Comparing Program Comprehension of Physically and Virtually Separated Concerns

Janet Siegmund*
University of Magdeburg, Germany

Christian Kästner
Philipps University Marburg, Germany

Jörg Liebig, Sven Apel
University of Passau, Germany

ABSTRACT

It is common believe that separating source code along concerns or features improves program comprehension of source code. However, empirical evidence is mostly missing. In this paper, we design a controlled experiment to evaluate that believe for feature-oriented programming based on maintenance tasks with human participants. We validate our experiment with a pilot study, which already preliminarily confirms that students use different strategies to complete maintenance tasks.

Keywords

Separation of Concerns, Program Comprehension, FeatureHouse, Ifdef

1. INTRODUCTION

Separation of concerns is an essential strategy to implement understandable and maintainable software [21]. Besides classic programming mechanisms, such as procedures and objects, many novel mechanisms for separation of concerns have been proposed in the past: components [10], aspects [14], hyper-modules [23], and so forth. Similarly, *feature-oriented programming (FOP)* advocates to structure software along the *features* it provides (i.e., user-visible characteristic of a software system) [5, 22]. That is, features are made explicit in design and code in the form of feature modules—one feature module implementing one feature.

In our field, it is common to believe that separating code along features improves program comprehension. However, program comprehension is an internal cognitive process that we cannot observe directly [15]. Thus, it is not sufficient to rely on plausibility arguments in the debate of whether some concept or mechanism improves program comprehension. Instead, we need controlled experiments to measure it [3, 8].

In this paper, we set out to evaluate whether separating features into separate feature modules improves program comprehension. In particular, we concentrate on the mechanism of FOP as implemented in the tool FeatureHouse [2]. In FOP, developers can trace each feature to one *physically separated* feature module. We compare the effect of physical separation on program comprehension to an implementation in which features are annotated with conditional-compilation directives such as `#ifdef`s. We speak of *virtual separation*, because the `#ifdef` directives allow developers to trace a feature to its scattered implementation throughout the source code.

To this end, we designed a controlled experiment, in which we observe how participants comprehend source code during maintenance tasks. As material, we used two comparable software systems—one decomposed physically in terms of feature modules and one annotated with preprocessor directives. Based on experimental results, we can give recommendations on which technique of separating code along features is suitable for which task and how to improve them.

Our contributions are twofold:

- We design a reusable experiment to evaluate the impact of physical separation with FeatureHouse on program comprehension.
- We conducted a pilot study to validate the experiment and prepare a large scale run.

We plan to execute the experiment with a larger sample in the fall term. We appreciate feedback and additional research questions to evaluate. We also invite others to conduct this or similar experiments. Therefore, we provide all necessary material online at `http://fosd.net/experiments`.

2. PHYSICAL VS. VIRTUAL SEPARATION

To separate crosscutting concerns, several programming techniques were developed, including *aspect-oriented programming* [14] and FOP [22], which aim at dividing the source code into modules regarding concerns or features.

FOP as implemented by AHEAD [5] and FeatureHouse [2] separates code belonging to different features *physically* into separate folders, one folder per feature (and per interaction). Each folder may contain multiple packages and (partial) classes that implement the corresponding feature. To generate a product for a specific feature selection, the code from the selected features is composed, such that classes and methods that have the same name are merged by superimposition [2].

As a base line for comparison, we use *virtual separation* with `#ifdef` directives, in which features are merely mapped to code fragments with annotations in the source code. A

*The author published previous work as Janet Feigenspan.

FOSD'12, September 24–25, 2012, Dresden, Germany.

common mechanism is to use `#ifdef` directives in the source code to indicate which code fragments belong to which features. To generate a product for a specific feature selection, a preprocessor removes the code of all deselected features. In this approach, code belonging to a feature may be scattered over multiple classes and may be tangled with code of other features. The name *virtual* separation comes from separate tools that can create views on the source code of specific features, thus emulating modules [13]; these views are not further considered in this paper, because they deserve an evaluation of their own.

Both strategies, physical and virtual separation, allow a tracing from features to code fragments. Using physical separation, each feature can be traced to one directory, whereas, using virtual separation, we can trace a feature to multiple code locations using a global search.

