How to Compare Program Comprehension in FOSD Empirically – An Experience Report

Janet Feigenspan
Metop Research Center
Magdeburg, Germany
janet.feigenspan@metop.de

Christian Kästner
University of Magdeburg
Magdeburg, Germany
ckaestne@ovgu.de

Sven Apel
University of Passau
Passau, Germany
apel@uni-passau.de

Thomas Leich
Metop Research Center
Magdeburg, Germany
thomas.leich@metop.de

ABSTRACT

There are many different implementation approaches to realize the vision of feature oriented software development, ranging from simple preprocessors, over feature-oriented programming, to sophisticated aspect-oriented mechanisms. Their impact on readability and maintainability (or program comprehension in general) has caused a debate among researchers, but sound empirical results are missing. We report experience from our endeavor to conduct experiments to measure the influence of different implementation mechanisms on program comprehension. We describe how to design such experiments and report from possibilities and pitfalls we encountered. Finally, we present some early results of our first experiment on comparing CPP with CIDE.

Categories and Subject Descriptors. D.2.13 [**Software Engineering**]: Resuable Software; D.3.3 [**Software Engineering**]: Language Constructs and Features

General Terms. Experimentation, Human Factors, Languages

Keywords. Program Comprehension, Empirical Software Engineering, FOSD, Preprocessor, CIDE

1. INTRODUCTION

In software development, a large amount of money is spent on software maintenance [28]. One major part of maintaining software is understanding code [41]. Therefore, one important goal in software engineering is to develop concepts, languages, and tools that aid understanding in order to reduce maintenance costs.

One paradigm that aims at increasing understandability is *feature-oriented software development (FOSD)* [4]. The key abstraction of FOSD is a *feature*, which represents a product characteristic or domain abstraction relevant to stakeholders. FOSD aims at separation of concerns in terms of features, even for crosscutting and interacting features, and provides corresponding abstraction and implementation mechanisms [4, 8, 35]. Modularizing software in terms of features promises improved understandability, because concerns can be traced directly from the problem space (domain description) to the solution space (implementation) [4, 29].

There are numerous implementation approaches for FOSD. Examples are preprocessor-based implementations with the C preprocessor [19], XVCL [20], or CIDE [23]; *aspect-oriented programming (AOP)* [25] with languages like AspectJ or AspectC; and *feature-oriented programming (FOP)* [35] with languages or tools like Jak/AHEAD [8] or FeatureC++ [3]. Although all approaches aim at the common goal of separation of concerns, they use very different mechanisms, ranging from annotations with `#ifdef` directives, over superimposition, to sophisticated weaving mechanisms. Which of the FOSD approaches has the best effect on understandability? Due to the immense differences between all these approaches, this cannot be answered easily. If we asked developers, we would get very different opinions [5]. Is there a way to evaluate understandability of different FOSD approaches in a sound way that is not just based on subjective opinions?

Recently, we naively set out to conduct an experiment to compare FOSD approaches empirically. Initially, we wanted to measure understandability, from here on referred to as *program comprehension*, for all common FOSD approaches and provide a ranking or results like "developers are able to understand an AspectJ-based implementation by 35 % faster than an equivalent preprocessor-based implementation" [5]. We soon realized that this would not be so simple: (a) comparing complete FOSD approaches to derive a ranking, e.g., AOP vs. FOP, is nearly impossible, because of the number of parameters that would have to be considered, and thus (b) only few aspects of program comprehension can be measured feasibly.

In this paper, we report our experience of designing such experiment, and possibilities and pitfalls we encountered. We present some early results of our first experiment on comparing CPP with CIDE. This way, we want to convey some intuition on the boundaries of measuring program comprehension, to show what can be evaluated empirically and what requires an unrealistically high amount of effort. We hope that other researchers pick up empirical evaluations, so that we can eventually combine the results to a larger picture on program comprehension in FOSD approaches.

2. BACKGROUND

In this section, we give a brief overview of different implementation mechanisms. We selected four mechanisms: (1) aspect-

FOSD'09, October 6, 2009, Denver, Colorado, USA.

```
1 public class Stack {
2   LinkedList items =  new LinkedList();
3   public void push(Object i) { items.addFirst(i); }
4   public Object pop () { return items.removeFirst(); }
5 }
```

Figure 1: Base implementation of a stack.

```
1 public aspect Safe {
2   pointcut safePop(Stack stack):
3     execution(Object pop()) && this(stack);
4   Object around(Stack stack): safePop(stack) {
5     if (stack.items.size() > 0) return proceed();
6     return null;
7   }
8 }
```

Figure 2: Aspect-oriented implementation of *Safe*.

oriented programming with AspectJ, (2) feature-oriented programming with Jak, (3) annotating feature code with CPP's `#ifdef` directive, and (4) annotating features in CIDE. We selected AspectJ, Jak, and CPP because their influence of program comprehension is controversially discussed [5, 13, 29, 30, 40]. CIDE was selected because it is part of our own research.

