference on Software Engineering (ICSE)*. IEEE, 9–19.

[56] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 2010. An Examination of Software Engineering Work Practices. In *CASCON First Decade High Impact Papers (CASCON)*. IBM, 174–188.

[57] Donna Spencer. 2009. *Card Sorting: Designing Usable Categories.* Rosenfeld Media.

[58] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *International Conference on Software Engineering (ICSE)*. ACM, 107–119.

[59] Rebecca Tiarks. 2011. What Maintenance Programmers Really do: An Observational Study. In *Workshop on Software Reengineering (WSR)*.

[60] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program Understanding Behaviour during Enhancement of Large-Scale Software. *Journal of Software Maintenance: Research and Practice* 9, 5 (1997), 299–327.

[61] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented

Exploratory Study. *Journal of Software: Evolution and Process* 25, 11 (2013), 1193–1224.

[62] Tianxia Wang and Yan Liu. 2017. Jsea: A Program Comprehension Tool Adopting LDA-Based Topic Modeling. *International Journal of Advanced Computer Science and Applications* 2, 3 (2017).

[63] Ronald L. Wasserstein, Allen L. Schirm, and Nicole A. Lazar. 2019. Moving to a World Beyond "p < 0.05". *The American Statistician* 73 (2019), 1–19.

[64] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer.

[65] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. 1981. The Effect of Modularization and Comments on Program Comprehension. In *International Conference on Software Engineering (ICSE)*. IEEE, 215–223.

[66] Trevor J. Young. 2005. *Using AspectJ to Build a Software Product Line for Mobile Devices*. Master's thesis. University of British Columbia.