To illustrate virtual and physical separation, we show example in Figure 1. Both excerpts show code from MobileMedia, a software for the manipulation of media on mobile devices [9]. On the left, we show virtual separation implemented with `#ifdef` directives; on the right, an implementation of the same code with FeatureHouse.

In prior work, we and others discussed trade offs between physical and virtual separation [3, 12, 13, 17, 18]. Physical separation has been claimed to improve code comprehension, because, by separating features into folders, the amount of information is limited; only relevant code of a feature is present. Hence, developers might be less distracted and can focus on the code of a single feature during maintenance tasks. However, we also made the experience that (potentially due to the lack of interfaces), to understand code of a feature, base code also has to be understood. Hence, there might be important information missing, which developers have to look up in different folders. This might slow developers down compared to virtual separation, in which information of base code and feature code (but also code of other features) is present in one file. To evaluate whether physical separation of concerns indeed improves program comprehension, we designed a controlled experiment, described next.

3. EXPERIMENT DEFINITION

To evaluate whether physical separation of concerns à la FeatureHouse has a benefit on program comprehension, we designed a controlled experiment. To describe the settings and results, we use the guidelines provided by Jedlitschka and others [11]. To support replication, we provide all material of the experiment at the project's website (`http://fosd.net/experiments`).

3.1 Objective

With our experiment, we target the question whether participants understand physically separated source code (feature modules) different than source code that is virtually separated (preprocessor directives). To understand our research question, we need to understand how humans process information. To process information from the outside world, we use our working memory, which holds information we perceive and makes it available for further processing [4]. However, working memory capacity is limited to only few *items*, which are units of information, for example, digits of a telephone number or objects on a shopping list [19]. By structuring information, we can store more information. For example, we can group information of a shopping list into groceries and clothing and then memorize few items of the groceries and few items of the clothing category. In physically separated source code, the amount of information presented in one place is smaller and more clearly structured, so the working memory of participants might not be stressed too much.

However, when the present information is not enough to understand code, participants need to search relevant information. Hence, they might need more time, and during their search, they have to keep in mind where their search started. For that, they need more working memory capacity. Our first research questions are the following:

RQ$_{1/2}$: Does physical separation of concerns improve program comprehension in terms of correctness/response time?

Additionally, we are interested in the search behavior of participants. In virtually separated code, files are larger, because they typically contain code of several features. Thus, participants may use the search function often to find information. In physically separated code, one file contains information of only one feature; hence, relevant code may be easier to find without using the search or using the search less frequently. However, the information presented in one file might not be enough to understand the code, so participants might use a global search (i.e., across modules) more often. Thus, we state a second research question:

RQ$_3$: Is there a difference in the search behavior between physically and virtually separated concerns?

Furthermore, there might be a difference in the strategy participants use to find a bug. Different strategies require different amount of time and cognitive resources, so an efficient strategy can improve program comprehension. In the ifdef version, participants might start by using the global search function to locate code of the relevant feature, because according code is scattered across the project. In the FeatureHouse version, participants might start by opening a file in the relevant feature module, because according code is located only in that module and files are short compared to the ifdef version. Thus, we state a third research question:

RQ$_4$: Is there a difference in the first action to find a bug?

3.2 Material

As material, we use MobileMedia, which was implemented by Figueiredo and others with the help of students in Java ME with the preprocessor Antenna, which enables ifdef directives in Java ME code [9]. We use MobileMedia, because it was carefully developed and evaluated regarding standard coding techniques and design principles, so we can be sure to have minimized confounding effects due to badly implemented code. Furthermore, MobileMedia is often used in research to compare physically and virtually separated code (e.g., [9]). Thus, our results provide further data on the effect of physically and virtually separated code based on MobileMedia or similar systems. Of course, in future work, we need to consider additional software systems to generalize our results.