While explaining the approaches, we use a running example of a class *Stack* in Figure 1, which we subsequently extend with a feature *Safe* to ensure that no elements can be popped of an empty stack.[1]

Aspect-oriented programming with AspectJ.

AOP was developed to modularize otherwise scattered and tangled code of crosscutting concerns in aspects [25]. An aspect can alter the structure of the base program by means of inter-type declarations and can alter the behavior of the base program by means of advice, which is executed at certain join points selected by pointcuts. AspectJ is a popular aspect-oriented language based on Java that implements these concepts and provides a compiler to weave aspects into Java code [26]. Several researchers have shown that AOP is suitable to implement features (e.g., [15]).

In order to illustrate AOP, we show one possible AspectJ implementation of the feature *Safe* in Figure 2. The extension to the base stack is encapsulated in an aspect called `Safe` that defines the pointcut `safePop`. The pointcut captures the execution of the method `pop`. Advice, declared with `around()`, ensures that the method pop is only executed if the stack is not empty; otherwise, it returns `null`.

Feature-oriented programming with Jak.

FOP has a similar goal to modularize crosscutting concerns, but uses a different implementation strategy: Classes are split into class fragments according to features and all class fragments of a feature are encapsulated in a feature module [8, 35]. Each class fragment contains only the part of the class that is necessary to implement the corresponding feature. To derive a program, all class fragments of a class are composed during compilation.

In Figure 3, we show the implementation of the feature *Safe* with Jak, an FOP dialect of Java implemented as part of the AHEAD tool suite [8]. The keyword `refines` is used to indicate that the declaration specifies not a full class but only a fragment (called class refinement) and the keyword `Super` is used to specify how to methods can be merged. To compile a program that includes the feature *Safe*, the class fragments of Figures 1 and 3 are composed

[1] We omit the usual method `top` for brevity.

```
1 refines class Stack {
2   Object pop () {
3     if (items.getSize() > 0) return Super.pop();
4     return null;
5   }
6 }
```

Figure 3: Feature-oriented implementation of *Safe*.

```
1 public class Stack { //...
2   public Object pop() {
3 #ifdef SAFE
4     if (items.size == 0) return null;
5 #endif
6     return items.removeFirst();
7   }
8 }
```

Figure 4: CPP implementation of the stack.

(members are merged into a single class, methods with the same name are composed by inlining the original behavior at the *Super* call. During composition, members are merged into a single class and methods have the same name are merged (the original method is inlined at the *Super* call)).

Annotations with the C preprocessor (CPP).

A different way to implement features comes from textual preprocessors like CPP [19] or those in various product line tools like XVCL [20], pure::variants[2], or Gears[3]. Instead of separating features into distinct files (physical separation of concerns), they are only annotated in a common tangled implementation. With these annotations, we can trace each feature from the problem space to its scattered implementation.

With CPP, developers use `#ifdef` and `#endif` directives to annotate code. Since the CPP is a text-based processor, it can also be applied for other programming languages than C. Furthermore, everything can be annotated, even just an opening bracket, which is very flexible but leaves creating reasonably annotate source code to the discipline of the programmer. Academics often criticize preprocessor annotations for negative effects on readability and maintainability [13, 40], but due to their simplicity they are common in industrial practice.

In Figure 4, we show the implementation of *Safe* using CPP. Lines 7–9 assure that elements can only be popped from a non-empty stack.

Annotations with CIDE.

CIDE was developed at the University of Magdeburg [23]. Instead of textual annotations like with CPP, in CIDE, features are annotated in a tool infrastructure and are represented with background colors, one color per feature. Additionally, modularity is emulated by providing views on the source code (show only the code of feature X) [24]. A further difference to CPP is the kind of allowed annotations: While CPP works on plain text, thus allowing us to annotate everything, CIDE uses the underlying structure of the according source code file to enforce disciplined annotations, assuring that not arbitrary text, but only classes, methods, or statements can be annotated [23].

In Figure 5, we show our stack example a last time, this time with a gray background color to denote the feature *Safe*.

[2] `http://www.pure-systems.com`

[3] `http://www.biglever.com/`

```
1 public class Stack { //...
2   public Object pop() {
3     if (elements.size == 0) return null;
4     return elements.removeFirst();
5   }
6 }
```

Figure 5: CIDE implementation of the stack.

Which approach is the most understandable?.

So, which of the presented FOSD approaches provides most benefit on program comprehension? As the examples demonstrate, the approaches differ considerably, so this question cannot be answered easily. For example, AOP uses a sophisticated join point model, whereas FOP creates class fragments inside feature modules. CPP and CIDE even use very simple mechanisms and do not separate feature code at all. The syntax of AspectJ has a large number of new concepts and keywords, whereas in Jak and CPP, two keywords suffice. On the other hand, compared to Jak, AspectJ is more expressive to extend a base program, and CPP allows almost every kind of changes. Do these constructs provide a benefit on program comprehension or a drawback? Do circumstances exist in which AspectJ is more comprehensible than Jak and vice versa? What about CPP and CIDE? In the remainder of the paper, we show our experience in determining this empirically and present first results.