From the preprocessor version of MobileMedia, we created another version based on FeatureHouse.[1] We selected FeatureHouse, because we had the opportunity to work with

[1]There is also an AspectJ version of MobileMedia, which uses physical separation of concerns. However, AspectJ syn-

```
1  // #if includeMusic || includeVideo
2  ...
3  public class MusicMediaUtil extends MediaUtil {
4    public byte[] getBytesFromMediaInfo(MediaData ii)
5        throws InvalidImageDataException {
6      try {
7        byte[] mediadata = super.
8          getBytesFromMediaInfo(ii);
9        if (ii.getTypeMedia() != null) {
10 //#if (includeMusic && includeVideo)
11         if ((ii.getTypeMedia().equals(MediaData.MUSIC)) ||
12           (ii.getTypeMedia().equals(MediaData.VIDEO)))
13 //#elif includeMusic
14         if (ii.getTypeMedia().equals(MediaData.MUSIC))
15 //#elif includeVideo
16         if (ii.getTypeMedia().equals(MediaData.VIDEO))
17 //#endif
18           { ... }
19       }
20       return mediadata;
21     } catch (Exception e) { ... }
22   }
23 //...
24 //#endif
```

(a) Virtual Separation

```
1  public class MusicMediaUtil extends MediaUtil {
2    private boolean isSupportedMediaType( MediaData ii) {
3      return false;
4    }
5
6    public byte[] getBytesFromMediaInfo( MediaData ii)
7        throws InvalidImageDataException {
8      try {
9        byte[] mediadata = super.getBytesFromMediaInfo(ii);
10       if (ii.getTypeMedia() != null) {
11         if (isSupportedMediaType(ii))
12         { ... }
13       }
14       return mediadata;
15     } catch (Exception e) { ... }
16   }
17   ...
18 }
```

(b) Feature Music_OR_Video

```
19 class MusicMediaUtil {
20   private boolean isSupportedMediaType(MediaData ii) {
21     return original(ii) ||
22         ii.getTypeMedia().equals(MediaData.MUSIC);
23   }
24 }
```

(c) Feature Music

```
25 class MusicMediaUtil {
26   private boolean isSupportedMediaType(MediaData ii) {
27     return original(ii) ||
28       ii.getTypeMedia().equals(MediaData.VIDEO);
29   }
30 }
```

(d) Feature Video

Figure 1: Virtual and physical separation using the preprocessor Antenna (a) and FeatureHouse (b-d).

students who are familiar with it at the same level as with preprocessors. Thus, we do not need a training session and can keep the time for the experiment as short as possible.

To ensure that both versions differ only in the underlying programming technique, two reviewers realized the refactorings. They evaluated the work of the other reviewer on few code fragments. We explicitly encourage other experimenters to evaluate the comparability of both versions and give us feedback.

An important difference between both versions is caused by the technique, such that in the FeatureHouse version, there are more folders, because for every feature or feature combination, a new folder is created, in which files are stored according to the declared packages. In the ifdef version, there are no folders for features or feature combinations, but only those folders defined by the package declarations (which are also present in the FeatureHouse version).

To evaluate our research questions, we use a between-subjects design, so we give one group of participants the ifdef version, and the other group the FeatureHouse version. This way, we can compare the performance of participants of both groups. For the first research question, we analyze response time and correctness for maintenance tasks. Response time is logged automatically, and correctness determined manually by an expert.

For the second research question (regarding the search behavior), we log how participants use the search function during solving maintenance tasks. Participants can use either a local search, that is, within a file, or a global search, that is, in all files and folders of the complete project. Both searches uses strings (no pattern matching or syntactical search).

For the third research question, we log the behavior of participants, that is opening and closing files, switching between files, and using local or global search including the search term.

To control for programming experience, one of the major confounding parameters in program comprehension experiments, we apply a questionnaire to measure it [6]. Based on the value in the questionnaire, we can apply a control technique (e.g., create two groups with comparable programming experience). In addition to measuring program comprehension, the search behavior, and first action for a task, we use a questionnaire to assess the opinion of participants regarding difficulty of tasks and motivation to solve a task (both on a five-point Likert scale [16]). This way, we get more information to interpret our data.

To present source code, tasks, and the questionnaire to participants, we use the tool PROPHET [7]. It lets experimenters create tasks, specify how participants see source code, and logs the data (e.g., response time, actions of participants). Furthermore, it automatically sends the data to a specified e-mail address.

3.3 Tasks

We developed five bug fixing tasks, such that we can evaluate the claimed benefit of physical separation of concerns. Hence, the class in the FeatureHouse version that contains

tax requires considerable training, so we use FeatureHouse instead, and leave evaluation of physical separation of concerns á la AspectJ for future work.

the bug is relatively small compared to the ifdef version. To get an impression of how short source code has to be to provide a benefit (if any), we introduced the bugs in classes of different size. All tasks were designed to have comparable difficulty, so that it does not confound the results. We encourage other researchers to evaluate the comparability of both tasks and give us feedback. Additionally, we evaluate whether comparing similar statements of different features helps to find a bug (Task 2). Furthermore, we analyze how the need to consider two classes of different features affects program comprehension (Task 5). We designed only 5 tasks to avoid a too long duration. In our experience, 2 hours is the upper limit for an experiment; after that, participants lose motivation and/or concentration, and/or become fatigued.

To present the tasks, we gave participants a bug description as a user might provide it. Additionally, we provided the feature that is present when the bug occurs, so that participants can focus on feature code. This way, we can evaluate our research question, because cohesion refers to feature code only. In Table 1, we provide an overview of all tasks. To complete a task, participants are instructed to determine the class and line number of the bug, describe why the problem occurs, and suggest a solution as verbal description. We use all information to determine whether a task was solved correctly. To measure comprehension, we analyze correctness and response time of a solution. The more correct answers and the smaller the response time of participants, the better they understood source code. Next, we describe each task in detail, show relevant code fragments with bugs highlighted, and discuss whether the FeatureHouse or ifdef version might provide benefits for comprehension.

Task 1

In this task, instead of setting the counter to the actual value, it is set to 0. To illustrate this bug, we show relevant source code in Fig. 2. The class that contains the bug is considerably smaller in the FeatureHouse version, such that the complete class fits on one screen. However, the original method definition in the base feature might be relevant to understand the bug. Thus, participants of the FeatureHouse group might be faster, if they do not look at the base code, or slower, if they do not look at the base code.

Task 2

In Task 2, a false identifier is used (`SHOWPHOTO` instead of `PLAYVIDEO`). We show an excerpt in Figure 3. Like in Task 1, the FeatureHouse version is considerably shorter. However, in the Ifdef version, source code for other features (e.g., Photo) are visible, which participants might compare with feature Video and, thus, may help them to recognize that `SHOWPHOTO` is the wrong identifier to play a video. Another difference is the location at which the command is defined. In the FeatureHouse version, command definition and usage appears on the same screen, but not in the ifdef version. Thus, we can argue both in favor of and against a benefit for program comprehension in the FeatureHouse version.

Task 3 and 4

Task 3 and 4 are similar to Task 1, so we do not show source code here. In Task 3, the target is class in the FeatureHouse version is too large to fit on one screen. Thus, a possible benefit due to shorter classes might not occur here or be weaker.

```
1  public class MediaUtil {
2  // 73 lines of additional code
3      public MediaData getMediaInfoFromBytes(byte[] bytes)
4              throws InvalidArrayFormatException {
5  // 64 lines of additional code
6          MediaData ii = new MediaData(x.intValue(),
7              albumLabel, imageLabel);
8  // 5 lines of additional code
9
10         // #ifdef includeSorting
11         ii.setNumberOfViews(0);
12         // #endif
13 // 62 lines of additional code
```

(a) Ifdef

```
1  class MediaUtil{
2      private MediaData createMediaData(String iiString, String fidString,
3              String albumLabel, String imageLabel) {
4
5  // 16 Lines of additional code
6          MediaData ii = original(iiString, fidString,
7                  albumLabel, imageLabel);
8
9          ii.setNumberOfViews(0);
10         return ii;
11     }
12
13 // 10 lines of additional code
```

(b) FeatureHouse–Sorting

```
1  public class MediaUtil {
2  // 121 additional lines of code
3    private MediaData createMediaData(String iiString, String fidString,
4        String albumLabel, String imageLabel) {
5
6      Integer x = Integer.valueOf(fidString);
7      MediaData ii = new MediaData(x.intValue(), albumLabel, imageLabel);
8
9      return ii;
10   }
11 // 47 additional lines of code
```

(c) FeatureHouse–Base

Figure 2: Bug location for Task 1 (bug highlighted).