3. DESIGNING EXPERIMENTS ON PROGRAM COMPREHENSION

There is an overwhelming body of literature on how to conduct experiments in such a way that they are sound. If we are not careful how to conduct the experiments, we can easily get biased results. For example, when asking subjects (the individuals that participate in our experiment) how well they understand an implementation in each of our four languages, we may get subjective results that are heavily influenced by their background and personal opinion (some may have learned to use AspectJ, others may be experienced with C++ development and preprocessor usage). Therefore, we must be very careful how to design and conduct experiments. We started by briefly reviewing the literature on controlled experiments in general and give a very short overview of the most important concepts[4]. Readers already familiar with conducting experiments may skip this section.

An *experiment* is a systematic research study in which the investigator directly and intentionally varies one or more factors (*independent variables*) while holding everything else constant and observes the results (*dependent variables*) of the systematic variation [16]. From this definition, three criteria for experiments can be deduced. Firstly, an experiment must be designed such that other researchers can replicate it (*replication*). This is an important control technique in empirical research. Secondly, the variations of the factors must be intended by the investigator (*intention*). Random variations should be avoided because they prevent replication. Thirdly, it is important that factors can be varied (*variability*). Otherwise, an effect on the result cannot be observed depending on the variations of the factors.

The process of experimental research can be divided into five stages. In the first two stages, *objective definition* and *design*, the experiment is prepared. During *execution*, we run the experiment and collect data, which we analyze during *analysis*. Finally, we interpret our results during *interpretation*. We introduce these stages and relevant terms [21].

[4]A comprehensive discussion can be found in [14, 38].

3.1 Objective Definition

The first two steps when starting an experiment is to define the variables of the experiment and to specify hypotheses that should be tested. In our case, we have one independent variable: the FOSD approach. Since we want to assess the understandability of AspectJ, Jak, CPP, and CIDE, our independent variable has four *levels*. Our dependent variable is program comprehension, because we want to assess whether and how different FOSD approaches influence program comprehension.

A *hypothesis* can be defined as an educated guess about what should happen under certain circumstances [16]. One important criterion is that hypotheses are *falsifiable*, i.e., that we can reject them [33]. Hypotheses that are continually resistant to be falsified are assumed to be true, yet it is not proven that they are. The only claim we can make is that we found *no evidence to reject* our hypotheses.

Program comprehension is 'the process of understanding a program code unfamiliar to the programmer' [27]. Depending on the amount of domain knowledge, there are different models for how program code is understood. In *bottom up models*, a program is analyzed by examining statements and grouping them to chunks, which are iteratively abstracted to a high level understanding of source code. A programmer uses a bottom up approach when he has no knowledge of the program's domain (e.g., [32]). Otherwise, he can use his domain knowledge to create hypotheses about a program's purpose and verifies or rejects them by examining the code (top down or integrated models) (e.g., [9]).

So, how can we *measure program comprehension*? Several techniques and measures exist in the literature with differing reliability and effort in applying them (see [11] for a comprehensive survey). Typical techniques include maintenance tasks, mental simulation (e.g., pen-and-paper execution of the source code) and think-aloud protocols (i.e., subjects verbalize their thoughts during comprehending a program [1]). Usually, correctness, completeness, and time to solve a task are used as measure for program comprehension. Which measure to chose depends on the experiment, we will discuss our choice later.

So, how could our hypotheses look like? An example could be 'The number of errors of a maintenance task is lower for Jak than for AspectJ with bottom up program comprehension', which exactly defines the technique, measure, and program comprehension model to which our hypothesis is applicable.

Finally, it is imperative that we define our hypotheses before designing or actually executing the experiment, because decisions in subsequent stages depend on the hypotheses (e.g., the subjects we include or the analysis methods we apply) [16]. In addition, it prevents us from the bad practice of 'fishing for results' in our data and thus discovering random relationships between variables [12].

3.2 Design

The next step is to design our experiment so that we are able to evaluate our hypotheses. During this stage, internal and external validity as well as confounding variables have to be considered.

- A *confounding variable* is a parameter that influences the dependent variable besides variations of an independent variable [16]. In order to soundly measure the influence of FOSD approaches on program comprehension, we need to identify and control the influence of confounding variables. For example, programming experience could influence program comprehension more than using different FOSD approaches. If we accidentally distribute all the experienced programmers in one group and all the novices in another, this could overshadow our measures and cause biased results.

- *Internal validity* describes the degree to which the value of the dependent variable can be assigned to the manipulation of the independent variable [38]. This means that we have to control the influence of all confounding parameters (e.g., noise level or programming experience).
- *External validity* is the degree to which the results gained in one experiment can be generalized to other subjects and settings [38]. The more realistic an experimental setting is, the higher is its external validity. Hence, we could conduct our experiment in a company under real working conditions with employees of the company. Now, however, our internal validity is threatened, because we cannot control the influence of confounding variables like programming experience.