Task 5

In Task 5, we implemented the additional feature AccessControl to observe how participants can trace source code. The feature introduces rights to manage pictures, so if users have no rights to delete a picture, they cannot delete it. As bug, we use a wrong label for deleting a picture, such that the check for according rights is never executed and a user can delete a picture without according rights (Figure 4). The definition of the correct label is in another class, so two classes have to be looked at to locate the bug. In the FeatureHouse version, the two classes are located in different feature modules, which might slow down participants.

Additionally, we designed a warming up task to let participants familiarize with the experimental setting. In this task, participants should count the occurrence of a feature (ifdef version) or how often a class is refined (FeatureHouse version). The result of this task is not analyzed.

3.4 Analysis Methods

To analyze the data, we use descriptive statistics (mean, standard deviation, frequencies, and boxplots) to describe response time, correctness, search behavior, and first action for a task. This way, we get an overview of how that data are

Task	Bug Description	Feature
1	When converting media, the counter that describes how often a medium was looked at is always set to 0 instead of the actual value.	Sorting
2	When a video should be played, the according screen ("Play Video") is not shown. Nothing is happening	Video
3	When clicking on "View Favorites" in the menu, no favorites are shown, although there are favorites and the according functionality is implemented	Favourites
4	When pictures should be shown sorted by number of views, they appear unsorted anyway.	Sorting
5	Although a user has no rights to delete a picture, she can delete it anyway.	AccessControl

Table 1: Overview of maintenance tasks

```
1  public class MediaListScreen extends List {
2    // #ifdef includePhoto
3    public static final int SHOWPHOTO = 1;
4    //#endif
5    // #ifdef includeVideo
6    public static final int PLAYVIDEO = 3;
7    //#endif
8  // 64 additional lines of code
9      public void initMenu() {
10         // #ifdef includePhoto
11         if (typeOfScreen == SHOWPHOTO)
12             this.addCommand(viewCommand);
13         //#endif
14 // 7 additional lines of code
15         // #ifdef includeVideo
16         // [NC] Added in the scenario 08
17         if (typeOfScreen == SHOWPHOTO)
18             this.addCommand(playVideoCommand);
19         //#endif
20 // 32 additional lines of code
```

(a) Ifdef

```
1  class MediaListScreen {
2      public static final Command playVideoCommand =
3              new Command("Play Video", Command.ITEM, 1);
4      public static final int PLAYVIDEO = 3;
5
6      public void initMenu() {
7          original();
8
9          if (typeOfScreen == SHOWPHOTO)
10             this.addCommand(playVideoCommand);
11     }
12 }
```

(b) FeatureHouse

Figure 3: Bug location for Task 2 (bug highlighted).

distributed. To evaluate the first research question, we analyze whether there is a difference in correctness and response time. For correctness, we use a χ^2 test, since we compare frequencies. For response time, we use either a t test, or, if our sample is smaller than 30 participants and response times are not normally distributed, a Mann-Whitney-U test (all tests are described in Anderson and Finn [1]).

For the second research questions, we compare the frequencies of local and global search within groups and between groups with a χ^2 test. For the third research question, we can either use a qualitative analysis, or compare frequencies of different actions with a χ^2 test, if expected frequencies are larger than 3.5 (cf. [1]).

```
1  public class MediaController extends MediaListController {
2  // 14 additional lines of code
3      public boolean handleCommand(Command command) {
4          // #ifdef includeAccessControl
5          if (label.equals("Delete Label"))
6              if (!AccessController.hasDeleteRights()) {
7                  gotoAccessDeniedScreen();
8                  return true;
9  // 467 additional lines of code
```

(a) Ifdef

```
1  class MediaController {
2      public boolean handleCommand(Command command) {
3          if (label.equals("Delete Label"))
4              if (!AccessController.hasDeleteRights()) {
5                  gotoAccessDeniedScreen();
6                  return true;
7  // 16 additional lines of code
```

(b) FeatureHouse–AccessControl

```
1  public class MediaController extends MediaListController {
2  // 8 additional lines of code
3    public boolean handleCommand(Command command) {
4  // 43 additional lines of code
5        /** Case: Delete selected Photo from recordstore * */
6      } else if (label.equals("Delete")) {
7        String selectedMediaName = getSelectedMediaName();
8  // 169 additional lines of code
```

(c) FeatureHouse–Base

Figure 4: Bug location for Task 5 (bug highlighted).

4. PILOT STUDY

To evaluate the feasibility of our design and provide some first data to evaluate our research question, we conducted a pilot study. Our participants were 8 students (graduates and undergraduates) from the University of Passau with a mean age of 23. They were enrolled in the course *Contemporary Programming Paradigms*, in which preprocessors and FeatureHouse were taught with comparable level of detail. Thus, participants have comparable, necessary knowledge regarding both technqiues to complete the tasks. No participant was familiar with MobileMedia. All were aware that they took part in an experiment and that their performance does not affect their grade for the course. Participants volunteered to take part and did not receive compensation for their participation.

To create two comparable groups, we applied a programming-experience questionnaire a few weeks before the experiment [6]. Not all participants who completed the questionnaire showed up for the experiment. Thus, both groups differ in their programming experience. We discuss this prob-

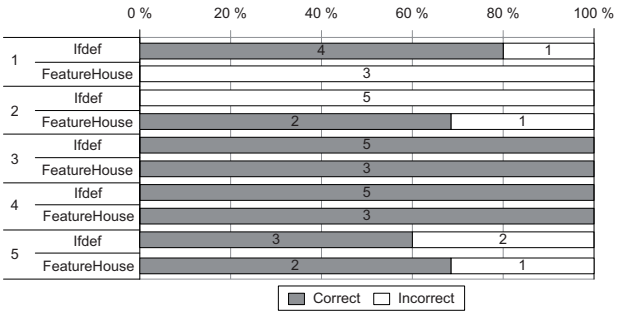

Figure 5: Number of correct answers per group and task.

lem in Section 5. Furthermore, we assessed participants' experience with Java on a scale from 1 to 5; both groups have a medium experience (3).

We conducted the experiment at the University of Passau in one computer lab instead of a lecture session. Before the experiment, we gave participants an introduction about what to expect. After all questions were answered, participants started to work on the tasks on their own.

4.1 Results

First, we evaluate program comprehension by analyzing correctness, response time, search behavior, and first action for each task to shortly address the research questions. To separate reporting data from interpreting them, we only report the data here and discuss them in Section 4.2, in which we also discuss the feasibility of our design.

4.1.1 Correctness

First, we look at correctness. In Figure 5, we give an overview of the number of correct solutions. The third and fourth task appear to be easy, because all participants found the correct solution. The first task appears to be too difficult for the FeatureHouse group, because no participant found the correct solution. The same counts for the second task for participants of the ifdef group.

4.1.2 Response Time

Second, we look at the response times. In Table 2, we show how long participants needed to solve each task and all tasks together (in minutes).[2] For most of the tasks, the ifdef group was faster; only for the second task, the FeatureHouse group was faster. The difficulty seems to vary, because the response times differ between tasks.

4.1.3 Search Behavior

In Table 4, we show how often participants used the search feature (local, global, and combined). Participants of the ifdef group used the search considerably more often than participants of the FeatureHouse group. For the local search, participants always used it more often than the global search.

4.1.4 First Action

In Table 3, we summarize how participants started to solve a task. Participants of the ifdef group most often used a global search to find code fragments of the relevant feature, whereas participants of the FeatureHouse group most often opened a file in the relevant feature. Additionally, in tasks where a label of a button is mentioned in the bug description, some participants searched for that label. However, they did not start to search for the label in the first task where it is mentioned (Task 2), but only for the subsequent tasks. Furthermore, two participants of the FeatureHouse group started in a wrong feature (SortPhoto). We believe this is caused by the fact that also feature SortPhoto (in addition to Sorting) sounds relevant for the task.

[2]Since our sample consists of only 8 participants, we do not compute standard deviations. Instead, the interval between minimal and maximal value can be used as estimator for dispersion.

Task	Group	RT	Min	Max
1	Ifdef	12.41	3.86	16.17
	FH	14.03	7.84	17.42
2	Ifdef	22.79	9.53	48.14
	FH	13.06	10.86	14.41
3	Ifdef	8.2	7.29	9.49
	FH	12.77	8.98	16.53
4	Ifdef	4.16	2.14	7.86
	FH	9.53	6.47	11.42
5	Ifdef	7.27	2.95	12.27
	FH	12.38	6.08	18.08
All	Ifdef	54.83	42.17	66.58
	FH	61.77	53.99	69.60

RT: response time in minutes, Min: fastest response time, Max: slowest response time, All (last row): response time for all task combined.