When designing experiments, we have to find a compromise between both kinds of validity. For example, if we do not know how our variables interact or our resources are rather limited, we can start with experiments that maximize internal validity (e.g., by using only unpaid students of the same university of the same programming course). This is also the path we take in our experiments. Once we have established a hypothesis, we can design experiments that are more realistic.

A first step in controlling confounding variables is to identify them. Since the number of confounding parameters on program comprehension is large, we discuss them separately in Section 4. When identified, we need to control them. In literature there are a number of approaches to control confounding variables: randomized sampling, keeping the parameter constant, including parameter as independent variable, ex post analysis of the parameter, or experimental designs [2, 16]. In our experiments, we use a simple experimental design and usually randomization, assuming that statistical errors even out with a large enough sample.

3.3 Execution and Analysis

If we carefully design our experiment, running it usually the easier part. In this stage, we recruit subjects, let them complete our tasks, and collect our data as planned.

Having collected the data, we need to describe and analyze them. For describing our sample and data, we can compute some descriptive statistics, e.g., frequencies, means, or standard deviation. This information is necessary for replicating our experiment [2].

After describing the data, we can apply significance tests to evaluate our hypotheses [2]. Those tests are necessary in order to determine whether a difference we encountered is significant or just appeared randomly. Depending on the data, we can apply different tests. For example, if we want to check whether frequencies of correct answers differ between Jak and AspectJ, we use a χ^2-test [2]. If we want the check whether measured times to complete a task differ between Jak and AspectJ, we can use a t-test or Mann-Whitney-U-test, depending on how our data are distributed [2]. For analyzing two independent variables, e.g., programming experience and FOSD approach, there are further tests like ANOVA [2]. An overview of significance test and their requirements and application can be found in [2]. Statistical tools like *SPSS* or *R*[5] help to analyze the data.

4. CONFOUNDING VARIABLES ON PROGRAM COMPREHENSION

After our brief overview of controlled experiments in general, we discuss confounding variables (also called confounding parameters) on program comprehension. Identifying and controlling confounding parameters is necessary to allow us to draw sound conclusions from our result and avoid bias. During our design, we found a high number of confounding parameters (by literature review and consulting experts). Due to space limitations, we group the parameters into three categories and pick one parameter per category to explain its influence and how it could be controlled. The remaining parameters are discussed in detail in [14].

Personal parameters.

Personal parameters are related to the subjects of an experiment. As example parameter, we discuss programming experience.

Consider an expert and a novice programmer, who both solve a task in an experiment. The chance that the expert programmer has dealt with a program similar to the task at hand is considerably higher than for a novice programmer. In the case an expert knows the kind of problem, but a novice does not, the cognitive processes for understanding the source code of the task are not the same. Whereas the expert *uses* his knowledge to solve a task, the novice *acquires* knowledge. Hence, to avoid biased results (we would not solely measure whether one program is more understandable than another, but additionally the effect of programming experience on program comprehension), we control the influence of programming experience in our experiment. Of the control strategies discussed in Section 3.2, we use pseudo randomization, because of the high influence of programming experience on program comprehension.

To control the influence of programming experience, we need to measure it. In the literature, programming experience is diversely understood. We found several aspects that were used, for example years of practical programming or number of programming courses at college. Since there is no common definition or questionnaire, we need to consider relevant aspects of programming experience depending on the hypotheses of an experiment. For our experiment, we used Java code and asked in how many Java projects the subjects have participated in a preliminary survey. Details of this survey can be found in [14]. Based on the measured experience, we divided the subjects into two even groups.

Besides programming experience, we identified domain knowledge, intelligence, and education as confounding parameters. Except for domain knowledge, measuring personal parameters is hard, because either no common understanding exists (programming experience, intelligence) or the measurement is difficult (intelligence, education). Depending on the hypotheses as well as human and financial resources, according means to control the influence need to be defined. We kept domain knowledge and education constant and randomized intelligence in our experiment.

Environmental parameters.

The second group of confounding parameters is specific to experimental situations that assess program comprehension as well as experiments in general. We discuss tool support in more detail.

Nowadays, software development is supported by tools that foster program comprehension. However, before functionalities of a tool can be used, persons need to familiarize with it. Even after an equal amount of time, persons can use different sets of features of the same IDE. Hence, letting persons use the same IDE does not control the influence of tool support sufficiently: Some subjects need to familiarize with it, whereas others may not know how to use a certain functionality. Letting subjects use their preferred IDE prevents that subjects have to familiarize with an unknown tool, but introduces variations in tool support.