Table 2: Response times of participants per task.

Task	Group	Local	Global	Combined
1	Ifdef	166	21	187
	FH	32	13	45
2	Ifdef	152	25	177
	FH	28	13	41
3	Ifdef	106	11	117
	FH	39	19	58
4	Ifdef	21	5	34
	FH	16	7	23
5	Ifdef	73	8	91
	FH	25	12	37

Table 4: Search behavior of participants per task.

4.1.5 Opinion of Participants

Regarding the opinion of participants, we find a tendency that the ifdef group found the tasks easier to solve, except for Task 2. For motivation, there is a tendency that participants of the ifdef group are more motivated to solve a task. This tendency might be caused by the fact that two participants of the FeatureHouse group were unhappy to be in that group (as they told us). Thus, the FeatureHouse version appears more difficult to participants and they did not like it. This can affect their performance, such that they work slower [20].

4.2 Interpretation

Since our sample is too small and the ifdef group is more experienced, we cannot meaningfully interpret the effect of physically and virtually separated concerns. Except for Task 2, the faster response time of the ifdef group could be caused

Task	Group	Open file in base	Open file in relevant feature	Open file in wrong feature	Global search for relevant feature	Global search for label
1	Ifdef	-	-	-	5	-
	FH	1	1	-	1	-
2	Ifdef	1	-	-	4	-
	FH	-	2	-	1	-
3	Ifdef	-	-	-	3	2
	FH	-	1	-	-	2
4	Ifdef	-	-	-	2	3
	FH	-	-	2	-	1
5	Ifdef	-	-	-	5	-
	FH	-	2	-	1	-

Table 3: First action participants used to solve each task.

by the higher experience. Thus, our interpretation is only a suggestion for future experiments.

Regarding the search behavior, we found that participants of the ifdef group used the search function considerably more often than participants of the FeatureHouse group. Additionally, all participants used the local search more often than the global search. There are two interesting facets regarding the search behavior of the FeatureHouse group. First, for the second task, in which the class containing the bug consists of only few lines, participants used the global search more often. Second, for the last task, in which two classes in two different folders needed to be located to find the bug, the global search is used only half as much as the local search (similar to the search behavior for the other tasks). Thus, this tracing task seems to have comparable effort compared to the other tasks. Based on our data, we can split our third research question regarding the search behavior into three questions:

RQ_{3-1}: Do participants of the ifdef group use more search than participants of the FeatureHouse group?

RQ_{3-2}: Do participants of the ifdef group use more local search than participants of the FeatureHouse group?

RQ_{3-3}: Do participants of the ifdef group use less global search than participants of the FeatureHouse group?

Regarding the first action to solve a task, participants of the ifdef group most often search for feature code with a global search, whereas participants of the FeatureHouse group opened a file in the relevant feature (or features that appear relevant). Thus, we might conclude that participants use different strategies to solve a task.

Nevertheless, we found evidence about the feasibility of our design. Participants always understood the tasks and questionnaire and knew what they had to do. Only on two occasions, participants talked to each other, but the experimenter reminded them to work for themselves. Furthermore, two participants mentioned being unhappy to be in the FeatureHouse group. Thus, when conducting the experiment, it might be useful to motivate participants of the FeatureHouse group about the benefits of FeatureHouse. However, we have to take care not to bias participants toward preferring FeatureHouse or preprocessors, because this might bias the results. Besides that, no problems occurred. Thus, the task descriptions and questionnaires seem to be clear to participants.

However, we found that two tasks (3 and 4) appear to be too easy, because all participants solved it correctly. Hence, when replicating the experiments, it might be useful to increase the difficulty of these tasks. For example, for Task 3, providing the label might have made the task too easy, because it occurs only 2 times in the complete project. For Task 4, we can provide an erroneous implementation of bubble sort, instead of a `TODO` in the empty method body. Furthermore, we found that one participant spent 48 minutes on Task 2. Thus, it might be useful to set a time limit for each task, for which the response times in Table 2 can be used as orientation.