Depending on our hypotheses, we need to control tool support: If the comprehensibility of two languages should be compared, tool support would confound the result, because not the comprehensibility of the language alone, but also how the language is supported

[5] see `www.spss.com` and `www.r-project.org`

by the tool is measured. On the other hand, if skills of subjects should be measured, letting them use their preferred tool is more advisable, because they do not have to familiarize with a new one. In our experiment, we eliminated tool support.

Other environmental parameters beyond tool support include training and motivation of subjects, noise level, position effects, ordering effects, test effects, the Hawthorne effect [36], and the Rosenthal effect [37]. All of them can be more or less easily controlled. For example, we could pay our subjects for good performance in our experiment, *if* we have according financial resources. In our experiment, we used a warming up task to let subjects familiarize with the experimental setting and tried to keep all other parameters constant.

Task-related parameters.

Task-related parameters are caused by the experimental tasks and source code. As an example, we discuss comments.

As shown by Prechelt et al. [34], commented code is significantly easier to understand than uncommented code. Hence, for controlling the influence of comments in our experiments, comments must be comparable in different versions and must have the intended effect (e.g., support subjects in comprehension, not confusing them). We can conduct pretests, consult experts, or assess the opinion of subjects to control the influence of comments.

Further parameters include structure of source code, coding conventions, difficulty of the task, syntax highlighting and documentation. We can control the influence of most of them like the influence of comments (i.e., conduct pretests, consult experts, or assess the opinion of subjects).

5. COMPARING FOSD APPROACHES

Now, we come back to our initial goal to compare FOSD approaches. We show that due to the sheer number of confounding parameters discussed in the previous section, the scope of experiments that soundly *and* feasibly assess the influence of FOSD on program comprehension is very small. We started by naively designing an experiment to assess the understandability of AspectJ, Jak, CPP, and CIDE in one experiment, but soon found out that we could not realistically conduct it. Then, we tried a more narrow approach by comparing AspectJ and Jak, which also turned our as unrealistic. Finally, we found that we have to proceed in small steps, as we will show.

5.1 Four approaches

Initially, we naively wanted to compare AOP vs. FOP vs. preprocessors in general [5]. However, we found that the programming language is an important confounding parameter. Are the results the same whether we use AspectJ [26], CaesarJ [6], or Alpha [31]? Is there a difference when we use Java, or C or some other language as host language? To explore these effects, we need to consider programming languages as independent variables as well, and there can easily be dozens of languages for AOP and FOP and even preprocessors. Optimistically assuming just five programming languages per approach, we already have to create $5 \cdot 3 = 15$ versions of a program, which all must be comparable regarding difficulty, commenting style, structuring, etc. (i.e., confounding task-related parameters). Besides creating 15 comparable programs, we must recruit subjects for 15 groups. Alternatively, we could 're-use' our subjects such that one subject works with all programming languages of one FOSD approach, but this way each subject needs considerably longer to complete our task (five programs instead of one) and we need to control several test effects.

At this point, we have not even started with other confounding parameters like programming experience (maybe Jak is easier for novices and AspectJ is faster to understand for experts) or tool support (maybe without tool support Jak is better than AspectJ but with tool support it is the other way around), and can already show that the experiment will become extremely complex and require many subjects. If we also include the effect of programming experience, tool support or other parameters, we easily reach designs where we need thousands of subjects or tasks that take several days. Hence, we cannot feasibly compare FOSD approaches as general as AOP vs. FOP in one experiment.

5.2 Two approaches

If we restrict our comparison to AspectJ and Jak, we restrict ourselves to just two approaches with one language each. We reduce external validity, because we can only provide results about these two languages but not about AOP or FOP in general.

Still, there are many confounding variables left. In order to be able to generalize our result, we need to include several levels of programming experience, tool support, comments, etc. as independent variable. If we include just experts and novices, IDE and text editor versions, as well as commented and uncommented versions, our required number of subjects would be too large again ($2 \cdot 2 \cdot 2 \cdot 2 = 16$ groups).

Even if we fix all this, we need two comparable programs, one written in AspectJ and one in Jak. How can we make sure that they are comparable regarding their structure, comments, difficulty, etc., despite the significant differences between both languages? AspectJ and AHEAD differ considerably, for example, regarding the keywords, structure of source code, composition mechanism, etc. There are numerous ways of implementing the same problem in AspectJ (of course, the same is true for Jak). So, is the AspectJ program just difficult to understand due to the given implementation, or due to AspectJ's mechanisms in general? How can we eliminate the effect of different implementations of the same problem? Furthermore, we only would compare the implementation of *one* problem. Maybe for some other problems, the outcome would be reversed?

Hence, even comparing two programming languages, is nearly impossible. We can only derive results for specific implementations of specific problems. So, what aspects of an FOSD approach and its effect on program comprehension can we feasibly measure?

5.3 Realistic comparison

In order to feasibly design experiments, we need to restrict our external validity even more. This means that we cannot test *everything* with *one* experiment in which we analyze the effect of all levels of our independent variable and all confounding parameters on program comprehension. Instead, we have to choose the scope of our experiment so small that we can reliably measure program comprehension *and* do not exceed our available resources.

Hence, a first step is to keep most confounding parameters constant. This way, our experiment has a low degree of external validity, but it can be feasibly conducted and we can draw sound conclusions. Second, we restrict our experimental design to the simplest comparison: one independent variable with two levels. This reduces the number of subjects we need. Furthermore, we can compare two levels that are rather similar. This helps us to create comparable conditions in our experiment (e.g., two versions of source code that differ only in few aspects).

Examples of feasible comparisons are the effect of a few keywords on understandability (e.g., does it make a difference to have or not have *cflow* in AspectJ, does it make a different whether Jak uses the keyword *refines* or *layer*?) or comparing programs in the

same programming language, but with different annotations (CIDE vs. CPP). Furthermore, we could only use male students as subjects that have completed the same programming courses at the same university, are familiar with the same programming languages and domains and have an average IQ. In this case, we can only generalize our results to individuals with the same characteristics. The challenge is to find the right balance.

6. DEMONSTRATION EXPERIMENT

In this section, we describe an experiment comparing the effect of CPP and CIDE on program comprehension. The purpose of this description is not provide all necessary information to replicate our experiment (which is described in [14]), but to illustrate the small scope of feasible *and* sound experiments, which we derived in the previous section.

Our goal is to measure whether using colors in CIDE instead of textual annotations à la CPP has an effect on program comprehension. As shown in Section 2, both approaches are quite similar, and we expected that, when ignoring all tool support like views, the kind of annotation has no effect on program comprehension.

6.1 Objective definition

Our independent variable has two levels: textual annotations à la CPP and annotations using background colors à la CIDE. Our dependent variable, program comprehension, was measured with four maintenance tasks (given a bug description, subjects had to find the cause in the source code and fix it). We assessed the time to solve a task and whether a task was completed successfully.

Our hypotheses are:

- There are no differences in solving time between Java-CPP-annotated and Java-CIDE-annotated source code with bottom up program comprehension.
- There are no differences in the number of completed tasks between Java-CPP-annotated and Java-CIDE-annotated source code with bottom up program comprehension.

We expect no differences, because for all tasks, subjects need to analyze source code on a textual basis. Hence, it should be irrelevant how the according source code statements are annotated.

6.2 Controlling confounding variables

In order to control the influence of personal parameters, we measured programming experience in a pre-test and used matching to create homogeneous groups according to programming experience (and gender). Since our sample was large enough (about 50 subjects), we assume that both groups are homogeneous according to intelligence, too.

We chose the domain of software for mobile devices, which was unfamiliar to all subjects (ensured in pre-test), thus enforcing bottom up program comprehension. Regarding education, we selected subjects that took an advanced programming course at the University of Passau, which required several basic programming courses.

In order to control environmental parameters, we conducted the experiment in a browser, not in an IDE, thus excluded an influence of tool support. We created a HTML file for every source code file. A link to every file was displayed at the left side of the screen (similar to the package explorer of Eclipse). Subjects were not allowed to use the search function of the browser.

As training, we gave one neutral introduction in one room for all subjects to CPP and CIDE with familiar source code examples. The experiment was also conducted in one room (i.e., same noise level, etc.). For controlling the influence of motivation, subjects were required to participate in our experiment to complete their course and could enter a raffle for an Amazon gift card. All subjects knew that they participated in an experiment. Due to our limited resources, we did not conduct a repeated measure (hence: excluded test effects) or switched the order of the task. To control position and ordering effect, we used a warming up task, which took about ten minutes and should subjects familiarize with the source code. Furthermore, the tasks were arranged with increasing difficulty, so that, with each task, subjects were more familiar with the source code.

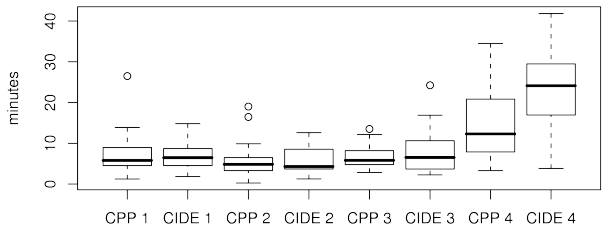

Figure 6: Response times with CPP and CIDE for all four tasks.

In order to control task-related parameters, we used a code-reviewed Java source code for an application for mobile devices, developed by others [15]. The code has about 3800 lines of code, 37 classes, and 4 features were already annotated using textual *#ifdef* statements. For creating the CIDE version, we deleted all lines that contained preprocessor statements and colored the background of all feature source code with the same colors as CIDE uses (see [14] for details). Since we used a code-reviewed version, structure, coding conventions, comments, documentation was already approved by experts, and our changes for CIDE did not affect them. For syntax highlighting, we used the same style as Eclipse, because all of our subjects were familiar with it.