5. THREATS TO VALIDITY

When designing and conducting experiments, threats to validity are unavoidable and have to be reported. In our design, several threats occur. One threat is how we obtained the modular version of MobileMedia. Basically, it was derived from the AspectJ version by refactoring. Although the refactorings and the resulting code have been reviewed carefully, it is unclear whether designing and implementing a system like MobileMedia from scratch in a feature-oriented way would have led to a different more favorable decomposition, possibly making use of more effective modularization patterns. Exploring such patterns and related anti-patterns empirically is an avenue of further work.

A second threat is caused by the sample. When comparing techniques, we have to ensure that participants have comparable familiarity with them. Otherwise, we would measure differences in familiarity, not in the comprehensibility of both techniques. To control this threat, we recruited students from a course in which FeatureHouse and preprocessors were taught. Thus, we can assume that all participants have comparable knowledge of the evaluated techniques. However, we have to be aware that recruiting students means that our results are only valid for students. If we want to draw conclusions about experts on FeatureHouse and preprocessors, we need to recruit expert programmers.

For the pilot study we conducted, our sample is too small to draw sound conclusions regarding how research questions. Thus, we used the data as evidence for the feasibility of our design rather than to evaluate our research questions. Furthermore, the FeatureHouse group is less experienced than the ifdef group and mostly unhappy to work with the FeatureHouse version. Thus, worse program comprehension of the FeatureHouse group may be caused by lower experience

or happiness, not the underlying technique. To avoid misinterpreting the data, we only carefully described tendencies regarding benefits and drawbacks of physically and virtually separated code.

6. CONCLUSIONS

Separation of concerns is supposed to improve program comprehension. However, there are no empirical studies that evaluate comprehensibility of physically separated code. To close this gap, we presented an experimental design to compare program comprehension of physical and virtual separation of concerns. We refactored the ifdef version of MobileMedia (virtually separated) to a FeatureHouse version (physically separated). In a pilot study with 8 students, we showed the feasibility of our design. Our next step is to replicate the experiment with a larger sample. Furthermore, we encourage other researchers to replicate our experiment. With sound empirical results, we can give recommendation on which technique of separating code is suitable for which task and how separation of concerns can be improved.

7. ACKNOWLEDGMENTS

Thanks to the reviewers who helped to improve this paper. Siegmund's work is funded by BMBF project 01IM10002B, Kästner's work partly by ERC grant #203099, and Apel's work by the German Research Foundation (AP 206/2, AP 206/4, and LE 912/13).

8. REFERENCES

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

[2] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 221–231. IEEE, 2009.

[3] S. Apel, C. Kästner, and S. Trujillo. On the Necessity of Empirical Studies in the Assessment of Modularization Mechanisms for Crosscutting Concerns. In *ACoM '07: Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques*, pages 1–7. IEEE, 2007.

[4] A. D. Baddeley. Is Working Memory still Working? *The American Psychologist*, 56(11):851–864, 2001.

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

[6] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring Programming Experience. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 73–82. IEEE, 2012.

[7] J. Feigenspan and N. Siegmund. Supporting Comprehension Experiments with Human Subjects. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 244–246. IEEE, 2012.

[8] J. Feigenspan, N. Siegmund, and J. Fruth. On the Role of Program Comprehension in Embedded Systems. In *Proc. Workshop Software Reengineering (WSR)*, pages 34–35, 2011.

[9] E. Figueiredo, N. Cacho, M. Monteiro, U. Kulesza, R. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270. ACM, 2008.

[10] G. Heineman and W. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

[11] A. Jedlitschka, M. Ciolkowski, and D. Pfahl. Reporting Experiments in Software Engineering. In *Guide to Advanced Empirical Software Engineering*, pages 201–228. Springer, 2008.

[12] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *McGPLE '08: Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40. Department of Informatics and Mathematics, University of Passau, 2008.

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

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Mae-da, C. Lopez, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

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

[16] R. Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[17] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 169–194. Springer, 2005.

[18] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *FSE '04: Proceedings of the 12th International Symposium on Foundations of Software Engineering*, pages 127–136. ACM, 2004.

[19] G. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63(2):81–97, 1956.

[20] D. Mook. *Motivation: The Organization of Action*. W.W. Norton & Co., second edition, 1996.

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

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

[23] P. Tarr and H. Ossher. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 729–730. IEEE, 2001.