Since we had the same source code, we could use the same tasks for both versions. All four maintenance tasks we created by introducing bugs that occurred during runtime (forcing the subjects to examine the control flow of the program) in the source code of a specific feature. Du to space limitations we have to defer the interested reader to [14] for details on these tasks. Before the experiment, we confirmed that the bugs we produced can be found by subjects in a reasonable amount of time (within two hours) with a pre-test with some students from the University of Magdeburg.

6.3 Results

We report our results, before we interpret them. This separation is standard practice [7] to ensure that readers can distinguish results from interpretation, which helps to understand the consequences we draw from our result.

For testing our first hypothesis (no difference in solving time), we conducted a Mann-Whitney-U-test [2] to compare the mean solving time of both versions. For the first three maintenance tasks, we found no differences in solving time. For the last task, CPP subjects were significantly faster, which means that we have to reject our hypothesis. For visualization, we show box plots[6] of the times for all four maintenance tasks for each CPP and CIDE in Figure 6.

We tested our second hypothesis (no difference in number of

[6]A box plot is a common form to depict groups of numerical data and their dispersion. It plots the median as thick line and the quartiles as thin line, so that 50 % of all measurements are inside the box. Values that strongly deviate from the median are outliers and drawn as separate dots.

completed task) with a χ^2 test, which checks whether observed frequencies significantly differ from expected frequencies. We found no significant differences in the number of completed task, which confirms our hypothesis.

How can those results be interpreted? Since we found a difference in response times for one task, we must reject our according hypothesis. Now, we have to interpret what this means. Why did a difference for the last task occur? The bug in the last task was located in a class that was entirely annotated with red as background color. We suspect that this color was the main reason for the performance difference in the task. To confirm this suspicion, we look through the comments subjects were encouraged to give us after the experiment. Some subjects indeed marked this as problem and wished they could have adjusted the intensity of the background color to their needs.

Note that although we did not observe significant differences in our data (except for the last maintenance task), this does not mean that there are none. Instead, what our results show is that we have found no evidence of differences. It is possible that there are indeed effects on response time or number of correctly solved tasks, yet the effect could be too small for our sample to reveal or that other confounding variables we did not think of eliminated the effect. This is one reason why replication is crucial to empirical research: Only if a hypothesis is continually resistant to be rejected, we can assume that the relationship it describes indeed exists.

Hence, next steps in assessing the understandability of CIDE and CPP are to replicate our experiment and confirm our second hypothesis (i.e., that the kind of annotation has no effect on the number of completed tasks). Furthermore, we will explore whether the reason for the performance difference in the last maintenance task was the background color of the according source code file or something else.

To summarize, we learned from this experiment that, yes, it is possible to measure program comprehension given a sufficiently small scope. This encouraged us to keep on evaluating program comprehension regarding further factors, e.g., tool support in IDEs or disciplined annotations. In this experiment, we found that contrary to our initial expectations coloring code does not significantly increase program comprehension (at least not when searching for a bug in a feature), but on the contrary can even hinder it. This gave us a new perspective on our tool and encouraged us to search for other visualizations or make them adjustable by the user.

7. FURTHER WORK

So, what are next steps in measuring program comprehension? If we think of the demonstration experiment, we can extend our independent variable to other programming languages or create other programs with other degrees of complexity. We can use programming experts as subjects instead of novices. However, we cannot vary all parameters at once, but have choose very few (otherwise, we would exceed our resources).

For annotations with CPP and CIDE, it was relatively easy to create comparable programs, because we ignored tool support and the underlying programming languages are identical, so that the kind of annotation is the only difference between the versions. In a next step we will evaluate tool support.

But how can we start to compare AOP and FOP, which differ so much? The answer is, that we have to start even smaller, for example compare two programs that only differ in their extension (`refines` in Jak vs. an inter-type declaration in AspectJ). When we have collected enough data to explain small differences between Jak and AspectJ, we can incrementally increase complexity and differences of programs and integrate the knowledge into a theory of understandability of Jak and AspectJ. In order to assess the understandability of AOP vs. FOP, we have to generalize our knowledge to other programming languages.

Since the steps in comparing AOP and FOP are rather small, it may take a decade until we have a sound body of knowledge concerning program comprehension of AOP and FOP, let alone all FOSD approaches. Nobody knows whether there is still interest in FOSD in ten years or whether other programming paradigms have emerged by then. It is impossible for one research group to assess the understandability of FOSD approach in a reasonable amount of time with a reasonable amount of financial resources. Hence, with this work, we want to encourage others to take up empirical research and establish a community for measuring program comprehension of FOSD approaches. This way, we have more people working on creating a body of knowledge, which speeds up the process and reduces the errors we make along the process.

8. RELATED WORK

We are not aware of empirical research on program comprehension in the context of FOSD. However, empirical results can be found in other domains, from which we can learn. For example, with the development of the object-oriented paradigm, researchers were curious about the benefit of object orientation compared to procedural languages. Daly et al. [10] assessed the effect of inheritance on understandability. They found performance differences in favor of object-oriented source code (although the opinion of subjects was that maintainability of procedural source code was better). A similar result was found by Henry et al. [17], who compared C and Objective C source code.

In the last couple of years, several systematic reviews about the status of empirical software engineering were published [18, 22, 39]. Although they are not focused on program comprehension, they provide useful advice for empirical research in general (e.g., include programming experts in experiments to ensure external validity or refer to disciplines like cognitive psychology, because comparable problems occurred there along with solutions due to the age of this discipline).

9. CONCLUSION

We reported how we set out to compare FOSD approaches like AOP, FOP, or preprocessor-based implementations empirically regarding program comprehension. We learned that, in order to be able to draw sound conclusions from an experiment (internal validity), it is important to control confounding parameters on program comprehension. However, their sheer number makes large-scoped experiments difficult. It is practically impossible to compare all FOSD approaches at once. Instead, a hypothesis should focus on few aspects of FOSD approaches, because this allows us to feasibly test it. With a small experiment comparing different forms of preprocessor annotations, we demonstrated the feasibility of small-scale experiments.

The next steps in assessing the understandability of FOSD approaches are to define small hypothesis and evaluate them empirically. Once results for those small hypotheses are clear, we can work on more complex hypotheses. In order to speed up this tedious process, it is necessary to establish a research community for empirically assessing the understandability of FOSD approaches. With our work, we hope to motivate some researches to join us.

Acknowledgments. We thank Jörg Liebig for his help on organizing and conducting the experiment in Passau. We thank METOP GmbH for the Amazon gift card for our subjects. Feigenspan's work is supported in part by BMBF project 01IM08003C (ViER-

forES). Apel's work is supported in part by DFG project #AP 206/2-1.

10. REFERENCES

[1] J. R. Anderson. *Cognitive Psychology and its Implications*. Worth Publishers, 2000.

[2] T. W. Anderson and J. D. Finn. *The New Statistical Analysis of Data*. Springer, 1996.

[3] S. Apel et al. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, pages 125–140, 2005.

[4] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[5] S. Apel, C. Kästner, and S. Trujillo. On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In *Proc. Int'l Workshop on Assessment of Contemporary Modularization Techniques*, pages 1–7. 2007.

[6] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[7] A. P. Association. *Publication Manual of the American Psychological Association*. American Psychological Association, 2001.

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

[9] R. E. Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering*, pages 196–201. 1978.

[10] J. Daly et al. The effect of inheritance on the maintainability of object-oriented software: an empirical study. In *Proc. Int'l Conf. Software Maintenance*, pages 20–29. 1995.

[11] A. Dunsmore and M. Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS-35-2000, University of Strathclyde, 2000.

[12] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.

[13] J. Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. 1997.

[14] J. Feigenspan. Empirical Comparison of FOSD Approaches Regarding Program Comprehension – A Feasibility Study. Master's thesis, University of Magdeburg, 2009.

[15] E. Figueiredo et al. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering*, pages 261–270. 2008.

[16] C. J. Goodwin. *Research In Psychology: Methods and Design*. Wiley Publishing, Inc., 1998.

[17] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the Maintainability of Object-Oriented Software. In *Proc. TENCON*, pages 404–409. 1990.

[18] A. Höfer and W. Tichy. *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, chapter Status of Empirical Research in Software Engineering, pages 10–19. Springer, 2007.

[19] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, 1999.

[20] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. Software Engineering*, pages 810–811. 2003.

[21] N. Juristo and A. M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer, 2001.

[22] V. B. Kampenes et al. A Systematic Review of Quasi-Experiments in Software Engineering. *Information and Software Technology*, 51(1):71–82, 2009.

[23] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering*, pages 311–320. 2008.

[24] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. Workshop Visualization in Software Product Line Engineering*, 2008.

[25] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 220–242. 1997.

[26] G. Kiczales et al. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 327–353. 2001.

[27] J. Koenemann and S. P. Robertson. Expert Problem Solving Strategies for Program Comprehension. In *Proc. Conf. Human Factors in Computing Systems*, pages 125–130. 1991.

[28] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[29] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 169–194. 2005.

[30] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proc. Int'l Symposium Foundations of Software Engineering*, pages 127–136. 2004.

[31] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 214–240, 2005.

[32] N. Pennington. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychologys*, 19(3):295–341, 1987.

[33] K. Popper. *The Logic of Scientif Discovery*. Routledge, 1959.

[34] L. Prechelt et al. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, 2002.

[35] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, pages 419–443. 1997.

[36] F. J. Roethlisberger. *Management and the Worker*. Harvard University Press, 1939.

[37] R. Rosenthal and L. Jacobson. Teachers' Expectancies: Determinants of Pupils' IQ Gains. *Psychological Reports*, 19(1):115–118, 1966.

[38] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin, 2002.

[39] D. I. K. Sjoberg et al. A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.

[40] H. Spencer and G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198, 1992.

[41] A. von Mayrhauser and A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.